零 认清本质

新加坡国立大学计算机系有两门课：CS 1101 / 1102。

几乎所有的大学计算机系课程都有两门类似的课程；但几乎所有的学生都误解了这两门课；以为前者是教C，后者是教Java；但实际上前者是 Programming Methodology 后者是 Data Structure and Algorithm。

所以这两门课可以有选择，1101c 或者 1101s，使用不同的语言作为媒介。语言并不重要。

只要掌握了编程的思想、数据结构、算法，使用不同的语言去表达是很容易的。

会了很多种电脑语言后，学一门新的编程语言，几乎只要花一个晚上看看官方的语法文档就可以立刻开始使用做东西了。最多就一个星期。

基本上，那些说长时间说自己在学C#，学java的程序员，都是2B程序员，他们完全不懂得程序开发中“思想”、“数据结构”、“算法”的本质，而将大量的时间耗费在语言实现的细枝末梢中，纯粹浪费自己时间。

不同的语言会有不同的特性，有一些特性是比较重要的，普遍存在于多种语言当中的，“学习”一种新语言，实际上仅需要查看文档，看这种语言是以怎样的语法支持这些特性而已。

=========

OO是影响很广的编程概念，基本上，是Enterprise Developer（注：企业级开发者）的圣经、法则。

ED认为，越OO越好。

基本上，计算机业界有两批人，一批是真正的程序员，或者说hacker，一批就是ED。

ED实际上是企业的工具，他们很少有自己创新的想法；企业说啥米，就做啥米。所以，会有大量的vender，提供工具、支持、新技术，去train这些ED。

典型的vender有微软、IBM、Oracle等等；这些vender为了向企业推销产品，他们就经常会鼓吹一些新的“技术”，然后打包成为解决方案，推销给企业。

为了鼓吹、宣传这些技术，还有一批企业是专门在“布道”的，他们是所谓的“咨询公司”。

这样的咨询公司，他们会专门聘用一些所谓“Evangelist”，屁事不做，整天四处布道，名头都很牛逼，如XX金牌讲师。

他们实质上，就是推销员，只是，他们推销的产品，是所谓的“新技术”而已。

微软在新加坡好像就招了不少Evangelist 。每隔几年，微软所推广的技术就会“革新”一次，Evangelist们就不断的四处去宣传新技术改变了一切，能够提高效率无数倍。

Evangelist本身的技术，很多是很差的；就好像推销员本身，是不会做产品开发、不懂技术的。他们仅仅是会宣传、鼓吹新技术而已；满口各种新技术名词，但他们本身，可能仅仅只是会使用这些技术写一个Hello World。

因为他们本身素质很差，所以，他们是无法分辨他们所推广的技术本身是否好，他们只是复读机。有时候，vender本身在推的技术也其实不错，但复读机们也会把它夸张到荒谬的地步。

OO就是一个典型。

OO仅仅是无数编程模型中的一种而已，但它被过度的夸张，诠释。

Hacker们写程序，基本不会去追求程序本身是否符合OO规范。Hack这个词的意义本身就在于打破规范。

但是，大多数的ED是很笨的，他们缺乏独立思考的能力，他们需要被Train，而无法自学。Hacker的那套，他们接受不来。

所以，才会有vender / consultant / 培训学校一系列的产业，去鼓吹：

OO、XML、SOAP、Web Service、Silverlight等等一系列伪技术。

有的ED，一辈子都无法意识到他们实际上是中了vender的圈套；无法掌握真正的编程技术，而沉迷于vender们所鼓吹的“新技术”，一代接一代。

然后，只要有其中的一代技术ED没能掌握，ED就立刻被淘汰了；因为这种ED，穷其一生都没有学会真正的编程；他们仅仅是学会了一代又一代的被封装的伪技术使用技巧而已。

伪技术的典型特征是封装。

它本身没有任何新的东西，只是把旧的技术封装一下，换汤不换药而已。

OO是最好的封装技术；所以它被无底线的推崇。

封装很重要；但是，对于程序员来说，掌握封装技术本身，跟学习使用别人封装好的技术工具；是两回事。

“程序员从此不再需要关心XXX”，这是evangelist最常用的宣传语句；2B ED，看了就很高兴，然后拼命去学习新的“技术”，把他们曾经掌握的XXX底层技术给忘掉。

微软所宣传的理念被Hacker理解为“Even monkeys can code”。ED被evangelist鼓吹的新技术洗脑，最终就是成为[monkey](http://en.wikipedia.org/wiki/Code_monkey)而已；所做的工作，毫无技术含量；很容易被淘汰。

所谓的程序员30岁必须转行这种说法，便是源于ED被洗脑。

这种ED，从未掌握真正的编程技术，是必然被淘汰的。

=========

而这种ED，在大学时，就是把cs 1101 / 1102理解成为教 c / 教 java的那群人。

他们，从一开始就走错了。

=========

作业（编辑说明：在技术宅和他老婆的故事中，只有女主人公完成作业之后，男主人公才会发出新课程。当然，身为看客的您可以无需完成这些作业，但如果您仍是学生，或者您正在带学生或小弟的话，倒是可以做个参考）：

1. 用500字讲述什么是Programming Methodology?

2. 列举10种Data Structure.

3. 列举10种Algorithm.

一 认清实质

“算法”、“数据结构”等，是本质；很重要，需要掌握，但一般开发时，很少需要自己去实现。

觉得多数开发，是“拚积木”。

即便是业务逻辑需要对一些数据进行排序，也不可能自己去实现一个quicksort算法；而是直接调用quicksort的现成类库。

这也直接造成了2B ED穷其一生都不能掌握真正的编程能力。

他们认为，能够“解决”问题就好，至于问题是怎么解决的，他们并不关心。

对于细节的认识、掌控能力，直接造成了水平的天渊之别。

以拍照为例子，以前人们用傻瓜相机，现在人们用iPhone去拍照；很快，很方便，还可以加滤镜。

但是，普通人们在不了解什么是光圈、精深、背光等概念的情况下，是没有可能成为摄影师的。

即便他们放下iPhone拿起DSLR。

普通人跟摄影师拍摄同样的东西；出来的照片也许会差不多，但如果深入去比较，景深、角度、光线、取景等等等等细节，则都会有差别，而这些差别积累起来，就造成了普通照片与摄影作品的差别。

画家要画好画，必然要对画笔、颜料、纸张的特性有深入的了解。

厨师要做好菜，必然要了解食材的特性，对调味料、厨具等有娴熟的掌控。

ED的“解决问题就好”，跟没有下过厨房的千金小姐拿着菜谱使用微波炉做菜没啥区别。

在大厨手里，微波炉也可以是神器；但：

“有的人，纵然神刀在手，亦无法成为刀中之神。”

程序员要“拚好积木”，那必然需要对积木的种类、材质、特性，有深入的了解。

总得对quicksort的实现有认识，才能够用好quicksort。在有的场景下，quicksort的性能反而是最差的。如果不了解，就无法去把quicksort用好。

程序开发中，有一个著名的 80 / 20 原则。

我想，这个原则也可以适用于ED。

程序员只要花20%的努力就可以成为一个混日子的ED；80%的程序员均是如此。

但如果要成为一个优秀的程序员甚至hacker，那么，需要花多至少4倍的努力。

**有什么积木可以用？积木本身是怎么做的？积木A比积木B好在哪里**？

这些，是需要花大量的时间去了解。

全部都是实在的经验积累，没有捷径。

都是.NET语言，C# 跟 VB.Net的差别在哪里？对于ED，他们偶尔也会对这样的问题感兴趣，然后，他们会去看介绍，看比较文章。。。。但其实，这事完全是木有用的。

