二零一四年自述——“坐下来能写，站起来能说”

Sadapplelc

“如果你很喜欢一件事情，怎么能忍受自己不全情投入做到最好呢？说简单点，他们就是目标明确并且拼命向前跑的人。没有那么多的顾虑，那么多的患得患失，心无旁骛要达到目标的人。”

题记

在去年末的时候接触到了 Lyx 和 Texmacs，发现两个软件都非常有趣而且好用。针对 Lyx 的特点，觉得拿来新年里记录自己的随感这个想法可以实践一下，所以有了这份文档，用来记录二零一四年自己的经历和想法。

要说 2014 年自己心底最想达成的想法，说出来也就一句话——磨练好的气质和条件反射，积累重要的技能，成为更好的人。

什么样的气质是好的呢？

- 判断准确、取舍豁达、之后的行动迅速而不拖泥带水（所谓“智与勇”）
- 锲而不舍地专注，不找到自己的正解决不罢休（所谓“大力”）
- 耐心、平常心（所谓“恬淡”）
- 善于控制和调整自我
- 和强者交往，和别人做有益的分享和交流

什么样的条件反射是好的呢？

- 做任何事的时候都明白“心”、“技”、“体”三位一体，密不可分。
- 一件事，重要吗？急迫吗？问自己
- 问好的问题，形成好的感觉比盲目地做事更重要
- 永远惜时、惜人、惜福
- 很多事都可以分解清楚再来做，找到正确的顺序来安排！
- 追二兔不得一兔，所以，一心一意

- 勤能补拙

什么样的技能（对于现在的我来说）是重要的技能？

- 熟练掌握 R，尽快学习、熟悉 C++、Phython、SAS 等有用的语言
- Latex（Lyx）、Texmacs 等学术写作软件的熟练运用（比如，用以 CV 的写作）
- Excel 要相对熟悉
- 金融和数据挖掘领域的知识积累
- 清晰准确地表达自己观点

无须再多说了，只需记得《劝学》里的话。平时多思考多表达自我，把重要的东西积土成山。希望到年末，我能慢慢接近“坐下来能写（思），站起来能说”的境界。

Jan

1.1

I made a promise to myself on the first day of 2014.

1.2

志气，志气。

1.11

Q：为什么自行车两轮在行进中比三轮车能更好的保持平衡？试做转弯时的受力和力矩分析

1.12

今天学习《Lyx Essential》一文的文本 layout 和数学输入基础部分，剩下如何做 reference、如何制表、如何插图、如何插入代码（比如 R）留待下次学习。

符号读法

; 分号 semicolon ” 双引号 quotation marks/double quote ‘单引号/撇号 apostrophe/single quote ‘ 重音号 backquote/grave accent * 星号 asterisk/star

/ 斜线 slash \ 反斜线 backslash/escape | 竖线 bar/pipe/vertical bar _ 下划线 underline/underscore

井号 crosshatch/sharp/hash % 百分号 percent sign/mod ^ 折音号 circumflex/caret ~ 波浪号 tilde

{} （左右）花括号/大括号 (left/right|open/close) braces [] （左右）方括号/中括号 (left/right|open/close) brackets () （左右）圆括号/小括号 (left/right|open/close) parentheses <> 尖括号 angle brackets

多动手写程序

注册了 csdn 和 github，以后没事多转有益有趣的网站社区。今年啊一定要在编程上有大大的进步！

走出门去！

很多学生在网络或者游戏等虚拟世界里投入了过多的时间，但忽视了身边的同学朋友。他们还没有意识到他们身边的同学、师兄师姐、朋友甚至朋友的朋友，都有可能是今后职业发展的重要资源。因此在做 JA 讲师的时候，我会鼓励大学生朋友能通过各种途径参与社会活动，建立人际关系网，为今后的职业发展积累资源。

善用时间

1. 明确任务与目标
2. 细化目标，执行计划
3. 善用闲散时间
4. 区分使用小段时间和大段时间

“一旦下定决心完成某一任务，就要在确定总体目标之后细化每一周甚至每一天的工作量，尽量确保每个时间节点都能按时完成。由于副业挤占了休息时间，可能会很累。虽然不是说完全不能有休闲活动，但如果有可能还是可以把闲散时间利用起来，做一些工作相关的事情，这样既放松了大脑，又有效利用了时间。如果能坚持一段时间，就会小有所成。”

试建表格：

有用且急迫	无用但急迫
有用但不急	无用且不急

1.14

Texmacs 学习

cyan 青色，蓝绿色

1.15

荆轲

咏荆轲陶渊明 (如何居中？)

燕丹善养士，志在报强嬴。　　招集百夫良，岁暮得荆卿。　　君子死知己，提剑出燕京；　　素骥鸣广陌，慷慨送我行。　　雄发指危冠，猛气冲长缨。　　饮饯易水上，四座列群英。　　渐离击悲筑，宋意唱高声。　　萧萧哀风逝，淡淡寒波生。　　商音更流涕，羽奏壮士惊。　　心知去不归，且有后世名。　　登车何时顾，飞盖入秦庭。　　凌厉越万里，逶迤过千城。　　图穷事自至，豪主正怔营。　　惜哉剑术疏，奇功遂不成。　　其人虽已没，千载有馀情。

注释

荆轲：战国时卫国人，为燕太子丹报仇，以送地图为名，藏匕首刺秦王，不成被杀。　　燕丹：战国时燕王喜的太子，名丹。　　强嬴：秦国。　　荆卿：指荆轲。　　渐离：高渐离，战国时燕国人，与荆轲友善，善击筑 (古时的一种乐器)。　　宋意：燕国的勇士。　　商音、羽奏：商声和羽声。商声凄凉，羽声较激昂。

译文

燕国太子喜欢收养门客，目的是对秦国报仇雪恨。　　他到处招集有本领的人，这一年年底募得了荆卿。　　君子重义气为知己而死，荆轲仗剑就要辞别燕京。　　白色骏马在大路上鸣叫，众人意气激昂为他送行。

个个同仇敌忾怒发冲冠，勇猛之气似要冲断帽缨。　　易水边摆下盛

大的别宴，在座的都是人中的精英。　　渐离击筑筑声慷慨悲壮，宋意唱歌歌声响遏行云。　　座席中吹过萧萧的哀风，水面上漾起淡淡的波纹。

唱到商音听者无不流泪，奏到羽音荆轲格外惊心。　　他明知这一去不再回返，留下的姓名将万古长存。　　登车而去何曾有所眷顾，飞车直驰那秦国的官廷。　　勇往直前行程超过万里，曲折行进所经何止千城。

翻完地图忽地现出匕首，秦王一见不由胆颤心惊。　　可惜呀！只可惜剑术欠佳，奇功伟绩终于未能完成。　　荆轲其人虽然早已死去，他的精神永远激励后人。

已亥杂诗龚自珍

陶潜诗喜说荆轲，想见停云发浩歌。吟到恩仇心事涌，江湖侠骨恐无多。

关于 CFA

因为二级最早也只能明年考，而今年暑假想争取实习，所以选择十二月底考一级可能更好。这样上半年也会更有时间提升自己的 computing 技能和对统计的理解，这才学统计的我以后找工作的关键。这几天学习生活的感觉还不错，对编程和电脑技术有了更强的兴趣，我所要做的就是继续 motivate myself。不足就是还是会有分心的时刻，不得不说，我还是想形成一种强烈的气质来推动自己学习生活，这样能够更惜时更有效率。所以还要加油！

Reminder

早点做出一份简历来！

1.16

排序算法

- 冒泡排序
- 选择排序

1.18

Grovel：卑躬屈膝
注意 recipe 的重音

一月剩余时间的小计划

1. 完成《R 编程艺术》一书 Ch1~8 并完成 simple function 一节的所有习题
2. ESL 完成 Ch3,4,7 并完成 587 相应作业
3. 通读《非线性优化基础》的 Ch2,3,4 并尽可能完成 project 剩下的证明
4. 复习时间序列

1.21

火花天龙剑卷首语

塞林格写过一部名作叫《麦田里的守望者》，里面的主人公是一个被学校开除的中学生，他貌似玩世不恭，厌倦现存的平庸的一切，但他并非没有理想。他想象悬崖边有一块大麦田，一大群孩子在麦田里玩，而他的理想就是站在麦田边作一个守望者，专门捕捉朝悬崖边上乱跑的孩子，防止他们掉下悬崖。

我很喜欢“守望者”这个名称，它使我想起守林人。守林人的心境总是非常宁静的，他长年与树木、松鼠、啄木鸟这样一些最单纯的生命为伴，他自己的生命也变得单纯了。他的全部生活就是守护森林，瞭望云天，这守望的生涯使他心明眼亮，不染尘嚣。“守望者”的名称还使我想起守灯塔人。在奔流的江河中，守灯塔人日夜守护灯塔，瞭望潮汛，保护着船只的安全航行。当然，与都市人相比，守林人的生活未免冷清。与弄潮儿相比，守灯塔人的工作未免平凡。可是，你决不能说他们是人类中可有可无的一员。如果没有这些守望者的默默守望，森林消失，地球化为沙漠，都市人到哪里去寻欢作乐，灯塔熄灭，航道变为墓穴，弄潮儿如何还能大出风头？

人做事情，或出于利益，或出于性情。凡出于性情做的事情，大都仅仅是为了满足心灵需求，如写作、艺术欣赏、交友等等。我们做这个主页

也是如此，愉快是最基本的标准。如果什么时候我们在这里找不到愉快，就必须怀疑是否有利益的强制在其中起着作用，使性情生活蜕变成了功利行为。火花是这样一个地方，它不想在踌躇满志的游戏精英网站中挤自己的一块地盘，而是很安静的，与世无争的，但也因此在普遍的热闹和竞争中有了存在的价值。我们只想开辟一块园地，让火炎玩家们找到回家的感觉。

如果一个人有自己的心灵追求，又在世界上闯荡了一番，有了相当的人生阅历，那么，他就会逐渐意识到自己在这个世界上的位置。世界无限广阔，诱惑永无止境，然而，属于每一个人的现实可能性终究是有限的。你不妨对一切可能性保持着开放的心态，因为那是人生魅力的源泉，但同时你也要早一些在世界之海上抛下自己的锚，找到最合适自己的领域。一个人不论伟大还是平凡，只要他顺应自己的天性，找到了自己真正喜欢做的事，并且一心把自己喜欢做的事做到尽善尽美，他在这个世界上就有了牢不可破的家园。于是，他不但会有足够的勇气去承受外界的压力，并且会有足够的清醒来面对形形色色的机会的诱惑。我们当然没有理由怀疑，这样的一个人必能获得生活的充实和心灵的宁静。

1.22

blunt: 钝的，迟钝的

车的 Alignment：

Alignment 是四轮定位；balance 是动平衡。

alignment 是调校悬挂的，与轮胎不能说完全无关，但调好后什么轮胎装上都不影响；吊起来后看两个轮子，是倒八字的，而不是平行的。

balance 是关于轮胎的，把轮胎放到平衡机上，平衡后可以装到任何合适的车子上。

每次扒轮胎换 rim 都必须做平衡；而如果没什么事，不需要做定位。

平时开车注意感觉车的状况，尤其是加减速的时候、过坑、过包的时候，有没有异常的情况。

你的状况，轮胎异常磨损，很有可能是 alignment 的问题。应该还伴随着方向盘上传来有规律的震动，而且这种震动是随着车速的增加而加强的。

还有一种方向盘上的震动，只在某个速度区间，例如 80-100 之间，低了高了都不震，这通常是轮胎的动平衡不好，而与 alignment 无关。

车胎：

groove: 沟，槽

注意花纹（*tire tread pattern*）和型号。注意花纹深度和磨损标记。

NOTE: According to most states' laws, tires are legally worn out when they have worn down to 2/32" of remaining tread depth. To help warn drivers that their tires have reached that point, tires sold in North America are required to have indicators molded into their tread design called "wear bars" which run across their tread pattern from their outside shoulder to inside shoulder. Wear bars are designed to visually connect the elements of the tire's tread pattern and warn drivers when their tires no longer meet minimum tread depth requirements.

一般来讲，轮胎花纹即轮胎胎面上各种纵向、横向、斜向组成的沟槽。别看这些横着竖着的花纹很乱，其实他们可是有着明确的分工。纵向花纹因为具有纵向连续性的特点，所以主要承担雨天排水的功能，并且对于轮胎的散热也很有帮助，但抓地力不足。横向花纹则有着较大的抓地能力，从而可以弥补纵向花纹的先天缺陷。

与纵向花纹相对，横向花纹纵向断开，横向连续，这样的设计使得轮胎横面刚度大，摩擦力以及制动效果较好。

普通的混合花纹轮胎使用限制较少，前后左右均可调换，是一种综合平衡性十分出色的轮胎。

不对称花纹即左右两边拥有不同的花纹结构。由于左右花纹结构不同，所以通常设计时会更注重设计增大转弯时外侧花纹的着地压力的花纹。这也是为什么不对称轮胎在某些情况下过弯时性能会相对出色的原因。并且考虑到日常使用的情况，不对称轮胎外侧花纹的耐磨性也能会得到相应提高。不仅如此，由于两边的花纹采用不同的结构形式以及橡胶配方，所以轮胎的整体性能可以更强更全面，比如一边花纹可以侧重排水能力，一边花纹可以侧重抓地能力等。目前不对称花纹轮胎已经被得到广泛运用，并且取得了市场得肯定，是多数车主购买轮胎时的首选。不仅如此，由于两边的花纹采用不同的结构形式以及橡胶配方，所以轮胎的整体性能可以更强更全面，比如一边花纹可以侧重排水能力，一边花纹可以侧重抓地能力

等。目前不对称花纹轮胎已经被得到广泛运用，并且取得了市场得肯定，是多数车主购买轮胎时的首选。需要注意的是由于该种轮胎内外花纹不同，在安装时必须确定轮胎的内外是否正确。在采用这种花纹的轮胎在轮胎对调的时候没有特殊要求，前后左右都可以，只要主要轮胎的内外即可。

单导向花纹轮胎的花纹沟之间相互连接，拥有较强的公路性能以及极强的排水能力。这种花纹的轮胎有固定的滚动方向，也就是说这种轮胎只能朝着一个方向跑，这样的优点在于轮胎的滚动阻力会相对较小，车辆操控性更加优秀。但也因为这样的设计单导向轮胎不能左右对调只能同侧轮胎互换，这也是为什么叫单导向轮胎的原因。

块状轮胎被广泛用在越野车上面，一般根据使用条件的不同主要分为三种：公路轮胎、全地形轮胎和泥地胎。

泥地轮胎简称 MT(Mud-Terrain) 轮胎，一般只有越野发烧友和特殊路段工作者才会选用。它与公路 (HT) 轮胎正相反，MT 轮胎壁坚硬，胎牙夸张，胎牙之间的距离明显偏大，便于泥地行驶的时候慢速排泥或高速甩泥，另外在一些恶劣的地面上更容易增加附着力，如凸凹不平的岩石地面。

纵向花纹轮胎顾名思义，当然是以纵向花纹为主的轮胎。这种轮胎的花纹与轮胎方向一致，绕呈一条或者多条连续的圆圈。这种轮胎拥有良好的车头指向性以及排水性能，高速行驶时噪音控制很好，并且受摩擦力所带来的向前的阻力也相对小很多。所以纵向花纹轮胎可以提供极好的运动操控性。但这种轮胎自身的缺陷也很明显，由于只有纵向花纹，所以其提供的抓地力并不充足，所以其提供的制动能力以及驱动能力都不算优秀。像这种单一的纵向花纹轮胎尽管有着出色的性能但因为存在着不少先天缺陷，所以日常使用中并不常见。

横向花纹恰恰与纵向花纹相反，能够提供极好的抓地力，所以其在制动能力与牵引力方面有着得天独厚的优势。这种花纹的轮胎在专业领域用途比较广，比如一些农业机械，如收割机，拖拉机等，以及一些工程机械和大型牵引车等都可能使用到横向花纹轮胎。但这种花纹的轮胎因为胎噪大，易磨损，耗油高，高速性能差等缺点，所以在民用领域里几乎没有市场。

光头胎因为没有排水沟槽以及其他任何花纹，所以它能够得到一条轮胎最大的接地面积。在干地抓干地路面上可以得到最大的摩擦力，从而让轮胎紧紧吸住地面。因为有如此之强的地能力，所以装配光头胎的赛车能以更高的速度过弯。也正因于此，FIA 为了安全考虑，从 1998 年之后禁止

F1 赛事使用光头胎，必须使用带有直沟花纹的轮胎。从而降低赛车过弯时的速度。虽然光头胎性能很强，但它并不适合民用。及软的胎质，并且多数为热熔胎，跑上几百公里一条数千元的轮胎就宣布寿终正寝，这绝对不是一般人能玩的。而且这种轮胎也并非可以全天候使用，一旦遇上下雪下雨，那他也只有打滑的份儿。

1.23

我们常常追求结果，而忘了过程。而如果局限于结果，我们就会选择急迫，而失去了平常心。这就像拘泥于获胜，就会选择安全，可是为这个避开战斗，就会失去真正的勇气。你们的正解请你们保留，我要寻找我自己的正解。——藤泽秀行先生原作，我用我自己的体会重新表述

十年学会程序设计：

以下是我在编程上成功的秘诀:

- 对编程产生感兴趣并因为乐趣而写程序。确信你自始至终都能乐在其中，这样你才愿意将十年光阴投入编程事业
- 与其他程序员交流；阅读别人的代码。这比任何书任何培训都重要。
- 不断地编写。最好的学习方法是在实践中学习。从技术角度说，在特定领域的个人最高效率并不因为经验够多就会自动获得；但若有意识的通过努力去提升经验，个人效率会变高。而高效的学习一般需要明确的任务和因人而异的适当难度，以及及时的反馈和重复或者修正错误的机会。

《如果让我重做一次研究生》：

- 做为研究生不再是对于各种新奇的课照单全收，而是要重视问题取向的安排，就是在硕士或博士的阶段里面，所有的精力、所有修课以及读的书里面都应该要有一个关注的焦点，而不能像大学那般漫无目标。大学生时代是因为你要尽量开创自己接受任何东西，但是到了硕士生和博士生，有一个最终的目的，就是要完成论文，那篇论文是你个人所有武功的总集合，所以这时候必须要有个问题取向的学习。我

常常跟学生讲，选对一个领域和选对一个问题是成败的关键，而你自己本身必须是带着问题来探究无限的学问世界，因为你不再像大学时代一样泛滥无所归。所以这段时间内，必须选定一个有兴趣与关注的主题为出发点，来探究这些知识，产生有机的循环。由于你是自发性的对这个问题产生好奇和兴趣，所以你的态度和大学部的学生是截然不同的，你慢慢从被动的接受者变成是一个主动的探索者，并学会悠游在这学术的领域。

- 我常说英文 research 这个字非常有意义，search 是寻找，而 research 是再寻找，所以每个人都要 research，不断的一遍一遍再寻找，并进而使你的生活和学习成为一体。中国近代兵学大师蒋百里在他的兵学书中曾说：「生活条件要跟战斗条件一致，近代欧洲凡生活与战斗条件一致者强，凡生活与战斗条件不一致者弱。」我就是藉由这个来说明研究生的生活，你的生活条件与你的战斗条件要一致，你的生活是跟着老师与同学共同成长的，当中你所听到的每一句话，都可能带给你无限的启发。

- 回想当时我在美国念书的研究生生活，只要随便在楼梯口碰到任何一个人，他都有办法帮忙解答你语言上的困难，不管是英文、拉丁文、德文、希腊文……等。所以能帮助解决问题的不单只是你的老师，还包括所有同学以及学习团体。你的学习是跟生活合在一起的。当我看到有学生呈现被动或是懈怠的时候，我就会用毛泽东的「革命不是请客吃饭!」来跟他讲：「作研究生不是请客吃饭。」

- 怎样进入一个领域最好，我个人觉得只有两条路，其中一条就是让他不停的念书、不停的报告，这是进入一个陌生的领域最快，又最方便的方法，到最后不知不觉学生就会知道这个领域有些什么，我们在不停念书的时候常常可能会沉溺在细节里不能自拔，进而失去全景，导致见树不见林，或是被那几句英文困住，而忘记全局在讲什么。藉由学生的报告，老师可以讲述或是厘清其中的精华内容，经由老师几句提点，就会慢慢打通任督二脉，逐渐发展一种自发学习的能力，同时也知道碰到问题可以看哪些东西。就像是我在美国念书的时候，我修过一些我完全没有背景知识的国家的历史，所以我就不停的念书、不停的逼着自己吸收，而老师也只是不停的开书目，运用这样的方式慢慢训练，有一天我不再研究它时，我发现自己仍然有自我生产及蓄发

的能力，因为我知道这个学问大概是什么样的轮廓，碰到问题也有能力可以去查询相关的资料。所以努力让自己的学习产生自发的延展性是很重要的。

- 写论文时很重要的一点是，文笔一定要清楚，不要花俏、不必漂亮，「清楚」是最高指导原则，经过慢慢练习会使你的文笔跟思考产生一致的连贯性。我常跟学生讲不必写的花俏，不必展现你散文的才能，因为这是学术论文，所以关键在于要写得非常清楚，如果有好的文笔当然更棒，但那是可遇不可求的，文彩像个人的生命一样，英文叫 style，style 本身就像个人一样带有一点点天生。因此最重要的还是把内容陈述清楚，从一万字到最后十万字的东西，都要架构井然、论述清楚、文笔清晰。

- 做研究生的时代，固然应该把所有的心思都放在学业上，探索你所要探索的那些问题，可是那只是你的一只脚，另外还有一只脚是要学习培养一、两种兴趣。很多人后来会发现他的右脚特别肥重（包括我自己在内），也就是因为忘了培养左脚。很多很有名的大学者最后都陷入极度的精神困扰之中，就是因为他只是培养他的右脚，他忘了培养他的左脚，他忘了人生用两只脚走路，他少了一个小小的兴趣或嗜好，用来好好的调解或是排遣自己。

- 做一个研究生或一个学者，有两个感觉最重要 责任感与罪恶感。你一定要有很大的责任感，去写出好的东西，如果责任感还不够强，还要有一个罪恶感，你会觉得如果今天没有好好做几个小时的工作的话，会有很大的罪恶感。除非是了不得的天才，不然即使爱因斯坦也是需要很努力的。所以为什么说要赶快选定题目？因为如果太晚选定一个题目，只有一年的时间可以好好耕耘那个题目，早点选定可以有二、三年耕耘那个题目，是三年做出的东西好，还是一年的东西好？如果我们的才智都一样的话，将三年的努力与思考都灌在上面，当然比一年还要好。

如何训练自己：

（一）**尝试接受挑战，勇于克服**　研究生如何训练自己？就是每天、每周或每个月给自己一个挑战，要每隔一段时间就给自己一个挑战，挑战一个你做不到的东西，你不一定要求自己每次都能顺利克服那个挑战，但

是要努力去尝试。我在我求学的生涯中，碰到太多聪明但却一无所成的人，因为他们很容易困在自己的障碍里面，举例来说，我在普林斯顿大学碰到一个很聪明的人，他就是没办法克服他给自己的挑战，他就总是东看西看，虽然我也有这个毛病，可是我会定期给我自己一个挑战，例如：我会告诉自己，在某一个期限内，无论如何一定要把这三行字改掉，或是这个礼拜一定要把这篇草稿写完，虽然我仍然常常写不完，但是有这个挑战跟没这个挑战是不一样的，因为我挑战三次总会完成一次，完成一次就够了，就足以表示克服了自己，如果觉得每一个礼拜的挑战，可行性太低，可以把时间延长为一个月的挑战，去挑战原来的你，不一定能做到的事情。不过也要切记，硕士生是刚开始进入这一个领域的新手，如果一开始问题太小，或是问题大到不能控制，都会造成以后研究的困难。

（二）论文的写作是个训练过程，不能苛求完成精典之作

要有这是一个训练过程的信念，应该清楚知道从哪里开始，也要知道从哪里放手，不要无限的追下去。

（三）论文的正式写作

1. 学习有所取舍　　到了写论文的时候，要能取也要能舍，因为现在信息爆炸，可以看的书太多，所以一定要建构一个属于自己的知识树，首先，要有一棵自己的知识树，才能在那棵树挂相关的东西，但千万不要不断的挂不相关的东西，而且要慢慢的舍掉一些挂不上去的东西，再随着你的问题跟关心的领域，让这棵知识树有主干和枝叶。然而这棵知识树要如何形成？第一步你必须对所关心的领域中，有用的书籍或是数据非常熟悉。

2. 形成你的知识树　　我昨天还请教林毓生院士，他今年已经七十几岁了，我告诉他我今天要来作演讲，就问他：「你如果讲这个题目你要怎么讲？」他说：「只有一点，就是那重要的五、六本书要读好几遍。」因为林毓生先生是海耶克，还有几位近代思想大师在芝加哥大学的学生，他们受的训练中很重要的一部份是精读原典。这句话很有道理，虽然你不可能只读那几本重要的书，但是那五、六本书将逐渐形成你知识树的主干，此后的东西要挂在上面，都可以参照这一个架构，然后把不相干的东西暂放一边。生也有涯，知也无涯，你不可能读遍天下所有的好书，所以要学习取舍，了解自己无法看遍所有有兴趣的书，而且一但看遍所有有兴趣的书，很可能就会落得普林斯顿街上的那位旧书店的老板一般，因为阅读太多不是自己所关心的领域的知识，它对于你来说只是一地的散钱。

3. 掌握工具　　在这个阶段一定要掌握语文与合适的工具。要有一个外语可以非常流畅的阅读，要有另外一个语文至少可以看得懂文章的标题，能学更多当然更好，但是至少要有一个语文，不管是英文、日文、法文……等，一定要有一个语文能够非常流畅的阅读相关书籍，这是起码的前提。一旦这个工具没有了，你的视野就会因此大受限制，因为语文就如同是一扇天窗，没有这个天窗你这房间就封闭住了。为什么你要看得懂标题？因为这样才不会有重要的文章而你不知道，如果你连标题都看不懂，你就不知道如何找人来帮你或是自己查相关的数据。其它的工具，不管是统计或是其它的任何工具，你也一定要多掌握，因为你将来没有时间再把这样的工具学会。　　4. 突破学科间的界线　　应该要把跨学科的学习当作是一件很重要的事，但是跨学科涉及到的东西必须要对你这棵知识树有帮助，要学会到别的领域稍微偷打几枪，到别的领域去摄取一些概念，对于本身关心的问题产生另一种不同的启发，可是不要泛滥无所归。为什么要去偷打那几枪？近几十年来，人们发现不管是科学或人文，最有创新的部份是发生在学科交会的地方。

5. 论文题目要有延展性　　对一个硕士生或博士生来说，如果选错了题目，就是失败，题目选对了，还有百分之七十胜利的机会。这个问题值得研一、博一的学生好好思考。你的第一年其实就是要花在这上面，你要不断的跟老师商量寻找一个有意义、有延展性的问题，而且不要太难。我在国科会当过人文处长，当我离开的时候，每次就有七千件申请案，就有一万四千个袋子，就要送给一万四千个教授审查。我当然不可能看那么多，可是我有个重要的任务，就是要看申诉。有些申诉者认为:「我的研究计划很好，我的著作很好，所以我来申诉。」申诉通过的大概只有百分之十，那么我的责任就是在百分之九十未通过的案子正式判决前，再拿来看一看。有几个印象最深常常被拿出来讨论的，就是这个题目不必再做了、这个题目本身没有发展性，所以使我更加确认选对一个有意义、有延展性、可控制、可以经营的题目是非常重要的。

6. 养成遵照学术格式的写作习惯　　另一个最基本的训练，就是平时不管你写一万字、三万字、五万字都要养成遵照学术规范的习惯，要让他自然天成，就是说你论文的脚注、格式，在一开始进入研究生的阶段就要培养成为你生命中的一个部份，如果这个习惯没有养成，人家就会觉得这个论文不严谨，之后修改也要花很多时间，因为你的论文规模很大，可能几百页，如果一开始弄错了，后来再重头改到尾，一定很耗时费力，因此

要在一开始就养成习惯，因为我们是在写论文而不是在写散文，哪一个逗点应该在哪里、哪一个书名号该在哪里、哪一个地方要用引号、哪一个要什么标点符号，都有一定的规定，用中文写还好，用英文有一大堆简称。

7. 善用图书馆

不过切记不重要的不要花时间去看，你们生活在信息泛滥的时代，跟我生长在信息贫乏的时代是不同的，所以生长在这一个时代的你，要能有所取舍。我常常看我的学生引用一些三流的论文，却引得津津有味，我都替他感到难过，因为我强调要读有用、有价值的东西。

8. 留下时间，精致思考　　还要记得给自己保留一些思考的时间。一篇论文能不能出神入化、能不能引人入胜，很重要的是在现象之上作概念性的思考，但我不是说一定要走理论的路线，而是提醒大家要在一般的层次再提升两三步，**conceptualize**你所看到的东西。真切去了解，你所看到的东西是什么？整体意义是什么？整体的轮廓是什么？千万不要被枝节淹没，虽然枝节是你最重要的开始，但是你一天总也要留一些时间好好思考、慢慢沉淀。conceptualize 是一种非常难教的东西，我记得我念书时，有位老师信誓旦旦说要开一门课，教学生如何 conceptualize，可是从来都没开成，因为这非常难教。我要提醒的是，在被很多材料和枝节淹没的时候，要适时跳出来想一想，所看到的东西有哪些意义？这个意义有没有广泛连结到更大层面的知识价值。　　傅斯年先生来到台湾以后，同时担任中央研究院历史语言研究所的所长及台大的校长。台大有个傅钟每小时钟声有二十一响、敲二十一次。以前有一个人，写了一本书叫《钟声二十一响》，当时很轰动。他当时对这二十一响解释是说：因为台大的学生都很好，所以二十一响是欢迎国家元首二十一响的礼炮。不久前我发现台大在每一个重要的古迹下面竖一个铜牌，我仔细看看傅钟下的解释，才知道原来是因为傅斯年当台大校长的时候，曾经说过一句话：「人一天只有二十一个小时，另外三小时是要思考的。」所以才叫二十一响。我觉得这句话大有道理，可是我觉得三小时可能太多，因为研究生是非常忙的，但至少每天要留个三十分钟、一小时思考，想一想你看到了什么？学习跳到比你所看到的东西更高一点的层次去思考。

9.找到学习的楷模（Motivation）　　我刚到美国念书的时候，每次写报告头皮就重的不得了，因为我们的英文报告三、四十页，一个学期有四门课的话就有一百六十页，可是你连脚注都要从头学习。后来我找到一个好办法，就是我每次要写的时候，把一篇我最喜欢的论文放在旁边，虽然

他写的题目跟我写的都没关系，不过我每次都看他如何写，看看他的注脚、读几行，然后我就开始写。就像最有名的男高音 Pavarotti 唱歌剧的时候都会捏着一条手帕，因为他说：「上舞台就像下地狱，太紧张了。」他为了克服紧张，他有习惯性的动作，就是捏着白手帕。我想当年那一篇论文抽印本就像是我的白手帕一样，能让我开始好好写这篇报告，我学习它里面如何思考、如何构思、如何照顾全体、如何用英文作脚注。好好的把一位大师的作品读完，开始模仿和学习他，是入门最好的方法，逐步的，你也开始写出自己的东西。

1.26

从去年九月以来偶有咳嗽，今天从此事又联想到其他一些事（比如母亲最近动的手术），开始觉得我现在也应该开始注意去了解自己的身体状况了，有一个好的作息生活习惯是最终的目标。现在的休息时间要嘛太少要嘛太多不太有规律，准备改进。保证每天六小时睡眠是必要的，在此基础上根据自己的情况来决定增加与否。休息时间的话在 1:30am 以前为宜。与生活作息习惯相关的是平常做事生活的态度，如能做到恬淡勇敢必然会有帮助。

晚上偶然逛进范建同学在统计之都的博客thinkffan，有点意思，自己以后如果回去，统计方向就业上的一些情况可以问问他。

1.28

instantiation 具体化，实例化

1.29

“庾信平生最萧瑟，暮年诗赋动江关”——杜甫原诗，见于去年九月 VOA 对张益唐之采访

Feb

2.7

TIL “TIL”....