**他们看了别人的介绍，以为自己懂的，但实际上，他们只是在复读而已，完全木有懂**。

作为一个ED，要了解C#跟VB.Net的差别在哪里，最好的方式，就是花时间去把两种语言都学了。用这两种语言分别去写个几万行程序，然后就懂了。

当某天ED成为Hacker的时候，那就反倒可以去看各种介绍，看一眼，然后瞬间就可以悟了。

**这也就是为什么很牛程序员学习新语言可以那么快，因为有太多的知识可以复用；而这些知识的积累，必然是需要通过在实际中，无数行的实际编码，无数篇的资料阅读中得来的。**

没有捷径。

**很多初学者，或者说，编程的伪爱好者，他们，会热衷于去四处请教大师，下载各种经典书籍，企图读一本编程圣经，然后一夜脱胎换骨。**

**这是，不可能的。**

这种伪爱好者，永远不可能成事；在学习的过程中，抱着去“走捷径”的心态，本身就已经是入了歧途；最终会花更多的时间。

原来Ruby / 现在 Python的一个光头大牛Zed A. Shaw，为了表达“没有捷径”这样的观点，特意写了本《Learn Python The Hard Way》：  
<http://learnpythonthehardway.org/>

甚至有一个系列：<http://learncodethehardway.org/>

从长远来看：The Hard Way Is Easier。

我完全同意。

作业：

1. 列举10个Python Web框架

2. Python有多少种不同的解释器？

3. Perl 跟 Python 有什么不同？

二 怎么看待牛人

请看这个帖子：  
http://blog.csdn.net/hu\_zhenghui/article/details/7184799

快速浏览即可，无需细读；浏览过后再继续往下看。

读后的感觉是不是：

**“虽然不知道在说什么，但是看起来很厉害的样子！”**

整篇文章的关键是在这句：

“作者胡某某。曾任完美时空（现更名为完美世界）顾问，承担互联网方面的部分管理工作。现在主要精力研究互联网产品设计，是Axure授权的高级咨询顾问和高级培训讲师。”

这也就是，我在第一课中提到的“啥事不做，整天四处布道，名头都很响亮，如XX金牌讲师”，“Evangelist本身的技术，很多是很差的；就好像推销员本身，是不会做产品开发、不懂技术的。他们仅仅是会宣传、鼓吹新技术而已”。

碰巧今天看到这个非常有代表性的帖子；整个帖子看下来，作者毫无海量数据处理实际开发经验，纯粹堆砌这些流行技术名词而已。他没有用过这些技术，随便乱丢技术名词，整篇似是而非，必然的结果就是：“虽然不知道在说什么，但是看起来很厉害的样子！”

学习技术的人，如果受了这种“看起来很厉害的样子！”的蒙骗，会走很多很多弯路。

**那么，如何识别“看起来很厉害”跟“真的很厉害”？**

就好像，CSDN虽然有些忽悠人的文章，但也是有些好的文章在里面，如何辨别？

**1. 看得多了，自然会分辨。**

研发知识的最好来源之一是技术博客，就我自己而言，看了博客园自创办伊始前5年的所有首页文章；外加常年订阅400+博客，twitter fo 400余人等。

我这么做，主要是因为看得快；没有“看不过来”的问题；但实际上是个很笨的办法。

要保持最新技术的了解，确实是需要看很多blog；**除此之外，我想不出别的途径；但这并非必要。**

**2. 看书**

多看，最大的好处是了解最新技术，而且这是很土的方法。很多时候，并不需要了解很多“最新技术”；很多“最新技术”都是属于第一课中所讲的“封装技术”，不了解，也完全没有关系。

计算机的经典好书并不多，好书是公认、经得起时间考验的。

看完这个豆列也就差不多了：  
<http://book.douban.com/doulist/995755/>

完全可以不去理解“最新”的浮躁，去上面的豆列挑几本看，仔细的看，就可以脱胎换骨了。

就我自己而言，对我技术影响最大的一本书倒不在上面豆列的20本书中，而是：  
<http://book.douban.com/subject/1467587/>

经典书，是必须看，并且反复看的；如果说有什么“捷径”的话，看经典书就是最快的捷径了。

这些经典书中的思想，是永远不会过时的；任何时候看，都不会太晚。

给ED看的书也有经典：  
<http://book.douban.com/subject/1229954/>

首先，这是本好书；而且这本500多页书的传奇在于它讲了无数企业开发的模式，但其中的一页半讲述的：Active Record Pattern影响了过去5年多6年的Web开发潮流。

**3. 写代码 + 看代码**

学习编程，是一定要去编程的。

书、资料再好，光看不练；也很容易把自己看成傻子。

在实际项目中写代码；然后看别人是怎么做的。

别人，指的往往是开源项目；而不是Google搜来的某个不知名博客中贴的代码。哪个开源项目比较厉害，同样是有目共睹的。

做Web开发，几乎所有人都会去造ORM的轮子，没事，就去造一个；然后比较自己的版本，跟优秀的开源ORM在API风格、架构设计、实现细节上，有何不同。

三 架构比较

承接上文，12306的案例是蛮不错的题材；看过咨询师“很厉害的样子”，那么，究竟要如何做好 「海量事务高速处理系统」 这个方案？

“Hacker”提出了方案：

caoz，出自百度的超低调牛人：  
<http://hi.baidu.com/caoz/blog/item/f4f1d7caee09b558f21fe780.html>

云风，原网易杭州研究中心总监：  
<http://blog.codingnow.com/2012/01/ticket_queue.html>

同样的，也有另外一些“ED”在讨论方案：

林仕鼎，百度首席架构师，曾任微软亚洲研究院研究员：  
<http://qing.weibo.com/2244218960/85c41050330009xm.html>  
<http://weibo.com/2244218960/y0l4S7Y1d>

白硕sse，上海证券交易所总工程师：  
<http://weibo.com/1922397344/y0jMo9IaD>  
<http://weibo.com/1922397344/y0jP6jNRB>  
<http://weibo.com/1922397344/y0jUy2rkf>

且不论“Hacker”跟“ED”谁更加牛，从他们的**解决问题的手法、角度上看就非常不同**。

“Hacker”所追求的是解决问题，只要是问题被解决，怎么解决的无所谓；并发流量太大，系统处理不过来；caoz / 云风两种的方案，实质上都是直接去处理源头 - 避免并发。

caoz把高并发的请求直接分流去非主业务服务器，主业务服务器无需面临高并发；云凤则提出排队系统，避免高并发的出现。

而林仕鼎、白硕则是正儿八经的去讨论在有这样高并发的前提下，要怎么处理。

哥伦布的鸡蛋。

能够用手去扶住鸡蛋，“Hacker”绝对不会犹豫；而“ED”则努力的去把鸡蛋竖起来。

注意，牛“ED”未必就不懂得可以用手。

这样“Hacker”精神，在云风的blog上，还有另一个体现：屏蔽垃圾评论的验证码。

博客有很多垃圾评论，需要屏蔽，有很多很多种方式，各种神奇的验证码，叶贝斯规则过滤等等。

“ED”可以设计出来很多方案，并实现。

云风肿么做呢？

他在评论发表的时候，增加了一个项目：为了验证您是人类，请将六加一的结果（阿拉伯数字七）填写在下面

**“只要能解决问题，就采用最简单的设计。”**

这个验证码插件是我自己写的，只有一行 perl 代码。就是判断输入是不是 '7' 。

结果它很管用。从后台 log 看，拦截了几万条 spam 。”  
<http://blog.codingnow.com/2012/01/dev_note_7.html#comment-42161>

注意，牛的“Hacker”未必就不懂得做出庞大架构并实现。

“要如何做好「海量事务高速处理系统」这个方案”本身就可能是个伪命题，

「海量事务高速处理系统」这个需求本身可能根本就不存在。

**作业：**

1. 林仕鼎是百度首席架构师吗？

2. 看完caoz所有的blog。

四 SQL vs NoSQL

在几乎所有的web应用中，数据库都是核心的一环。

Web应用往往都是“Database driven”，业务、数据都是由数据库完成，而前端页面仅仅是演示、修改数据的一个“壳”。

因此很多web框架，都会标榜自己能够兼容多少多少数据库，做CRUD多么多么容易。

一般上，提到数据库的时候，指的都是关系型数据库；但关系型数据库并非唯一的一种数据库类型。

关系型数据库，一开始便是设计为通用，并有ACID支持的。

Atomicity 原子性、 Consistency 一致性、Isolation 隔绝性、Durability 持久性

杀手欧阳盆栽说：“**每件事都有它的代价**”。上述四个特性，都是有代价的。

对于严谨的商业应用，如银行、交易系统；为求业务的安全，他们不得不，或者说，能够并且愿意付出这些代价。

回到12306，后端数据库传说使用的是Oracle，而站出来说吐槽12306的行家往往都会提到 redis \ mysql 这样的替代。

有些菜鸟“ED”看到这些吐槽就出来喷了，说这些行家不懂神马业务安全性的重要，这帮做互联网的弱爆了，票务是必须使用 Oracle才能搞定云云。

好像还有人专门去喷了Fenng，这实在是太讽刺了。Fenng实际上是Orcale ACE Director<http://www.hudong.com/wiki/%E5%86%AF%E5%A4%A7%E8%BE%89>，国内屈指可数的Oracle专家。

很多人，特别是弱“ED”、“专家教授”，沉寂在自己所在的领域，然后就以为“悟”了；实际上，仅是把自己变成了井底之蛙。

知识的广博、全面性非常重要。

在**某个领域，通用的东西成熟之后，往往就会有专用的解决方案出现。而专用的解决方案多了之后，又会有新的通用解决方案出现。**

**天下大势，分久必合，合久必分。**

计算机，最早有很多专用系统，如王安打字机；个人电脑通用之后，这些专用设备就湮灭了；而iPad、手机的涌现，则又是专用系统。

是的，传统上需要去购买 Orcale、DB2 等巨贵无比的数据库系统，去满足业务需求；不是因为它们把问题解决到了极致，而是因为没有别的选择。时代已经变了，井底之蛙若把Oracle当成是王道，那只能被时代淘汰。

关系型数据库作为通用解决方案，是非常非常好的；它是一把神刀。

但是，它有以下问题：

**===== ED总是要写烂SQL ====**

首先. 还是那句话，有的人纵然神刀在手，亦无法成为刀中之神。关系型数据库提供的SQL能力，是高度抽象的，封装了无数层的。写SQL的人，太多太多根本不了解SQL背后所执行的事情；烂“ED”都是如此。

这甚至**造就了一个职业：DBA**。DBA去负责数据库微调、优化，听起来很高级，但实质上，就是给滥用SQL的“ED”**擦屁股**而已。

对于庞大的企业来说，管理者是知道大部分ED都弱爆了，他不期望也不需要ED去了解数据库，他只需要ED去完成最基本的业务功能，然后让DBA去给ED擦屁股。

大部分的ED，并没有意识到这一点；他们拼命去追求方便快捷的“搞定”；滥用SQL的各种高级功能；甚至，他们把分享SQL的复杂使用当成是乐事。

ED所努力的，是把自己变笨，把活尽可能的都交给神奇的数据库去解决；数据库怎么解决的，他们不关心；这实质上，是在削弱自己工作的技术含量，自我贬值而已。

工程师如果能够把数据库给用好了，哪里还有DBA什么事？

**DBA所谓的数据库优化，往往就是把工程师不负责任写下的2B SQL查询找出来，然后改写为文艺方式罢了。**

不要滥用数据库，一点都不难。

**===== 通用数据库性能有极限 =====**

其次，关系型数据库作为通用解决方案，它提供了太多的东西，它做了太多的事，而所有的事情，都有它的代价，直接而言，就是牺牲性能了。

所以，DBA的另一个职责，则是把数据库的各种参数调配好，让其能够发挥最高的性能。

从这个意义上去说，DBA的工作就不仅仅是给ED擦屁股了。

但，这样的微调，是有极限的。**DBA拚了命去把鸡蛋竖立起来，考虑了桌面摩擦、空气流动、手指颤抖等等因素，鸡蛋虽然可以竖立一会，但终究还是会倒下去；**这也就是微调的极限。

在某些场景下，是可以用手的：把业务中没有用到的数据库功能都去掉；甚至，去掉完整的ACID支持。

这样一来，数据库的性能就可以有数量级的改善了。

**===== 关键在于灵活性 ====**

最后，上面两点，其实都是跟性能相关的；关系型数据库即便作为通用方案，它的性能有极限，但也能够满足绝大多数应用场景了。关系型数据库的软肋，是在灵活性上。

数据库有表、而表有结构；而表的结构，在应用上线之后，很难修改。

这同样造就了一些专业学问：严密的业务分析、设计数据库结构、如何在数据库上线之后修改结构等等。

这些问题或者说学问之所以存在，是植根于关系型数据库表结构不灵活的前提之上。

再次”Think out of the box“，如果数据库可以做到灵活、随时修改的表结构呢？

**====== NoSQL ======**

关系型数据库的三个问题，被NoSQL全部解决了。

（同样的，所有事情都有它的代价；NoSQL在解决SQL固有问题的同时，也引入了新的问题；另一方面，NoSQL解决的也不仅仅是SQL的这三个问题。）

ED要写烂SQL？没有关系，彻底不让他们写SQL好了。

数据库支持功能太多？砍功能还不容易么？

Schema不灵活？那就schema-less好了。

目前，NoSQL的实现方案很多，MongoDB、Redis、Carssendra等等等等；每一个都可以非常不同，是专用解决方案：有自己独有的特性，去解决特定场景的特定问题。

（当然，像MongoDB，已经很有NoSQL通用解决方案的意味了。）

普通程序员用SQL，文艺程序员用NoSQL，2B程序员把NoSQL当SQL用。

普通程序员在从SQL切换去NoSQL时，会受固有的SQL知识限制，总有把NoSQL当成SQL去用的冲动，但这是非常2B的行为。

从微观的角度讲，原来SQL查询中所支持的各种神奇joining / groupby都不见了；拼命的想要去找在NoSQL数据库环境下同样的神奇工具。

这彻底违背了使用NoSQL的初衷：为了就是不让你滥用SQL的这些神奇功能。

滥用SQL会造成严重的性能问题，而在性能问题浮现之后，需要耗费更大的力气去纠正。

是的，信用卡透支购物很方便；但付账单的时候就傻逼了；所以，换成无法透支的借记卡。

固然没有了透支的便利，会有不方便，但彻底杜绝了还不起账单，被收取高额利息的问题。

要透支的便利？ED，请先去掌握好理财技能，彻底了解透支的影响，然后我们再来谈便利。

从宏观的角度讲，会有人企图去给NoSQL做封装，让NoSQL表现得跟SQL一样；甚至，去把NoSQL去掉的那些SQL功能加回去。

SQL已经是一个非常非常成熟的方案，它所能够解决的问题范畴里面，几乎没有办法做得比SQL更好。

在NoSQL的基础上，去试图模拟SQL，只能成为SQL的蹩脚模拟；还不如直接用回SQL。

在网路编程里面也有类似的例子，TCP跟UDP。可以把SQL看成是TCP，它是可靠、神奇的。UDP虽然不可靠，但是会比TCP更快。要做视频、语音通讯，UDP是更好的选择。**但要去做“不丢包、不失帧”的可靠视频通讯，选择在UDP的基础上添加确认、重发机制模拟TCP，那就是2B了。不是天才，没法做得比TCP更好，直接用TCP就好。**