不浮躁的社会是怎样的？

这个回答有意思：

“不浮躁就是该吃饭吃饭，该睡觉睡觉。该看书看书，该洗澡洗澡。聊事时聊事，陪朋友时陪朋友。万事各得其所，专心在此时此刻，做每一件事。

而不是吃饭时想着别人的鱼翅海参，睡觉时想着发票报销，看书时想着如何炫耀，洗澡时想着喜酒凑份子，聊事时情不自禁总想谈钱，陪朋友时总是手痒想刷一刷微信朋友圈——所谓浮躁，也就是时时刻刻，希望以最短的时间，博取最多的存在感、优越感和自我认同。”

2.9 严格的训练！

“传播负能量：你缺的根本不是什么鼓励和赞扬，你那臭水平犯的低级错误根本没法通过鼓励和赞扬改正过来。你缺的是训练！严格的重复训练！可惜高中毕业之后，就没人花心思来训练你了，就更别提严格了。”

——振聋发聩

2.14 何谓成熟

什么样的男人算是成熟?

不需要靠一切阿谀外界的行为来获得安全感。不需要靠一切贬损外界的行为来获得优越感。不需要靠外界的一切褒扬来获得存在感。安静公平的面对一切。确实知道自己该做什么，而且更明确自己不该做什么，以及可以不必做什么。

到这地步，差不多可以算是成熟了。——张佳玮

2.15 男儿当自强

老爹上次批我现在圈子里女生太多了，少了阳刚之气，时间长了不好。有一定道理，其实不只是人，还有读的书、听得音乐这些所有亦然。

男儿要活的豁达、潇洒。男儿当自强。

2.16

有时间可以想想玩 UNO 和 Texas Poker 的基本策略和技巧

2.17 思考范式

如何提升行动力：人类行动心理学的有效 Hack

人为什么会拖延、人为什么行动力很差。近些年心理学已经取得了突破性研究成果。普通人思考目标的时候，使用的是目标意图, 是：

我要做什么....

但是，有位天才心理学家 Peter Gollwitzer 发现目标意图这样的思考范式，反而很难达成目标，于是，他对自己的实验对象，使用了一种替代范式。强迫实验对象，使用一种称之为：执行意图的思考范式来思考。结果令人惊讶，人们更容易克服拖延症、达成目标。

什么是执行意图？就是使用 if...then... 的思考范式。比如，不要再说，我要学 Ruby。而是说，如果我要学习 Ruby，那么，今天晚上就装上环境。

当你关于行动与目标，长年累月这么思考，最终建立自动化机制，那么行动力慢慢就变强大了。如果... 那么... 成为生命中的一部分。我要... 这种句式，就从自己的语言体系中死掉了。

学习编程

刚开始，不要:

- 忙于去社交
- 泡各种论坛、
- 发无聊帖子
- 下载各种盗版电子书（自己花钱买来的你才会格外心疼）
- 争议哪种语言更好

每个人的世界都是如此不同，不争议，用它做点作品，或者是送给自己的女朋友，或者是赚点外快。慢慢地，就成长了：）

2.18 志当存高远

今天值得记录的事有两件：

1. 余杰的这篇《薄酒与丑妻》很有意思。
2. 读到肖寒老师的一些链接，尤其是对他母亲的一篇采访和他高中语文老师对他的一篇回忆《志当存高远》让我印象深刻，不由想起自己的读书经历。

顺手把两篇文章记录在这。

家庭教育专访

按：一九九九年高考，六安二中十八岁的肖寒同学以 694 分的成绩夺得安徽省理科状元之桂冠，被北京大学数学系录取。为此，安徽省妇女联合会儿童部的秦丽珠同志对肖寒的母亲——六安纺织厂生产技术科朱毅峰同志进行了专访，摘录如下：

秦：是不是您的孩子先天条件比别的孩子好？

朱：不，我的肖寒先天不足，未足月就出生了，体重不足 2 公斤，还不会吮奶，我们家长硬是用自制的恒温床，用勺柄向孩子嘴里一点点滴母奶才把他从死神手里夺了回来，我当时已经 30 岁了。

秦：对肖寒的早期教育，您认为突出了哪些重点？

朱：习惯成自然，首先抓了孩子的习惯养成。如生活有规律，按时睡，到时起床；讲卫生；劳动习惯，凡自己能干的事自己做，刚 3 岁时自己穿衣、脱衣、洗手、洗脸；吃饭不挑食物。其次是爱护并培养孩子的好奇心及求知欲。肖寒二三岁时，每天晚都要听了故事才肯睡觉，为此，我家买了订了许多少儿刊物，爸爸妈妈讲给他听后，有时还要他复述，或故意留下故事结尾让孩子猜。直到现在，他的生活习惯依旧良好，求知欲旺盛。

秦：你们夫妇是不是一刚开始就要把孩子培养成高材生，将来上名牌大学？

朱：虽然高龄得独子，当初并没有考虑孩子将来考高分，上名牌大学，只是想培养一个各方面素质较高，德、智、体全面发展，将来能自食其力的、对国家四化建设有用的人。近代物理学家爱因斯坦说：“智力上的成就，在很大程度上依赖于心格的伟大。”这话对我们影响很大。另外，通过学习后我们懂得了，在同等智力水平上，一个人能否成才，他的非智力因素如抱负、自信心、兴趣、毅力、自制力、心理承受能力等起决定性作用，而这些因素往往是家庭从小培养训练才能具备的。十几年来，我们就非常注意训练孩子具有良好的非智力方面的素质。

秦：你们是如何训练非智力因素，举几个例子吧。

朱：孩子很小时我们就引导他树立远大理想抱负。一次孩子说：“我的脸要黑一点就好了，从古到今的英雄不是黑脸大汉就是红脸大汉，我的脸不黑又不红，哪能做英雄呢。”我立刻引导他说：“你想做英雄是好样的，英雄之所以是英雄，不是脸红脸黑决定的，是由他们的志向、本领和为国家、人民作的贡献决定的。你呀，要好好学知识，有本领、长大能为祖国强大做贡献，同样可以成为英雄。”我们经常这样趁势引导他确立远大理想、抱负。例如自制力的培养，在让孩子了解家庭经济困难的前提下买些

零食或苹果，说明要求孩子多长时间吃完或一天吃多少，尽管苹果就在放在他的房里，孩子从来按规定来吃。

一次孩子刷鞋子因刷不干净急得又蹦又跳，我严厉地批评了他。孩子不理解我为什么发那么大的火，感到很委屈，事后我因势利导地对他说：“我发火并不是因为你鞋子没刷好，而是你对待困难的态度，人生道路长得很，会有许许多多意想不到的困难的态度，人生道路长得很，会有许多多意想不到的困难的挫折，只有意志坚强的人，有毅力战胜困难并确信自己能战胜困难的人，才能达到胜利的目的地。你这点小事做不好就生气，将来怎么面对人生？”这事对孩子印象很深，他在作文、日记中多次提到。

孩子小学时品学兼优，在班上担任了班干、队干，是深受老师和同学喜爱的“红人”。上初中选班干部却落选了，再加上仅似 1.5 分之差未考上省重点中学，孩子情绪低落，心里承受不小的压力。我们及时引导，让他正确认识选班干部落选和未考上省重点中学，把压力化为动力。告诉他当不当班干部、上不上重点中学并不重要，重要的是你的积极进取的精神，自信心要永远拥有，战胜困难不怕挫折的毅力要求永远拥有，这样才会立于不败之地。听了这些话，孩子放下了包袱，有了一种较好的心态。第二学期就被选为学习委员。应该说，这是对他心理承受能力的一次锻炼。

秦：在教育方法上，您认为哪点较成功，达到了预期效果？

朱：在教育方法上，我们特别注意，态度要温和亲切，象对待朋友一样，对孩子多观察、少指示命令、不唠叨。让孩子在一个宽松、亲切、平等的氛围中成长。孩子下午放学较迟，我下班后急急忙忙回家做饭，他爸爸经常骑上自行车去接孩子。我们考虑到孩子在校品学兼优且心地善良，文体活动十分活跃，还颇有组织才能，在同学中很有吸引力。但他毕竟处于花季的年龄，面对的是太多太多的诱惑，如社会上音像媒体方面的、小书摊上的低级趣味的读物，游乐场、电子游艺机……甚至有来自异性的爱慕追求等等。做父亲的接到经过一天紧张学习的儿子后，边走边聊，像朋友样的随意轻松。这样既使孩子远离身边的种种诱惑，又让孩子精神上得以放松，同时还加深了亲情。所以我们一 要求孩子的，他都能很好地接受或照办。孩子的意见、要求我们也多站在他的角度想一想，或采纳，或解释。

秦：十八年的家庭教育，你们作家长的很辛苦，谈谈这方面的感想好吗？

朱：好的。十八年的含辛茹苦，我无怨无悔。我们夫妇想到要把孩子培养成对国家有用的人才，我们必须为孩子树立好榜样，凡要求孩子做到的，我们必须率先做到。我们要求孩子好好学习，多阅读课外书籍，我们俩经常手不离卷和孩子一起学习。我自己 35 岁那年考进了六安师专夜大学学习，孩子爸爸后来也参加了成人高等自学考试。我们两人都从基层走进了企业的技术部门，成了厂里的技术骨干。我们要求孩子上进、守纪、认真、文明、礼貌、乐于助人，我们自己在工厂里严格遵守厂纪厂规，工作一贯认真负责，不管遇到多大困难，在家从不发牢骚，怨天尤人。邻里间互相帮助，亲密无间；夫妻间相敬如宾。身教重于言教啊。

秦：既要上班工作，又要挤时间提高家长的文化、业务水平而自学，还要照顾好孩子，你们是如何安排的？

朱：在家庭里，我们夫妻分工协作，每个阶段我俩都商量拟定出育儿方案。一般是一个陪伴孩子并兼顾自己学习，另一个做好柴米油盐，做饭等后勤工作。孩子自小学高中十二年来，我几乎每天早晨都是五点多起床，为孩子做早餐，他爸爸每晚都要等儿子睡下后才肯休息，尽管孩子并不需要大人作什么。

秦：真是舐犊情深啊！谢谢您给我们介绍了宝贵的经验。再见！

朱：再见！

志当存高远——记 1999 年安徽省高考理科状元肖寒

和肖寒在一起读过书的同学，都觉得他是一个高高个子、经常剃着平头、着装极其普通的同学；教过肖寒课的老师，都清晰地记得他是一位上课昂着脑袋、两眼紧盯黑板、积极思考发言的学生。作为他的高中语文老师，虽然他已毕业十几年，但留给我印象最深的就是他立志高远、勤奋刻苦。

肖寒的初中生活是在六安二中度过的，因成绩优异被留校就读高中。正巧，上高一后的一天，他在报纸上看到一篇刊登当年安徽省高考理科第名的报道，于是小心翼翼地把这则报道剪了下来，并端端正正地放在写字台板下。晚上，他郑重地对爸爸妈妈说：“我 定要好好学习，三年后争当全省高考理科第一名!” 在 篇周记中，他这样写到：“小鸡飞不高, 因为它的心中只有一把秕糠；小鸟飞不高，因为它的心中只有三尺茅檐；而雄鹰之所以能搏击长空；那是因为它胸怀万里!”

他是这样想的，也是这样做的。高中三年，他用“勤奋”二字来实践

自己的诺言。高二时，安徽省首届物理奥林匹克夏令营在六安二中举行。学校里第 个报名的是他，上辅导课时听得最认真的是他，活动时积极参加的也是他。为了巩固夏令营活动所学的知识，迎接下一年的全国奥林匹克竞赛，他自己买了一本厚厚的《高中物理金牌之路》。一个夏天, 他忘记了酷暑炎热，忘记了蚊叮虫咬，有时一干就到半夜，硬是用一个暑假靠自学把这本书给啃了下来。功夫不负有心人。第二年，他获得了全国第 15 届高中物理奥林匹克安徽省一等奖。

还有一件让我记忆犹新的事情。1999 年 7 月 10 日，我被抽到安徽大学高考阅卷。正当我在安大校园内匆匆行走时, 突然听到一声“陈老师，您好”的问候声。这不是肖寒吗？只见他站在一个水泥桌边，面前放着一本名叫《辩证唯物主义和历史唯物主义》的大学教材，我惊讶地问到：“你在这里干什么?”他回答道：“表哥在这里读书, 我乘暑假来体验体验生活, 早学 点大学知识。”好一个肖寒！

凭着自己高远的志向、不懈的努力，在 1999 年的高考中，肖寒终于以 694 分的成绩荣获当年安徽省理科状元！上了大学后，肖寒更没有懈怠。他曾经写信对他的父母说：“这里人才济济，高手云集，竞争非常激烈，我只有更加勤奋才行。”每当谈到这些，他妈妈总是自豪地说：“肖寒在大学非常努力，成绩优异，每学年都获得学校特等奖学金，大二被评为‘北京大学三好学生’。”他的爸爸也曾高兴地说：“他们本科四年只要完成 I50 个学分就可毕业，肖寒两年就学完 98 个学分。他准备大三就完成本科学业, 接着就开始准备考研、留学。”

以后的一切，应该说是顺理成章。2004 年，进入新加坡国立大学统计与应用概率系，2006 年获理学硕士学位，2006 年进入芝加哥大学统计系，2011 年获统计博士，2011 年开始在美国罗格斯大学担任教授。

要问人生之路有多长，就看你的理想有多高；要问人生之路怎么走，首先看你是否勤奋。这些，肖寒用行动作出了很漂亮的回答，也给我们中学生树立了精彩的示范和学习的榜样！

2.19~2.22 Plan

1) Write down the progress of current project completely.
2) Review some needed time series material for 565
3) If time allows, study the stat learning online course

PS: From today, I'll manage to form a habbit of solving math/programming problems alternatively by day.

2.20 为了自己的正解

今早，回忆起过去的事，对过往的一些情愫都有了一个平静的理解。周慧超在我的人生中是个重要人物，高中 + 复读两年半，还有大学前一两年，前后大概四年，暗恋过她。自己与人交往时候的一些情愫，应该和那个时期暗恋她密不可分。虽然今日若问我回到那时还会不会去做那些有些傻傻的事情，我的答案是绝对否定。应该说，我生于世这近二十七年，为很多没必要的情愫浪费了很多时间，这些宝贵的时间不会再倒回来了，我觉悟到了这点，从今往后不会苛责自己那些年没有选择一条努力提升自己的路，但是，我一定会通过强烈的努力来珍惜我现在的时光和知己。

梁启超先生说过，我总不惜以今日之我挑战昨日之我。自今天起我决意像先生一样去实践一番，试着去把所有无意义的情愫都抛弃了，不是说为抛弃而抛弃，而是说向着自己设定的目标坚定的向前跑，把那些情愫自然而然的抛下！这对我现在追冬冬也有意义，因为我想把一个纯粹的、有好奇心求知欲、成熟的自己展现给她，前两者也是当年我遇到她时我的模样：）

2.22 切尔西，老男孩

shingle 砂砾，鹅卵石

重温切尔西老男孩们 2012 的欧冠之路。

2.25 少年人，青年人，老年人

“三十岁的梁启超作《少年中国说》，纵谈人之老少，气吞长鲸，好不痛快! 我引申之，人有老少，文章亦有老少。少年之文章，如烈酒，使人有拔剑斫地不可一世之慨，有引吭高歌怒发冲冠之气；老年之文章，如清茶，使人有清风徐来水波不兴之感，有手挥五弦目送鸿之致。少年之文章，使人忧，使人怒，使人热血沸腾；老年文章，使人闲，使人静，使人冷眼旁观。少年之文章是流出来的，老年之文章是挤出来的。少年之文章可舒

张万物，老年之文章则无可奈何。少年之文章如“壮志饥餐胡虏肉，笑谈渴饮匈奴血”；老年之文章如“白头宫女在，闲话说玄宗”。少年之文章写未来之事，在幻想中纵横驰骋；老年之文章写过去之事，在回忆里昏昏欲睡。”——余杰《少年气盛说文章》

March

3.3 Decide to Learn Emacs

"Very often Meta characters are used for operations related to the units defined by language (words, sentences, paragraphs), while Control characters operate on basic units that are independent of what you are editing (characters, lines, etc). "

"Remember that most Emacs commands can be given a repeat count; this includes text characters. Repeating a text character inserts it several times."

"The distinction between killing something and deleting it affects whether you can yank it with C-y; it makes no difference for undo."

"If you make changes to the text of one file, then find another file, this does not save the first file. Its changes remain inside Emacs, in that file's buffer. The creation or editing of the second file's buffer has no effect on the first file's buffer. This is very useful, but it also means that you need a convenient way to save the first file's buffer. Having to switch back to that buffer, in order to save it with C-x C-s, would be a nuisance. So we have

C-x s Save some buffers"

3.4 Customizing Emacs

Easiest way to find where the user init file is:

C-h v user-init-file

Easiest way to open it is (in the scratch buffer):

(find-file user-init-file) and hit *C-j* to eval

3.5

优秀人士的特点：
勇于接受新事物
追求更强的过程本身就是目的
举一反三

3.6 足间的红线

重读草纹《足间的红线》，就是要以这样的意志来面对冬冬。或者说，if you know the direction, then the to-do list quite simple.

穆帅这话实诚：

“我得承认，当代的年轻人都梦想着一夜暴富，这是现代社会的一个问题。我怎样才能告诉球员们别瞎折腾？如果换成是我的孩子，我会怎么做？我会告诉他，年轻人得脚踏实地的感悟与适应这个世界，需要去努力工作。”

3.8 此身何惧

微博上碰到陶凌同学，我无意中坚持去做的事情却让她印象深刻。她对我说“博士，我认识你是刘同学生病，你的事情我听哭了，早就知道你但不认识. 我和我妈说你在我心里地位颇高。希望你和我最爱的蕾蕾能修成正果!”，“博士，我哭点不高，但是我只和我看得上的人交往。你是我欣赏的人！我不伪装！你有啥事我能帮忙的你说话，我不遗余力!”，“感动我的是你做而不是你写。老牛生病让我看清很多人。蕾蕾是特别好的女孩子，只有你那么好的人我才能放心”。读到这几句时我真不知道说什么好。为了她这几句知心还有鼓励的话，当浮一大白后用心酬答!

因她的话也又想起了老牛，想到现在的我能不能做的更好。因为最近天冷剪头戴起了当年谭望送我的帽子，加上老牛这事，于是特别 qq 上对她说了几句感谢的话，之后又和她聊了两句。我会记得当年她批判对的那些缺点的（主要是钻牛角尖、有时生活不够注意）。近一个月真的是把所有过去感情的结都彻彻底底的解了（不再封存!）。既然对于过去该说的都说了，那么对于未来，“敝屣榮華，浮雲生死，此身何懼”!

3.9 算法如性格，数据结构如体格，而语言只是外衣

读《C 语言点滴》

API：应用程序接口。就是软件系统不同组成部分衔接的约定。由于近年来软件的规模日益庞大，常常需要把复杂的系统划分成小的组成部分，编程接口的设计十分重要。程序设计的实践中，编程接口的设计首先要使软件系统的职责得到合理划分。良好的接口设计可以降低系统各部分的相互依赖，提高组成单元的内聚性，降低组成单元间的耦合程度，从而提高系统的维护性和扩展性。

Anchor 锚点

onus 责任，负担

overhead 管理费用，开销

We can envision（想象，展望）RAM as a long line of boxes, where each has an address.

另：《C 语言点滴》中的《熬夜指南》不错，长时间脑力劳动注意补充维生素 B 和 C，水果和干果应该都不错，另外就是一定时候后可以放松放松肌肉走动走动。

一家之言：

无论你用什么语言，什么工具，什么系统，算法和设计模式才是编程的根本。

How to search in GOOGLE?

用好 +，-，~，OR（或者 |），.. ，“ ”，* 这些基本的操作符。特别搜索命令比如 filetype, site, link, related, cache, info, location, source, imagesize, intitle, inurl, in anchor, intext.....

需要的时候，也可以用用 google 的计算器。

以后用 computer 做统计相关时要去注意的关键点：

Data 是怎么来的（如果有的话），其背后的物理或者现实意义是什么，变量间有什么已知的关联，做 model 前有预先 expect 什么没有？

而在得到某个返回值后，又可以问：这个返回的值计算了什么？是怎么算的又该怎样解读？是否是之前所预计的？下一步该做什么？

换句话说，要去弄明白得到的结果如何 *interpret*，背后有没有什么有意义或者有趣的东西可以继续做。

3.10 Apply function families in R

Q: Why R has to store all objects in memory as a consequence of *lexical scoping*?

Note

The things to consider when choosing an apply function are basically:

What class is my input data? vector, matrix, data frame…

On which subsets of that data do I want the function to act? rows, columns, all values…

What class will the function return? How is the original data structure transformed?

It's the usual input-process-output story: what do I have, what do I want and what lies inbetween?

曹薰铉谈天才

“围棋首先需要天才，而且这个天才必须焚膏继晷地付出努力。但是，眼下韩国棋坛看不到这样的天才。李世石不是天才，他属于棋风独特的‘天才型’棋手。我的师兄吴清源（1914~）是天才，而且极其勤奋。他从小捧书打谱，以致左手手指弯曲变形了。一次，濑越先生看师兄太勤奋，就打发他去看棒球休息休息，没想到师兄不看棒球仰头看天，他是把天空当做棋盘继续研究棋。师兄今年 101 岁，他依然到棋战的观战室研讨，拿出自己的见解。如果不是天才，培养没有用。就是拼了命学，也不会有长进的就是围棋。”

PS：在关西棋院网站上看到几个棋手资料，结城聪喜欢的东西是“一生懸命な姿を見ること”，喜欢的话是“初心忘れるべからず“，大概是勿忘初心的意思；坂井秀至喜欢的话是“なせばなる”，大概是觉得自己能做的事就要勇敢去做，有志者事竟成的意思；而村川大介喜欢的则是”無心“

Coursera 基础光学结课信

在教學上，重要的不是教了多少細節，而是能不能讓學生覺得這個學問有點趣味。學生必須要能夠培養自己主動學習的精神和能力，這才是能長久伴隨著自己的能力。如果作業是上完課透過機械性的操作就可以做完的，相關印象很快就會隨風而逝。

3.11 最近很快乐

感觉到快乐，应该把快乐的源头找出来，这样以后可能更容易再进入这样的状态。感到不快乐或者烦恼也是这样，把源头弄清楚了，取舍决断起来也会更容易。比如说我这个三月以来就一直感觉很快乐，原因我想了想很简单，就是我找到了我觉得很有意思真正想去尝试的东西（比如对 emacs，linux，regular expression，编程、算法乃至对整个 computer system 的基本构建和运行原理），而另外 Research 也一直有在做没有拖拖拉拉或者虚掷光阴。以前有时很想试着改变一些坏毛病，不过总是有反复，我想一是因为没有进入一个好的心理状态，二是因为没有找到一个人生大海中可以坚定抛锚的坐标，而这两点其实又是相辅相成的。以后不管做任何事，我都会尽力去找到支持我去做的 motivation，然后剩下的事就是全身心地投入了。

现在的我确确实实体会到了“凡生活与战斗条件一致者强”的意味，我觉得 emacs 在慢慢成为我的一个生活方式，我得感谢 Richard Stall 还有参与 GNU 的人们，你们让我由 emacs 开始发觉到了许许多多有意思我非常想去了解尝试甚至掌握的东西。我希望还有更多有趣的东西能慢慢进入我的生活和习惯。

诚挚的建议（台大物理系朱士维）

- 【由终而始】比【有始有终】更重要

-事有终始，知所先后，则近道矣

- 【用以致学】比【学以致用】更重要

-与时俱进地面对科技的破坏式创新

- 【价值】比价格更重要

-先培养自己能提供的价值，而不是先问自己的价格

spurious 欺骗性的，伪造的

3.12 解放你的学习思考

“知识是从猜想开始，而猜想往往是错误的。

回到你们的学习上谈，题目来了，发现自己不会，你们应该要高兴，猜想的机会来了；看到题目不会，不怕！用瞎的，就是猜想的意思。第一步就是不能一直想去追求答案，要给自己瞎蒙的空间。第二部，是要疼爱自己的猜想，作为一个猜想，最大的期待就是被反驳。

很多人可能觉得，这么忙，没有时间慢慢想。那就是选择的问题，看我们是要追求程度还是成绩，程度是长久的，成绩是短暂的。猜想与反驳的过程，才会让你感到愉快让你（的学习）被解放。”

——台大朱士维

以上观点非常有趣，我怎么没有早十年明白猜想和反驳的重要性？我其实这很多年也是一个以标准答案为主的人，初高中还常会尝试一题多解，现在已经绝少了，更别说猜想和反驳。马上就我生日了，正好把这点加入新年的 To-Do-List 吧！

3.13 祝自己二十七岁生日快乐

我决定在暑假搬离现在的住处，我会寻找一个环境相对简单有自己独立空间的地方继续住，全身心地投入 research，为明年毕业努力。现在的住的客厅没有封闭，很难做到这点。室友焦扬人还不错，我和他相处地还可以但可惜不够深，他很爱整洁，和他相处这些方面自己也随之更注意，但人生学业上讨论的很少，也很少能从他处感觉到真知的启发（这也和讨论本身很少，还有两人的气质不同有关）。再考虑到相对我客厅的环境，房租确实贵多了，而下学期的 funding 还不确定。综合这三点，我要搬出去。古有孟母三迁，我也这么做。

生日这一天，专门祝我生日快乐的有爸妈、佘头（+ 他拿 offer 的好消息）、冬冬、三妹、雪吨、李简、霜晚、李老师、王头、贾娃、孙栎、陈汫这些人（晚上又陆续收到小羊群里大家、杨兄进伟、陈秋月、冯龙、李承芮四人的祝福）。大部分是亲人老友，有几位现在联系的很少但相识也算很

久了。谢谢他们还记得我。人活于世，能报答的人真心不多，也因此一定要有选择。我的原则其实就一个——“范、中行氏皆众人遇我，我故众人报之。至于智伯，国士遇我，我故国士报之。”

最近最大的改变是 motivation 和人生态度，很开心很快乐，同时也有意识到仍有一些浪费时间的无意义习惯存在，在新的一岁，我非改变某一些不可。第一期 to-eliminate-list：

1）周中非急需查询的事不上微博

2）少发朋友圈状态，且上一条同样 apply 于微信朋友圈

3）少做无意义的思考，此条最为重要

有破就当有立，第一期 to-establish-list 如下：

0）凡事，寻找自己的正解并忠实地表达自我！喜欢的不喜欢的不要憋着，能直接表达的都要直接表达，不要再顾虑实现无意思的思虑那么多该不该啦！考虑到自己有时候会压抑自己的心性和想法，这条目前最为重要

1）形成 If……then……的思考习惯

2）做不同事形成不同的 mode，比如自己静思的时候就要能抛开其他所有一切；而待人接物时候要落落大方，不是别人说一句你被动应一句，而要同时主动地思考判断

3）【由终而始】+【用以致学】+【猜想和反驳】，实践朱士维老师所提的这有趣的几点

4）做事永远要问问自己 motivation 在哪？处事有缓急轻重，可以轻妙时当轻妙，无法轻妙时就当以弘毅自勉

5）解题、编程实践尽量每天 alternatively 地交互进行

其他努力什么这些很自然的东西就不多说了。

3.14 春假开始

下午与杨婷聊了挺久的，也让我得以多了解了一下她。以后有 Data Mining 的问题可以找她讨论。

3.17 状态不好或是 motivation 不足的时候怎么办？

真正的 motivation 不会那么弱的，如果还没有就不停地找寻它！在状态确实不好的时候，不妨做一些自己觉得真正快乐的事情，比如跑到球场独自练球、翻阅一本喜欢的书、回忆过去经历的一些有趣的事情。

3.18 持之以恒，润物无声

想起贾思敏小姐的话：“我觉得追蔡东磊持之以恒润物无声偶尔死缠烂打偶尔若即若离，保持距离之余又让她觉得你一直在身边……默默提升自己也不能放弃无声的电话，哪怕相对沉默呢。。。让她变成习惯。你知道她都是被动接受一切的，不过不要为了找话题变成 drama king。”

梁萧

“光阴寸箭，一发三载。吾性拙驽，穷先人之智，兀自耿耿，落魄西去，以求解脱。朝夕得君眷顾，惶惶然无以为报。人生聚散，譬如朝露，洒泪而别，莫如悄归。梁萧再三顿首，不知所言。”

三月剩余时间的计划

1）完成 R Exercise 中的所有习题

2）阅读《R 编程艺术》Ch1-Ch8，重点读扩展案例

3）有目的地准备 587 期中考试，涉及 logistic regression 的时候可以阅读 ESL 相关章节

4）读完老板关于 LARS 的 notes

最近觉得比较 meaningful 的事情

1）Multivariate 的课本不错，一定要把一二章的题好好写写

2）王头之前和我提 C++ 和 JAVA 中先学后者比较好，我比较了一下之后还是觉得学 C++ 比较好，因为有助于了解计算机系统的底层。

3）最近 Courera 要开一门通过游戏学 Phython 的课，到时会看一下，未必会有时间跟着学下去，但觉得这种学语言的方式非常 motivated，于是也进而想我现在学的 R 和 C++ 有木有这样类似可以做的东西呢？

4）想试着看能不能多用用Feymann 技巧

3.19 勤能补拙是良训，一分辛苦一分才

晚上把办公室基本整成了自己的书房，以后有啥要读的书基本都可以在办公室阅读了。

自我缺点以及优点（特质）的小结：

缺点：浅尝辄止眼高手低不求甚解、有时候心躁（无意义的思考和忧虑过多），有时候不知道拒绝、懒惰

优点：有时候骨子里的认真（重诺也可归此一类），目前也只能想到这一点了 =。=

我想，我应该多一点再多一点那样的认真时刻，因为唯有认真，才能够展现一种非如此不可的气质。

回文平方数

罗丹向我提了一个找出一个 32 位的回文平方数的问题，网上看到有人有 C++ 找出来了，用了 400 多秒，我在想那人的 code 会是怎样的。

3.20 How to write great codes? How to think like a programmer?

Starch 淀粉；形式主义

esoteric 深奥难解的

3.21 凸分析, 态度

今天认真阅读了《非线性优化基础》的凸分析一章，感觉不错。意识到这部分内容中的重要结果和技巧（比如凸函数（包括加了一阶或二阶连

续可微条件的时候）的几个充要条件是怎么倒来倒去的？）对未来应该会有帮助，所以有些 result 一定要自己再写一写。另外自己还要多想想这里接触到的一些 notion 的几何意义和性质，比如分离超平面、锥和极锥、（凸函数的）共轭函数等。今天阅读还有一个初步印象是——在凸意义下，下半连续往往可以 imply 很多闭的结果。

态度

今天读书的状态特好，重新找回了以平静的心态读书时那种安宁自得的感觉。心底始终还存着那个关于 altitude 的故事，“altitude, after all, is everything”，这话我由衷赞同。自己因那年纠结所形成的心理反应肯定还会不时跑出来折腾我，不过我如果能够平静（并略带自嘲:））地面对，应该能轻松跨越每一次的。

又想起《统计学习那些事》的结尾语：世间是否此山最高，或者另有高处比天高？答曰：在世间自有山比此山高，*Open-mind* 比天高。

相棒 S12E19

晚上看了相棒 12 的最后一集，小野田最后还是保护了那个证人，这集值得回味的是法律与权力的关系，还有父爱。

多尝试多比较

睡前比较了 round,trunc,signif,floor,ceiling 这几个 R 函数的不同

3.22 一以贯之，读《Linux 桌面使用之道》

Test for Goodness of Fit

学习了如何检验一个 data 是否近似服从一个泊松分布。

Prussian 普鲁士的

读《Linux 桌面使用之道》

如题，这篇文章写的真好，有一些观点如醍醐灌顶。希望我慢慢也能弄清如何用 *emacs* 看网页、收邮件、做笔记、管理文件，用命令行听音乐、

看视频、备份系统，还有如何使用分布式版本控制软件 *git*。另一个体会是在软件使用上“选择最好并一以贯之”很重要。下面是该文的摘记：

1. 为什么你的 Linux 水准原地踏步？

1.1 消费主义

Linux 下的免费软件很多, 许多人就要把所有有趣的软件都玩遍. 这就和高富帅收集跑车, 白富美收集包包一个概念. 但是问题的关键是时间都被浪费了, 学习 Linux 的初衷被丢之脑后.