作业：  
1. NoSQL的方案，如MongoDB还解决了SQL的什么问题？

2. NoSQL的应用场景有啥米？

五 设计模式

设计模式，应该是很多ED心目中牛B的编程方式。

上回说到ED的好书POEE，实际上便是一本专门讲企业开发中使用的设计模式中的书。

设计模式，并不多，基本上看完GoF的这边《Design Pattern》便可以有足够了解了。

而实际开发中常用的设计模式更是屈指可数，Singleton，Factory，Facade，Active Record、Provider等等。

就那么几个，花花功夫，仔细了解一下这几个，然后在实际编码中应用一下，便可以算是掌握了。

**设计模式，并不难。**

它是开发中非常必要的知识，实际上，是非常基础的知识，并不牛B。

开发的时候，需要时刻明确自己的目标：解决问题。

解决问题才是最重要的。

设计模式的存在，是为了更好的维护、管理代码，或者是为了扩展性；绝对不可以为了设计模式而模式。

在Java程序中，**为了模式而模式**的现象蛮普遍的。

我猜想这是因为Java是一门工业语言，有大量的ED使用的缘故。

ED把设计模式的使用，当成是一种可以炫耀的编程技巧，或者说，他们遵从的编码规范中，要求他们去使用某某设计模式。

至于为什么要使用设计模式，最常见的理由便是：为了将来可以XX，或者YY。

程序开发中，有一句名言：“Pre-mature optimization is the root of all evil”。

**过早优化，是万恶之源。**

非常多的情况下，将来预想中的XX或者YY；并不会发生。大部分代码，写了之后就会被丢弃掉，再也不会有人去维护。

当XX或者YY发生的时候，往往，都是会推倒重来。

除非你很牛B，只有牛到一定程度，才有可能对将来可能发生的情况做好合理的预测，并预留出改善、调整的空间。

但非常讽刺的是，**为将来做设计的最好方法就是：什么都不做。**

**只有空白，才能够留下最大的发挥空间。**

现在为将来可能发生的情况，做了编码，任何一行编码，都是很可能是在为将来添加限制、制造麻烦。

现在写下去的代码，将来，都是要被删掉的；能够不写，就不写。

在任何时候，都应该保持代码简洁。

函数，尽可能的短；当一个函数的长度，超过一个屏幕的时候，便应该考虑重构、拆分。

牛B的程序，都应该是简单、易懂的；采用大量的设计模式，复杂得让人无法直接看懂，或许有它的意义以及应用场景，但这绝对不是编程功力牛B的表现。

打个比方，设计模式就是武术招式。

学徒，必然需要熟悉什么“有风来仪”或者“屁股朝后平沙落雁式”。

但更高的境界是：无招胜有招。

简单、直接的把代码搞定。

Python大牛沈崴有云：“得道的程序员，既不封装，也没有重复代码。”  
<http://eishn.blog.163.com/blog/static/6523182010102342531684/>

**作业**：

1. 使用一种编译语言实现 Singleton 模式

2. 使用一种动态语言实现 Singleton 模式

3. 说说对 Provider 模式的理解。

六 模板引擎

设计模式再“高级”一点，便是所谓的“框架”了。

从事Web开发，一般都会接触到MVC框架这个概念。

M：也就是Model，直接跟网站数据库相关。

V：也就是View，是网页的模版，跟显示数据相关。

C：则是Controller，相当于网站的业务逻辑。

MVC也不仅仅是应用于网站开发，它的概念实际上植根于桌面软件，并且在手机软件开发上也有应用。

MVC本身是一个设计模式，是一个被验证过的，可以用来很好归纳、管理代码的软件开发方式。

基于这样的设计模式，提供了很多相关的类库实现，则“设计模式”升级为“框架”。

MVC的任何一个方面，扩展出去讲，都可以讲上几天几夜。

今天只讲V。

传统的ASP / PHP网站开发，V是很混乱的。

默认只有一种文件，html与业务逻辑代码混杂在同一个文件；相当难以维护。

ASP.NET相对于asp做出了很大改进，提出了code-behine的概念：默认将html的模版代码，以及c#或者vb.net的逻辑代码切分到两个不同的文件。

这样的方式算是有很大进步。

微软平台上做开发是比较苦逼的，微软掌控了整个开发平台的前进速度。

asp跟PHP在开始的时候，是相似的技术。有类似的便利，以及类似的麻烦。

微软推出了.net，推广了code-behind的模式；然后，所有的微软程序员都超着微软指定的这个方向去迈进。

asp被抛弃了，自从ASP.NET诞生之后，就不再有任何改进。

而PHP，在开源世界中，则不断的得到各式各样的改进。

各种模版引擎层出不穷；不仅可以实现code-behind这样简单的模版、业务代码分割；很多还直接引入了MVC的概念；实现了三层的分割。

而ASP.NET，则长期止步于web form的code-behind，在开源世界中的MVC方案大放光彩若干年后，才推出 ASP.NET MVC。

模版技术，最初的目的就是要把业务代码，也就是说，把获得数据的代码跟html分割。

在模版实现上，因此涌现了不少不同的设计哲学。

Python的Django框架中的模版，是一种典型。

它彻底的禁止程序员在模版中嵌入任何代码；模版中，只可以出现html；以及一些跟业务逻辑无关的控制标签，如：

1. {% If XXXX %} foo {% **else** %} bar {% end %}

条件XXXX，必须是一个数据值，不可以是一个复杂表达式、不可以包涵函数调用等等。

模版中，也不可以声明任何新的变量，下面的做法是被禁止的：

1. {% **set** i = 0 %}
2. {% **foreach** item **in** items %}
3. {% i += 1%}
4. <div>
5. {{ item }}
6. {% **if** i mod 2 == 0 %}
7. <hr />
8. {% end %}
9. </div>
10. {% next %}

Django的模版，从技术上彻底禁止程序员添加任何逻辑，强迫程序员必须在controller中去写各种逻辑，以确保模版内容的纯洁干净。

所以Django的模版，一般都非常简单，有很好的移植性，并且可以让网页设计人员直接编辑。

ASP.NET则是另一种典型；虽然有了code-behind，但是它没有对前端代码，以及后端代码做任何限制。

在前端aspx页面中，可以嵌入任意的逻辑代码，而code-behind的code，为空白；这种伪“code-behind”的方式，跟原来的asp没啥区别。

ASP.NET从框架本身，并不阻止程序员去做这样的事情，实际上，它还标榜它这样的特性：方便原有的asp项目直接升级到.NET的平台上。

也有另外一种奇葩的做法，前端aspx页面保持空白，然后在code-behind的code中去拼接所有的html。这样的方式，ASP.NET框架本身也不禁止。

只要ASP.NET程序员喜欢，没有什么不可以的。

ASP.NET把对模版使用方式的选择权留给了程序员，如果程序员自律，他们可以按Django模版那样的方式去使用模版，并拥有Django一样的优点；如果程序员自律？！

在某些可以通过嵌入代码去快速处理的场景，ASP.NET的模版也保留了程序员去hack的能力。

还有一些模版技术，则是折衷的（如tornado的模版）：允许嵌入单行代码，如声明变量，调用函数等等；但是不允许整块、整块的业务代码出现模版中。

上述**三种模版设计哲学，各有它们的道理，以及应用场景。**

**需要根据具体的业务、应用场景，才能说其中哪种比较合适。**

开发人员的能力也是直接相关的，如果团队中，普遍不自律；缺乏将业务、模版代码分割、以提高代码可维护性的意识，那么Django的做法是最好的，它直接禁止去滥用模版，强迫他们去使用更好的开发风格；即便在某些场景下会更麻烦。