1.2 自我主义

例如, 拒绝使用更好的软件, 因为" 我习惯了老的软件" 或者" 我写了脚本可以实现类似功能" 或者" 我的人生哲学等等"

1.3 缺乏经验, 轻信他人

典型的就是如果有个新的最酷的文本编辑器出来了. 你立即丢弃了你正在用的 Emacs, 因为你轻信了新软件的广告. 一旦那个软件停止升级, 发觉你在其上时间精力投资也就白费了. 也许在转投阵营前调查软件后面的人 (用户和开发者) 可能更好一点. 又比如有个开源界的知名人士宣称 Unix 的原则就是很多小的独立的程序通过纯文本协议相互作用. 在你把这当成黄金法则到处乱应用之前, 也许你应该了解一下该大牛讲话的风格就是比较偏激的.

1.4 自我管理差

年青的我就是最好的反面典型. 当时我自认为自己智商高, 记忆力好, 所以学到有用的知识后没有记下来, 后来全忘了. 现在我醒悟了, 我会写下心得, 公开发表, 同时增量备份到全世界至少三个服务器上. 在若干年内不断的维护改进. 比如我写的一年成为 Emacs 高手就是一个活生生的例子.

2. 哲学

2.1 长期投资

软件对我而言是长期投资. 所以我会花很多时间学习软件的快捷键 (快捷键长期来说操作比鼠标高效), 那种只能用鼠标操作的软件对我来说就是"坏" 软件.

2.2 理性思维

以文件管理器来说, 在 Windows 平台下我长期使用并热爱 Total Commander, 在 Linux 下平台下也有几款文件管理器在界面和功能上和 Total Commander 上非常接近, 自然从感情上讲我会爱屋及乌. 但是我最后还是选择了以 Emacs 作为主要的文件管理器, 以丑陋的 Midnight Commander 为辅助, 因我已确定选择软件的总的原则是尽可能用少的的软件做尽可能多的事. 具体说来就是以 Emacs 做任何事, 如果一定要用其他软件辅助 Emacs, 快捷键也应是 Emacs 的.Midnight Commander 符合该条件

2.3 科学方法

细节见后文, 我要强调的是我的工具和方法都是有高手验证过, 有官方文档支持. 对我而言, 科学的含义, 就是前人反复验证过的, 过去 100% 成功也许能保证现在将来可能成功 (因工具环境不断在变化, 所以只是有可能). 除此以外都不叫科学.

2.4 以人为本

在后文我会列出一些筛选软件的方法. 也会推荐一些软件. 但你决不要认为只要用了这些软件或者记住了我的方法就能成为高手了. 这只是敲门砖而已. 实际上我推荐的不是软件, 而是人. 这些软件和用户和开发者是一个精英荟萃的圈子. 重要的是你要理解这个圈子的风格.

3. 少而精

软件数量少, 品质高, 功能强, 依赖小, 界面通用.

3.1 数量少

这点很重要, 只有需要使用的软件少 (但是完成的工作一分都不能少), 才可能精通. 软件少的另外一个好处是维护省心, 例如你发觉某个软件的快捷键和另一未知软件有冲突, 找出另一个未知软件的工作量和你在系统上安装的软件数量有重大关系.

3.2 品质高

我对于品质的要求很简单:

久经考验 (例如 Emacs 的开发历史有 35 年以上)

聪明人认同

何为久经考验, 何为聪明人, 我相信是可以找出很多客观的指标的. 指标越严格, 筛选结果就越少.

3.3 功能强

原谅我又用 Emacs 举例, 用 Emacs 可以看网页, 收邮件, 写程序, 做笔记, 写博客, 读 rss. 附带说明一下, 在我罗列的这些功能中,Emacs 都是极为优秀, 很少有同类软件可以媲美. 事实上, 功能强的终极境界就是提供了一个无所不能的可编程平台.

3.4 依赖小

例如, 命令行软件不依赖于 QT,GTK 之类图形界面库, 安装包小很多, 运行时消耗内存也小, 启动快.

3.5 界面通用

例如, 我写程序用 emacs, 一般操作用 bash shell, 文件管理用 Midnight Commander(mc), 上网用 Firefox, 但是通过适当的设置 (Firefox 需要安装插件 keysnail, 其他软件只要用最新版即可), 他们的快捷键都是 Emacs 的.

(后文软件推荐各节略）

Subsetting Principles

OOB: out of bounds

holistic: 全盘的，整体的

trade in 以旧换新

meta 元

Remember to use the vector boolean operators & and |, not the short-circuiting scalar operators && and || which are more useful inside if statements. Don't forget [De Morgan's laws], which can be useful to simplify negations: !(X & Y) is the same as !X | !Y，!(X | Y) is the same as !X & !Y. For example, !(X & !(Y | Z)) simplifies to !X | !!(Y|Z), and then to !X | Y | Z.

其他

今天趁着修车的空闲时间把 simple function 一节写 autocorrelation 函数的问题搞定了，当然答案还是比我的 succint，用的是 sapply(1:k,FUN) 的形式。下午阅读了 Wickham 的 Subsetting 一节，收获不少，不过内容很多需要慢慢消化。晚上陈秋月问了我好几个 master exam 的问题，其中有一个由 X 与 Ycorrelation 为 1 的信息导出 X-Y 分布的题还挺有趣的，另有一题涉及到 three way contingency table 的自己不懂。

3.23 故书不厌百回读，熟读深思子自知

做题有益

上午为一个江西老乡做了 master exam 的 tutor，帮她解答了几个问题。一个是 unequal scale parameter 指数分布 mix 的问题，要求 P（X-Y），where X~exp(a), Y~exp(b), 且 X、Y 独立，我猜想应该一般的有 P（X-Y）=b/(a+b)。一个是略带技巧性的求由特定字符 pattern 出现次数而定义之随机变量的期望的问题，其第二问如果 approximate P（X>=2）值得再想想。还有个问题基于 Yij=Xi+eij(i 表示个体，j=1,2 表示 test 的次数，eij 相互独立) 的 model, 问 estimate eij 的方差用什么统计量好，由此还涉及 unequal variance 的 testing。虽然是做 tutor，但自己也发觉有一些值得回头翻书或者思考的东西。

有些书常读常新

下午重新翻了翻《非线性优化基础》的第二章，重温的时候有不少新收获（比如对 co、cl、ri 还有求极锥等几种运算的理解，从支撑超平面的角度如何自然引出次梯度的定义，不足的是对共轭函数、方向导数与次梯度的关系还有不少疑惑）还有为什么真是觉得“故书不厌百回读，熟读深思子自知”。

几日小结

读数学的东西非得能静下心来细思、动手用笔在纸上试算方有真提高，单单一时记住几个概念、性质和定理是不会有大帮助的。这点和学习自然语言和计算机语言是一样的，记住单词和语法并不足以写出好文字和好程序，必须不断自己动手去写，不时用高于自己现有水平的东西来 challenge 自己才可能有所提升。而好的直觉和条件反射也只有在这样的过程中慢慢找到。最近还有一点觉得要注意的是，觉得有趣的东西很多是好事，但在某一段时间看的太杂太多也没必要，反倒是要尽量优先掌握和理解重要内容的 *gist* 并试着用自己的语言精炼地表达出来，慢慢使之成为自己的一部分。最后一点是，多用 Feymann 技巧来 review 所学。

3.24 不自觉做错事

理解听讲比记笔记更重要

下午上多元统计分析，觉得自己现在上课抄笔记有时候会影响到听讲，这样的倒不如专心于听讲，笔记能记则记，没时间记那便作罢。忘了从什么时间开始上课有时候会因为记笔记而忽略听课，所以决心从现在开始改变。有时候课上的细节并不需要记，记在脑子里或者课后查书即可，重要的是理解演讲者的思路和强调的重要部分。

不自觉做错事

晚上才知道昨天上午给老乡答疑的时候居然忽略了陈秋月的敲门声 =。= 加上之前易兰学姐的外卖等事件，我以后能注意还是尽量多注意，一是说话不要说太快太随意, 二是有些事先措施能避免误会的话何妨去做呢（陈秋月的 case 比如把办公室门打开）?

3.25 先成为生活上的强者，才能成为棋盘上的强者

贯彻信念

有些事你相信不对，有些标准你认同，可若不去在行为上贯彻你所相信的这些，那你自己本身的很多行为也会变得苍白。我不做这样的人，也因此从今往后，我会坚定地拒绝一些坏东西。

友情、爱情和亲情

碰到王侃，聊了不少，也把我现在的好状态还有对冬冬的心态告诉了他。他和我说，友情爱情这些的啊发展到最后都成为了亲情。或许吧。我说我还有几年觉悟来贯彻我的爱情理想，他说我痴，是的，也没几年好痴了不是？就让我无条件的再实践几年吧。而且，说曹操曹操到，聊到冬冬的时候就看到她在微信里 post 了新状态，说是在纠结装修的事情。她说她之前觉得简约就够了，实际操作起来才发现远不是这样。

生活上的强者和棋盘上的强者

晚上看了一会围棋 TV 一期王雷陈一鸣对最近时越 VS 朴廷桓一局的视频解说，中间王雷谈到围棋手们对于心态的调节的时候说了一句，“先成为生活上的强者，才能成为棋盘上的强者”，并同时举了时越、古力、李昌镐的例子，说得很好。另外他还提到状态好的时候和不好的时候棋力的差距有两子，我也深表认同，自己的体会也是状态（心态）好的时候的效率完全可以数倍于状态（心态）不好的时候。

我的一个毛病是拖拉的时候做事经常要弄到最后一两天最后几个小时才来解决问题（比如好几次 meet 老板的时候，惭愧 =。=），最近因为 motivatived 在学和生活，所以不觉得拖拉，不过昨晚到今天（现在！）一旦稍缓下来，又有拖拉的迹象 =。= 我必须跨越！

PS：最近有时忍不住跑去看围棋视频，按自己之前的想法是该放下围棋一段时间才是的。。Anyway，忙起来这些就慢慢放下了吧，不过还真不敢说不去看视频了 =。=

Stack 的三种含义（数据结构，代码运行方式，内存区域）

见阮一峰《Stack 的三种含义》一文，明白了线程和进程、stack 与 heap 的关系。看了文后读者的评论亦进一步知道实际上后两种含义都是第一种含义（数据结构）的应用，也进一步得到阅读 CSAPP（*Computer System: A Programmer's Perspective*) 的推荐。

其另有《进程与线程的一个简单解释》一文，用 CPU 和工厂、进程和车间、线程和工人 etc 类比，比喻的很生动。最后这段评论也让我印象深刻：

操作系统的设计，因此可以归结为三点：

（1）以多进程形式，允许多个任务同时运行

（2）以多线程形式，允许单个任务分成不同的部分运行

（3）提供协调机制，一方面防止进程之间和线程之间产生冲突，另一方面允许进程之间和线程之间共享资源

3.26 你做事做人彻底吗？

“所有的懦弱都出自于没有爱，或爱地不彻底，这两者一样。”

第一次听外甥女叫我舅舅

今天看到外甥女金京叫舅舅的音频，我依稀能从中辨出“舅舅啊……舅舅呢……我在等着你……蹦蹦跳跳”这些语音，后来还在微信上看到一张金京和兔子对视的照片，真是太可爱了。

3.27 弱国无外交，懒人无选择

知乎上有个这样的问题——「人生的成败和努力往往无关，只和关键时刻的关键选择有关」怎么看待这句话？

马伯庸的回答并无特别之处，总体的意思无非是“若不努力那可能连选择的机会都没有”，但还是击中了我，尤其是重读到这幅对联的时候：

有志者事竟成，破釜沉舟，百二秦关终属楚

苦心人天不负，卧薪尝胆，三千越甲可吞吴

蓝色字，是文眼。正可能因为我现在离这两处文眼还有距离，所有方读来有戚戚然的感觉吧。

3.28 CI 的构造没学好

今晚陈秋月问了我两个有意思的问题，分别是有关 poisson 和 exponential 分布参数的 CI 的构造，其中 poisson 的 exact CI 的构造涉及泊松与 Gamma 分布的一个重要联系，而 exponential 的 CI 涉及其 variance 用 MLE 还是样本方差估计好的问题。

由此，我感觉很有必要温习常用分布的性质和联系。

3.31 非先有气质不可

“人并没有什么可值得骄做的。

我不大喜欢大道理，不过上述确是我的人生哲学。人，没有什么可骄做的，说什么都没有，有点儿过分，也许零七八碎地多少有那么一点点儿吧。那么我来订正一下：也许有那么一两点可以使人感到自豪的东西，但是顶多不过如此。

我觉得，决定事情成败的是极其微妙的因素，一盘棋如此，社会上所有事情的运行不也是如此吗？这是我在漫长的围棋生涯中痛切地感受到的。这些微小的东西积少成多，就可以造成巨大的成果。

你说什么呀！这不是很简单的道理吗？作为日本人，这是古往今来老幼皆知的。“积土成山”。

是的，不用说那些高深的道理。确实，有这个成语就足够了，尘土就是那些微小的东西。我们棋手苦心积虑、力求掌握的，正是这些微小的东西。这时，我们自己也是尘土，自始既不是伟人，也不是强人。但是，如果挖山不止，有一天，也许真的能够移动大山，成为一个伟人。”

——赵治勋《超越实地和模样》

人生，要活得潇洒，豪爽和侠义。男人的价值不是金钱、名誉, 比起女人来更由“人生哲学”来决定。

在年轻时，不要去为自己的力量不足而烦恼，只要全力以赴去拼就行了，反正也是不知道自己的潜力，也没必要去深深地苦恼。

在这世上，拥有不同才能的人很多。为了让这个才能开花，我想，需要环境，需要学习的姿势，需要努力。

世界上包罗万象的所有，都有可以学的东西，以这样的想法理解人生的话，不久，这个人所拥有的才能就一定能开花。

我的想法与这种看法不一样。我教出我的知识，没有什么可以值得可惜的，如果多少被人吸收，并且有一定收获收益了的话，我是最欢迎的了。增加一个强手，也是增加围棋世界的一份财产。而且，他们即使渐渐变强，我也不认为我会轻易地就输给他们。（*从这句话可见，藤泽也有着 Richard Stallman 所珍视的那种分享精神*）

——藤泽秀行

Note: Bear in mind the plotting scale, it's important how you decide on what scale to use.

April

4.1 平常心与男儿胸怀

鸡汤两则：

1）学技术不要死记硬背，一定要多问“为什么要这样”？“为什么是这样”？原理搞通了一通百通。千万不要遇到什么“难以解释的事”就说自己被坑了。

2）**阿里的新同学入职的时候问我，怎么学好技术？我说一定要学好基础，比如:node.js，nginx，以及 Java 的 NIO，虽然这些东西表面上不同，但是如果你把基础知识——操作系统的 I/O 模型搞懂了，这些东西就一下子就只剩下 read the fucking manual 了。**人与人的学习能力强不强主要是看基础扎不扎实。

四月读书计划

上月春假期间计划完成的不错（虽然其实一月就一度计划过 =。=），R 编程上提高不少，主要因为做了不少编程实践。另外非线性规划方面也学了不少东西，可惜的是没有投入时间巩固做题之类的，忘得比较快。前两天开始继续学习 Hastie 他们的 online course，收益不少，看能够这周把课程完成拿到 certificate。

这个月主要的想法是提高对统计分析的感觉，因此主要想把 ESL 中的三五章给好好读下来。以下是粗略的阅读计划：

1. ESL Ch 3,4,7,8. CV，Bootstrap, EM 这些想法都要好好体会。还有就是不妨围绕 587 的 project 来积累相关 technique(比如 Trees，Clustering，Bayesian Network)。

2. 做一定数量的 nonlinear programming 习题并积累这方面的一些几何直觉和例子。

3. Time Series 的 Best Linear Prediction 还有 MLE、conditional MLE 的实现等内容有时间的话弄清楚。这月将要学的 State Space Model, Kalman Filter，VAR 也会很有用。

4. 学习 emacs 下的 auctex 的使用。

平常心

我有时候做着做着事情容易想到其他的事而分心，或者说暂时的 lose motivation 而左看看右看看磨蹭时间。这是很恶的习惯，要改。这里，不妨以吴清源先生那句“追二兔不得一兔”来比照自己的所为。一段时间内要尽量目标清晰的投入地做好一件事情，坚持做下去习惯就会更好。

男儿胸怀

另一个感触是昨天在读藤泽秀行先生传记时候得来。有一节说到，“男儿的胸怀，几经生与死的磨难修炼而来。人若没有经受生活苦难后的觉悟，便会被淘汰”。说的真好，我现在虽不太可能经历生与死的磨难，不过也确实明白每天做的每一件事情都有关男儿胸怀的大与小，所以对生活中的每件事，都应当作为难得的历练去经历。

4.2 Get your hands dirty!

tail off: to dwindle to nothing
parse: 解析，语义分析
mega: 宏大的，精彩的
inventory: 库存

From the Boss

Get Your Hands Dirty: to involve yourself in all parts of a job, including the parts that are unpleasant, or involve hard, practical work

ESS R package installing issue

Two approach:

1. Run chooseCRANmirror(graphics = FALSE) first to choose the mirror
2. In ~/.Rprofile file, put such: local({r <- getOption("repos") ; r["CRAN"] <- "http://rweb.quant.ku.edu/cran"> > options(repos=r)})

I would like to know more about *how to configure the Rprofile file.*

如何学习编程语言

基本成分不外四种：数据成分，运算成分，控制成分，传输成分

- 数据成分：有哪些数据类型？如何使用？
- 运算成分：运算符号有哪些？如何运用？
- 控制成分：三种类型的控制语句是如何写的？
- 传输成分：程序中如何输入和输出数据？

4.3 I realize my duty

One day I would be a father, and all the habits I have formed and gonna form, all the roads I have taken and gonna take, have a meaning.To be a GOOD father, I still have some characters and habits to form. Cheers!

4.4 *Go Back and Forth*

昨晚梦见了冬冬，出现在一次我和我朋友的聚会场合，和大家处的很融洽。这或许是我潜意识中的她成为我女朋友后的日子。最近最美好的梦，没有之一 ~

在统计、数学、编程上的中期想法（两三个月的打算）

- 统计上，想藉由对 ESL 的阅读，还有平时积累的 real data analysis 的积累，形成统计分析上更好的感觉。

- 数学上，想完成 multivariate 书上前两章的习题以及对常用分布性质的积累还有对 conditional prob 的更熟练运用，还有就是 research 相关的 nonlinear prog 尽可能多学习多算题（算题上自己还有些没弄清怎么做）

- 编程上，在完成对《R 编程艺术》一书的阅读并完成《R Exercise》上所有习题之后，不间断地用合适的编程问题 challenge 自己用 R 实现。另外，感觉单读《C++ Primer》来学 C++ 的话可能周期会比较长，不如在合适的时间从李戈老师的课入手，快速的切入 C++ 的学习，能用这门语言做基本的小型编程。

读书之法

这段时间学了不少东西，也发现了自己在学习习惯上的一些问题。东学学西学学并不好，这时候就该沉下心来figure out 重要性和优先级来进行有选择的学习，只有把重要和基础的东西理解好，前进起来才会更有力。另一个感觉是，在状态好的时候，自己的联想能力很强，在状态差的时候，阅读和上课都比较被动，读着读着就困了或者被动的听着这样情况偶有发生。看来得找法子把好的状态下的条件发射（怎么提问、怎么联想、怎么类比）之类的 build in 到日常当中来，成为一种生活状态。

前进，但要不时回去以增强你的信心

如题，想起我本科时记在《代数学引论》开卷第一页上的一段话。大意是，当时阅读枯燥段落的时候不妨越过它们到后面去寻找 motivation。*Go Back and Forth* 应该是学习的常态。

Q: 最近有时间的话试着小结 Cross Validation 在我已知的统计方法中的应用，比如 Lasso，Forward Stepwise Selection, SVM, etc.

4.5

Mac 下查看隐藏文件

In terminal, enter as follows:
defaults write com.apple.finder AppleShowAllFiles YES

Q: What's the difference between semantic and syntax, etc?

4.6

法正

"今游侠，其行虽不轨于正义，然其言必信，其行必果，已诺必诚，不爱其躯。"法正骨子里和刘备一样，都是游侠秉性。爱憎分明，意气任性。

4.7 The Hacker Altitude

Python 配置好了

在 OS X 和 Win8 上都为 emacs 配置好了 Python 2.7，以后需要或者有时间的时候随时可以开始学。

feat 功绩，伟业；卓越的手艺、本领
imperative 必要的；命令的
procotol 协议，草案
sap 使衰竭
ecstacy 迷幻药
deja vu 似曾相识的感觉
holistic 整体的，全盘的

Lisp and Non-Lisp

- If Lisp is the result of taking syntax away, Perl is the result of taking syntax all the way.
- Lisp is the greatest programming language because its minimal (i.e. practically non-existent) syntax makes Lisp macro programming powerful.

以后有机会的话我会学学 Emacs Lisp 来了解这个世界。

The Hacker Attitude——by Eric Raymond

1. The world is full of fascinating problems waiting to be solved.
2. No problem should ever have to be solved twice.
3. Boredom and drudgery are evil. (drudgery 单调沉闷的工作）
4. Freedom is good.
5. Attitude is no substitute for competence.

Learn how to program

This, of course, is the fundamental hacking skill. If you don't know any computer languages, I recommend starting with Python. It is cleanly designed, well documented, and relatively kind to beginners. Despite being a good first language, it is not just a toy; it is very powerful and flexible and well suited for large projects. I have written a more detailed evaluation of Python. Good tutorials are available at the Python web site.

I used to recommend Java as a good language to learn early, but this critique has changed my mind (search for "The Pitfalls of Java as a First Programming Language" within it). A hacker cannot, as they devastatingly put it "approach problem-solving like a plumber in a hardware store"; you have to know what the components actually do. Now I think it is probably best to learn C and Lisp first, then Java.

There is perhaps a more general point here. If a language does too much for you, it may be simultaneously a good tool for production and a bad one for learning. It's not only languages that have this problem; web application frameworks like RubyOnRails, CakePHP, Django may make it too easy to reach a superficial sort of understanding that will leave you without resources when you have to tackle a hard problem, or even just debug the solution to an easy one.

If you get into serious programming, you will have to learn C, the core language of Unix. C++ is very closely related to C; if you know one, learning the other will not be difficult. Neither language is a good one to

try learning as your first, however. And, actually, the more you can avoid programming in C the more productive you will be.

C is very efficient, and very sparing of your machine's resources. Unfortunately, C gets that efficiency by requiring you to do a lot of low-level management of resources (like memory) by hand. All that low-level code is complex and bug-prone, and will soak up huge amounts of your time on debugging. With today's machines as powerful as they are, this is usually a bad tradeoff —it's smarter to use a language that uses the machine's time less efficiently, but your time much more efficiently. Thus, Python.

Other languages of particular importance to hackers include Perl and LISP. Perl is worth learning for practical reasons; it's very widely used for active web pages and system administration, so that even if you never write Perl you should learn to read it. Many people use Perl in the way I suggest you should use Python, to avoid C programming on jobs that don't require C's machine efficiency. You will need to be able to understand their code.

LISP is worth learning for a different reason —the profound enlightenment experience you will have when you finally get it. That experience will make you a better programmer for the rest of your days, even if you never actually use LISP itself a lot. (You can get some beginning experience with LISP fairly easily by writing and modifying editing modes for the Emacs text editor, or Script-Fu plugins for the GIMP.)

It's best, actually, to learn all five of Python, C/C++, Java, Perl, and LISP. Besides being the most important hacking languages, they represent very different approaches to programming, and each will educate you in valuable ways.

But be aware that *you won't reach the skill level of a hacker or even merely a programmer simply by accumulating languages —you need to learn how to think about programming problems in a general way, independent of any one language. To be a real hacker, you need to get to the point where you can learn a new language in days by relating what's in the manual to what you already know. This means you should learn several very different languages.*

I can't give complete instructions on how to learn to program here —

it's a complex skill. But I can tell you that books and courses won't do it —many, maybe most of the best hackers are self-taught. You can learn language features —bits of knowledge —from books, *but the mind-set that makes that knowledge into living skill can be learned only by practice and apprenticeship. What will do it is (a) reading code and (b) writing code.*

Peter Norvig, who is one of Google's top hackers and the co-author of the most widely used textbook on AI, has written an excellent essay called Teach Yourself Programming in Ten Years. His "recipe for programming success" is worth careful attention.

Learning to program is like learning to write good natural language. The best way to do it is to read some stuff written by masters of the form, write some things yourself, read a lot more, write a little more, read a lot more, write some more ... and repeat until your writing begins to develop the kind of strength and economy you see in your models.

Finding good code to read used to be hard, because there were few large programs available in source for fledgeling hackers to read and tinker with. This has changed dramatically; open-source software, programming tools, and operating systems (all built by hackers) are now widely available. Which brings me neatly to our next topic...

The world is full of fascinating problems waiting to be solved.

Being a hacker is lots of fun, but it's a kind of fun that takes lots of effort. The effort takes motivation. Successful athletes get their motivation from a kind of physical delight in making their bodies perform, in pushing themselves past their own physical limits. Similarly, to be a hacker you have to get a basic thrill from solving problems, sharpening your skills, and exercising your intelligence.

If you aren't the kind of person that feels this way naturally, you'll need to become one in order to make it as a hacker. *Otherwise you'll find your hacking energy is sapped by distractions like sex, money, and social approval.*

(*You also have to develop a kind of faith in your own learning capacity*

—a belief that even though you may not know all of what you need to solve a problem, if you tackle just a piece of it and learn from that, you'll learn enough to solve the next piece —and so on, until you're done.)