武断的认为任何一种模版设计哲学是“最佳”的想法是极其肤浅的。

各种成熟的模版技术，一般也都会有包括以下特性：

1. 嵌入

也就是说，编写各种可以复用的小模版块，然后供多个不同地方调用；比方说，用户头像（甚至名片）的显示。

具体页面不需要重复编写这些重复的模块。

并且，这些模块需要调整时，只需要修改一个地方，便可以在所有地方生效。

2. 继承

能够编写一些基础模版，定义常见的页面结构。

具体页面继承这些基础模版，便不需要重复编写那些结构代码。

同样的，当页面结构需要调整时，也是修改一处，所有生效。

3. i18n

网页模版的国际化支持是一个模版引擎是否成熟的表现。

如果没有，当网站需要同时提供多种不同语言支持的时候，会很麻烦。

成熟的模版，都会提供内置的支持。

因为网页模版实现实在是太多了，大家功能也都差不多，那么性能，也就成为了相当重要的比较指标。

有的模版，能够“编译“，渲染起来快些。

一般可以简单认为，功能越多的模版，性能会约低。有的模版，甚至将i18n的支持变成可配置的，不需要的时候就可以关闭，以提高性能。

也有的模版认为，写 {% %} <%%> {{}} 这样的符号太麻烦了，可以直接忽略，它可以自动聪明的识别 html，以及模版控制代码。简单的说，就是以极其华丽的方式，去方便程序员少打几个字符。

还有的模版，在实现嵌入功能的时候，还可以选择所依赖的的css / js文件。

比方说，要显示用户的名片，需要引入 namecard.css；那么，可以在 namecard的模块文件中指定这个依赖，然后模版渲染的时候，自动把这个css的引用，放在html的头部。

直接在模块文件中写 namecard.css 的引用是很傻的，因为模块可以在模版中引用多次。重复引用同一个css文件是没有道理的。

种种模版功能细节，实际上，都是可以在没有模版支持的框架中去实现。

想想PHP，它本来是非常简单的，默认只能够在同一个文件中混杂逻辑与代码。

但一旦程序员有了追求，它也可以有模版实现。

模版不支持 i18n，程序员一般也是有办法在现有模版实现中添加相应的支持的。

并不复杂，关键是看程序员的态度；看程序员是否有把事情做得更好、更优雅的态度。

一般情况下，程序员选择去实现更多的模版功能的时候，必须先看看别人是怎么做的。比方说，如果完全不知道什么是gettext就去自行实现模版的 i18n 功能，是非常2B的。

绝大多数情况下，程序员面临的问题，都不是自己独有的，一定是别人已经解决过的问题。

是否有足够的见识，有足够的知识广度，了解别人的解决同样问题的做法是程序员能力的表现。

是否有快速的搜索出类似的解决方案，也是能力的表现。

1. PHP的Smarty 模版的设计哲学是什么？

2. Perl的Mason 模版的设计哲学是什么？

3. 什么是gettext?

4. 前端Javascript实现的模版中，目前最成熟的是哪个引擎？

七 运维的重要性

先摘录一段话勉励一下生日宝：

截止2010月6月，Facebook接近2000雇员。10个月时间从1100人增长到2000，一年时间员工人数翻了一番！

最大的两个团队是开发工程师和运维，都是400-500人的规模

猪头宝，在Facebook，运维跟开发是一样重要的。运维才不是用vender提供的软件，然后按manual去step by step的做事情。

有很多创造性的工作可以做。

猪宝你知道twitter是肿么更新服务器的么？

Twitter有几千台服务器，一旦网站要跟新，这几千台服务器上面的代码部署都要更新。

肿么让这几千台服务器快速的获得新代码呢？逐台服务器下载太慢了，数千台服务器同时向代码中央服务器获取新代码又会把中央服务器的带宽挤爆。

肿么办？

Twitter的运维工程师直接用了BT的协议，使用p2p下载来解决这个问题：  
<http://engineering.twitter.com/2010/07/murder-fast-datacenter-code-deploys.html>

它们就这样把部署的时间从原来的40分钟大幅减少到只要12秒～～

运维，很多时候都是要编写脚本，把很多原本需要人手工做的事情通过脚本自动化管理起来。

这些脚本乃至系统的编写与开发，都是需要能力的。

运维的投入，都是为了节约别人的时间；而时间节约、效率提高、稳定性提高，这些都是有意义的。

之前猪宝去广州参加技术沙龙，有一个人人网之类的运维去讲自己的工作木有意义，是“花卖白粉的心，赚卖白菜的钱”；当场就被金山的过程改进经理周琦 Zoom.Quiet给吐槽了。

说他的态度不对，运维部门应该是一个盈利的部门，而不应该是一个被无视的部门；运维，应该是通过提高技术水平，提高效率，节约成本；以达到“赚钱”的目的。

caoz很推崇的一个技术牛人杨建；便是做运维的，在新浪、腾讯呆过，现在应该是被caoz收去4399了：<http://blog.sina.com.cn/iyangjian>

如果木有杨建这样的运维高手，新浪是木有可能支撑起一小时近20亿实际http请求处理量的：<http://blog.sina.com.cn/s/blog_466c66400100cfrj.html>

Facebook的有9个级别的代码发布流程：<http://www.dbanotes.net/arch/facebook_how_facebook_ships_code.html> 这些都是运维的工程师牛B才有可能的；并且也确实解决了实际业务问题。

如果仅仅是做普通的SA，那么工作是很routine，很无聊，很没有技术含量的；但是如果能够提高，面临的问题是完全不一样的。

这跟烂ED与Hacker的区别也是一样。

实际上，很多**职业都是一样；如果是做那底层的普通工作，都必然是无聊的，木有意义的。但一旦有进步，层次提高，面临的就是完全不一样的环境**。

有做文书工作，收集、整理资料的律师；也有Alan Shore。

工作是否有意义，在于职位的层次。

八 控制器

设计模式再“高级”一点，便是所谓的“框架”了。

从事Web开发，一般都会接触到MVC框架这个概念。

M：也就是Model，直接跟网站数据库相关。

V：也就是View，是网页的模版，跟显示数据相关。

C：则是Controller，相当于网站的业务逻辑。

MVC也不仅仅是应用于网站开发，它的概念实际上植根于桌面软件，并且在手机软件开发上也有应用。

MVC本身是一个设计模式，是一个被验证过的，可以用来很好归纳、管理代码的软件开发方式。

基于这样的设计模式，提供了很多相关的类库实现，则“设计模式”升级为“框架”。

MVC的任何一个方面，扩展出去讲，都可以讲上几天几夜。

今天只讲C。

传统上，php / asp / asp.net web form等，使用的是所谓的 Page Controller Patterns:<http://martinfowler.com/eaaCatalog/pageController.html>

Page Controller简单的说，便是一个网址对应一个程序文件。

所以，我们会看到大量类似： show.php / show.asp / show.aspx 的网址存在，这样的网址，背后都有相应同名的文件。

这样的模式，是网站从静态转向动态是最自然的改变方便，也最为容易让初学者接受。

但随着网站的复杂化，这样的模式会慢慢显得不够方便；比方说，多个不同的网址，映射到相同的处理；比方说，处理的时候，复用共同的资源。

页面内容的动态化，同一个程序文件，显示的内容是动态生成的 - 根据不同的query string，生成不同的内容，如：show.php?id=1234

网页程序内部，实际上是需要解析网址中的query string，并做不同的操作。

这实际上是一个映射的过程，将网址映射到相应的处理。

为了方便做这样的映射，慢慢的出现了所谓的 Front Controller Patterns:  
  <http://martinfowler.com/eaaCatalog/frontController.html>

这是通过某种机制，将符合各种规则的网址请求映射到程序中的一个类，或者是一个函数处理。

一般上，是使用正则表达式解析网址，并映射。

将网址映射到一个类；

1. urls = ("/home", "hello")
2. app = web.application(urls, globals())
4. **class** hello:
5. def GET(self):
6. **return** 'Hello, world!'

将网址请求映射到类，是相对较“重”的处理方式，比方说，需要处理类的初始化等等。

有的框架，也可以是一个函数，则相对“轻量”一些：

1. (r'^$', 'home'),
3. def home(request):
4. **return** HttpResponse("Hello, world.")