Teach Yourself Programming in Ten Years ——Peter Norvig

此文常读常新。

谢枋得

“人可回天地之心，天地不能夺人之心。大丈夫行事，论是非不论利害；论逆顺不论成败；论万世不论一生。志之所在，气亦随之；气之所在，天地鬼神亦随之。”

一个网友谈到读到这段话后的感觉时说，“看了这段文字，我重塑了三观。有的人能为正义付出一切，而我却总是斤斤计较得失、权衡利弊；有的人能活得直指本心，而我却时时被功名利禄左右；有的人眼里是千百年的朝代兴衰、世事变迁，而我目光短浅，只看得见眼前。这也许就是圣人和凡人的区别吧。”

这里也简略谈谈现在的我的感受。我相信这个世界上有所谓正义的存在，但正义的标准并不那么绝对，当某个人某些人以正义之名做事的时候，须得好好辨识清楚。真正胸怀正义的人，我想不会在嘴上多言，而是果断地行动。同时，我觉得，是非要论，利害也要论，人与人、社区与社区、族群与族群、国与国这些的相处，若只有是非观而没有互相理解、互相妥协的 idea，那这世界或许会更糟糕。而且我觉得某种程度上，科学研究的基本对象也可以说是“利害”，从这个角度讲，不论“利害”则不足以论大是大非。至于“逆顺”，我个人觉得不足论，当论的是“过程”。而“论万世不论一生”一句我基本不认同，一个人不论“一生”，不弄清自己这一生想要什么，不要什么，何足以论“万世”？见了“一生”，放可能有“万世”的眼光。当然，谢枋得先生可能突出的是要有论“万世”的眼光和气质，而不是说不去论“一生”。

我最喜欢的，实为最后一句——“志之所在，气亦随之；气之所在，天地鬼神亦随之”。真性情的人、完成大事业的人都有属于 TA 们自己的志气，纵刀斧加身，不可夺也。

知友赵一鸣一回答，契合现在的自己

“还有一种爱，需要更多的努力，你除了要享受，还要奉献。

比如恋爱。

我喜欢上了一个姑娘，她在欧洲留学，我跑去找她表白，当时非常急。哎刚才教育题主一副义正言辞样子，到了自己还是关心则乱。我想过如果我们的关系能够自然过度下去该多好，可惜我们都在国内的时候只见过一面，现在隔着大西洋实在很是遗憾。可是表白之后我觉得还是不要太急了。我希望的是她能够不断变好，平心而论，异地的情况下，我很难帮助她，她只能在很多时候独自承担。可是我还想她，身上是软肋和盔甲。我受过这种煎熬，现在的态度是，还是保持那种相信，相信我们最后能在一起，然后等待，不断为之努力，完善自己，争取对面和自己都是不断变好。互动要有，但也不要太急，急了还是那个问题，太狭隘，失去了很多自己变得更好的地方。我们都变得更好，才更有能力相爱。很多故事里，干柴烈火似的一烧完，却发现两人之间各种问题的爆发已经超越了之前自己觉得能负的责任，那才真遗憾。我们反正也急不来了 *(_)*，就慢慢做准备吧。

我也想过最后还是不能在一起，我想还是淡然一点好。再不能相逢大家都老了，你嫁别人我娶别人，也没什么。到很多年之后我们真的老了，我们若能再相见，我能对你说：

“我认识你，永远记得你。那时候，你还很年轻，人人都说你美，现在，我是特意来告诉你，对我来说，我觉得现在你比年轻的时候更美，那时你是年轻女人，与你那时的面貌相比，我更爱你现在备受摧残的面容。”

我把我的知乎 *ID* 和密码都给你了，不知道你上不上，你会不会看到。

没什么必须做不做就来不及的事，除了完善自己。”

我感觉到了冬冬对我没有什么依赖，这样的状态下去是无法开始的，可又何必怨艾呢？如这位知友所说——把自己变得越来越好。现在能做的改变是逐渐摸索着和她正常地聊天打电话，那就去做呗！哪怕到某一天自己觉得自己足够好了她仍然不回应，哪怕到了那天意味着离别和前行。反正，我是不会让我和她的关系变得连朋友都不是的。

函数式和命令式编程

和命令式编程相比，函数式编程强调程序的执行结果比执行过程更重要，倡导利用若干简单的执行单元让计算结果不断渐进，逐层推导复杂的运算，而不是设计一个复杂的执行过程。函数式编程经常使用递归。纯函

数式的程序没有变量和副作用。因为纯函数式程序设计语言没有变量，函数没有副作用，编写出的程序可以利用记忆化、公共子表达式消除和并发计算在运行时和编译时得到大量优化。

The distribution fo sample median

The distribution of both the sample mean and the sample median were determined by Laplace. The distribution of the sample median from a population with a density function f(x) is asymptotically normal with mean m and variance \frac{ 1 }{ 4n f(m)^2 } where m is the median value of distribution and n is the sample size. In practice this may be difficult to estimate as the density function is usually unknown. These results have also been extended.[14] It is now known that for the p-th quartile that the distribution of the sample p-th quartile is distributed normally around the p-th quartile with variance equal to \frac{p(1 - p)}{nf(x_p)^2} where f(x_{p}) is the value of the distribution at the p-th quartile.

4.9

久坐不动不好，改！

4.10 more time on research

昨天的 meet 拖到了今天，其实也没有很忙，这两天想多了其他事（+鼻塞不舒服）。我想学习编程这些有趣的东西之外，我需要在 research 上多花时间。最近具体的计划依然参见 4.1. 的记录。自己现在的两点想法是

1. 把思考 *research* 中的小问题养成一个习惯，可以用来填充零散时间。
2. 心算和记忆力应该长期培养！（怎么做呢？）

4.11 报税完毕

mesmerize 催眠，迷惑
uncanny 离奇的，神秘的

Install R package tclust locally

尝试本地安装 tclust 的 package，配置好了系统 path 变量依然出错，后来终于意识到可能是没有安装 windows 下用来 bulid r package 的 Rtools 还有 Rbatch（这个的作用我还不太清楚）的原因，尝试下载安装前两者之后果然本地安装顺利完成。

where.exe

今天才知道 windows 下有个 where.exe 的存在，难怪那天看冯龙装某 R 包的时候有个错误觉得很奇怪，原来却是没有把这个 exe 放到 path 系统变量中所导致。

4.13 Monument Valley

Monument Valley

It is a really good game which might free your imagination in the 3-dim world. Also there is a real view site named monument valley in US.

听别人讲照片的故事

晚上和孙蝶一起看了陈秋月给我们翻看她这些年的一些照片，她边听边给我们讲，是很有趣的经历。TX 的州花、向日葵田以及历史，新墨西哥的白沙滩，犹他的 bryce canyon（看了她照片我才知道有条有趣的 trail 我没去），西雅图的 Needle，她在成都的小表妹～有时候不必身至，看别人的照片就能有很大的收获和震撼。

4.14 Linux is interesting!

directory 指南，目录
customize 定制
configuration 配置，结构，构造
miniature 微型的；微型、缩影
arcane 秘密的，晦涩的；奥义（n.)

encrypted 加密的
starch 形式主义
consolation 安慰，慰藉
vendor 卖主，供应商
erector 建设者，装配工
pulley 滑轮组
gear 齿轮，传动装置
motor 马达，发动机
concatenate 把……联系在一起
alias 化名
floppy 松散、懒散的；软盘（n.）

Cofigure Sudo in Fedora

usermod sampleusername -a -G wheel

Linux 网络基础

- IP 编址乃双层编址方案，一个 IP 地址标识一个网卡接口。现在往 IPv6 发展。
- IPv4 地址分网络部分和主机部分两部分，共 32 位。通过子网掩码来确定网络部分位数。
- 同网络内主机通信需要用到 MAC 地址（Via ARP）。
- 不同网络间的通信需通过网关/路由器，其上的数据传输功能叫路由功能，一般有多个接口，连接到不同网络中，并通过路由表进行数据转发。
- IP 地址较难记，故引入域名系统。分三部分，类型（如 net、com、edu、org、gov）、域名、主机名（如 mail、www、ftp）。
- 每个域名代表一个 IP，DNS 服务（由 DNS 服务器提供解析）用来在 IP 与域名之间进行转换。Q：*qq 能用但网络打开不了可能表示 DNS 服务器挂了*。

- 以太网连接常用命令：lspci, lsusb, ifconfig -a(或者特定接口), ifup ifdown 用来启用或禁用

4.15 LCP Problem

stale 陈腐的，不新鲜的

What is PID?

the pid files contains the process id (a number) of a given program. For example, Apache HTTPD may write it's main process number to a pid file - which is a regular text file, nothing more than that -, and later use the information there contained to stop itself. You can also use that information (just do a cat filename.pid) to kill the process yourself, using kill <the number in the .pid file>.

PID & TID

It is complicated: pid is process identifier; tid is thread identifier.

But as it happens, the kernel doesn't make a real distinction between them: threads are just like processes but they share some things (memory, fds...) with other instances of the same group.

So, a tid is actually the identifier of the schedulable object in the kernel (thread), while the pid is the identifier of the group of schedulable objects that share memory and fds (process).

But to make things more interesting, when a process has only one thread (the initial situation and in the good old times the only one) the pid and the tid are always the same. So any function that works with a tid will automatically work with a pid.

It is worth noting that many functions/system calls/command line utilities documented to work with pid actually use tids. But if the effect is process-wide you will simply not notice the difference.

技术的未来

消费是人类生存和生活的前提，本身并没有错。但是，当我们鼓吹消费文化，企图建立一个消费社会、消费人生的时候，事情就不一样了。所谓刺激消费，很大程度上是在鼓吹和放大人类享乐纵欲、贪婪懒惰的本性，以及金钱至上、及时行乐的价值观，将人类异化成消费工具。——阮一峰《技术有什么未来?》

4.18 Heading for a better Research Mode

matzoh: a kind of Jewish bread
cuttlefish 乌贼，墨鱼

Good ways of Research

- *Form the habbit of expressing yourself faithfully with a pen and scratch paper, or with a keyboard, emacs and a computer language.*
- Form a habbit to think deep, make analogy from difference sources and build your own knowledge(ability) net.
- Keep an good appetite of solving problems and some small problems could be fill in a tiny time gap.
- Learn to read papers effectively. Know where to work for detail, where to grasp the main idea, and where to obtain a holistic view.
- Have different ideas/approaches/projects stored in brain at a time but *FOCUS* on only one at a time.
- Communicate with other people and try to know what research they are doing as well.
- Learn from the advisor and take the initiative. Bring good/dumb questions and ideas to him and see how he approaches them.

Things I wanna change for better Research Mode

1. Whenner feel unmotivated, drag a pen or computer, solve some problems!!
2. Everyday you are expected to think about your research for a while. Ask questions like: how can I march further in this problem/project? etc.
3. Time and tide wait for no man. Save time as much as you can, spent time on important things, avoid unnecessary things.
4. If ..., then .... Eg. If you have some idea or feel the urge to work something out, you'd better start right now, at least today.
5. Write in English as much as you can. Speak in English as much as you can.

4.19 (Except a Few Emacs problems) My Ubuntu is ready to roll!

mediocre/mi:di'oker/ normal, just so so
steering

Ubuntu

Maximize a window: Ct+Super+up arrow; Restore a window: Ct+ Super+down arrow

Disable online search in dash: wget -q -O - https://fixubuntu.com/fixubuntu.sh | bash

Enable recursive search in Nautilus: gsettings set org.gnome.nautilus.preferences enable-interactive-search false

Disable recursive search: gsettings set org.gnome.nautilus.preferences enable-interactive-search true

Q: what's the diff between these recursive search and type-ahead search?

Check the disk partition: sudo blkid

Note:Having problem enable auto-complete, autopair and smex mode by default in Emacs. Perhaps I haven't understand some mechanism behind the initialization of the .emacs file.

Qt Project

Dong dong told me he learned Cpp through Qt library a lot. I'm considering trying that in a not far future, first I need to be able to write some small programs in Cpp.

R package installing folder

use .libPaths() to get library location, search() to see currently loaded packages, library() to see installed packages.

4.20 能力、气度和胸襟

撂 liao
peripheral 外围的，次要的
tty:teletype 电报交换机
dll:dynamic link library
dired；扩展列表，操作目录
peephole 门镜，窥视孔
spool 线圈；缠绕（v.）
daemon 守护神，半人半神的精灵；系统守护进程

《海贼王》最让你感动的一个情节，喜欢张佳玮的以下回答

白胡子一击裂岛，让儿子们撤退；自己背对深渊，面向十万海军。
他的梦想是家庭。
但旧时代已经过去了，无法回头。
以自己的死，迎来一个全新的时代。
”One Piece 是确实存在的！“
要在大海上成为强者，气度和胸襟才是最重要的。

看这条鸿沟，那厢持刀而立的，就是正待以自己的死亡，开启时代之门的世上最强的男子白胡子。

推倒一世之智勇，开拓万古之心胸。

Daemon Program

Related to services on the computer.

4.21

How hard have you worked?

我觉得我很难说出我在某个阶段特别努力，包括现在。如果你看我的社交网站，大概也是吃喝玩乐的东西多些。但我今天拥有的一切，没有任何是运气所得，因为做好眼前事是我的生活状态，我不觉得那是努力，我觉得那是我分内的事。

以后我也会一直这样生活，并且我相信我能做的更好。有句话这么说的：“你必须一直努力，才能在别人眼里看起来毫不费力。”

4.22 Try Git for the first time

antipodal: completely opposite
commit: 保证，承诺
fork: 分支，分叉

Git Setup

git config –global
git config –list
Note: Need to know more about *how to set up remote, pull and push*?

Outlier Detection

Yi Fan gave a talk on outlier detection and functional data analysis today and introduce some interesting notions like data depth, centrol region, antipodal asym distribution.

Predictive Distribution

How to interpret the standardization via the Cholesky square root in the multivariate normal setting? Note that the standardized statistic form by the normal square root is not ancillary but the Cholesky version is.

April Reading Plan Revised

Seems unable to finish the initial plan set at the beginning of this month, se here I present a modified version.

1. Study some material for research. Like KKT condition, matrix/vector differetiation, etc.
2. Study some clustering and tree methods and try out some R implementation.
3. Read one or two papers from Tyler's Multivariate course.

Some thoughts for the summer

- Select several chapters for study in the Multivariate textbook. Try to work out the problems in those chapters.
- Study ESL as much as I can.
- Choose a book on nonlinear programming which is suitable for research and study hard!
- Challenge myself regularly on programming and math(especially on optimization).

Empirical Bayes, J-S Estimator and Tweedie's formula

See eeyang's ariticle < 那些年，我们一起追的 EB> and < 昔日因，今日意 >.

4.23

pithy 简练，精辟的

Simpson's Paradox

4.24 “推倒一世之智勇，开拓万古之心胸”

坂本龙马与西乡隆盛

如今日题首，此句原来出自陈亮，我之前的印象中以为是西乡隆盛。真是大赞的一句话。而提到西乡隆盛，wiki 的这则记录非常有趣：

“倒幕成功以前西乡隆盛先是支持藩主的“公武合体”，之后主张“尊皇攘夷”，但事实上也同英国合作，努力引进西方的技术。他在倒幕运动中纵横捭阖，运用权谋。明治政府成立之后他以宋代陈龙川的话“推倒一世之智勇，开拓万古之心胸”作为座右铭。他曾在自己的日记中写道：为人当学司马温公（司马光），无一事不可与他人道，努力要求自己向完全大公无私的方向发展。

西乡隆盛曾对坂本龙马说：“你前天所说的和今天所说的不一样，这样你怎么能取信于我呢？你作为天下名士必须有坚定的信念！”坂本说：“不是这样的。孔子说过，君子从时。时间在推移，社会形势在天天变化。因此，顺应时代潮流才是君子之道！西乡，你一旦决定一件事之后，就想贯彻始终。但这么做，将来你会落后于时代的。”西乡隆盛从始至终都代表下层武士的利益，其立场并不考量他国或是维新政府整体的利益。

西乡好恶分明、热情洋溢、时而采取不合理的行动。在维新三杰中，西乡隆盛最受日本人喜爱，却以悲剧收场。”

自己在豆瓣的卷首语

反应速度和学的速度更多取决于自己以前的积累，哪怕不是直接的。远的比如，一个人从小就练过很多运动，那么，即使学一个他过去完全没接触过的运动项目，他也比从小不怎么运动的人学得快很多。近的再如，有很好微分几何基础的人，看 hartshorne 肯定理解地比较快。而并不是因为他们大脑结构多么优于常人。试想，高考完了后，彻底不看书和出于兴趣自己学点东西的人，入学后的学习困难程度能一样吗？

【我不是说人的天资没有差异，我想说的是，天资有没有差异，这个不必去想了，因为改变不了，能改变的就是多下功夫。我也不是说一定要成为大神，成为大神固然好，可是可遇而不可求，不过，人一定要找到自己的归宿，让生命借此时刻处于奔跑的状态，把空虚赶到九霄云外。这一

点要是做不到，那就生命就真的灰暗了。

“Jea: 关键是得有东西做，我个人切身体验是，光学不做东西，不算东西，要把人给搞懵。me: 对现在我有的困惑就是自己的 idea 要怎么慢慢积淀出来自己还是觉得自己有点搓的你所说的要边做东西是歌好建议 Jea: 现在，感觉就比较不错，老板给了题目，平时会和我你一起讨论一些，或计算一些东西，但在这个过程中可以学会不少理论知识和计算方法，对我们的论文都是必须的。感觉非常不错。”

R 中的整型 L

“整数型在比较上可以精确比较是否相等；浮点型则由于硬件存储精度问题，不能用“==”准确比较。另外，R 语言里，类型转换会导致内存重新分配，所以有 C 语言强迫症的同学都喜欢尽可能明确指定变量类型，避免不必要的性能损失。”

4.25 钱这么纯洁的东西，有什么不可以谈的呢？

Remember to install Okular for pdf text highlighting on Ubuntu(Inkscape also supports pdf editing)[Done].

Org-mode

I hope sometime this summer I’ll learn more about this mode in emacs and form a habbit to write notes in it.

Lambda Calculus

Alonzo Church

《谈 Linux，Windows 和 Mac》——王垠

- Linux 和 Unix 里面包含了一些非常糟糕的设计。不要被 Unix 的教条主义者吓倒。学不会有些东西很多时候不是你的错，而是 Linux 的错，是“Unix 思想”的错。不要浪费时间去学习太多工具的用法，钻研稀奇古怪的命令行。那些貌似难的，复杂的东西，特别要小心分析。

- Windows 避免了 Unix，Linux 和 Mac OS X 的很多问题。微软是值得尊敬的公司，是真正在乎程序开发工具的公司。我收回曾经对微软的鄙视态度。请菜鸟们吸收 Windows 设计里面好的东西。另外 Visual Studio 是非常好的工具，会带来编程效率的大幅度提升。请不要歧视 IDE。要正视 Emacs，VIM 等文本编辑器的局限性。当然，这些正面评价不等于说你应该为微软工作。就像我喜欢 iPhone，但是却不一定想给 Apple 工作一样。

- 学习操作系统最好的办法是学会（真正的）程序设计思想，而不是去“学习”各种古怪的工具。所有操作系统，数据库，Internet，以至于 WEB 的设计思想（和缺陷），几乎都能用程序语言的思想简单的解释。

- 一个好的工具，应该只有少数几条需要记忆的规则，就像象棋一样。而这些源于 *Unix* 的工具却像是“魔鬼棋”或者“三国杀”，有太多的，无聊的，人造的规则。有些人鄙视图形界面，鄙视 IDE，鄙视含有垃圾回收的语言（比如 Java），鄙视一切“容易”的东西。他们却不知道，把自己沉浸在别人设计的繁复的规则中，是始终无法成为大师的。就像一个人，他有能力学会各种“魔鬼棋”的规则，却始终无法达到象棋大师的高度。所以，容易的东西不一定是坏的，而困难的东西也不一定是好的。学习计算机（或者任何其它工具），应该“只选对的，不选难的”。记忆一堆的命令，乌七八糟的工具用法，最后脑子里什么也不会留下。学习“原理性”的东西，才是永远不会过时的。

- 我只是想告诉新人们，去除头脑里的宗教，偏激，仇恨和鄙视。每次仇恨一个东西，你就失去了向它学习的机会。

Latex 页眉，页脚，页码的设置以及如何用 BibTEX管理参考文献

see packge *facyhdr*
\usepackage{fancyhdr}
\pagestyle{fancy}
\lhead{abc}
\chead{\thesection}

\cfoot{\thepage}

4.26 Solarized Theme successfully customized

Emacs package initialization

What does *(package-initialize)* do ? This line also solves some old problem about the smex, auto-complete, autopair package.

4.27

swap 交换
pivot 支点

4.29 说“你启发了我”!

fuddle 灌醉，使迷糊
primrose 最佳部分；樱草的，令人愉快的
bemoan 叹息，哀叹

你启发了我

读王垠《一个对 Dijkstra 的采访视频》,《学术腐败是历史的必然》,《什么是启发》,《什么是“脚本语言”》(这篇没怎么读懂),《对函数式语言的误解》,《Lisp 已死，Lisp 万岁!》(Great article),《程序语言的常见设计错误 (1) - 片面追求短小》(Interesting!),《“解决问题”与“消灭问题”》(Enlightening!)。一些摘记：

- 作曲家的工作不是写乐谱，而是构思音乐。最早的时候人们编程都是用汇编语言的，就跟写乐谱差不多。后来他们发明了高级语言，就以为这些语言把编程的问题解决了。但是你仔细一瞧，发现它们只是把编程最微不足道的问题解决了，但是困难的问题仍然困难。这些高级语言与越来越大的野心加在一起，反而让程序员头脑的负担更重了。
- 称职的程序员都知道自己头颅的尺寸是有限的，所以他们以谦逊的态度来对待工作，像回避瘟疫一样地回避小聪明。

- 1969 年，在阿波罗号登月之后不久，我在罗马的北约软件工程会议遇到了 Joel Aron，阿波罗计划的软件负责人。我知道每个阿波罗飞船上面的代码都会比前一个多 4 万行。我不知道“行”对于代码是个什么单位，但 4 万行肯定是很多了。我很惊讶他们能把这么多代码做对，所以我问 Joel：你们是怎么做到的？他说：做什么？我说：把那么多代码写正确。Joel 说：“正确?! 其实在发射前仅仅五天，我从登月器计算轨道的代码里发现一个错误，这代码把月球的重力方向算反了。本来该吸引的，结果写成了排斥。是一个偶然的机会让我发现了这个错误。”我的脸都白了，说：这些家伙运气真好？Joel 说：“是的。”

- 程序的优雅性不是可以或缺的奢侈品，而是决定成功还是失败的一个要素。优雅并不是一个美学的问题，也不是一个时尚品味的问题，优雅能够被翻译成可行的技术。牛津字典对 *elegant* 的解释是：*pleasingly ingenious and simple*。如果你的程序真的优雅，那么它就会容易管理。第一是因为它比其它的方案都要短，第二是因为它的组件都可以被换成另外的方案而不会影响其它的部分。很奇怪的是，最优雅的程序往往也是最高效的。

- 为什么这么少的人追求优雅？这就是现实。如果说优雅也有缺点的话，那就是你需要艰巨的工作才能得到它，需要良好的教育才能欣赏它。

- 我的母亲是一个优秀的数学家。有一次我问她几何难不难，她说一点也不难，只要你用“心”来理解所有的公式。如果你需要超过 5 行公式，那么你就走错路了。

- 我的程序的“短小”是建立在语义明确，概念清晰的基础上的。在此基础上，我力求去掉冗余的，绕弯子的，混淆的代码，让程序更加直接，更加高效的表达我心中设想的“模型”。这是一种在概念级别的优化，而程序的短小精悍只是它的一种“表象”。就像是整理一团电线，并不是把它们揉成一团然后塞进一个盒子里就好。这样的做法只会给你以后的工作带来更大的麻烦，而且还有安全隐患。

- 所以我的这种短小往往是在语义和逻辑层面的，而不是在语法上死抠几行代码。我绝不会为了程序显得短小而让它变得难以理解或者容易出错。相反，很多其它人所追求的短小，却是盲目的而没有原则

的。在很多时候这些小伎俩都只是在语法层面，比如想办法把两行代码“搓”成一行。可以说，这种“片面追求短小”的错误倾向，造就了一批语言设计上的错误，以及一批“擅长于”使用这些错误的程序员。

- 一个程序的“语义”通常是由另一个程序决定的，这另一个程序叫做“解释器”(interpreter)。程序只是一个数据结构，通常表示为语法树(abstract syntax tree) 或者指令序列。这个数据结构本身其实没有意义，是解释器让它产生了意义。对同一个程序可以有不同的解释，就像上面这幅图，对画面元素的不同解释，可以看到不同的内容（少女或者老妇）。

What people in history are overrated?

What Programming Laguages Do You Need?

我的主力编程语言是 Python，其次是 C++（大坑），由于项目需求了解过 Matlab、Java、C#、PHP、Javascript 的基本语法。俗话说贪多嚼不烂，建议你广撒网多捞鱼，多了解一些语言的语法机制、设计哲学、适用背景等，然后选一种或多种符合自己胃口的语言下手。

What's the difference between Data Structure and Data Type?

‘‘首先说下数据。数据的本质是差异，每一种差异是一个值。数据的意义是用来对比。

接下来考虑数据的表示。表示数据就要把所有的差异枚举出来，也就是确定值的范围。最简单的表示数据的方式是使用二值位串。这里强调一点：数据没有类型。如，“1”是一个数字还是字符？只有在被使用的时候才知道。数据的类型在于使用其的方式。人类认知事物的典型方式是加标签，或者说分类。这种方式便于人类理解，但同样也只是人类的主观意愿。

接下来考虑数据的解释。如何判断“123456”是一个数字还是两个数字？这里需要使用约定。需要事先约定好一个数字是三位还是六位。也就是说，在解释一组数据之前需要对数据的表示方式做好约定。

接下来考虑数据存储。受那位匿名同学的启发，数据的存储方式也就

是数据结构。回想我们在数据结构课上学到的东西，数据结构考虑的通常是效率和空间这些问题。而这些东西绝对是由数据的存储方式决定的。通常一种高效的算法会使用专门为这种算法设计的数据结构。这也就是数据结构的意义。

如果一定要问数据类型是什么东西，只能说是一种人为的限制。可能最开始数据类型出现的目的是为了减少由于程序员不小心造成的错误。还有可能就是受到面向对象思想的影响。或者干脆就是别人有我也应该有的想法。”

Interpreter and Compiler

Braid

With Monument Valley and Emacs Lisp Introduction Book, it seems that I’ve supported software development quite a lot this year....

<What Can Go Wrong: My Favorite Example>

This is a very good article about what might go wrong with your stat analysis if you fail to investigate the physics behind the data. Here are some quotes.

“I’ m one of many who bemoan the fact that statistics is typically thought of as —alas, even taught as —a set of formula plugging methods. One enters one’ s data, turns the key, and the proper answers pop out. This of course is not the case at all, and arguably statistics is as much an art as a science. Or as I like to put it, you can’ t be an effective number cruncher unless you know what the crunching means.”

“Effective application of statistics is far from automatic.”

May

5.1 你要勇猛

不必要的回忆多了

昨晚看到花花和八皮微信的状态，跟着一起回忆了一下对他们的记忆。然后就收不住了，直到今下午还想着些事没有开始改作业。这已然过了。现在的我回忆多了没啥意义，还是要多去做事、学习、行路和历练。不过，很高兴知道毛驴拿到了 p6 level 的 offer，要去阿里巴巴。

月初的改进想法

过去一周明显没有集中精力做事，由昨天到今天的回忆事件也感觉到自己那些老缺点还在——觉得怅惘的时候就开始东看看西看看，把正事也拖着不做。这是恶习，要改。怎么改呢？必须在一定时间里严格的规定哪些事绝对不做，哪些事一旦发觉自己无聊就试着去做。形成习惯和条件反射就好了，可形成的这个过程会无比艰难。目前的计划是：

- 减少微信和微博的时间。不到周末不刷朋友圈，上微博只能在需要查资料的时候。另外，也不再去 163，bbc 等网站刷新闻。知乎和 CV，COS 社区可以定期看看。
- 沉浸于虚无的自我感动仍然是我的大毛病。一旦感觉有这样或者是其他无聊和惆怅的苗子，就要马上开始埋头做事。比如，解题或者进行编程实践，脑子动不起来可以考虑积累衣食住行方面的见识，比如谈吃的书。其实我知道为改而改不容易成功，最好的方式就是形成一个与之相反的气质，雷厉风行之类的。

- 减少在办公室的时间（但不意味减少和人的交流，见下条），这段时间感觉在系里待多了，有几件本来计划做的事情没有完成。虽然理想的状态是不管在哪都能专注地思考和学习，不过如果难以做到那就要为自己创造安定的条件来那么做。
- 抓住机会，和厉害的人讨论，向可学习的人学习。这点能让人少走许多弯路。而实现这点，我需要成为一个更好的思考者（比如联想和类比的能力）和提问者（需要有洞察才能有好的问题）。
- 如年初开篇所说，最重要的是，历练自我表达自我，朝着“坐下来能写，站起来能说”的目标前进。
- 学期结束暑假开始，就要开始准备 CFA 了，那时候有财务方面的问题正好可以多问问冬子。
- 近期调整的小计划：明天的研讨会好好听，周末把 587 project 做好。

藤师

很久没翻旧时和藤师的通信。

“因有极诚挚的人生态度，可以预见到你必将有一个美好的未来。”——光诚挚是不够的，我有时还是太矫情太拖拉了，关心太多别人的事情，只会忘记自己要做怎样的人！改！

“为此我们不仅要乐观，还应该勇猛”——Yes！

Q：如何成为更好的思考者和提问者？如何更好的表达自己并理解别人？（长期有效的问题）

5.2 Learn the right way of social

screening 筛选，审查
retina 视网膜

Scientific Controls

Positive, Negative, Blind(Double Blind), Randomization

Symposium Ended

结束了，见到了不少牛人，意识到了 *learn the right way to social* 的意义。晚上搜了搜几个川大在美校友的情况，想想自己还真是这五年虚度了两年的时光的感觉。要没有决心追冬冬这事，还真不知道会不会仍在虚度的状态中。所幸这事重新点燃了我对美好事物的追求，这一年开始后尤其不错。

5.3 江左沉酣求名者，岂识浊醪妙理？

不必记录那么多感受，重要的确定了目标就一心一意地做好。从今天开始，我会减少记录感受方面的东西。对于冬冬，目前能做的也不多，就作为小小的理想，存在心底吧，无须反复想起。要鼓励自己的话，目前一句话就够了，“志之所在，气亦随之；气之所在，天地鬼神亦随之”。

The Hacker Attitude——by Eric Raymond（Repost)

1. The world is full of fascinating problems waiting to be solved.
2. No problem should ever have to be solved twice.
3. Boredom and drudgery are evil. (drudgery 单调沉闷的工作）
4. Freedom is good.
5. Attitude is no substitute for competence.

Good ways of Research （Repost）

- *Form the habbit of expressing yourself faithfully with a pen and scratch paper, or with a keyboard, emacs and a computer language.*
- Form a habbit to think deep, make analogy from difference sources and build your own knowledge(ability) net.
- Keep an good appetite of solving problems and some small problems could be fill in a tiny time gap.

- Learn to read papers effectively. Know where to work for detail, where to grasp the main idea, and where to obtain a holistic view.
- Have different ideas/approaches/projects stored in brain at a time but *FOCUS* on only one at a time.
- Communicate with other people and try to know what research they are doing as well.
- Learn from the advisor and take the initiative. Bring good/dumb questions and ideas to him and see how he approaches them.