类、函数，均各有优劣，但实际差异很小：

映射到类的方式，往往还会根据不同的HTTP header映射到类里面中相映的函数，比方说，将对 /home 的HTTP GET请求映射给 hello 类的 GET 函数；而对 /home 的 HTTP POST请求映射给 hello 类的POST函数。

这部分 url routing的设计与实现，各种语言、平台上的功能均向正则表达式靠拢，大同小异。

有的可能专门为 restful 做了优化，但即便木有，自行实现也并不复杂。

很多请求，都会有一些常用的默认处理，比方说，检查用户是否登陆，检查用户是否有权限等等。

这些业务控制逻辑，是完全可以复用的。

在Page Controller的场景下，一般是通过继承来实现；而Front Controller场景下，而一般通过函数修饰符的风格实现，如：

1. **class** UploadImgHandler(BaseHandler):
2. @tornado.web.authenticated
3. def post(self):
4. XXX

(上述代码，实际上既使用了继承，也使用了修饰符。)

Controller的改进，目的在于更加方便的维护代码、修改业务逻辑。

如果程序员有良好的开发风格，基本是使用最基础的php page controller，也可以达到类似的效果。

各种“先进框架”，实际上是将常用的模式抽象出来，并通过便利的约定方式向程序员开放；如果程序员缺乏维护代码的意识，也很可能将良好的约定习惯用滥。

需要了解的，是为什么各框架的controller设计会有这样的设计，并用好；而不是死板的遵循“开发指南”。

在简单业务场景下，实际上page controller会更加方便。

有这么一个“定理”：概念越简单的模式，在处理简单场景时，是越便利；但随着场景复杂化，简单的模式会越来越难以维护。

而概念相对复杂、高级的模式，处理简单场景时，会相对麻烦；但随着场景复杂化，则比简单的模式容易维护。

“复杂度是守恒”的：  
  模式简单，维护则复杂。  
  模式复杂，维护则简单。

一个复杂的地方变简单了，则另一个地方会变复杂；保持代码结构的清晰，不要自己给自己添麻烦。

什么叫自己给自己添麻烦？

普通复数形式，加s: pigs / cats / dogs

已经可以很好了，但偏生有人要增加不规则复数：

sheep / mice / wives

这种就是自己给自己添麻烦。

**作业**：

1. 说说对 restful 的理解

2. 什么是 reverse proxy ?

九 数据模型

这次来讲MVC中最后的M。

Model，几乎可以说是网页应用的核心。

之前课程提到过网页应用是由数据库驱动，而在很多场景，数据库 = M ； M = 数据库。

所谓的ORM； object relational mapping。

现在新的网页开发框架，特别是MVC框架，都会提供ORM支持，避免程序员直接写SQL、操作数据库。

传统上，ASP/ php臭名昭著的sql注入问题，便是因为菜鸟程序员直接在程序中根据用户输入拼接数据库造成的；而使用ORM框架，则可以彻底避免这种问题。

ORM有两种风格，一种是 R => O；一种是 O => R 。

====== R => O ======

传统上，程序员也都是先完成数据库设计（甚至是由DBA完成），然后再考虑相应的对象生成，也就是所谓的 R =＞　O。

在这样的场景下，整个软件的框架，还是以数据库为核心，业务的设计思维是以关系型数据库的表结构为基础去考虑的，具体应用实现上，会考虑很多关系型数据库的功能特性，比方说，外键，joining等等，并且，程序员需要直接考虑“数据库设计三范式”，以及冗余字段等面向数据库的优化手段。

并且，程序员也很可能会采用数据库的一些高级特性，如视图、存储过程、甚至触发器等等以方便使用。

O的存在，仅仅是为了方便操作数据库表。

====== O => R ======

这种设计哲学则是相反，程序员做业务分析、实现架构设计的时候，并不过多考虑数据库的特性与限制；程序仅考虑自己的业务对象：编程语言中的对象。

**数据库仅仅只是作为一个对象持久层来考虑：**

\* 程序运行的时候，对象是自动保持在内存中。

\* 但在对象状态改变、程序退出的时候，将对象保存进数据库。

\* 程序重新启动的时候，则从数据库中获得原先数据，并还原为内存中的对象。

在这样的场景下，数据库是一个可以被替换的存储层，它可以是关系型数据库，也可以是NoSQL，甚至是硬盘文件；所以，即便使用关系型数据库，一般也不会使用其高级功能。

设计哲学的不同直接造成了使用技术的不同。

====== 比较 ======

在ED开发圣经PEAA中，列举了下面三种方案：

1. Trascation Script

也就是直接拼接SQL啦～

2. Table Module

R => O；并且，O非常简单，直接以类似数组的方式读取表数据。

.net中ADO.net的DataTable / DataRow对象便是这种设计的典型实现。

3. Domain Model

O => R，直接设计业务领域(Domain)的对象，然后在考虑对象的持久化方案。

针对上面3中方案，Martin Fowler画了下面这张著名图解：  
<http://www.hamishgraham.net/page/Work-Habits.aspx/Architecture/Business-Layer>

他的结论是：

使用Table Module的方式，永远比直接写SQL简单；在简单的业务场景下，Table Module也会比Domain Model简单，但Table Module的方案复杂度会随着业务复杂化而快速增长。

反之，Domain Model的复杂度跟业务复杂度相比始终保持水平增长；它虽然一开始最复杂，但随着业务复杂度超过一定程度后，它反而会成为最简单的方案。

就我自己的开发经验，基本与Fowler的描述吻合；但随着ORM技术的成熟，Domain Model，未必如他在图中画的那样，一开始就有那么高的复杂度。

关键是看程序员是否习惯于关系型数据库的实现方案，如果是，那么，切换去Domain Model，确实会比较麻烦，各种不适应。

但如果是一个没有关系型数据库经验的程序员，或者说，没有强制使用SQL思维习惯的程序员，使用Domain Model，也可以是很自然的方案。

下一课，讲继续详细说明Domain Model。

作业：

1. ED开发圣经PEAA究竟是哪本书？

2. 数据库三范式是什么？

3. 关于Domain Model，什么是充血模型？什么是贫血模型？

十 做，就对了

学以致用，很多时候，学习一样东西最好需要能够在实际中应用起来。

所以我在第2课"怎么看待牛人"中强调的必须“看代码 + 写代码”。

不过我在里面提到的例子“ORM”却并不好，ORM太过庞大。实际编码，应该是从小开始。

运维工作中更经常使用的是脚本语言，脚本程序甚至是shell命令都可以完成很多有意义的事情。

这些猪头应该在工作中体验很多；但作为程序员，程序能够发挥的作用也可以体现在生活上。

玩Draw Something单词想不出来，是完全可以写个程序来输出单词列表的。

上网下载一个英文单词词库；然后甚至可以用最傻X的方式去逐个单词检查，看Draw Something给出的字母是否能够组成各个单词。

程序首先是要完成需求，这里的需求仅仅是要方便玩游戏，猜出朋友的单词谜语。

程序运行慢点完全无所谓，千分之一秒输出结果，还是10秒输出结果，都不会影响这个需求的实现。

（当然，如果是玩Facebook上的限时拚单词游戏那需求又是不同。）

这种“程序”是所谓的Throw-away code，写完就扔。

像Draw Something这样的游戏，乐趣就在于努力去想、努力猜成功之后的成就感。有了这样一个程序，那就不用努力去想，游戏的乐趣也就会在瞬间丧失，“破解工具”自然也就得扔掉了。

即便写完就扔，但写这样的程序却有其意义。写与不写是差别是0与1的差别，这是本质的区别。

我会非常鄙视那些热衷于看各种语言的介绍但却一行程序都不写的人。

有的人，听说erlang很牛B，上网搜了一堆介绍，不断的感叹“哇～Erlang确实很牛！”，“哦耶！Facebook Chat跟Web QQ都是在用erlang，果然erlang才是王道！”

但是，他自己却不写任何一行erlang程序；有时，还会抱怨公司的管理层都是傻逼，这个项目用erlang再合适不过，为什么不用，为什么不给团队使用erlang的机会呢？

一定要写程序，没有机会，也要创造机会。

而在我看来，生活中这种“玩游戏”的机会再合适不过。

写了Draw Something的“破解工具”，会使得猜单词没有成就感，丧失游戏的乐趣；但，完成了一个程序去破解一个游戏，这本身也是一件有成就感的事情啊～

并且，游戏的乐趣会转移为编程的乐趣；而乐趣，是让自己变厉害的最大动力。

Geek享受这样的机会；而ED则等待别人享受这样的机会。

“做，就对了” - 慈济宗创始人 证严法师

**作业：**

1. 使用Perl 实现一个程序输入若干字母，输出这些字母所能组成的所有单词列表。素，就是要写个 Draw Something的“破解工具”。

2. 比较Perl的实现跟云风的lua实现有何不同：<https://github.com/cloudwu/guess-word>

十一 域模型

在前面的课程中提到PEAA中只有一页半的Active Record Pattern (<http://martinfowler.com/eaaCatalog/activeRecord.html> )影响了过去5年多6年的Web开发潮流。

这个潮流是由Ruby On Rails引领的。

RoR的作者DHH David Heinemeier Hansson是Hacker，他因为RoR在2005年被Google跟O'Reilly选为年度黑客。

他在设计RoR时，选用了Active Record作为RoR的M层。

Active Record非常简单，一个类对应一个表，一个对象实例对应一行数据；并且有简单的有Save / Delete以及查询等简单的函数操作。

严格的说，Active Record不是福勒所推崇的充血Domain Object模型 （<http://martinfowler.com/bliki/AnemicDomainModel.html> ），Active Record对象提供的功能函数太少，只有通用的数据操作方法，而不包涵业务逻辑；但它又不像POJO （<http://martinfowler.com/bliki/POJO.html> ） 那样完全的贫血。

（充血、贫血Domain Object之争，可以去iteye翻帖子）

从福勒 AnemicDomainModel 一文看，他在当年（2003）是推荐了充血Domain对象跟POJO，但过去几年在Web开发领域所流行的却是 Active Record这样两边都沾点，但却又不全是的中间妥协方案。

不搞教条主义，什么实用用什么，POJO不够，那么就加一点；充血太复杂，那么就减少一点。

从互联网的发展看，我一时间完全想不出有什么在理论上被设计得很好的模型，能够最终经历时间考验成为事实标准。

因特网的7层模型，实际用到的远不到7层；Java的EJB挂了；XML被JSON取代等等等等。

也许学院派提出的理论有他们的应用场景，只是，这样的场景，在快速发展互联网似乎很难找到例子。

互联网产品的业务相对简单，Active Record已经足够好，足够方便，因此大行其道。

另一方面，互联网产品做大后，也往往有着极大的性能要求，一个复杂的模型，是难以做性能优化的。

像Active Record，因为足够简单，Twitter在当年遇到性能问题的时候，便直接Hack掉RoR的实现，增加了 Cache-Money （ <https://github.com/nkallen/cache-money> ） 这么一个透明的缓存层。

如果RoR使用的是充血对象模型，对象中有复杂的业务逻辑，如何增加透明的缓存呢？

Active Record的实际上是对数据库操作做了抽象。

封装、抽象是一门艺术。

什么该封装，什么该暴露，什么彻底不可见，需要拿捏得很准确。

最容易犯的错误是过度封装，使得一些本来很简单的底层操作，到了上层变得完全不能用；或者说，很难用。

开发者需要用到hack的方式，才能去做这些简单的操作。

Active Record便是一个抽象封装得恰到好处的例子。过度设计、过度封装的数据操作层？EJB。

按照教科书对OO的定义，OO的核心特性之一是：encapsulation<http://en.wikipedia.org/wiki/Encapsulation_%28object-oriented_programming%29>

Private属性、方法，对象外部是完全不能访问的。

但如果遇到了需要访问的场景怎么办？！

有的人会说：“这样的场景本来就不应该出现，这是对象设计一开始没有做好造成的，错误的应该设成Public的属性设成了Private”。

ORM，采用O => R的映射的设计哲学，只考虑业务对象，完全不考虑底层数据库，数据库仅仅是一个可以被替换掉的持久层，它可以是关系型数据库、也可以是NoSQL，甚至是硬盘文件。

也就是说，Domain Object是把后端数据库给设成“Private”了，即便底层是关系型数据库，你也不可以直接去写SQL。

即便你使用的是MS SQL Server，你也不能去调用它特有的SQL特性。

Asp.Net刚出来的时候，微软曾经鼓吹过一个叫 N-Tiers 的架构：<http://msdn.microsoft.com/en-us/library/ms973279.aspx> /<http://msdn.microsoft.com/en-us/library/bb384398.aspx> 。

我曾经以为这是王道，直到我膝盖中了一箭……呃，不，直到我看了Joel Spolsky写的 The Law of Leaky Abstractions：<http://www.joelonsoftware.com/articles/LeakyAbstractions.html>

理想很丰满，现实很骨感。

ORM工具再怎么封装都好，底层用了数据库，就是用了数据库。

开发者必然需要了解数据库的特性，能否直接调用数据库的特性，是一个选择。

是否要彻底对上层屏蔽掉数据库的存在，也是一个选择。

N-tiers架构推荐一层又一层的封装，如果错误使用，把选择当成教条，是会有噩梦的。

========

Python是一门很有趣的语言，它支持继承，能实现OO，但是缺乏 encapsulation 的语言支持。

Python根本就没有public / private这样的关键字，然后呢？

然后可以回过头再去看：“这样的场景本来就不应该出现，这是对象设计一开始没有做好造成的，错误的应该设成Public的属性设成了Private”。

这句话，这话说得对嘛？

**作业：**

1. N-tiers架构的噩梦场景是？

2. 什么系统/场景需要充分使用特定数据库的特性？

十二 作业点评

**h1. 作业分析**

作业是课程的一部分，实际上，还是这个课程最重要的部分。

如我在前面课程中提到的一样：

很多初学者，或者说，编程的伪爱好者，他们，会热衷于去四处请教大师，下载各种经典书籍，企图读一本编程圣经，然后一夜脱胎换骨。

这是，不可能的。

同样的，如果仅仅是看了这个课程，而不做作业，那么在看课程前后，个人的能力是不可能有变化的。

充其量，跟看了一部或许好玩的小说差不多。

作业并不是考试，而是课程的延伸，是没有可能参照着课程的内容，然后对作业做出回答。

每节课，仅仅只是指出一个方向，然后需要大量时间的去朝这个方向做学习、探索，然后以作业的形式做出对这个方向的回答。

这才是学习。

花几分钟看几眼课程，然后就期待自己技术能力有变化？能够有改变，从不会做作业变成会做作业？

别开玩笑了，如果能够这样，那么程序开发会是一门非常没有技术含量，非常没有含金量的行业。

只有用心好好完成了作业之后，才有可能获得知识。

这个课程的作业，也完全不是：

小明有5个苹果，他吃了一个。然后给小寒了一个，求太阳到地球的距离。

这样无厘头的题目。

每节课的作业，都是跟课程有直接关系的。

**h2. 第一课**

1. 用500字讲述什么是Programming Methodology?

2. 列举10种Data Structure.

3. 列举10种Algorithm.