Things I wanna change for better Research Mode（Repost）

1. Whenner feel unmotivated, drag a pen or computer, solve some problems!!
2. Everyday you are expected to think about your research for a while. Ask questions like: how can I march further in this problem/project? etc.
3. Time and tide wait for no man. Save time as much as you can, spent time on important things, avoid unnecessary things.
4. If ..., then .... Eg. If you have some idea or feel the urge to work something out, you'd better start right now, at least today.
5. Write in English as much as you can. Speak in English as much as you can.

5.4 May the force(4th) be with you~

笨问题：为什么努力？

当你不努力的时候，你会产生一个幻觉：
“老子只要努力一下，分分钟打败你们这些平庸的人。”
而当你开始努力，并且坚持下去的时候，你就会发现：
“老子他妈再不努力，就分分钟要被人打败了。”

而当你努力到了一定程度，你可能会发现：

“老子已经拼到尽头了，但我确实不可能超越他了。但，哪怕多接近一点也是好的。”又或者“老子如果不努力，这个世界上还有什么有趣的事情？”

当然，可能还有别的，不过我没经历过，所以我吹不出来。

久而久之，有些人一直没有开始努力过，所以总会以为自己随时可以打败别人——嗯，打不败的时候会有很多“场外因素”，包括有个好爹。

有些人被打败了，放弃了，所以他们的牛停在某个层次上。

还有那么些人，被打败了，爬起来，再击倒，再爬起来。直到最后，真的爬不动了，一般走到这一步，回过头看一眼的时候，会发现自己已经超越了很多很多人。

也有那么一小撮人，因为某些原因，他们比别人能爬起来更多次，或者被击倒的次数更少，或许他们走的更快。

然后，他们走到了顶峰。

至于是什么支持我一直努力？

我还没吃过寿司之神在银座的寿司，也没吃过 Fat Duck，就连在隔壁香港的龙景轩，都还没去过。

我也还没吃过 biang biang 面，没尝试过新鲜的松茸，甚至还没彻底的体会过全国各地的美味食材，更不用说各种大师手艺。

嗯，关键是，我所认可的每一个朋友们，还没过上自己想要的生活呢。

嗯，听起来比较奇怪，好像没怎么在知乎上说过。不过，我自己的国家，我的中国，还有很多东西可以继续努力呢。我或许，还能出一份力，去改变一些东西。

嗯，我还有更多更多的事没有做完。我还想做好多好多的事情

我既然活的好好的，为什么不一直努力呢？信念这玩意太虚无了，我还是更愿意谈一些实在的东西。

简洁的人生建议（From Quora)

1. 和你最好的朋友结婚：如果让你最放松的的人不是你的配偶，那就让他成为吧！（假设这个人不是你家庭成员）

2. 不要试图成为一个“成熟的人”：永远保持乐趣（不需要药物或酒精）。在泥中摔跤，唱歌，用枕头打架，在任何时候都要保持童心。

3. 不要停止学习：如果你在生活中仅仅沿着惯性前进，你将会输掉一切。让你的智力不断伸展。
4. 不要总想着成为原创：去讲故事或者在画布上涂上颜料或者其他。
5. 纠结于“公平”只会导致停滞不前。
6. 如果你没有失败过，那你就做错了。（犯错是很正常的）
7. 不要试图和盲目狭隘的人讲道理。
8. 不要太强调新闻时讯和“被告知”。
9. 做一些不为金钱利益的事情。
10. 幸福的关键是去建设，而不是去得到。
11. 真正的爱情和美满的婚姻总是不平静的，充满了争执。
12. 时间流逝的比你想象的快得多，而且这种效应只会随着年龄增长而加速。
13. 财富，相对来说是不重要的。
14. 有些事情无法传授，只能体验。
15. 明白你是谁，接受自己，成为自己。
16. 不要等待许可，肯定自己，积极行动。
17. 不要欺骗自己。
18. 尽可能的原谅，积怨收效甚微。
19. 保持谦卑（尤其是对于那些“小人物”）
20. 只有你自己才能控制你想要的快乐。
21. 找到一个良师益友并且成为一个良师益友。
22. 找到你真正喜欢的东西并且投入其中。
23. 如果你必须去洗手间，停止你在做的一切去吧，不要总尝试“在挺一会”

24. 你不必吃完你盘子里的一切。
25. 你不必拿起每一个在响的电话。
26. 牙齿健康很重要。
27. 多采取行动，让人遗憾的事情无所作为远远多于立即行动。

English Version:

1. Marry your best friend: If the person you're the most comfortable with isn't your spouse, make them! (Assuming this person is not your family member)
2. Don't try and be a "grown up": Always have fun (without drugs or alcohol). Mud-wrestle, sing, pillow-fight, any age.
3. Don't stop learning: If you start coasting through life, you're gonna lose. Always stretch your intellect.
4. Don't always try to be original: Just tell the story or paint the canvas or whatever.
5. Focusing on "fairness" will lead to stagnation.
6. If you're not failing, you're doing it wrong. (It's OK to make mistakes.)
7. Don't try and reason with mindless, irrational people.
8. Don't stress yourself out with news and "staying informed" too much.
9. Do something that's not for money.
10. The key to happiness is BUILDING stuff, not GETTING stuff.
11. True love and a good marriage involve a lot of fighting, not serenity.
12. Time passes by a lot faster than you'd think. This effect accelerates with age.
13. Wealth is relatively unimportant.
14. Some things can't be learned; they can only be experienced.

15. Figure out who you are, then ACCEPT that person, and then BE that person.
16. Don't wait for permission. Give yourself the okay.
17. Don't lie to yourself.
18. Forgive as much as possible. Grudges achieve little.
19. Be humble (especially to the "little" people).
20. You and you alone control how happy you allow yourself to be.
21. Find a mentor and BE a mentor.
22. Find what you like and let it kill you.
23. If you have to go to the bathroom, stop what you are doing and go. Don't try to "hold it."
24. You don't have to eat everything that's on your plate.
25. You don't have to pick up a phone that's ringing.
26. Flossing teeth is very important.
27. Always take action on things. People regret inaction more than action.

简洁人生建议逆版

1. 不要和你最好的朋友结婚：如果让你最放松的的人是你的配偶，那么你的知己在什么地方？（假设这个人不是你家庭成员）
2. 成为一个“成熟的人”：不要永远保持像孩子一样（不需要药物或酒精）。在泥中摔跤，唱歌，用枕头打架，这只是你过去的状态。你要成为一个男人，男人的身上永远背负着责任。
3. 不要停止学习：但你在生活中需要沿着惯性前进，这样你会赢得一切。要学会让你的智力休息一下。
4. 要不停的尝试成为最原始最真诚的：在演讲上保持疯狂或者在画布上涂上颜料或者其他。

5. 不要太纠结于“公平”，这只会导致你的停滞不前。
6. 如果你真正的失败过，那意味着你真的做错了。（犯真正的错并不总是正常）
7. 可以试着和盲目狭隘的人讲道理，你不说他们可能真不懂。
8. 要强调新闻时讯和“被告知”，你会发现强势而非凡的人总是使用这招。
9. 做一些为金钱利益的事情，人首先要学会照顾自己。1
10. 不幸福的关键是总去建设，而不是去得到。
11. 真正的爱情和美满的婚姻总是平静的，它很少充满了争执。
12. 时间流逝的比你想象的慢得多，而且这种效应其实只会随着年龄增长而减慢。所以，不要误导自己，把握时间。
13. 财富，是重要的，尤其当你蓦然回首的时候。还有，白富美旁边总是高富帅。
14. 有些事情可以传授，但不能体验。所以。。。勇敢的出发。
15. 明白你不是谁，不接受自己，学习他人。
16. 要等待，要勇敢否定自己，要积极行动。
17. 要学会适当欺骗自己。
18. 孔子说，以德报怨，何以报德？要学会反抗和学会说不。
19. 保持高傲（不论对谁，内心的高傲是强大力量的源泉）
20. 人是社会性动物，别人总是会影响你想要的快乐，所以礼貌待人。
21. 不要当良师益友，但你可以做一个给别人启迪的人。
22. 不要耗费时间寻找你真正喜欢的东西，它就在身边，你只需要发现。
23. 如果你必须去洗手间，你要尝试对自己说“再挺一会儿，只挺一会儿”
24. 吃完你盘子里的一切，这个地球上有很多人盘子里没有东西。

25. 拿起每一个在响的电话。
26. 牙齿健康很重要，但胃和消化系统更重要。
27. 不要多采取行动，让人遗憾的事情远远就是因为没有思考立即行动造成的恶果。

结论：构造这个逆版的朋友说的好："不要相信所谓的简洁的人生建议，这些所谓的简短的人生建议只不过是逻辑学和心理学的游戏"。任何人给你的建议都有其适用的情景。慎重地看待它们，它们中有些可能完全不适用于你或者当时的情况。永远勇敢地做自己的分析判断，寻找自己的正解。

5.5

set off 引起，点燃

TAR & State Space Model

5.6

（程序员）练级建议：

- 不要乱买书，不要乱追新技术新名词，基础的东西经过很长时间积累而且还会在未来至少 10 年通用。
- 回顾一下历史，看看历史上时间线上技术的发展，你才能明白明天会是什么样。
- 一定要动手，例子不管多么简单，建议至少自己手敲一遍看看是否理解了里头的细枝末节。
- 一定要学会思考，思考为什么要这样，而不是那样。还要举一反三地思考。

In Ubuntu, Alt+F10 focus on the menu

5.7 人生贵得适意尔！何能羁宦数千里以要名爵？

知乎文一篇（赞最后一段的态度！）

解决办法说起来很简单，换路，革了自己的命。很多高级公务员、高级职员、企业中层、高管都是这么做的（下海、创业、读书等等）。但是你得有换路的内在实力，找得到机会，还要有抓住机会的能力。最最重要的是要有运气。不具备以上要素的，还是老老实实做现有的工作，偶尔吐吐槽发泄下就好了。毕竟，进入中层和上层社会的概率奇低。学生党更是要好好的工作，好好的搬砖。通过几年工作收获的阅历、达到的视野、积累的人脉岂是读读文章即可学来的？

不想让文章太消极，最后跟我一起温故下千古名篇《滕王阁序》的一段：冯唐易老，李广难封。屈贾谊于长沙，非无圣主；窜梁鸿于海曲，岂乏明时？所赖君子安贫，达人知命。老当益壮，宁移白首之心？穷且益坚，不坠青云之志。酌贪泉而觉爽，处涸辙以犹欢。北海虽赊，扶摇可接；东隅已逝，桑榆非晚。孟尝高洁，空余报国之情；阮籍猖狂，岂效穷途之哭！

是不是今天才发现真正读懂了这篇文章？脑补一下这些知知名人物的一生，想想自己，是不是又快流泪了？哎，我连鸡汤都熬糊了。

Note: 上文提到“落差感”这个概念，从这个角度看，我欣赏陈秋月同学的性格。真心不在乎别人的年薪、买房、结婚啊之类事，现在我真的只有在状态很好的时候才能做到 =。= 后来又读了知乎上张佳玮的自述，说初中时期语文老师的讲的一句“人生贵得适意尔！何能羁宦数千里以要名爵？”对他影响很深，我今天重读也是这样的感觉，这句话也可算是对“落差感”的一个最好回答。

一个爱情故事——张佳玮

2001 年，各种机缘巧合，网上认识一个女孩儿。那时我在无锡，她在重庆。我 18 岁，她 13 岁。

聊。电话聊。邮件聊。手写信聊。

2002 年我上大学，到了上海。写东西，认识了些人。2004 年时出了第一本书，自觉略有薄名。轻狂度日，不着边际。2005 年出了二、三本书，更沾沾自喜，不知天高地厚起来。

2005 年春天，以她为女主角名字写了个长篇，2006 年初出版了，不提。

到 2005 年秋天吧，经历了一件很摧折人的事。简而言之，险些就寻短见，被人送进医院那种。那时在黑暗里自己躲着，确切说。已经崩溃了，对自己的一切方面——真的是一切方面——都产生了巨大的怀疑。那时跟她打电话，说这事，说得她都哭了。她哭了会儿，就静下来，很安稳的说，你别担心，我就要过来啦。

2006 年夏天，她高考完了，违了父母安排（这事至今有后患），自己填了上海某大学的志愿，来了上海。在此之前，我们没见过任何一面，照片都没递过，纯是聊天。9 月初见，之后就在一起了。

当年秋天，两人过得穷愁潦倒——一半是因为我那时没有固定收入，一半是因为她是官家小姐，锦衣玉食习惯了，一开始俩人都没准备，把钱花没了——所以到了要数硬币吃麻辣烫、为了省地铁钱宁愿到处骑车往来的境况。早饭曾经要靠扫沙发下、墙角边的硬币来凑。那年 11 月，她回学校考试前，我们把车票钱算罢，最后剩了些钱，俩人都饿了大半天，买了俩肉夹馍。十一月午后晴暖，两个决定天不怕地不怕过穷日子的人在丁字路口，坐靠着消防栓，边晒太阳，边欢天喜地分吃肉夹馍。我一直觉得，后来吃过的一切，没一样能和当时的肉夹馍相比。那是第一次，看见她金堂玉马家出来的一姑娘，坐消防栓上认真吃肉夹馍，我就觉得，这辈子没什么可说的了。

自此而后，算是决定了一件事。但如果以“爱上" 为标准，事情远还没结束。

自那以来，六年有余。我性子急，她性子倔，两人在一起很容易吵架，但都记性不好，没隔夜仇。兴趣彼此交接，所以到现在，两个人都是有话聊不完。她被我熏陶到喜欢上了历史、游戏、体育、绘画；我被她熏陶到了从只听木管乐和大提琴变成了钢琴协奏曲迷和拉赫玛尼诺夫迷。她爱吃无锡的馄饨和小笼包了，我学会吃辣了。她和我组成 PS3 若干游戏的小搭档了。我被她教会打帝国时代联机了。等等，等等。

2007 年她想来巴黎留学，我就开始攒钱、学法语。因为她不想动家里的资源，而且是纯 DIY，我又是没任何工作经历的自由撰稿人，纯靠稿费

和银行流水做保，也没什么人际关系可投托，所以 2010 年正经开始办出国时，两个人焦头烂额，走了不少弯路。但每次到艰难时，两人都还能彼此安慰和扶助。每次过一关卡，就觉得感情更上一层楼。就像双人打网球一样，打的对手越多，经的险情越吓人，越会在事后彼此安慰：真是运气好，真是天注定啊……

爱情这个事情，最初很大程度是心理暗示，“啊，因为某细节，我爱上了她”，这样想的人很多。进入爱情状态后，容易越想越迷，情人眼里出西施。但我觉得，爱情不是从“爱”或“不爱”来计算，而是“有多爱”。爱情不是是非题，而是程度深浅。最初进入爱情状态的时候很是美好，但之后的经营和维持，漫无止境的。

比如，离开上海来巴黎前几天，两个人搬家时，企图合力搬 12 个箱子的书去小区门口，合计 840 斤。我扛出门一箱子，回来搬第二个，看到她不知怎么，已经把个比她还庞大的箱子推到楼门口了，那时候，我除了赞美她是盖亚，别无他想。比如，来巴黎第一顿饭，我炒得了扬州炒饭，进房间发现她（完全没学过）用土豆、蘑菇不知怎么弄出了份味道正宗的酸辣汤。比如，三月去日本，她知道我的兴趣，提前一天把京都的地图给背熟了，一路拽我走。这种时候，很容易有种感恩之情——不是对她，而是对命运。许多时候，日常生活琐碎的细节都会让人，诸如早上先醒写东西，顺手给她整被子床头放杯水时，看到她熟睡的容颜就会心头一动。诸如两人合力收拾完房间，摆好茶几、毯子、灯、电视、书，然后倚在沙发上开始看球赛，一边吃葡萄时，就会觉得很幸福。

以前写到过，一个人会爱另一个人，通常是因为，他或她代表着某段你拥有过的最好的心情（闲适、欢乐、恋爱中），这些心情可能寄存在某首歌、某份吃食、某本书、某种天气里。每个人都有这么种瘾，翻开来就是一长串苦甜交加的故事。所以每个人的爱情，追溯到最后，总关乎梦或者爱或者一些纯粹时光的美好事物。对我来说，因为已经发生过那么多事，所以这段感情格外饱满；而未来的每一天都在发生一些新的可爱的细节，所以我每天都会觉得，比之前的爱又多了一点点。所以如果以爱上为标准，那我每天都在继续爱上她。就是这样。

5.8 做想做的人，不做好人

冯龙的话和陈秋月的 presentation

晚上在办公室和冯龙聊天的时候，他说到他觉得我人太好了，大家都会找我帮忙，但是也正是这点使我有时候关心自己的事不够多。我觉得他说的很对，这一年也有试着去改变——有些该拒绝的会果断拒绝，继续努力吧。做好人没意思，做自己想做人，追寻自己的正解才有意思。

另外，和冯龙偷听了一下陈秋月做 presentation，觉得她英语表达很不错啊，不像我来了美国没感觉自己进步了多少，对很多人演讲的时候还是容易紧张（熟悉的人前说英语就不会，所以这更多的是心理心态的问题，要改变条件反射还是要多练）。这刺激到我了，我呀，目指他们两个，一定要进化到能更好更多的表达自己的状态！

火热的青春——《灌篮高手》

知乎上的这篇文章写的真好！这部漫画里几乎每个人都有那么几段让我难忘的情节，它深刻地影响了也必然会继续影响我这一辈子。

刷好牙

有段时间没特别注意刷牙的情况了，也就是说刷的比较马虎。今晚刷完一看，好像左下的智齿上有一黑点以为是有点蛀了，仔细再刷了一遍发现没有了，虚惊一场 =。= 以后啊还是多注意，刷牙认真一点，为了减少中老年的痛苦 =。=

Phase 1, 2, 3 and Nilson

Using a example about his current project, Nilson explained to me what is phase 1, phase 2 and phase 3. It's really helpful. He also said if I want to go to industry in the future, his wife have some connections in the finance field.

Late Night Call from Father

Also talked with Zhiheng Xu and hope I could pay him a visit some time this summer in CA.

5.9

I just know that cortisol(皮质素) is closely related to the dealing pressure within human body. The high level of it might cause high pressure, low denfense functionality, bad memory, etc.... I was wondering whether my level sometimes is higher than normal =.=

5.10 Algorithms

Seriously Thinking about start with a good book(like *Algorithm Design Manual*) to learn some data structure and algorithms.

Find the Place to Move in this Mid July

A town house in society hill, which is very close to Busch campus(5 minutes drive). The other two roomates are all freshman of Rutgers and seems very good. Looking forward to the new environment!

5.11 Summer First Half Tentative Plan, 任坤

The Plan

Before I leave for SF on June 17th, I still have five weeks. After I come back around July 5th, I have eight weeks for the summer. Here are the things I plan to do in the first half:

- Research in a wise and persistent way(like summarize some papers you've read and don't limit the vision to the paper that the advisor gave me). Accumulate optimization technique. Finish Ch1 of the

book *Nonlinar Programming*. *A small motivation is the Kalman Filter in the least square section*. If time allows, study the multivariate textbook.

- Roughly finish reading *Art of R Programming*. Focus on the data types, *scoping rule*, *functional programming*, data input and output, string manipulation parts. Try to finish all the problems in the *R Exercise* file. If still have time, learn some C++ and prepare some questions to Zhiheng Xu.
- Five weeks, five chapters in the ESL. Tentatively Ch 4,7,8,9,12. Not only read, but also try to solve as many problems as possible at the back of each chapter.

Some Caveat

- *To submerge in meditation, stay away from weibo, wechat etc. Do not care much about the trifles.*
- Exercise regularly.
- Self-Control ability is important the next leap.
- Prepare well for the mid-summer travel. From 衣食住行. Like for dressing, why not try some jeans?

如何迅速成长成为一名数据分析师？

之前在统计之都认识的任坤同学的回答如下：

“数据分析最重要的可能并不是你熟悉的编程工具、分析软件，或者统计学知识，而是清楚你所使用的统计知识（统计学、计量、时间序列、非参数等等）背后的原理、假设及其局限性，知道各种数据分析工具（例如数据挖掘）能带来什么，不能带来什么，看到一组统计检验的结果你能言说什么，不能言说什么。这一切的背后，需要一套完整的「科学」逻辑框架，让你了解自己手中的工具的本质，你才能从数据中「正确地」发现有效的信息，而不是胡乱地使用一大堆自己都搞不清楚的工具来堆砌分析结果，这样得到分析结果不仅无用，而且有害。

知道了这些后，希望成长为「数据分析师」，就需要着手训练自己的能力和洞察力。既然是「数据分析师」，那就分别从「数据」和「分析」两方面入手。

「数据」当然包含了数据收集、处理、可视化等内容，每个环节对于最后的结果都有关键性的影响。其中涉及的技术性内容只是一部分而已，更重要的是你要理解数据收集（是否存在采样偏差？如何纠正或者改进？）、处理（是否有漏洞或异常情况没有考虑？）背后的逻辑。

例如：如果分析股票数据用于设计交易策略，那么你不仅需要明白数据处理本身的问题，还要清楚金融市场的基本知识。例如，使用股票价格时，到底要用收盘价，还是复权价；复权价的话要用前复权价还是后复权价。这些选择与数据分析没有太大的关系，纯粹决定于你分析的目的是什么。因此你要充分了解这些概念背后的逻辑、动机是什么，才能正确地根据自己的目的作出选择。

数据可视化更多的是一门艺术：如何把信息以最恰当的方式呈现给希望获得这些信息的人。首先，你要充分理解这些信息究竟是什么，有什么特点，你才能较为恰当的选择采用的可视化工具。

另外一部分就是「分析」。当然就是各种分析模型，还是需要了解这些模型背后的逻辑，要放到整个项目的上下文中去看，而不是单纯地在模型中看。

总而言之，「理解」数据以及其中的信息是非常重要的，这决定了你的分析和呈现的方法是否合适，决定了最后的结论是否可靠。

现在可以回答题主的问题了：成长为一个数据分析师，要注意「理解」你的知识，形成一个系统，而不是像机器人一样机械地胡乱套用模型。在这个理念下训练你的编程能力，了解你所分析对象的原理和尽可能多的细节。在这个基础上，才能谈数据分析。”

他的另一个对“金融专业学生如何从基础开始学习计算机？”的回答也值得一看，其中有段关于他从小到大是如何学习编程的简单回顾。

“首先说一下熟练运用计算机的必要性和意义。

在这个时代，如果从事专业性较强的职业，计算机应用的熟练程度、开发水平能很大程度的影响你的思维方式的严谨性和工作生产力。

思维方式方面，直接的体现就是，分析一个项目，学习计算机编程和从事涉及编程的项目开发（例如金融中的程序化交易、量化交易或者其他涉及大量数据处理和分析的研究型项目）能培养较为严谨的系统性思维，

在处理各种项目时都大有裨益。

工作生产力方面，同样是要处理几十个 Excel 表格，无法熟练运用的人可能在很小的问题上就卡一天解决不了，枯燥的手动操作各种复制粘贴、改来改去，上司的一个需求变动就要大动干戈重新做一遍，这样的生产力是极其低下的。目前国内据了解，可能大部分较为专业的金融从业人员都要和 Excel 等软件打交道，但很少能熟练运用，只知道一些最基本的功能，大部分能够通过编程或者聪明的操作方式完成的许多人还是手动完成。如果掌握了一定的编程技能，这些耗费大部分人一天甚至几周时间的琐碎的重复性工作（比如合并 100 个 Excel 表格、为 2000 人打印每份收信人名字都不一样但内容相同的邀请函、按照特定方式处理较大的金融数据，从 400 页的网页上抓取空气质量数据用来设计某种产品），到了懂编程的人手里就只要几分钟或几个小时就能完美的解决，并且遇到需求改变时也能从容淡定的换个参数重新执行即可。甚至在一些没有计算机可用的情况中，编程仍能帮你解决问题，例如让你按照名字字母顺序排序 1000 页文件，如果不懂任何排序算法，那么你可能会称为无头苍蝇；如果你懂得排序算法，那么你就可以手动地来模拟计算机排序算法，把这个任务按照比别人快很多倍的速度手动完成。

计算机在许多工作中都不是必要的，但很有可能帮助你大幅提升生产力。编程教会你如何思考（Programming teaches you how to think）和解决问题（problem solving）。

明确了非计算机专业的学生，尤其是金融专业的学好计算机的必要性和意义之后，我就主要讲讲我作为非计算机专业的金融学生学习计算机的经历和一些想法，希望对题主有帮助。（对故事没兴趣的直接略过，跳到最后）

我是第一批「90 后」，6 岁以前根本没有听说过「计算机」这种东西，或者任何相关的概念，6-7 岁上小学时听说有个同学家有台「电脑」，但不知道那是什么。后来邻居家里有了台电脑，才终于见到了庐山真面目，一个笨重的显示器、主机箱、滑轮鼠标和键盘。于是我经常往隔壁家跑，和隔壁小孩在电脑上随便玩一切能玩的东西，似乎只有画图等等。那时觉得电脑是个很神奇的东西，觉得我们家里也应该有一台电脑，就跟父母说：「宁愿 3 个月不吃饭，也要买一台电脑」。感谢父母的远见卓识，攒了攒钱，在我 9 岁时家里买了一台电脑，上面装着 Windows 98，一个专业人员过来把电脑装好后我第一个做的事就是赶紧把画图打开，随便画了画，

那种兴奋感无以言表。

后来，发现街上有一些卖盗版碟的小店，光盘密密麻麻的摆在两个分类中。一看光盘外包装上的画面就知道这是游戏，另一些画了一些抽象的图形的就是软件。一开始，我对软件没有什么太多概念，基本就是买点游戏盘随便装着玩。现在看起来值得庆幸的是，我没有陷入游戏世界无法自拔，处于一次偶然的想法，觉得游戏玩一玩也就那样，不如看看旁边那些「画着抽象图标的奇怪光盘」里面究竟是什么。不看不要紧，一看我就此进入了计算机和编程的世界。那时随便买了点软件光盘，里面有个叫 VB6，不知道什么意思，回去电脑上装着看看。结果发现，装完打开后什么也没有，自己摸索着创建了一个「工程」，发现外面的大窗口里面居然有个小窗口，旁边的工具箱里面还有一大堆小图标。随便点了个画着按钮的小图标进去，发现居然小的窗口里面出现一个图标上画的按钮，又经过一阵摸索，发现修改 Caption 这个里面的内容按钮上的字就变成了修改后的内容。

双击那个按钮就切换到了另一个试图，里面是一些英文词，自己改了改都说写的错的，错误信息也不知道什么意思。后来发现这个按钮有一个属性叫作 Name，胡乱实验后终于可以让点击按钮后按钮的文字变成另一串字了！这项发现对当时的我来说是个巨大的冲击，从未接触过编程的我发现，自己居然可以用代码来控制这台机器中的窗口和里面显示的东西。后来又拖了个进度条进去，想写一个跟「复制文件」窗口差不多的进度条不断增加的那样一种小程序。东凑西凑，终于把计时器（Timer）、按钮（Button）、进度条（ProgressBar）凑到一起，自己发现了 x = x + 1 这样的方式能让进度条的值增加。用 Timer 0.1 秒加一格进度条，终于实现了一个自动加进度条的小程序！那时还 10 岁的我，异常兴奋。后来就摸索的越来越多，看到微软发布了 C#，觉得名字很酷，看样例代码发现跟以前用的 VB 代码差异很大，后来出于好奇和探索欲转换到了 C# 上，转换的过程很痛苦，本来 VB 弄得就是一知半解，大量从未听说过的概念，莫名其妙的错误。后来经常喜欢用 C# 做一些自己有兴趣的小东西，也逐渐开始看书，Beginner，Professional 等等都看过，凡是在电脑上需要重复性工作的东西我都先考虑能不能用 C# 编个小程序解决。

初中一次学校比赛，每个班要制作自己的一个小网站。我用自己学得网页设计、制作、HTML, CSS 用 http://ASP.NET 做了一个小网站，弄了个小空间和子域名，交给学校评比，获得了一等奖。看到其他班级做的，都还停留在 FrontPage 生成的、不经修饰的难看网页中，我非常得意，并

且在不断地学习新东西。在这个过程中，我上了初中和高中。

后来上大学，选择了厦门大学金融系，那时并没有想读计算机专业，因为觉得计算机就像我的朋友，专门学计算机反而没意思了。上大学时，给学生机构做了个网站，给学院也做了几个网站，另外还接触到了平面设计、视频剪辑、特效等等东西，因为学院里面并没有多少这方面熟练的同学，于是得着一个计算机看起来不错的同学就拼命用，正所谓「能者多劳」，我也觉得是学点新东西的机会，于是就做了不少这方面的东西，简单地学了些 Adobe Photoshop，Illustrator, After Effects, Premiere Pro 等等，后来还下载了不少 http://Lynda.com 上的教学视频看看，受益颇丰。

大三时选专业，我选了金融工程专业。那时开始更多地使用 Excel 做财务数据的简单整理和分析（例如计算 WACC, EVA，做财务预测等等），在完成课程案例项目的过程中逐渐探索和完善自己做的表格，最后形成了美观、易用、用户友好、冗余性低、灵活度高的财务分析表格。后来学习的东西涉及到了计量经济学和一些统计学模型，逐渐开始接触 Matlab 和 R，发现国内许多老师都用 Matlab，海归老师基本都用 R。

在大学中，我主要的生产语言是 C#，写了个基于本地网络的公司财务决策博弈的游戏，原理类似于 MESE，就是服务器端程序开启一场游戏，定义经济体和多种资源数量，客户端连接到服务器端，每个客户端都代表一个公司，每一期内公司向市场买资源（实时的），可以决定借贷、建立工厂、部署生产线，每一期结束前在限定的时间内提交生产产品的数量和价格，以及推广、研发投入的决策，服务器端根据所有客户端提供的决策汇总起来根据一个自己想出来的市场模型决定下一期的劳工成本、借贷利率，并且根据需求和供给决定产品价格、各个公司售出多少，推广费用是否奏效，研发投入带来了技术提升有多少，等等，几乎给每个变量设计了内在的矛盾和博弈，例如大家推广费都很高那就没什么效果，一个公司市场份额变化与每期该公司出的推广费占所有公司推广费之和的比值成正比，等等。每期结束后服务器计算好各种数据，为每个公司形成一个财务报表发布给各个客户端，各个客户端根据这些信息再做下一期的决策。写好之后，找来一些同学测试，或者来玩。看着几个同学在绞尽脑汁设计自己的公司运营策略，我觉得很有成就感。

虽然一直紧跟着 C# 的发展，每出一个新版本就把新特性都学一下，比如匿名方法、Lambda expression，LINQ 等等，不断地更新自己的知识库和编程方法，逐渐形成了一套比较系统的写程序的方法。后来发现，