这课的作业实际上是在问，你对“编程本质”的内容掌握了多少，如果不够熟悉，了解得不够多，要赶快去学习。

**h2. 第二课**

1. 列举10个Python Web框架

2. Python有多少种不同的解释器？

3. Perl 跟 Python 有什么不同？

这课的作业，同样是在问具体到Python这个语言平台，在实际开发中可供挑选的现成工具有哪些？问的是对自身工作所使用的平台熟悉程度。这课的作业，也完全可以根据使用的语言不同，而改成别的技术题目。

这课讲的是实际中对工具掌控的熟悉程度这个方向，如果熟悉，那么这三个问题是很容易回答的，如果不熟悉，而为了做作业去打开Google，搜“python web框架”，然后填名字。那么就完全木有做作业的意义。

**h2. 第三课**

1. 找出一篇看上去很厉害的文章。

2. 找一本书，开始看，作为期中考书目。

这课讲的是阅读的重要性，两项作业，一个要求阅读的广度，一个是要求阅读的深度。

作业是要做的。OK，这课讲了阅读的重要，明白了，然后就洗洗睡了？自身的阅读的东西，无论是广度还是深度，都跟以前一样，那学这课程有个毛用？

宣称喜欢这个课程，并且表示关注、期待的同学，请问，你选择的期中考书目，已经翻了几页？

如果一页还没有翻；那么请好好问一下你自己，你究竟是不是要学习提高改变自己的？

**h2. 第四课**

1. 林仕鼎是百度首席架构师吗？

2. 看完曹政所有的blog。

这一课其实还是在讲阅读的重要性，以及对事物的好奇心。

如果，你对技术有热情，有追求，课程中居然出现了“百度首席架构师”这样的字眼，你必然会对他有无限的好奇，会去刨根问底的了解他。

那么，是很容易就发现林仕鼎根本就不是百度首席架构师，相反，caoz曾经更符合这个身份。

我列举了两个hacker风格的IT人物，一个是caoz，一个是云风。

作业有一项是看完caoz的所有blog，他的blog很好看的。如果你真的看完了，那么，请问你是否有完成这课实际上还有另一个隐藏的“作业”，“看完云风的所有blog”？

如果没有，那是什么阻止了你？一个非常优秀的技术博客知识就放在你眼前，你，为什么不去看？

OK，没有时间，很忙，这些我很了解。

我只问一个：是否有过要把云风的blog也看完的念头？

如果连这基本的好奇心、求知欲都木有的话，那还是洗洗睡吧。

**h2. 第五课**

1. NoSQL的方案，如MongoDB还解决了SQL的什么问题？

2. NoSQL的应用场景有啥米？

这课是讲数据库，分析、比较了SQL、NoSQL，同样的，需要课后去做更加深入的了解并且思考SQL、NoSQL的适用场景。

**h2. 第六课**

1. 使用一种编译语言实现 Singleton 模式

2. 使用一种动态语言实现 Singleton 模式

3. 说说对 Provider 模式的理解。

如果连最简单的Singleton模式实现都是上网google的现成代码，那。。。还是那句话，洗洗睡吧。。。

这课讲的是设计模式的必要以及局限，如果只是看到后面对设计模式局限的调侃，而无视了前面提到的：“开发中非常必要的知识，实际上，是非常基础的知识”。

你究竟对非常基础的设计模式了解得多深入了？第三题换个模式，你说得出理解么？

**h2. 第七课**

1. php 的 Smarty 模版的设计哲学是什么？

2. perl 的 Mason 模版的设计哲学是什么？

3. 什么是gettext?

4. 前端javascript实现的模版中，目前最成熟的是哪个引擎？

这课是讲模版，模版有很多现成的实现，作业纯粹就是在要求去了解、认识各种模版技术的实现。

**h2. 第八课**

1. 说说对 restful 的理解

2. 什么是 reverse proxy ?

restful / reverse proxy等，都是跟controller相关，但延伸出去的相关知识。

相关性究竟在哪里？这个可以做为独立的一课去讲述，但也完全是可以自学了解的。但这绝对不是在跟小明讲了1+2=3后，问太阳与地球的距离。

**h2. 第九课**

没有作业。

**h2. 第十课**

1. ED开发圣经PEAA究竟是哪本书？

2. 数据库三范式是什么？

3. 关于Domain Model，什么是充血模型？什么是贫血模型？

第一题纯娱乐，第二题是确认课本知识掌握；第三题则又是在要求延伸阅读，实际上，也是在为下一课做预习。

**h2. 第十一课**

1. N-tiers架构的噩梦场景是？

2. 什么系统/场景需要充分使用特定数据库的特性？

这课作业是在要求对课程做思考，写课程时，我实际上是码了很多字，去描述N-tiers的噩梦场景。但后来我又全部删除。

因为，我前面已经讲了很多关于分层、封装的问题，也提供了The Law of Leaky Abstractions的连接，对N-tiers有了解，对分层的问题有了解，那么如果还不能认识到N-tiers这么一个多分层的技术的噩梦场景是什么的话；那么我还是只能说：洗洗睡吧。

整个课程，是在强调对数据库的封装。为了避免产生封装就是好的教条思想产生，所有我又加了“使用特定数据库的特性”这个作业，要求去思考一下相反的场景。

十三 再谈ED

ED是Enterprise Developer的缩写，我是刻意用这个缩写的，因为它还可以被解释为Ejeculation Disorder。尽管我使用的名称不一样，但类似的概念是一直都有的。

像Joel Spooky则是称他们为“In-house programmer”:<http://www.joelonsoftware.com/items/2007/12/04.html> 并极力劝说计算机系毕业生不要成为这样的程序员。

Joel认为“In-house programmer”是“恐怖的”，并列举了三条理由：

1. 永远不会有机会用正确的方法做事。  
2. 没有机会做优雅的产品  
3. 升职空间有限

我完全同意Joel的看法，实际上，我猜想很多“ED”也非常清楚了解他们的处境，经常可以看到一些网文，诸如：

程序员们 不要想一辈子靠技术混饭吃<http://developer.51cto.com/art/201111/299998.htm>

这位作者对ED职业发展的描述非常真实，对于那些走在ED路途上的人做出了诚挚建议：不要一辈子都做技术。

我觉得他的建议跟Joel建议毕业生不要成为"In-house programmer"的出发点是一致的。ED是死路一条，不同的是，Joel建议毕业生去参与那些优秀的技术公司的研发工作，而这位作者则干脆建议ED们彻底不要做技术。

任何行业，都会有做得好，跟做得不好的人；因为自己做得不好，而否定整个行业，我觉得是挺可悲的。

即便在中国，纯粹靠技术做得好，而获得“稳定的生活和高的薪水待遇”的程序员大有人在，当然，做市场开发而赚得盆满钵满的人也有无数。

365行，行行出状元。

在自己的行当里面混得不好，就指着别的行当的状元说，看，别的行业比我们好多了～这挺没出息的。

软件行业有软件行业的问题，也有它的优越；这构成了软件行业的“风格”。关键是，自己是否喜欢软件行业的风格。

若不喜欢，趁早换去别的行当；若是喜欢，那么应该努力的去做到最好；而不是发牢骚、抱怨这个行业中的不好。

关键在于自己的态度以及努力，想要有快感，首先是自己需要勃起啊！自己心态上不投入，没有激情，ED是必然的。

作为职业发展，首先必须是自己真心喜欢、热爱这个行业，即便没有薪酬，没有回报，自己也能够在做事的过程中获得精神上的满足。

只有这样才能够在技术发展路途上精益求精，不断进步；是的，即便“做到技术最强，也不可以成为100%受尊重的人”，但这并不是不把技术做到最强的理由。热爱技术，“做到技术最强”，本身就是一个充满诱惑的目标。

把技术视为敲门砖，自己本身就不尊重技术，那必然是没有办法成为别人尊重技术人。

无论外人怎么看，技术人都应当尊重自己。

猪头时常跟我辩解即便是在银行内部做IT也有升职空间，并且自己也不断的在银行内不断提升、转向更高的技术职位，其实是很厉害的。

Premier ne puis, second ne daigne, Mouton suis.

作业：  
1. 这篇课程最后的法文的下句是什么？

2. Almaviva是什么？

3. 猜我为什么出上面两道无厘头的作业？