整个计算机界还有那么多东西值得探索——算法、数据库、数据挖掘等等等等。除了 general-purpose 的编程，我还使用 Mathematica 做符号运算，并且略知 M 的主要编程范式是函数式编程。后来看到一篇文章说 C# 的新特性多半来自于 F#，于是出于好奇开始学 F# 语言，那是第一次正式接触到比较纯的「函数式编程」，觉得这个语言奇怪极了，完全不像是 C/C++/Java 这种体系的语言。从 imperative programming 转换到 functional programming 是痛苦的，两者的基本方法和世界观就不太一样，一开始很不适应。后来逐渐发现函数式编程实在是妙极了，能够用简短的语言快速地解决我原来很麻烦才能解决的问题。在这些过程中，又做了些网站，但每次做都有新的进步，例如引入了 MVC 架构，尝试了 AJAX，采用了 HTML5，jQuery 等等，并在这一过程中接触了更多的工具、编程语言，并且逐渐开始使用数据库来存储和调用数据，玩过 SQL Server, MySQL, SQLite 等等，甚至后来来自己部署了一个 Hadoop（虽然没起到作用）。

===

到了现在在读的金融研究生阶段，由于更多地涉及建模和数量化的东西，更多地学习统计编程。我们时间序列分析的课程和非参数计量经济学的课程都要求用 R 来完成，写东西最好用 LaTeX。虽然以前用 R 写过一点东西，不过是这些课程和一些研究项目逼得我逐渐开始使用 R 和学习 R，并且更多地接触了背后的开源社区，逐渐发现世界上有这么一大批人，无偿的贡献自己的力量，解决各种各样的问题。开源社区的力量是无穷的。后来我也在 GitHub 上注册帐号，并且把自己使用一些开源项目时遇到的问题和 bug 用英文写成 issue 供大家讨论。后来在 RProvider 项目上（一个允许在 F# 中使用 R 函数的 Type provider）发起的一个 issue，项目所有者建议我改一下代码发送一个 pull request，那时我还不了解 Git 版本控制系统的工作方式，赶紧导出找资料看什么是 Git，什么是 Fork, Branch, Merge, Pull, Push, Pull request 等等，发现「原来这些开源项目都是这样组织起来的!」，终于明白了上百人如何同时投入一个项目的开发而不造成大量的重复劳动和冲突，明白了各种我们用的软件的项目周期是如何推进的。这些对我而言就是很有意思的。

后来，在做一些金融量化交易策略时就把这些全部都用上了。多人同时开发时用的 Git 版本控制和 issue tracking 等等大幅提高了整个团队的生产力，在 R 中写了很多代码，有的项目重构了很多次，慢慢积累了很多有

用的扩展包，并且大量的看相关的文章和博客，还去 stackoverflow 提问等等，看到一个扩展包或者工具有问题或者需要改进的地方就去 GitHub 提出个 issue 建议改进并且参与讨论，或者直接自己 fork 发 pull request。虽然这些东西更偏向计算机的东西，和我本专业并没有太大关系，但是这些东西都提供了生产力和效率。

===

故事讲了这么多，该做个总结了。

从我自己的经历来看，非计算机专业的学生想学好计算机，要么就要有很强的兴趣、探索欲，要么就要有非学好不可的压力（比如作业无法完成），两者都有是最好的，会快速提升你的技能。另外，需要有「专业化」（*professionalism*）的精神，就是想把事情做专业一些，做完美一些，这样你的一个项目从一开始的简陋不断完善成美观、易用、交互体验良好、背后的设计思维一致，才有动力实现，在不断追求专业化的过程中你的水平才能进步。

对于小孩而言，他们有大量的时间探索、试错、思考、查阅，这样积累起得经验是最原始的，最直觉的，我就基本属于这种情况，就是探索——学习——探索——学习的过程。对于一个成年人，可能就没有那么多精力探索自己领域以外的东西了，因此建立起直觉可能会非常耗时。想学好的唯一的秘诀就是大量使用计算机和编程，所有东西都想想能不能用计算机来做，不会的东西多试验，或者多查资料，做的东西都不是做完就完了，多想想能不能改进——技术上、效率上、结构上等等，凡是能改进的全都改进，不断地改进，即使是一个 *Excel* 表格。在这个过程中，多激励自己，让自己有成就感。只看书、只上课学而不去大量的实践做各种小项目，纯属浪费时间。

小项目从哪里来？到处都是。如果一开使做不了我本科做的那种企业经营模拟对战，可以从课本上简单的东西来。自己实现一套二叉树欧式期权定价、美式期权定价、蒙特卡罗方法欧式期权定价，美式期权定价，不断地完善并且一般化、参数化，直到能轻松地做亚式期权、百慕大期权、敲入敲出期权、障碍期权定价。写的过程自己分析、自己设计，做出来后自己调试、自己测试，千万别看别人怎么做的。完成之后，看看别人怎么做的，能吸取到哪些教训，学了哪些新东西。你的下一个小项目，水平就更高了一点。不断地积累下去，是一切学习的秘诀。

那么，对于一个没有多少计算机基础的金融本科生，如何开始呢？我

提供一些建议：

1. 不要眼高手低，光看，不实践。把基本的办公软件学扎实，作为硕士生应该要掌握基本的 LaTeX，不喜欢这套体系可以用 LyX，总之文档排版系统是基本应该会用的。

2. 开始学编程。R，Python 等等都可以，把网上的简单教程看个遍，代码敲个遍，别偷懒，运行之前多预测会产生什么结果，多调整一下自己写的东西看看会产生什么不同，多思考为什么会这样。之后实现一些基本的算法，也可以实现你学的统计和计量模型，比如线性回归之类的，学一个实现一个，金融计算模型（二叉树、蒙特卡罗期权定价等等都实现一下），总之要大量写代码，遇到问题自己尝试解决，解决不了去社区看看。

3. 学习一下数据库，推荐 SQLite，免费轻量级，简单易用不用部署，在 R 和 Python 里面很方便调用。对你以后做数据密集型的研究和应用有好处，我现在做的股票数据、高频市场数据都放在 SQLite 数据库里面，学习一下 SQL 语言。

4. 如果涉及编程密集型的团队研究或者其他项目（例如设计和优化新的金融套利策略和相应的回测程序，或者开发一种最新论文中的非参数统计工具并检验），凡是有多人协作的项目，都建议学一下 Git 版本控制系统，开源用 GitHub，闭源用 Bitbucket 托管你的项目，写文档、issue tracking，wiki，都让你的项目井井有条，用软件工程的方式管理各种项目，是现代项目管理基本都要涉及的。”

他还对学习金融有如下中肯建议：

1. 理解金融市场运作的基本规则和事实以及建立基本的金融直觉。

2. 比较全面的掌握统计学的方法论、理论和应用，包括统计推断和时间序列分析。

3. 熟练地使用一种以上的编程语言和工具，能够做多种规模的数据处理、挖掘、统计分析、可视化。

4. 对文献的搜集和吸收能力。

记一首诠释何为“侠”的歌

“轻裘长剑，烈马狂歌，忠肝义胆，壮山河；
好一个风云来去江湖客，敢与帝王平起平坐。
柔情铁骨，千金一诺，生前身后起烟波；
好一个富贵如云奈我何，剑光闪处如泣如歌。
一腔血，流不尽英雄本色；俩只脚，踏破了大漠长河；
三声叹：叹，叹，叹，
———只为国家故园。
四方人，传诵着浩气长歌！”

听完不禁问自己，对于我们这个时代的男儿来说，什么样的男儿的气质和穿着算得上是“轻裘长剑，烈马狂歌”？

Distribution of Sample Median

For example, in the normal iid case.

5.12 士之怒

silhouette 轮廓，剪影

冰与火之歌 S4E6 里的 Bravos 和 Tyrion

这集很精彩，前有 Davos 为 Stannis 的慷慨陈词，后有 Tyrion 悲愤至极的爆发。豆瓣看到有人取用《唐雎不辱使命》里的这段作为观后短评：

“夫专诸之刺王僚也，彗星袭月；聂政之刺韩傀也，白虹贯日；要离之刺庆忌也，仓鹰击于殿上。此三子者，皆布衣之士也，怀怒未发，休祲降于天，与臣而将四矣。若士必怒，伏尸二人，流血五步，天下缟素，今日是也。”

真的是评的很有意思。原文更完整的 context 如下：

秦王怫然怒，谓唐雎曰：“公亦尝闻天子之怒乎？”唐雎对曰：“臣未尝闻也。”秦王曰：“天子之怒，伏尸百万，流血千里。”唐雎曰：“大王尝闻布衣之怒乎？”秦王曰：“布衣之怒，亦免冠徒跣，以头抢地耳。”唐雎曰：“此庸夫之怒也，非士之怒也。夫专诸之刺王僚也，彗星袭月；聂政之刺韩傀也，白虹贯日；要离之刺庆忌也，仓鹰击于殿上。此三子者，皆

布衣之士也，怀怒未发，休祲降于天，与臣而将四矣。若士必怒，伏尸二人，流血五步，天下缟素，今日是也。”挺剑而起。秦王色挠，长跪而谢之曰：“先生坐！何至于此！寡人谕矣：夫韩、魏灭亡，而安陵以五十里之地存者，徒以有先生也。”

虽然其事未必是真实历史，不过读来还是让人觉其凛凛生气。

5.15 穿着搭配

周六预定 Woodbury 买东西去，也顺便做点穿着搭配的笔记。

衣着笔记：

衬衫搭配心得：

1. 褐色西服，可配暗褐、灰色领带，穿白、灰、银色或明亮的褐色衬衫。
2. 灰西服，可配灰、绿、黄或砖色领带，穿以白色为主的淡色衬衫。
3. 蓝色西服，可以配暗蓝、灰、胭脂、黄色领带，穿粉红、乳黄、银灰或明亮蓝色的衬衫。
4. 黑色西服，穿以白色为主的浅色衬衫，配灰、蓝、绿等与衬衫色彩协调的领带。

衬衫领带搭配技巧：西装衬衫领带搭配 图案搭配

从图案搭配的角度讲，图案中的任何一种颜色能与衬衣或西装颜色一样的话，效果便会锦上添花。不过, 领带上的图案应该比衬衫上的更显眼。

1. 格子衬衣宜配斜纹领带。
2. 直纹衬衣宜配方格图案领带。
3. 暗格衬衣宜配花纹领带。
4. 同类型图案的衬衣、西装和领带不宜相配。
5. 印花或花型图案的领带最好配素色的衬衣。

衬衫搭配，领带搭配技巧：西装衬衫领带搭配 颜色搭配

1. 黑色西装，穿以白色为主的衬衫和浅色衬衫，配搭灰、蓝、绿等与衬衫色彩协调的领带。

2. 暗蓝色西装，可以配蓝、胭脂红和橙黄色领带，穿白色和亮蓝色的衬衫。

3. 灰西装，可配灰、绿、黄和砖色领带，穿白色为主的浅色衬衫。

4. 咖啡色西装，可以配暗褐、灰、绿和黄色领带，穿白、灰、银色和明亮的褐色衬衫。

5. 蓝色西装，可以配暗蓝、灰、黄和砖色领带，穿粉红、乳黄、银灰和亮蓝色的衬衫。

买西装时注意“两看”

1. 看做工。做工主要检查线迹、手工和夹面，查看西装口袋两条开线条是否一致，上袖处有无褶皱。如果是条纹或格子西装，则要看这两处的条格有没有对上。

2. 看质感与颜色。通常耐穿的西装都是天然的材质，比如纯半毛。最稳重的颜色多为藏青或是灰黑色，其他如咖啡色、深棕色都不太适合正式场合穿着。

选衬衫注意 6看”

看品牌、看质地、看领口、看袖口、看做工、看样式。

西服穿着要领：

- 西服袖口商标应拆除。
- 买回的西服应定期干洗，保持外形的平整干净。
- 就座之后，西服上衣的钮扣则要解开，以防其走样。
- 穿双排扣的西服，一般应将纽扣全部扣上。穿单排扣的西服，最底下的扣子永远不要扣上。如果是两粒扣，则只需扣上面那粒纽扣；如果是三粒扣，则扣中间那粒。非正式场合，可以不扣钮扣。
- 西服的衣袋不宜放太多物品，最好将物品放在西服左右两侧的内袋中。

- 马甲的扣子应全部扣上，其口袋多具装饰功能，除可以放置怀表外，不宜再放别的东西。
- 西裤的长度以坐下时能露出袜子为宜，其前面的两条裤线需烫挺烫直。
- 西裤两侧的口袋只能放纸巾、钥匙包，后侧的两只口袋，应不放任何物品。

衬衫穿着要领:

- 衬衫必须保持清洁，领口和袖口必须一尘不染。
- 领口和袖口必须扣上，袖口不能上卷；休闲场合可以去掉领带，打开领口纽扣，以示轻松。
- 领带是西服的灵魂，凡正式场合中，穿西服必定要系领带。
- 领带尾端的位置不应该低于皮带。
- 领带细的那端不应该比宽的那头长。
- 领带永远都不应该塞在裤子里。
- 永远都把细的那一头塞进领带背后的那一小片横的布里面。
- 衬衫和领带的颜色不能太接近，根据衬衫颜色再选配领带。
- 领带颜色永远比衬衫深。 • 领带的保存：挂起来，或卷起来。

在衬衫的领子的选择上：

1. 标准领（regular coIlar），由于领型普通，所以最容易搭配，无论什么领带都可以尝试与之搭配，而且还不必挑剔领带的图案。
2. 宽角领（wide 一 spread collar，也叫温莎领）这种领形适合系温莎结形的领带，而且一般与英国式的西服相搭配。是当年温莎公爵带头兴起的。但近年敞角领的衬衫流行与打得稍小的半温莎领结相配，这种搭配能于复古中反映近年来精致的现代思潮。
3. 带扣尖领（button 一 down collar），这种领形的领尖夹角一般等于或小于标准领形，因此适合系单温莎结或普通结。

4. 有襻领（tad collar），这种领形因夹角较小，所以一般系普通结。
5. 针孔领（pinholecolIar），适合系普通结。
6. 小方领（shrot point coIlar），一般系小温莎结或普通结。
7. 冀形领（wing collar），一般系蝴蝶结而不系普通领带。
8. 立领（standup collar），通常不系领带。

其他

- 记住三种颜色：白色、黑色、米色这三种颜色被称为“百搭色”。
- 新买的衬衫，如果你在脖子与领子之间插进两个手指，说明这件衬衫洗过后仍然会很合适。
- 衬衫的合身与否将直接影响到整体着装的贴合程度，尤其对于一些面料轻盈的西装来说，如果穿在里面的衬衫过于宽松，也很难保证上装的服帖整洁。衬衫肩线应该落于肩膀骨外侧约 1-2 厘米处，过宽会产生慵懒没有活力的视觉感受，过窄则显得人瘦小、不够庄重。因为正装衬衫主要以精纺纯棉、纯毛制品为主要面料，在水洗后可能会有不同程度的缩水，因此在选购衬衫时要有意识地将肩线宽度略微放宽 0.5 厘米。
- 颜色方面白色及蓝色衬衫一向是男士们挑选衬衫的主流色彩。蓝色给人冷静沉稳、红色给人热情大方的感觉。白底条纹或格子衬衫显得轻松一些，这种衬衫商务感较强；深色衬衫和花纹衬衫一般不适合正规活动和高级商务活动，只有在较松、休闲的气氛中或参加联欢会时才选用此类衬衫；八字领衬衫较大气，尖领的则较秀气，但这要看当时流行何种领型；短袖衬衫适合中下阶层人士穿用，中上层人士则应选择长袖衬衫。
- 衣领带钮扣的衬衫不能搭配双排扣西服。（Why？）
- 如果拥有一件大衣，那么应该是灰色的，第二件大衣应该是黑色的，第三件是咖啡色的，第四件是藏青色的。

- 穿西服一般要系领带。领带的色彩应与西服、衬衫和谐相配。按照西服——衬衫——领带这三者的顺序是：深—浅—深，浅—中—浅或深—中—浅的配色方法。
- 穿休闲衬衫时不要忘记配衬卡其布（一般为浅褐色的布料，常用于军服）、细帆布等同样的休闲风格的裤子以及休闲鞋，颜色的选择你就可以随心所欲。
- 洞察衣料 衬衫同为棉质衣料。所用棉线的粗细及织法的不同，制成的衬衫衣料大不一样，触觉和视觉也各异。
- 小 tips：在衬衫颜色的选择上不妨多下点工夫，观察学习法是不错的穿衣方式。除此之外如领带的搭配，袖口真的是样好东西，西裤的颜色也都需要一并作整体考量，找到正确的搭配之后，保证让你随手打开衣柜，怎么穿都好。
- 知乎有个《男生如何找准自己的穿衣风格，提升衣着品味?》的问答不错
- 如果你有 3 件上衣，3 条裤子，3 双鞋，你可以穿出 27 种不同的组合，一个月天天都有新鲜感，天呐！每两天换一种搭配是适宜的频率，这样你可以穿整整 2 个月！然后从头来过！

牛仔裤选择：

Note 1:

笔者身高 172cm，体重达到了 72kg。标准的上半身微胖下半身不长的杯具型人物。所以在购买牛仔裤的时候几乎没有合适的，除了在 G-STAR 买过 w32.L30 的比较适合之外，别的裤子没有列外的都是偏长。因为下半身不长，所以购买时我劲量买锥形裤，这种版型基本上能避免我大腿粗而且腿不长的问题，而且比较舒服。购买时，如果长度有问题，那就最好倾向于买原色裤。因为这样的裤子修改裤脚时比较方便，而且原色裤基本上在偏正式的场合也可以穿。腰围是关键，我个人的观点是穿着不要系腰带的大前提下可以放一到两根手指的裕度。裤长最好是穿鞋坐下的时候不露出袜子。此外就是穿着蹲下试试看，要舒服基本上就可以了。

5.16 批判的价值

《批判的价值》——王垠

总是有人告诉我，我不应该批评一些技术，特别是不应该在一些公认的“大牛”或者流行的技术头上动土。要做出自己的“成果”，这样才可以得到大家的“尊重”。首先，你可能没有发现，被我批判得最厉害的技术和人，其实也是最牛气哄哄，最以自己的地位压制其他人的（比如 Unix 和 Go 语言）。我从来没有尖锐地批评过一个朋友经过自己的努力，做出来的难看的小板凳。不管他如何幼稚，得到的总是我的鼓励和帮助，除非有一天他也变得牛气哄哄。而且，我觉得我的成果已经够多了，都是很多人想都想不到的东西，够写好几篇博士论文了。多得在我脑子里已经堆不下了，而且还在不断地产生。我从来不公开自己最新的想法，所以你们在这里看到的“惊人”的想法，全都是经过多年的思考和经验才得到的，全都是我司空见惯的。我的顶级水准的代码很多都放在网上，可是有谁知道它们是用来做什么的，它们的价值？所以我觉得虽然这种某些人希望我采用的“默默无闻”的方式，作为个人的策略，不失为一个混进“上流社会”好办法，然而这对于 IT 业和学术界的大环境，却并不能达到我希望的效果。其实恰恰相反，这种做法只会让我自己也成为这大环境的一部分，并且为维护它的各种谎言而耗费我的生命。

我难道还不够“专心学术”吗？在 Cornell 和 Indiana 的日子里，有哪一天我不是沉迷于我梦想中的学术呢？在那段时间里，我从老师那里学会，外加自己的摸索，掌握了很多人难以想象的本质性的知识，纯净而美好的想法。我把学术都已经做到头了，做到超过世界的“前沿”很多年以后了。可是当我进入公司实习，进入业界工作，我看到了差距。这不是很多人所谓的“理想与现实的差距”，因为我所学到的并不是空洞而难以实现的“理想”，而是切实可行的，并且被证明为非常有效的做法。你能理解那种曾经生活在“未来”，现在却受累于各种历史包袱的感觉吗？我拥有“未来”的技术，为了它们我付出了巨额的生命和汗水。然而这种“未来”却是相对的，这些未来技术本应已经被广泛采用，却由于业界对各种低劣技术的商业炒作和宗教崇拜，显得离我们很远。或者更可怕一点，由于对低劣技术的商业炒作和宗教崇拜，被人认为是过时的东西。

像很多有同样经历的人一样，我为了这些美好的理论和想法能被世人接受做出了努力。像他们一样，我不愿意得罪人，我做过好好先生，甚至

说过奉承话。我以为只要我的想法比现有的好，就会逐渐被人接受，我觉得没有必要去批评不好的想法。有好几次在对一些事情进行尖锐的批评之后，我心理都在嘀咕，因为我深深地知道，我得罪了人，我烧毁了自己通向“成功”的桥梁，甚至也许会活不下去。有多少次把之前写过的文章藏起来，其实是为了让自己还可以继续在这个混世里面活下去。可是我发现这样下去，自己就活得越来越像一条狗，就像很多其他人一样。或者叫做沉默的羔羊。

在这个技术的数字游戏里，只做加法而不做减法，是永远得不到正确的答案的，因为你的大作，将建立在摇摇欲坠的基础之上。你向混世妥协，然而混世并不会因此对你好一些。IT 公司里总是有人认为自己什么都懂，他们追随某些“大牛”的思想，因为那样他们就可以显得很牛，可以被重用。如果你有真知灼见，而这种人做了你的上司，那几乎是必然的矛盾，因为你将眼睁睁的看着自己憧憬中的项目一步步的走上一条不归的绝路。你的生命开始在“妥协”和“沟通”之间徘徊。你不好意思对上司的作法直接提出异议，然而当你想方设法进行“沟通”的时候，你发现虽然自己具有深入的洞察力，你的想法却总是被人们头脑中的各位“大牛”所压倒。你的洞察力和智力，比起那些水平其实不如你的大牛们的名气来说，简直一文不值。

这些大牛哪里来的如此大的威力？因为他们实际做出过什么“成功”的技术吗？在经过调查之后你惊讶的发现，根据对“成功”的定义，也许是的，也许不是的。他们也许做出过一些很多人在用的东西，然而很多人用的，却不一定是好东西，有可能是一个人人叫骂的东西。如果真有什么看得见摸得着的技术还好，可是很多所谓“牛人”其实不过是写过一本书，或者领导过一个“成功”的公司，或者只是有一个著名的 blog，上面写着一些和蔼可亲的入门级读物（我貌似也有嫌疑，然而我的 blog 非常有深度，而且我还有其他东西:P）。有些公司也许算是一个成功的公司，然而它有先进的技术吗？如果你把商业的成功和技术的成功混为一谈，那再来谈所谓“技术进步”，那还有什么意义呢？于是你就看到了很多人的“拜金主义”，一直拜到了技术那里：有钱人的技术都是好技术！

可惜的是，有钱人的钱，不是你的钱；有钱人的技术，不但不能让你也成功，却能让你坠入万丈深渊。除了羡慕他们以外，你还能从他们那里得到什么呢？工作？绿卡？那些就是拴住你的无形锁链，让你为了别人的野心而廉价的出卖自己的青春。不得不说，出于“战略”考虑，你有时候

确实可以妥协。然而不幸的是，自己妥协就算了，却总有人为了爬到更高处，把这种妥协作为教条，布道给更多的人。Blog 就是他们最重要的工具，写几个 post，抬出几个公司里的大牛，说自己跟他们关系多么接近，然后就把他们的“语录”下放给广大码农。最开头得到这信息的人，知道这只是临时的妥协，是这人往上爬的权宜之计。可是当这些东西越传越远，而没有人针锋相对，它就成为了真正的教条。所以“教主”们也许并不是罪魁祸首，而喜欢传教的人才是真正的罪魁祸首。不过也许，他们全都是罪魁祸首。这种罪，深深地植入到了每一个人的脑子里。

这种现象发展到最后，就使得 IT 公司成为了最苦逼的工作场所。在你耳边，总是环绕着人们对各位大牛的崇拜之情。吃饭时谈论的，总是那些人的逸闻趣事。你的梦想，就是有一天能成为他们那样的人，或者能跟他们在同一个办公室工作。你失去了自己，你不再理解，世界上其实还有其它的活法。你永远不会明白，你所谓的“知识”和“技术”，是多么的肤浅。

你低估了你自己的头脑的价值，这就是我想告诉你的。我批判你们心中的偶像，只不过是为了让我的身边存在几个拥有自己头脑的人，而不是每一次一起吃饭都无限卑微，让我慢慢地也感觉自己很卑微。你对大牛们的崇敬，你的“好学”精神，并不能得到我的好感。我只尊敬真正的勇士，那些尊敬他们自己的人。所以我写了这些东西，为了让你理解到你自己的价值，让你不再受蒙蔽，让你不再成为权威者们所谓的“知识”的奴隶，所谓的“技术”的学习者和执行者。

我希望你成为一个创造者。每个人刚生下来的时候都是一个创造者。是所谓的“教育”，让你们失去了自己的思想。而真正的教育，其实是当你忘记在学校学会的知识之后，剩下来的东西。在这种意义上，任何国家的教育都是失败的，而不只是中国。所有的知识都是肤浅的，唯有得到它们的途径，才是真正的精髓。

Kashmir 开司米

Ubuntu Unity 抽风

晚上回来 unity 抽风了，lauch bar 不见 +top menu 不见 +super 键失灵 +etc，一开始尝试 C+M+F1 调出 terminal，可死活忘了 reboot 这个命令（用 shutdown 需要 root 权限。。）。还好 gnome 模型没有问题，在

gnome 下搜索到有一个 unity-reset 的插件或许可以搞定这个问题，装后一输入命令，果然 everyting back to normal，哦耶！

5.17 Woodbury 归来

通过 Herschel 网站学习包类词汇

tote 手提，手提袋
pouch（wallet）小袋，皮夹子
hip pack 腰包
duffle (圆筒）行李袋
backpack 背包

5.18 Commencement Day

读阮一峰《计算机是如何启动的?》一文，对启动为什么使用 boot 一词，启动顺序（Boot Sequence），主引导记录（MBR）及其结构（如分区表，主引导记录签名），卷引导记录（Volume Boot Record），扩展分区和逻辑分区，启动管理器 (boot loader) 有了更清楚的认识。以下是一些笔记：

- boot 是 bootstrap（鞋带）的缩写，它来自一句谚语："pull oneself up by one's bootstraps"。字面意思是" 拽着鞋带把自己拉起来"，这当然是不可能的事情。最早的时候，工程师们用它来比喻，计算机启动是一个很矛盾的过程：必须先运行程序，然后计算机才能启动，但是计算机不启动就无法运行程序！早期真的是这样，必须想尽各种办法，把一小段程序装进内存，然后计算机才能正常运行。所以，工程师们把这个过程叫做" 拉鞋带"，久而久之就简称为 boot 了。

- BIOS 按照" 启动顺序"，把控制权转交给排在第一位的储存设备。这时，计算机读取该设备的第一个扇区，也就是读取最前面的 512 个字节。如果这 512 个字节的最后两个字节是 0x55 和 0xAA，表明这个设备可以用于启动；如果不是，表明设备不能用于启动，控制权于是被转交给" 启动顺序" 中的下一个设备。这最前面的 512 个字节，就叫做" 主引导记录"（Master boot record，缩写为 MBR）。

- " 主引导记录" 只有 512 个字节，放不了太多东西。它的主要作用是，告诉计算机到硬盘的哪一个位置去找操作系统。主引导记录由三个部分组成：（1）第 1-446 字节：调用操作系统的机器码。（2）第 447-510 字节：分区表（Partition table）。 （3）第 511-512 字节：主引导记录签名（0x55 和 0xAA）。其中，第二部分" 分区表" 的作用，是将硬盘分成若干个区。

- 控制权转交给操作系统后，操作系统的内核首先被载入内存。以 Linux 系统为例，先载入/boot 目录下面的 kernel。内核加载成功后，第一个运行的程序是/sbin/init。它根据配置文件（Debian 系统是/etc/initab）产生 init 进程。这是 Linux 启动后的第一个进程，pid 进程编号为 1，其他进程都是它的后代。然后，init 线程加载系统的各个模块，比如窗口程序和网络程序，直至执行/bin/login 程序，跳出登录界面，等待用户输入用户名和密码。至此，全部启动过程完成。

5.21 Ownership and Permission in Linux, Root Privilege

5.22 ‘‘数学好在学哪一门语言时更有优势”?

知乎一答

我的答案是 Lisp 和 Haskell，当然还包括其他 functional 语言。我这里保留的 functional 的英文，没有说是“函数式语言”，不是装逼，是因为这个词，在数学里有另外一个广为人知名字：“泛函”。（不过下面我直接说函数式语言了跟顺口一点。。。。）

之所以说数学好学函数式语言更有优势。是因为 1）相较其他语言，函数式语言和数学“更像”，2）函数式语言更依赖数学。

说函数式语言和数学更像，最基本的就是“函数”。要注意，虽然都叫函数，但是其实二者是完全不一样的。数学中的函数是一个从定义域到值域的映射，而计算机语言中的“函数”至少包括两个意思：procedure 和 function（Pascal 中就是分开的两个概念，呃。。。说出 Pascal 是不是就等于暴露了 OI 历史。。。），function 还可以粗略的等于从参数的空间到返回值

的空间的映射关系，但 procedure 完全就是程序跳转到另一个位置去执行，和“函数”这个东西基本没有任何关系。而在学 C 的时候，我们叫着函数，却基本都把它理解成 procedure。既是是有输入有输出的函数，还有和数学中函数的一个最大的不同：确定性，或者说，副作用。

一般来说，数学中的函数是一个从定义域到值域的确定的映射关系。换言之，确定的输入总产生确定的输出。但是过程式语言中这一点完全无法保证，例如允许对函数外的变量进行修改。而纯函数式语言中通常要求函数保证无副作用（比如通过强制变量不可变），这样保证了语言中函数和数学中的函数的一致性。

再举个可能不太恰当的例子：在纯函数语言中，通常变量不可变。于是，困惑了无数小白的诸如“a=a+1”这样违背数学常识的语句也不太可能出现了。

因为语言中的函数和数学中的函数统一了，递归这个概念也好理解多了。因为递归不再是“一段代码调用本身”这么拗口的概念，就是数学中的递归定义或者递推公式而已。

再者，配合函数式语言中常见的模式匹配，函数式语言仅仅是“看上去”，也更像数学。比如 Erlang 中实现斐波那契数列：

fib(0)->1;

fib(1)->1;

fib(N)->fib(N-1)+fib(N-2)

基本就是数学定义。

然后，泛函。数学上，泛函通常是指定义域为函数的函数，而函数式语言中，函数是 first-class 的，即和普通变量一样，函数可以操作函数。看，又和数学统一了。使用函数式语言，完全就是从数学的角度看问题了。这种操作在 C 和 Java 中不能直接实现。当然，C 可以使用指向函数的指针，Java 通常借助一个接口。但这样，都是从机器的角度思考，或者从面向对象的角度思考，而不是从数学的角度思考。

另外，学习函数式语言也确实需要数学基础。

过程式语言，包面向对象语言等等，都可以用图灵机来理解，但是函数式编程则不然，理解好函数式编程你需要理解 lambda 演算，然后不可避免需要形式系统的推理和证明，这些还真的需要数学基础。而 Haskell 更进一步，即使不碰 monad，至少需要集合论的知识甚至部分泛函（向量空间）的知识，而如果涉及到 monad，没有抽象代数和范畴论的概念基本很

难真正理解。（是的我要承认我还没有真正理解）

综上，如果数学好，“适合”学函数式语言，而我所举的，是其中两个典型。另外有人提到 clojure，我认为 clojure 可以视为 Lisp 的一种方言。其他推荐学习的函数式语言包括 Erlang 和 OCaml。

然后再来回答一下原来的问题，如果以后会搞计算机，那这两门语言都跳不过去，但是两者选其一，建议先学 Java，因为 C++ 太复杂（抱歉我是 C++ 黑，不争论）。但是，更建议 C 和 Lisp。

学 C，推荐的教程只有一本：《C 程序设计语言》，作者 K&R，不要去看任何其他的书。学 Lisp，推荐的教程也不用多说：《计算机程序的构造和解释》。

然后，虽然我大部分时间会使用 Python，但我坚决不同意一个专业的程序员从 Python 入门。虽然 Python 上手很快，但是无益于理解计算机的运作方式，也无益于理解程序的数学本质。当然，Python 是个好语言，（Perl？哈哈哈哈，逃 ~）。

说这么多，不是表达 Lisp 和 Haskell 比 Java 和 C++ 高端，或者 Java 和 C++ 是 inferior languages。C/Java/C++ 应用比函数式语言广泛毋庸置疑，而函数式语言学习的更大好处其实是学习其中的思想不是为了工程应用（大多数情况下，Lisp 在人工智能领域应用很广）。数学和机器都是计算机的一部分，对机器感兴趣，就去学 C，学体系，对数学感兴趣，就去学 Lisp，学计算理论，最后会发现，殊途同归。只不过，选择一样的时候，记得对另一样保持好奇心。

计算机很好玩的!!!!! 我们不要再（让别人乱）黑了好不好!!!!!

最后，也别全信《黑客与画家》，那就是一本 Lisp 软文集。。

5.24 超赞的一餐

今天和冯龙一起帮易兰搬了家，一室一厅的环境还不错，1070 的房租也很划算，不过再走远点就可能不安全。晚上在陈秋月家吃饭，莲藕龙骨汤、青椒土豆丝、宫爆鸡丁（辣味正好）、虾、卤牛肉，真是好久没吃这么赞的一席菜了 =。= 另外值得记录的两点：

- 第一次玩 Wii 上的打兔子游戏，后来的 Mario 过山车也挺好玩的。
- 晚上 UNO 赢了四五把，前所未有的事情 =。=

5.25 如何写好技术文档？

serendipity 机缘巧合；缘分

东杰师近文

一个坐在书房里沉湎于往事的人，如何向世人解释自己工作的意义？这种压迫感一直萦绕在史家心头，即使是声望卓著的马克·布洛赫，也不得不面对其幼子的天真盘查：“爸爸，告诉我，历史有什么用？”对此，英国马克思主义史学家埃里克·霍布斯鲍姆用一个书名作了回答：《史学家：历史神话的终结者》。按照这个看法，历史研究的用途是，戳穿无论是什么人出于什么目的、无论有意还是无意，编织出来的谎言，告诉世人真相。

历史学家是否以及在多大程度上能够复原真相，是另一问题。我这里关心的是，假如霍氏所言有理（它仍是今日大多数专业史家的共同信条），史家为何要“终结神话”？它会使我们的生活更美好吗？那么，生活美好又是什么意思？使人感到愉悦？历史研究有时能够使人愉悦，更多时候却正好相反：真相常会令人痛苦。日本当局删改教科书以掩盖侵略历史，显然是要逃避良心谴责，进而把自己打扮成唯一的无辜受害者，赢得道义的优势。在此意义上，是“神话”，而非“历史”，才使人愉悦。

既如此，历史学家为何还不依不饶，非把真相撕裂给大家看？

这是因为，不同人的感受有时会相互冲突，使一方愉悦的事，在另一方也许就是痛苦。因此，比愉悦更重要的，是正义和尊严，它们有时需以痛苦为代价去争取。一般来说，有能力建构神话并将其打扮成社会“共识”的，往往是握有强权的人，而这些神话是服务于他们自身利益的。当然，若说“终结神话”就能为弱势群体提供应有的权益和尊严，显然是妄想，但没有这至关重要的第一步，那些更实质的尊严又如何获得？这样，我们才明白，霍布斯鲍姆何以会在一篇讨论史家职责的演说中，突然提及那些“大多数普通人”：他们不够聪明，不够有趣，学历不高，“也注定不会功成名就”，然而，“任何值得人们在其中生活的社会都得为这些人着想，而不是为那些富人、精明人、杰出人物着想，尽管这样的社会也必须为这些少数人提供广阔的天地”。

但这并不意味着，终结神话仅对弱者才有意义。无论对谁，获知与自己有关的历史真相，都攸关其最基本的尊严。因此，对那些备受欺凌者来说，历史研究的目的是帮助他们拯救自己的尊严，而不是鼓励他们成为侵

略者。须知，在有些情形下，神话也是由弱势群体创造的。对此，霍氏并无偏袒之意：“我们确实不应忘记，在 1389 年有一场科索沃战役，塞尔维亚战士们和他们的盟军被穆斯林打败了，这在塞尔维亚一般人的记忆中留下了很深的伤痕，尽管如此，它并不去表明对现在占该地区总人口 90% 的阿尔巴尼亚人的压迫是合法的，也不表明声称该地区的领土基本上归他们所有的塞尔维亚人的要求是合法的。”

通过调动过去的资源，生活在今天的人们能够突破生活的坚壁和死角，无论向前或是退后，都是寻找海阔天空，给自己也给人家一条活路。在这个意义上，历史学确实会使我们生活得更美好。在讨论“科索沃战役”之前，霍布斯鲍姆还说了一句话：“几乎没有哪种褊狭的思想意识是基于单纯的谎言和毫无事实依据的虚构之上的”，提示出人类社会中一个更复杂的情势：在神话和真相之间，有时会存在一条混合道路；往往是这种混杂了谎言和事实的叙述，而不是那些完全的虚构，为“褊狭思想”提供了最具煽动力的基石，我们姑且称之为“半神话”（这里的“半”并非一个精确的计量概念）。

此言令我们想到陈寅恪先生曾讲到的一个现象：武则天为称帝颁行的《大云经》，决非如过去所说的伪经，相反，其经文全同旧本，奥秘只在疏证中：经过这番改造，原文增加了许多“新意”。盖伪造经典，其事既不易为，更难取信，不如在原文基础上“曲为比附”，反而事半功倍。这生动说明了“半神话”较“神话”更具煽惑力的原因：“半神话”本来包含一些事实，很容易激活有些人的片段经验，使他们倾向于肯定其整体判断。但实际上，个别事实一旦被放入一个有意歪曲了的叙述框架中，便完全可以用来服务于谎言。这一点，在诸如那些因为自己家里没有饿死人而否认大饥荒的言论中，很容易找到例证。

事实上，在大多数情形下，使历史学家付出更多精力和勇气去反抗的，正是这些“半神话”——为此，他们不只要得罪少数权贵，更可能得罪那些喜欢永远躲在一己经验中的“大多数普通人”。

如何写好技术文档？（陈斌）

我有十多年开发经验, 文科一直很好 (在上海最好的中学英语语文都拿过年级第一名), 从小热爱文学, 手不释卷. 我应有资格谈谈这个话题.

简洁最重要.

注意逻辑性.

实事求是. 要有例证.

与其拘泥技巧, 不如从根本着手, 提高个人修养和智力. 也就是从阅读开始, 读一流作品多了, 自然知道如何写.

我个人经验, 技术类阅读以作者本人是顶尖高手的开源文档最好. 如 http://tldp.org 上的 guide.

非技术类阅读, 从欧美古典非小说类作品开始比较好.

中国大陆解放后语文课本有很多二三流的东西, 容易误导年轻人外国作品好一点, 翻译因懂英文, 素质已高于一般人.

古典作品经过历史考验, 需专业知识少. 中国大陆的特点是解放后最好的中文作家都下放去做翻译了, 所以又多了一重质量保证. 这些老头老太翻译的通常是古典作品.

不要有先入为主, 以为经典就必然晦涩. 事实恰恰相反.

文献排版以专业工具处理, 如现在流行的 markdown. 不应该把时间浪费在排版上, 让工具去处理.

最好的文档就是不断维护的文档. 所以应少写文档, 写得越短越好. 否则你根本无精力提高质量.

5.26

Downton Abbey S1

已阅，挺精彩的一部剧，Anna 和 Bates 两人印象深刻。

如何高效且系统的学习金融知识？(知乎）

真心不太习惯先上结论：我经常对这些新员工说，不要指望着从零开始，一张白纸就能完整了解全部银行金融知识。应该从你自己学了这么多年的专业出发，寻找一个切入点，尝试以点带面，逐步扩张到整体。

——————— 假如你是会计专业 ———————

1、寻找一个切入点

学会计的，可以从各种财务报表、各种业务报表入手先了解借方、贷方的数据来源，各个会计科目的含义，并且跟各个业务经营部门的实际业务产品对上号。例如中间业务收入包括哪些产品的收入（理财、代发工资

诸如此类）。了解其收入起伏的原因和特点，占比等等。从这些内容你就基本可以建立一个粗线条的知识线了。从中可以了解银行主要业务和特点，以及阶段性发展的重心。

2、从知识线发散出去为什么信用卡收入全年有起伏，原因是什么？其主要收入来源是什么？（分期手续费还是刷卡分润）。然后再拓展出去，信用卡分期是什么东西？刷卡收入是什么？中间业务收入里面理财业务收入占比多少？理财包括哪些产品？这些产品是什么样子的。通过这些自学，你已基本了解银行各业务的基本内容啦，当然细节上还不够，就是要靠你自己在日常工作中逐步补充完善了。3、整理、重构一张知识网到第二步，你的知识线仅仅是二维的。一根粗线（知识线）上面有很多结点（产品点），但你需要开始了解各结点之间的关联。通常情况下，你可以随机抽两个结点进行比较，关联，思考。最后形成自己的一张知识网。例如：理财产品销售，个人储蓄存款之间的关系是什么？小额信用贷款，信用卡之间的异同点，关系是什么？

走到这里，恭喜你，基本上你已经是一个不错的业务能手了。

——————————— 最后的废话 ———————————

————- 嗯，其实说白了。就是从易到难，从自己最拿手的领域、角度出发，多思考，多阅读，多沟通。相信题主你能够成功的。

5.27 Open the Chase Checking and Saving

Casher's check and Personal check

When a customer asks a bank for a cashier's check, the bank debits the amount from the customer's account immediately, and then the bank assumes the responsibility for covering the cashier's check. This is in contrast with a personal check, where the bank does not debit the amount from the customer's account until the check is deposited or cashed by the recipient.

A cashier's check is different from a certified check, which is a personal check written by the customer and drawn on the customer's account, on which the bank certifies that the signature is genuine and that the customer has sufficient funds in the account to cover the check.

5.29 Hardware-Software Interface

hexadecimal
clobber 连续打击
proprietor 业主，所有人
senility 老年
electrical & electronic 电气的 & 电子的

How is memory organized? How do we find data in memory?

See the slides of the Hardware-Software Interface course.
MMU 内存管理单元

Jamie Zawinski

I recommend that you do what you love because you love doing it. If that means long hours, fantastic. If that means leaving the office by 6pm every day for your underwater basket-weaving class, also fantastic.

"Time always softens the pain and makes things look like more fun than they really were. But who said everything has to be fun? Pain builds character. (Sometimes it builds products, too.) "

SCIP & CSAPP

Somehow I really want to read the first three Chs of SCIP this year, I've ordered a used hardcover version online and let's see how far I could reach.

A Book Review of SCIP from Peter Norvig

I think its fascinating that there is such a split between those who love and hate this book. For most books, the review is a bell-shaped curve of star ratings; this one has a peak at 1, a peak at 5, and very little in between. How could this be? I think it is because SICP is a very personal message that works only if the reader is at heart a computer scientist (or willing to

become one). So I agree that the book's odds of success are better if you read it after having some experience.

To use an analogy, if SICP were about automobiles, it would be for the person who wants to know how cars work, how they are built, and how one might design fuel-efficient, safe, reliable vehicles for the 21st century. The people who hate SICP are the ones who just want to know how to drive their car on the highway, just like everyone else.

Those who hate SICP think it doesn't deliver enough tips and tricks for the amount of time it takes to read. But if you're like me, you're not looking for one more trick, rather you're looking for a way of synthesizing what you already know, and building a rich framework onto which you can add new learning over a career. That's what SICP has done for me. I read a draft version of the book around 1982, when I was in grad school, and it changed the way I think about my profession. If you're a thoughtful computer scientist (or want to be one), it will change your life too.

Some of the reviewers complain that SICP doesn't teach the basics of OO design, and so on. In a sense they are right. The book doesn't directly tell you how to design and write an object-oriented program using the subset of object-oriented principles that show up in the syntax of Java or C++. Rather, the book tells you what those principles are, how they came to be selected as worthwhile, how they can be implemented from the ground up, and how a different combination of principles might be more appropriate for some particular problems. This approach requires you to understand the range of possibilities, and to think about trade-offs as you go through the design process. Programming is a craft that is subject to frequent failure: many projects are started and abandoned because the designers do not have the flexibility, experience and understanding to come up with a suitable design and implementation. SICP gives you an approach that will succeed, but it is an approach based on principles and wisdom, not on a checklist. If you don't understand the principles, or if you are the kind of person who wants to be given a cookbook of what to do rather than to think creatively, or if you only want to work on problems that are pretty much like the problem you worked on last time, then this approach will not work

for you. There are other approaches that will be more reproducible for a limited range of simple problems, but there is no better way than SICP to learn how to address the truly hard problems.

Donald Knuth says he wrote his books for "the one person in 50 who has this strange way of thinking that makes a programmer". I think the most amazing thing about SICP is that there are so FEW people who hate it: if Knuth were right, then only 1 out of 50 people would be giving this 5 stars, instead of about 25 out of 50. Now, a big part of the explanation is that the audience is self-selected, and is not a representative sample. But I think part of it is because Sussman and Abelson have succeeded grandly in communicating "this strange way of thinking" to (some but not all) people who otherwise would never get there.

Another Review

The negative reviewers entirely missed the point of this book. The issues are not c++ versus scheme, nor the gap between the book's examples and real-world programs, nor that recursion is less intuitive than looping. The real point is to teach some very core foundations of computer science that show up everywhere. *For example, supposedly revolutionary XML looks a heck of a lot like a nested scheme list, first described in 1960. And processing an active server page (or Java server page) is very much like the textbook's specialized language evaluator. Finally, c++ polymorphism through vtables and part of Microsoft's COM mechanics are the exact same thing as the book's data-directed programming section. This is very deep material for a programming newbie to learn outside a course, but for an experienced nerd who's looking for a systematic framework, it's absolutely terrific.* I had done lots of lisp and compiler work before reading the book, so many of the concepts were not new. But it's with this framework in mind that I learn new technologies, and this approach greatly speeds up how long it takes to understand each week's "new" hot product/language/tool/mindset. Put another way: why do so many popular computer books take 1000 pages to describe a few trivial concepts?

Why SCIP matters?

SICP was revolutionary in many different ways. Most importantly, it dramatically raised the bar for the intellectual content of introductory computer science. Before SICP, the first CS course was almost always entirely filled with learning the details of some programming language. SICP is about standing back from the details to learn big-picture ways to think about the programming process. It focused attention on the central idea of abstraction – finding general patterns from specific problems and building software tools that embody each pattern. It made heavy use of the idea of functions as data, an idea that's hard to learn initially, but immensely powerful once learned. (This is the same idea, in a different form, that makes freshman calculus so notoriously hard even for students who've done well in earlier math classes.) It fit into the first CS course three different programming paradigms (functional, object oriented, and declarative), when most other courses didn't even really discuss even one paradigm.

Another revolution was the choice of Scheme as the programming language. To this day, most introductions to computer science use whatever is the "hot" language of the moment: from Pascal to C to C++ to Java to Python. Scheme has never been widely used in industry, but it's the perfect language for an introduction to CS. *For one thing, it has a very simple, uniform notation for everything. Other languages have one notation for variable assignment, another notation for conditional execution, two or three more for looping, and yet another for function calls. Courses that teach those languages spend at least half their time just on learning the notation. In my SICP-based course at Berkeley, we spend the first hour on notation and that's all we need; for the rest of the semester we're learning ideas, not syntax. Also, despite (or because of) its simplicity, Scheme is a very versatile language, making it possible for us to examine those three programming paradigms and, in particular, letting us see how object oriented programming is implemented, so OOP languages don't seem like magic to our students. Scheme is a dialect of Lisp, so it's great at handling functions as data, but it's a stripped-down version compared to the ones more commonly used for professional programming, with a minimum of bells and*

whistles. It was very brave of Abelson and Sussman to teach their introductory course in the best possible language for teaching, paying no attention to complaints that all the jobs were in some other language. Once you learned the big ideas, they thought, and this is my experience also, learning another programming language isn't a big deal; it's a chore for a weekend. I tell my students, "the language in which you'll spend most of your working life hasn't been invented yet, so we can't teach it to you. Instead we have to give you the skills you need to learn new languages as they appear."

Finally, SICP was firmly optimistic about what a college freshman can be expected to accomplish. SICP students write interpreters for programming languages, ordinarily considered more appropriate for juniors or seniors. The text itself isn't easy reading; it has none of the sidebars and colored boxes and interesting pictures that typify the modern textbook aimed at students with low attention spans. There are no redundant exercises; each exercise teaches an important new idea. It uses big words. But it repays a close reading; every sentence matters.

Statistically, SICP-based courses have been a small minority. But the book has had an influence beyond that minority. It inspired a number of later textbooks whose authors consciously tried to live up to SICP's standard. The use of Scheme as a language for learners has been extended by others over a range from middle school to graduate school. Even the more mainstream courses have become sensitive to the idea of programming paradigms, although most of them concentrate on object oriented programming. The idea that computer science should be about ideas, not entirely about programming practice, has since widened to include non-technical ideas about the context and social implications of computing.

SICP itself has had a longevity that's very unusual for introductory CS textbooks. Usually, a book lasts only as long as the language fad to which it is attached. SICP has been going strong for over 25 years and shows no sign of going out of print. Computing has changed enormously over that time, from giant mainframe computers to personal computers to the Internet on cell phones. *And yet the big ideas behind these changes remain the same, and they are well captured by SICP.*

I've been teaching a SICP-based course since 1987. The course has changed incrementally over that time; we've added sections on parallelism, concurrency control, user interface design, and the client/server paradigm. But it's still essentially the same course. Every five years or so, someone on the faculty suggests that our first course should use language X instead; each time, I say "when someone writes the best computer science book in the world using language X, that'll be fine" and so far the faculty have always voted to stay with the SICP course.

豆瓣书评

---时间 —

本书共有 5 章，每章都有近 100 道习题。前三章的习题我做了 90%，后两章太难，大概只做了 70%。　　这本书可以说是时间黑洞。每章分为 4-5 节，每节有几个小节，全书有一百小节（即 X.X.X）左右。我以小节为单位进行了估算，包括完成习题，每小节大约需要一个小时。当然不同小节难度不同，有的耗时长些，有的短些。于是读完本书并做完大部分习题需要上百个小时。再加上听课或看视频教程的时间则会更长。所以我觉得恐怕只有在校学生才有时间和精力来完成这本书的学习。

---内容 —

本书按照内容可以分为三个部分：过程抽象（第一章）；数据抽象（第二、三章）和语言抽象（第四、五章）。　　过程抽象部分比较简单，先介绍了 Scheme 的基本语法，让读者初步领略函数式编程的风采。对于有一定编程基础（相信国内极少有人入门就读这个）的读者来说，会有耳目一新的感觉，原来递归和迭代可以有另一种表现形式，但并不难理解。习题也比较简单，不会用掉太多的时间。过程抽象的概念也很简单，就是编程语言中的函数，目的是封装计算过程的细节。关于何时应该用过程抽象的原则是：一切可以定义为过程的计算片段都应该定义为过程。　　数据抽象是我认为的本书的核心，也是最值得我们仔细研读的部分。关于数据抽象最直接的理解就是面向对象编程，如 *C++*，而 *Java* 和 *C#* 则是更彻底的数据抽象。把一组过程抽象（类的方法）集中考虑，并加入内部状态（类的变量），就是一个数据抽象。每个数据抽象都应该把自己的内部对象状态和对象的实现隐藏起来，对外通过一组接口进行消息传递。这样听起来好像本书与一般的面向对象书没有区别，但实际上，这些都是我自己的

总结，书里面不会把这些概念直接罗列出来，而是通过一个个巧妙的例子，让读者一步步深入，感叹原来 A 还可以这样抽象，原来 B 还可以这样封装。个人认为如果时间有限，读完前三章已经可以领会本书大部分思想了，后两章可以不读。　　语言抽象是指自己发明一门语言，以解决某一特定应用领域的问题。在这一领域中，自己发明的语言会比其他通用语言更方便。定义了新语言的语法后，就要自己去实现该语言的编译器或解释器，可以通过现有的语言去构造。这一部分包含了许多编译方面的知识，但又与编译原理中的构造方法有不少区别，自己看书很容易看得云里雾里，听老师讲课才好一些。大部分习题很难做，一部分习题非常难。

---原因 —

为什么我们要学习这本书？因为这本书告诉我们如何抽象。为什么我们要学习如何抽象？因为抽象是我们控制软件复杂性的重要手段。

软件是人类有史以来最复杂的系统。其一、软件系统本身规模庞大，参与人手众多，难以管理；其二、环境和需求不断变化，且错误难以避免。

人类无法驾驭过于复杂的事物，于是只能寻找方法简化软件系统：把系统分为许多子部分，人们开发一个部分的时候，系统其他部分都是一种抽象，无需了解其细节。　　本书讨论的就是系统的组织和设计，有哪些方法可以帮助我们控制软件的复杂度。

---收获 —

除了贯穿全书的抽象思想外，我们还能从本书中学到（摘抄自老师 ppt。。。）：　　从另一个角度看程序和程序设计中的问题　　函数式程序设计　　多种多样的程序组织方式　　丰富多彩的编程模式　　对一些基础问题的理解

---资源 —

这本书相关的资料有不少，包括 MIT 的官方视频教程和非官方习题答案。　　另外就是书不好买，几大网上书店都没了，我还是在学校的教材中心买到的。

- MIT 的视频教程：　　http://swiss.csail.mit.edu/classes/6.001/abelson-sussman-lectures/
- 课后答案版本 1：　　全。很多题不是用书中的 Sheme 而是 Common Lisp，但没有太大影响。　　http://eli.thegreenplace.net/category/programming/lisp/sicp/

- 课后答案版本 2： 排版好，每一题给出多种语言的解答。 前 3 章比较全。第四章缺的太多，第五章完全没有。 http://sicp.org.ua/sicp/FrontPage

PS:

MIT 开始用 python 教授此课，代替了原本的 Scheme： http://www.wisdomandwonder.com/lin
mit-switched-from-scheme-to-python

The Perils of JavaSchools

Lazy kids.

Whatever happened to hard work?

A sure sign of my descent into senility is bitchin' and moanin' about "kids these days," and how they won't or can't do anything hard any more.

"You were lucky. We lived for three months in a brown paper bag in a septic tank. We had to get up at six in the morning, clean the bag, eat a crust of stale bread, go to work down the mill, fourteen hours a day, week-in week-out, and when we got home our Dad would thrash us to sleep with his belt." —Monty Python's Flying Circus, Four Yorkshiremen When I was a kid, I learned to program on punched cards. If you made a mistake, you didn't have any of these modern features like a backspace key to correct it. You threw away the card and started over.

When I started interviewing programmers in 1991, I would generally let them use any language they wanted to solve the coding problems I gave them. 99% of the time, they chose C.

Nowadays, they tend to choose Java.

Now, don't get me wrong: there's nothing wrong with Java as an implementation language.

Wait a minute, I want to modify that statement. I'm not claiming, in this particular article, that there's anything wrong with Java as an implementation language. There are lots of things wrong with it but those will have to wait for a different article.

Instead what I'd like to claim is that Java is not, generally, a hard enough programming language that it can be used to discriminate between great programmers and mediocre programmers. It may be a fine language

to work in, but that's not today's topic. I would even go so far as to say that the fact that Java is not hard enough is a feature, not a bug, but it does have this one problem.

If I may be so brash, it has been my humble experience that there are two things traditionally taught in universities as a part of a computer science curriculum which many people just never really fully comprehend: pointers and recursion.

You used to start out in college with a course in data structures, with linked lists and hash tables and whatnot, with extensive use of pointers. Those courses were often used as weedout courses: they were so hard that anyone that couldn't handle the mental challenge of a CS degree would give up, which was a good thing, because if you thought pointers are hard, wait until you try to prove things about fixed point theory.

All the kids who did great in high school writing pong games in BASIC for their Apple II would get to college, take CompSci 101, *a data structures course, and when they hit the pointers business their brains would just totally explode, and the next thing you knew, they were majoring in Political Science because law school seemed like a better idea.* I've seen all kinds of figures for drop-out rates in CS and they're usually between 40% and 70%. The universities tend to see this as a waste; I think it's just a necessary culling of the people who aren't going to be happy or successful in programming careers.

The other hard course for many young CS students was the course where you learned functional programming, including recursive programming. MIT set the bar very high for these courses, creating a required course (6.001) and a textbook (Abelson & Sussman's Structure and Interpretation of Computer Programs) which were used at dozens or even hundreds of top CS schools as the de facto introduction to computer science. (You can, and should, watch an older version of the lectures online.)

The difficulty of these courses is astonishing. In the first lecture you've learned pretty much all of Scheme, and you're already being introduced to a fixed-point function that takes another function as its input. When I struggled through such a course, CSE121 at Penn, I watched as many if not

most of the students just didn't make it. The material was too hard. I wrote a long sob email to the professor saying It Just Wasn't Fair. Somebody at Penn must have listened to me (or one of the other complainers), because that course is now taught in Java.

I wish they hadn't listened.

Think you have what it takes? Test Yourself Here! Therein lies the debate. Years of whinging by lazy CS undergrads like me, combined with complaints from industry about how few CS majors are graduating from American universities, have taken a toll, and in the last decade a large number of otherwise perfectly good schools have gone 100% Java. It's hip, the recruiters who use "grep" to evaluate resumes seem to like it, and, best of all, there's nothing hard enough about Java to really weed out the programmers without the part of the brain that does pointers or recursion, so the drop-out rates are lower, and the computer science departments have more students, and bigger budgets, and all is well.

The lucky kids of JavaSchools are never going to get weird segfaults trying to implement pointer-based hash tables. They're never going to go stark, raving mad trying to pack things into bits. They'll never have to get their head around how, in a purely functional program, the value of a variable never changes, and yet, it changes all the time! A paradox!

They don't need that part of the brain to get a 4.0 in major.

Am I just one of those old-fashioned curmudgeons, like the Four Yorkshiremen, bragging about how tough I was to survive all that hard stuff?

Heck, in 1900, Latin and Greek were required subjects in college, not because they served any purpose, but because they were sort of considered an obvious requirement for educated people. In some sense my argument is no different that the argument made by the pro-Latin people (all four of them). "[Latin] trains your mind. Trains your memory. Unraveling a Latin sentence is an excellent exercise in thought, a real intellectual puzzle, and a good introduction to logical thinking," writes Scott Barker. But I can't find a single university that requires Latin any more. Are pointers and recursion the Latin and Greek of Computer Science?

Now, I freely admit that programming with pointers is not needed in

90% of the code written today, and in fact, it's downright dangerous in production code. OK. That's fine. And functional programming is just not used much in practice. Agreed.

But it's still important for some of the most exciting programming jobs. Without pointers, for example, you'd never be able to work on the Linux kernel. You can't understand a line of code in Linux, or, indeed, any operating system, without really understanding pointers.

Without understanding functional programming, you can't invent MapReduce, the algorithm that makes Google so massively scalable. The terms Map and Reduce come from Lisp and functional programming. MapReduce is, in retrospect, obvious to anyone who remembers from their 6.001-equivalent programming class that purely functional programs have no side effects and are thus trivially parallelizable. The very fact that Google invented MapReduce, and Microsoft didn't, says something about why Microsoft is still playing catch up trying to get basic search features to work, while Google has moved on to the next problem: building Skynet^H^H^H^H^H^H the world's largest massively parallel supercomputer. I don't think Microsoft completely understands just how far behind they are on that wave.

But beyond the prima-facie importance of pointers and recursion, their real value is that building big systems requires the kind of mental flexibility you get from learning about them, and the mental aptitude you need to avoid being weeded out of the courses in which they are taught. Pointers and recursion require a certain ability to reason, to think in abstractions, and, most importantly, to view a problem at several levels of abstraction simultaneously. And thus, the ability to understand pointers and recursion is directly correlated with the ability to be a great programmer.

Nothing about an all-Java CS degree really weeds out the students who lack the mental agility to deal with these concepts. As an employer, I've seen that the 100% Java schools have started churning out quite a few CS graduates who are simply not smart enough to work as programmers on anything more sophisticated than Yet Another Java Accounting Application, although they did manage to squeak through the newly-dumbed-down coursework. These students would never survive 6.001 at MIT, or CS 323 at

Yale, and frankly, that is one reason why, as an employer, a CS degree from MIT or Yale carries more weight than a CS degree from Duke, which recently went All-Java, or U. Penn, which replaced Scheme and ML with Java in trying to teach the class that nearly killed me and my friends, CSE121. Not that I don't want to hire smart kids from Duke and Penn – I do – it's just a lot harder for me to figure out who they are. I used to be able to tell the smart kids because they could rip through a recursive algorithm in seconds, or implement linked-list manipulation functions using pointers as fast as they could write on the whiteboard. But with a JavaSchool Grad, I can't tell if they're struggling with these problems because they are undereducated or if they're struggling with these problems because they don't actually have that special part of the brain that they're going to need to do great programming work. Paul Graham calls them Blub Programmers.

It's bad enough that JavaSchools fail to weed out the kids who are never going to be great programmers, which the schools could justifiably say is not their problem. Industry, or, at least, the recruiters-who-use-grep, are surely clamoring for Java to be taught.

But JavaSchools also fail to train the brains of kids to be adept, agile, and flexible enough to do good software design (and I don't mean OO "design", where you spend countless hours rewriting your code to rejiggle your object hierarchy, or you fret about faux "problems" like has-a vs. is-a). You need training to think of things at multiple levels of abstraction simultaneously, and that kind of thinking is exactly what you need to design great software architecture.

You may be wondering if teaching object oriented programming (OOP) is a good weed-out substitute for pointers and recursion. The quick answer: no. Without debating OOP on the merits, it is just not hard enough to weed out mediocre programmers. OOP in school consists mostly of memorizing a bunch of vocabulary terms like "encapsulation" and "inheritance" and taking multiple-choice quizzicles on the difference between polymorphism and overloading. Not much harder than memorizing famous dates and names in a history class, OOP poses inadequate mental challenges to scare away first-year students. When you struggle with an OOP problem, your program still

works, it's just sort of hard to maintain. Allegedly. But when you struggle with pointers, your program produces the line Segmentation Fault and you have no idea what's going on, until you stop and take a deep breath and really try to force your mind to work at two different levels of abstraction simultaneously.

The recruiters-who-use-grep, by the way, are ridiculed here, and for good reason. I have never met anyone who can do Scheme, Haskell, and C pointers who can't pick up Java in two days, and create better Java code than people with five years of experience in Java, but try explaining that to the average HR drone.

But what about the CS mission of CS departments? They're not vocational schools! It shouldn't be their job to train people to work in industry. That's for community colleges and government retraining programs for displaced workers, they will tell you. They're supposed to be giving students the fundamental tools to live their lives, not preparing them for their first weeks on the job. Right?

Card Punch – yes, I learned Fortran on one of these when I was 12.Still. CS is proofs (recursion), algorithms (recursion), languages (lambda calculus), operating systems (pointers), compilers (lambda calculus) – and so the bottom line is that a JavaSchool that won't teach C and won't teach Scheme is not really teaching computer science, either. As useless as the concept of function currying may be to the real world, it's obviously a prereq for CS grad school. I can't understand why the professors on the curriculum committees at CS schools have allowed their programs to be dumbed down to the point where not only can't they produce working programmers, they can't even produce CS grad students who might get PhDs and compete for their jobs. Oh wait. Never mind. Maybe I do understand.

Actually if you go back and research the discussion that took place in academia during the Great Java Shift, you'll notice that the biggest concern was whether Java was simple enough to use as a teaching language.

My God, I thought, they're trying to dumb down the curriculum even further! Why not spoon feed everything to the students? Let's have the TAs take their tests for them, too, then nobody will switch to American

Studies. How is anyone supposed to learn anything if the curriculum has been carefully designed to make everything easier than it already is? There seems to be a task force underway (PDF) to figure out a simple subset of Java that can be taught to students, producing simplified documentation that carefully hides all that EJB/J2EE crap from their tender minds, so they don't have to worry their little heads with any classes that you don't need to do the ever-easier CS problem sets.

The most sympathetic interpretation of why CS departments are so enthusiastic to dumb down their classes is that it leaves them more time to teach actual CS concepts, if they don't need to spend two whole lectures unconfusing students about the difference between, say, a Java int and an Integer. Well, if that's the case, 6.001 has the perfect answer for you: Scheme, a teaching language so simple that the entire language can be taught to bright students in about ten minutes; then you can spend the rest of the semester on fixed points.

< 如何掌握程序语言 > by 王垠

学习程序语言是每个程序员的必经之路。可是这个世界上有太多的程序语言，每一种都号称具有最新的“特性”。所以程序员的苦恼就在于总是需要学习各种稀奇古怪的语言，而且必须紧跟“潮流”，否则就怕被时代所淘汰。

作为一个程序语言的研究者，我深深的知道这种心理产生的根源。程序语言里面其实有着非常简单，永恒不变的原理。看到了它们，就可以在很短的时间之内就能学会并且开始使用任何新的语言，而不是花费很多功夫去学习一个又一个的语言。

对程序语言的各种误解

学习程序语言的人，经常会出现以下几种心理，以至于他们会觉得有学不完的东西，或者走上错误的道路。以下我把这些心理简要分析一下。

1. 程序语言无用论。这是国内大学计算机系的教育常见的错误。教授们常常对学生灌输：“用什么程序语言不重要，重要的是算法。”而其实，程序语言却是比算法更加精髓的东西。任何算法以及它的复杂度

分析，都是相对于某种计算模型，而程序语言就是描述这种计算模型的符号系统。算法必须用某种语言表述出来，通常算法设计者使用伪码，这其实是不严谨的，容易出现推理漏洞。算法设计再好，如果不懂得程序语言的原理，也不可能高效的实现。即使实现了，也可能会在模块化和可扩展性上面有很大问题。某些算法专家或者数学家写出来的程序极其幼稚，就是因为他们忽视了程序语言的重要性。

2. 追求“新语言”。基本的哲学告诉我们，新出现的事物并不一定是“新事物”，它们有可能是历史的倒退。事实证明，新出现的语言，可能还不如早就存在的。其实，现代语言的多少“新概念”不存在于最老的一些语言里呢？程序语言就像商品，每一家都为了拉拢程序员作广告，而它们绝大多数的设计都可能是肤浅而短命的。如果你看不透这些东西的设计，就会被它们蒙蔽住。很多语言设计者其实并不真的懂得程序语言设计的原理，所以常常在设计中重复前人的错误。但是为了推销自己的语言和系统，他们必须夸夸其谈，进行宗教式的宣传。

3. “存在即是合理”。记得某人说过：“不能带来新的思维方式的语言，是没有必要存在的。”他说的是相当正确的。世界上有这么多的语言，有哪些带来了新的思维方式呢？其实非常少。绝大部分的语言给世界带来的其实是混乱。有人可能反驳说：“你怎么能说 A 语言没必要存在？我要用的那个库 L，别的语言不支持，只能用 A。”但是注意，他说的是存在的“必要性”。如果你把存在的“事实”作为存在的“必要性”，那就逻辑错乱了。就像如果二战时我们没能打败希特勒，现在都做了他的奴隶，然后你就说：“希特勒应该存在，因为他养活了我们。”你的逻辑显然有问题，因为如果历史走了另外一条路（即希特勒不存在），我们会过上自由幸福的生活，所以希特勒不应该存在。对比一个东西存在与不存在的两种可能的后果，然后做出判断，这才是正确的逻辑。按照这样的推理，如果设计糟糕的 A 语言不存在，那么设计更好的 B 语言很有可能就会得到更多的支持，从而实现甚至超越 L 库的功能。

4. 追求“新特性”。程序语言的设计者总是喜欢“发明”新的名词，喜欢炒作。普通程序员往往看不到，大部分这些“新概念”其实徒有高深而时髦的外表，却没有实质的内涵。常常是刚学会一个语言 A，又

来了另一个语言 B，说它有一个叫 XYZ 的新特性。于是你又开始学习 B，如此继续。在内行人看来，这些所谓的“新特性”绝大部分都是新瓶装老酒。很多人写论文喜欢起这样的标题：《XYZ：A Novel Method for ...》。这造成了概念的爆炸，却没有实质的进步。

5. 追求“小窍门”。很多编程书喜欢卖弄一些小窍门，教你如何让程序显得“短小”。比如它们会跟你讲 "(i++) - (++i)" 应该得到什么结果；或者追究运算符的优先级，说这样可以少打括号；要不就是告诉你“if 后面如果只有一行代码就可以不加花括号”，等等。殊不知这些小窍门，其实大部分都是程序语言设计的败笔。它们带来的不是清晰的思路，而是是逻辑的混乱和认知的负担。比如 C 语言的 ++ 运算符，它的出现是因为 C 语言设计者们当初用的计算机内存小的可怜，而 "i++" 显然比 "i=i+1" 少 2 个字符，所以他们觉得可以节省一些空间。现在我们再也不缺那点内存，可是 ++ 运算符带来的混乱和迷惑，却流传了下来。现在最新的一些语言，也喜欢耍这种语法上的小把戏。如果你追求这些小窍门，往往就抓不住精髓。

6. 针对“专门领域”。很多语言没有新的东西，为了占据一方土地，就号称自己适合某种特定的任务，比如文本处理，数据库查询，WEB 编程，游戏设计，并行计算。但是我们真的需要不同的语言来干这些事情吗？其实绝大部分这些事情都能用同一种通用语言来解决，或者在已有语言的基础上做很小的改动。只不过由于各种政治和商业原因，不同的语言被设计用来占领市场。就学习而言，它们其实是无关紧要的，而它们带来的“学习负担”，其实差不多掩盖了它们带来的好处。其实从一些设计良好的通用语言，你可以学会所有这些“专用语言”的精髓，而不用专门去学它们。

7. 宗教信仰。很多人对程序语言有宗教信仰。这跟人们对操作系统有宗教信仰很类似。其实如果你了解程序语言的本质，就会发现其实完全没必要跟人争论一些事情。某个语言有缺点，应该可以直接说出来，却被很多人忌讳，因为指出缺点总是招来争论和憎恨。这原因也许在于程序语言的设计不是科学，它类似于圣经，它没法被“证伪”。没有任何实验可以一下子断定那种语言是对的，那种是错的。所以虽然你觉得自己有理，却很难让人信服。没有人会去争论哪家的汉堡更好，却有很多人争论那种语言更好。因为很多人把程序语言当成自己

的神，如果你批评我的语言，你就是亵渎我的神。解决的办法也许是，不要把自己正在用的语言看得太重要。你现在认为是对的东西，也许不久就会被你认为是错的，反之亦然。

如何掌握程序语言

看到了一些常见的错误心理，那么我们来谈一下什么样的思维方式会更加容易的掌握程序语言。

1. 专注于“精华”和“原理”。就像所有的科学一样，程序语言最精华的原理其实只有很少数几个，它们却可以被用来构造出许许多多纷繁复杂的概念。但是人们往往忽视了简单原理的重要性，匆匆看过之后就去追求最新的，复杂的概念。他们却没有注意到，绝大部分最新的概念其实都可以用最简单的那些概念组合而成。而对基本概念的一知半解，导致了他们看不清那些复杂概念的实质。比如这些概念里面很重要的一个就是递归。国内很多学生对递归的理解只停留于汉诺塔这样的程序，而对递归的效率也有很大的误解，认为递归没有循环来得高效。而其实递归比循环表达能力强很多，而且效率几乎一样。有些程序比如解释器，不用递归的话基本没法完成。

2. 实现一个程序语言。学习使用一个工具的最好的方式就是制造它，所以学习程序语言的最好方式就是实现一个程序语言。这并不需要一个完整的编译器，而只需要写一些简单的解释器，实现最基本的功能（自注：好建议但不适用于现在的我）。之后你就会发现，所有语言的新特性你都大概知道可以如何实现，而不只停留在使用者的水平。实现程序语言最迅速的方式就是使用一种像 Scheme 这样代码可以被作为数据的语言。它能让你很快的写出新的语言的解释器。我的 GitHub 里面有一些我写的解释器的例子（比如这个短小的代码实现了 Haskell 的 lazy 语义）。

几种常见风格的语言

下面我简要的说一下几种常见风格的语言以及它们的问题。

1. 面向对象语言

事实说明，“面向对象”这整个概念基本是错误的。它的风靡是因为当初的“软件危机”（天知道是不是真的存在这危机）。设计的初衷是让“界面”和“实现”分离，从而使得下层实现的改动不影响上层的功能。可是大部分面向对象语言的设计都遵循一个根本错误的原则：“所有的东西都是对象（Everything is an object）。”以至于所有的函数都必须放在所谓的“对象”里面，而不能直接被作为参数或者变量传递。这导致很多时候需要使用繁琐的设计模式 (design patterns) 来达到甚至对于 C 语言都直接了当的事情。而其实“界面”和“实现”的分离，并不需要把所有函数都放进对象里。另外的一些概念，比如继承，重载，其实带来的问题比它们解决的还要多。

“面向对象方法”的过度使用，已经开始引起对整个业界的负面作用。很多公司里的程序员喜欢生搬硬套一些不必要的设计模式，其实什么好事情也没干，只是使得程序冗长难懂。

那么如何看待具备高阶函数的面向对象语言，比如 Python, JavaScript, Ruby, Scala? 当然有了高阶函数，你可以直截了当的表示很多东西，而不需要使用设计模式。但是由于设计模式思想的流毒，一些程序员居然在这些不需要设计模式的语言里也采用繁琐的设计模式，让人哭笑不得。所以在学习的时候，最好不要用这些语言，以免受到不必要的干扰。到时候必要的时候再回来使用它们，就可以取其精华，去其糟粕。

2. 低级过程式语言

那么是否 C 这样的“低级语言”就会好一些呢？其实也不是。很多人推崇 C，因为它可以让人接近“底层”，也就是接近机器的表示，这样就意味着它速度快。这里其实有三个问题：

1) 接近“底层”是否是好事？2)“速度快的语言”是什么意思？3) 接近底层的语言是否一定速度快？

对于第一个问题，答案是否定的。其实编程最重要的思想是高层的语义 (semantics)。语义构成了人关心的问题以及解决它们的算法。而具体的实现 (implementation)，比如一个整数用几个字节表示，虽然还是重要，但却不是至关重要的。如果把实现作为学习的主要目标，就本末倒置了。因为实现是可以改变的，而它们所表达的本质却不会变。所以很多人发现自己学会的东西，过不了多久就“过时”了。那就是因为他们学习的不是本

质，而只是具体的实现。

其次，谈语言的“速度”，其实是一句空话。语言只负责描述一个程序，而程序运行的速度，其实绝大部分不取决于语言。它主要取决于 *1)* 算法和 *2)* 编译器的质量。编译器和语言基本是两码事。同一个语言可以有很多不同的编译器实现，每个编译器生成的代码质量都可能不同，所以你没法说“A 语言比 B 语言快”。你只能说“A 语言的 X 编译器生成的代码，比 B 语言的 Y 编译器生成的代码高效”。这几乎等于什么也没说，因为 B 语言可能会有别的编译器，使得它生成更快的代码。

我举个例子吧。在历史上，Lisp 语言享有“龟速”的美名。有人说“Lisp 程序员知道每个东西的值，却不知道任何事情的代价”，讲的就是这个事情。但这已经是很久远的事情了，现代的 Lisp 系统能编译出非常高效的代码。比如商业的 Chez Scheme 编译器，能在 5 秒钟之内编译它自己，编译生成的目标代码非常高效。它可以直接把 Scheme 程序编译到多种处理器的机器指令，而不通过任何第三方软件。它内部的一些算法，其实比开源的 LLVM 之类的先进很多。

另外一些函数式语言也能生成高效的代码，比如 OCaml。在一次程序语言暑期班上，Cornell 的 Robert Constable 教授讲了一个故事，说是他们用 OCaml 重新实现了一个系统，结果发现 OCaml 的实现比原来的 C 语言实现快了 50 倍。经过 C 语言的那个小组对算法多次的优化，OCaml 的版本还是快好几倍。这里的原因其实在于两方面。第一是因为函数式语言把程序员从底层细节中解脱出来，让他们能够迅速的实现和修改自己的想法，所以他们能够迅速的找到更好的算法。第二是因为 OCaml 有高效的编译器实现，使得它能生成很好的代码。

从上面的例子，你也许已经可以看出，其实接近底层的语言不一定速度就快。因为编译器这种东西其实可以有很高级的“智能”，甚至可以超越任何人能做到的底层优化。但是编译器还没有发展到可以代替人来制造算法的地步。所以现在人需要做的，其实只是设计和优化自己的高层算法。

3. 高级过程式语言

很早的时候，国内计算机系学生的第一门编程课都是 Pascal。Pascal 是很不错的语言，可是很多人当时都没有意识到。上大学的时候，我的 Pascal 老师对我们说：“我们学校的教学太落后了。别的学校都开始教 C 或者 C++ 了，我们还在教 Pascal。”现在真正理解了程序语言的设计原

理以后我才真正的感觉到，原来 Pascal 是比 C 和 C++ 设计更好的语言。它不但把人从底层细节里解脱出来，没有面向对象的思维枷锁，而且有一些很好的设计，比如强类型检查，嵌套函数定义等等。可是计算机的世界真是谬论横行，有些人批评 Pascal，把优点都说成是缺点。比如 Brain Kernighan 的这篇《Why Pascal is Not My Favorite Programming Language》，现在看来真是谬误百出。Pascal 现在已经几乎没有人用了。这并不很可惜，因为它被错怪的“缺点”其实已经被正名，并且出现在当今最流行的一些语言里：Java, Python, C#, ……

4. 函数式语言

函数式语言相对来说是当今最好的设计，因为它们不但让人专注于算法和对问题的解决，而且没有面向对象语言那些思维的限制。但是需要注意的是并不是每个函数式语言的特性都是好东西。它们的支持者们经常把缺点也说成是优点，结果你其实还是被挂上一些不必要的枷锁。比如 OCaml 和 SML，因为它们的类型系统里面有很多不成熟的设计，导致你需要记住太多不必要的规则。

5. 逻辑式语言

逻辑式语言（比如 Prolog）是一种超越函数式语言的新的思想，所以需要一些特殊的训练。逻辑式语言写的程序，是能“反向运行”的。普通程序语言写的程序，如果你给它一个输入，它会给你一个输出。但是逻辑式语言很特别，如果你给它一个输出，它可以反过来给你所有可能的输入。其实通过很简单的方法，可以不费力气的把程序从函数式转换成逻辑式的。但是逻辑式语言一般要在“pure”的情况下（也就是没有复杂的赋值操作）才能反向运行。所以学习逻辑式语言最好是从函数式语言开始，在理解了递归，模式匹配等基本的函数式编程技巧之后再来看 Prolog，就会发现逻辑式编程简单了很多。

从何开始

可是学习编程总要从某种语言开始。那么哪种语言呢？就我的观点，首先可以从 *Scheme* 入门，然后学习一些 *Haskell* (但不是全部)，之后其它的也就触类旁通了。你并不需要学习它们的所有细枝末节，而只需要学习

最精华的部分。所有剩余的细节，会在实际使用中很容易的被填补上。现在我推荐几本比较好的书。

《The Little Schemer》(TLS)：我觉得 Dan Friedman 的 The Little Schemer 是目前最好，最精华的编程入门教材。这本书很薄，很精辟。它的前身叫《The Little Lisper》。很多资深的程序语言专家都是从这本书学会了 Lisp。虽然它叫“The Little Schemer”，但它并不使用 Scheme 所有的功能，而是忽略了 Scheme 的一些毛病，直接进入最关键的主题：递归和它的基本原则。

《Structure and Interpretation of Computer Programs》(SICP)：The Little Schemer 其实是比较难的读物，所以我建议把它作为下一步精通的读物。SICP 比较适合作为第一本教材。但是我需要提醒的是，你最多只需要看完前三章。因为从第四章开始，作者开始实现一个 Scheme 解释器，但是作者的实现并不是最好的方式。你可以从别的地方更好的学到这些东西。不过也许你可以看完 SICP 第一章之后就可以开始看 TLS。

《A Gentle Introduction to Haskell》：对于 Haskell，我最开头看的是 A Gentle Introduction to Haskell，因为它特别短小。当时我已经会了 Scheme，所以不需要再学习基本的函数式语言的东西。我从这个文档学到的只不过是 Haskell 对于类型和模式匹配的概念。

过度到面向对象语言

那么如果从函数式语言入门，如何过渡到面向对象语言呢？毕竟大部分的公司用的是面向对象语言。如果你真的学会了函数式语言，就会发现面向对象语言已经易如反掌。函数式语言的设计比面向对象语言简单和强大很多，而且几乎所有的函数式语言教材（比如 SICP）都会教你如何实现一个面向对象系统。你会深刻的看到面向对象的本质以及它存在的问题，所以你会很容易的搞清楚怎么写面向对象的程序，并且会发现一些窍门来避开它们的局限。你会发现，即使在实际的工作中必须使用面向对象语言，也可以避免面向对象的思维方式，因为面向对象的思想带来的大部分是混乱和冗余。

深入本质和底层

那么是不是完全不需要学习底层呢？当然不是。但是一开头就学习底层硬件，就会被纷繁复杂的硬件设计蒙蔽头脑，看不清楚本质上简单的原

理。在学会高层的语言之后，可以进行“语义学”和“编译原理”的学习。

简言之，语义学 (semantics) 就是研究程序的符号表示如何对机器产生“意义”，通常语义学的学习包含 lambda calculus 和各种解释器的实现。编译原理 (compilation) 就是研究如何把高级语言翻译成低级的机器指令。编译原理其实包含了计算机的组成原理，比如二进制的构造和算术，处理器的结构，内存寻址等等。但是结合了语义学和编译原理来学习这些东西，会事半功倍。因为你会直观的看到为什么现在的计算机系统会设计成这个样子：为什么处理器里面有寄存器 (register)，为什么需要堆栈 (stack)，为什么需要堆 (heap)，它们的本质是什么。这些甚至是很多硬件设计者都不明白的问题，所以它们的硬件里经常含有一些没必要的东西。因为他们不理解语义，所以经常不明白他们的硬件到底需要哪些部件和指令。但是从高层语义来解释它们，就会揭示出它们的本质，从而可以让你明白如何设计出更加优雅和高效的硬件。

这就是为什么一些程序语言专家后来也开始设计硬件。比如 Haskell 的创始人之一 Lennart Augustsson 后来设计了 BlueSpec，一种高级的硬件描述语言，可以 100% 的合成 (synthesis) 为硬件电路。Scheme 也被广泛的使用在硬件设计中，比如 Motorola, Cisco 和曾经的 Transmeta，它们的芯片设计里面含有很多 Scheme 程序。

这基本上就是我对学习程序语言的初步建议。以后可能会就其中一些内容进行更加详细的阐述。

Origins of the Linux Filesystem(Ruan Yifeng)

举例来说，根目录下面有一个子目录/bin，用于存放二进制程序。但是，/usr 子目录下面还有/usr/bin，以及/usr/local/bin，也用于存放二进制程序；某些系统甚至还有/opt/bin。它们有何区别？

长久以来，我也感到很费解，不明白为什么这样设计。像大多数人一样，我只是根据《Unix 文件系统结构标准》（Filesystem Hierarchy Standard），死记硬背不同目录的区别。昨天，我读到了 Rob Landley 的简短解释，这才恍然大悟，原来 Unix 目录结构是历史造成的。

话说 1969 年，Ken Thompson 和 Dennis Ritchie 在小型机 PDP-7 上发明了 Unix。1971 年，他们将主机升级到了 PDP-11。当时，他们使用一种叫做 RK05 的储存盘，一盘的容量大约是 1.5MB。

没过多久，操作系统（根目录）变得越来越大，一块盘已经装不下了。

于是，他们加上了第二盘 RK05，并且规定第一块盘专门放系统程序，第二块盘专门放用户自己的程序，因此挂载的目录点取名为/usr。也就是说，根目录"/" 挂载在第一块盘，"/usr" 目录挂载在第二块盘。除此之外，两块盘的目录结构完全相同，第一块盘的目录（/bin, /sbin, /lib, /tmp...）都在/usr 目录下重新出现一次。后来，第二块盘也满了，他们只好又加了第三盘 RK05，挂载的目录点取名为/home，并且规定/usr 用于存放用户的程序，/home 用于存放用户的数据。

从此，这种目录结构就延续了下来。随着硬盘容量越来越大，各个目录的含义进一步得到明确。

- /：存放系统程序，也就是 At&t 开发的 Unix 程序。
- /usr：存放 Unix 系统商（比如 IBM 和 HP）开发的程序。
- /usr/local：存放用户自己安装的程序。
- /opt：在某些系统，用于存放第三方厂商开发的程序，所以取名为 option，意为" 选装"。

Unix Philosophy(Ruan Yifeng)

先讲两个很老的小故事。

第一个故事。有一家日本最大的化妆品公司，收到了用户的投诉。用户抱怨买来的肥皂盒是空的。这家公司为了防止再发生这样的事故，很辛苦地发明了一台 X 光检查器，能够透视每一个出货的肥皂盒。同样的事故，发生在一家小公司。他们的解决方法是买一台强力的工业电扇，对着肥皂盒猛吹，被吹走的就是空肥皂盒。

第二个故事。美国太空总署（NASA）发现在太空失重状态下，航天员无法用墨水笔写字。于是，他们花了大量经费，研发出了一种可以在失重状态下写字的太空笔。猜猜看，俄国人是怎么解决的？（答案在本文结尾处。）

==========

这几天，我在看 Unix，发现很多人在谈"Unix 哲学"，也就是开发 Unix 系统的指导思想。Wikipedia 上列出了好几个版本，不同的人有不同的总结。发明管道命令的 Doug McIlroy 总结了三条，而 Eric S. Raymond 则在 The Art of Unix Programming 一书中，一口气总结了 17 条（英文版，

中文版）。但是我发现，所有人都同意，" 简单原则"—-尽量用简单的方法解决问题 —-是"Unix 哲学" 的根本原则。这也就是著名的*KISS（keep it simple, stupid）*，意思是" 保持简单和笨拙"。

下面就是我对" 简单原则" 的笔记。如果你想最简单地完成一项编程任务，我认为可以从四个方面入手：

1. 清晰原则。

代码要写得尽量清晰，避免晦涩难懂。清晰的代码不容易崩溃，而且容易理解和维护。重视注释。不为了性能的一丁点提升，而大幅增加技术的复杂性，因为复杂的技术会使得日后的阅读和维护更加艰难。

2. 模块原则。

每个程序只做一件事，不要试图在单个程序中完成多个任务。在程序的内部，面向用户的界面（前端）应该与运算机制（后端）分离，因为前端的变化往往快于后端。

3. 组合原则。

不同的程序之间通过接口相连。接口之间用文本格式进行通信，因为文本格式是最容易处理、最通用的格式。这就意味着尽量不要使用二进制数据进行通信，不要把二进制内容作为输出和输入。

4. 优化原则。

在功能实现之前，不要考虑对它优化。最重要的是让一切先能够运行，其次才是效率。*" 先求运行，再求正确，最后求快。"（Make it run, then make it right, then make it fast.）*90% 的功能现在能实现，比 100% 的功能永远实现不了强。先做出原型，然后找出哪些功能不必实现，那些不用写的代码显然无需优化。目前，最强大的优化工具恐怕是 Delete 键。

==========

答案是，俄国人用铅笔。

函数式编程初探 (Ruan Yifeng)

No statement, only expression. No side effect. Referential Transparency. Suite for Concurrency(并发编程).

Linux 的启动流程 (Ruan Yifeng)

Daemon, Run level, /etc/inittab, "rc"(run command), /etc/rcN.d, /etc/passwd, ssh 登录, /etc/profile ~/.bash_profile ~/.bash_login ~/.profile, non-login shell ~/.bashrc

如果你要手动关闭或重启某个进程，直接到目录/etc/init.d 中寻找启动脚本即可。比如，我要重启 Apache 服务器，就运行下面的命令：

$ sudo /etc/init.d/apache2 restart。

Bash 的设置之所以如此繁琐，是由于历史原因造成的。早期的时候，计算机运行速度很慢，载入配置文件需要很长时间，Bash 的作者只好把配置文件分成了几个部分，阶段性载入。系统的通用设置放在/etc/profile，用户个人的、需要被所有子进程继承的设置放在.profile，不需要被继承的设置放在.bashrc。

回车和换行 (Ruan Yifeng)

今天，我总算搞清楚" 回车"（carriage return）和" 换行"（line feed）这两个概念的来历和区别了。在计算机还没有出现之前，有一种叫做电传打字机（Teletype Model 33）的玩意，每秒钟可以打 10 个字符。但是它有一个问题，就是打完一行换行的时候，要用去 0.2 秒，正好可以打两个字符。要是在这 0.2 秒里面，又有新的字符传过来，那么这个字符将丢失。于是，研制人员想了个办法解决这个问题，就是在每行后面加两个表示结束的字符。一个叫做" 回车"，告诉打字机把打印头定位在左边界；另一个叫做" 换行"，告诉打字机把纸向下移一行。这就是" 换行" 和" 回车" 的来历，从它们的英语名字上也可以看出一二。后来，计算机发明了，这两个概念也就被般到了计算机上。那时，存储器很贵，一些科学家认为在每行结尾加两个字符太浪费了，加一个就可以。于是，就出现了分歧。Unix 系统里，每行结尾只有"< 换行 >"，即"\n"；Windows 系统里面，每行结尾是"< 回车 >< 换行 >"，即"\r\n"；Mac 系统里，每行结尾是"< 回车 >"。一个直接后果是，Unix/Mac 系统下的文件在 Windows 里打开的话，所有文字会

变成一行；而 Windows 里的文件在 Unix/Mac 下打开的话，在每行的结尾可能会多出一个 ^M 符号。

整理下：

【由来】这 2 个概念最初来自打字机，引入 2 个特殊字符以填补从当前行尾到下行行首的操作时间空白，后来应用于计算机的行分隔符，不同 OS 处理不同：

- Windows 回车换行 \r\n
- Unix 换行 \n
- Mac 回车 \r

【原义】

- 回车横向操作 carriage return CR，这个名字可能是指打印头像运作起来像奔跑的马车
- 换行纵向操作 line feed LF, 被吃掉一行

5.30

dust jacket 书皮
spine 书脊（spine is loose）

5.31 Scheme IDE setup

guile, racket, mit scheme, chez scheme, common lisp, emacs lisp... So many things about lisp...

Scheme Setup

mit scheme

In terminal, enter mit-scheme or mit-scheme –edit

Emacs

M-x load-library –> xscheme(eg.)
M-x run-scheme
Or C-x 2, C-x o, M-x run-scheme

Racket

Fail to install the .sh file manually...

So I install the earlier version directly via apt-get in Ubuntu together with the DrRacket IDE and it seems good.

June

6.1

The real world doesn't have a rewind button.

6.2

Why lisp is widely used in AI field?

After some searching effort, it seems like it was.....

整型和浮点型

浮点型的精度: float 为 7 位，double 为 15 位

Apple 的 Swift 语言

“所有的人都是零基础？此言差矣。编程语言发展到现在，很大一部分工作都是库作为支撑的，你做的只是搭积木的工作，再加上逻辑判断、循环、递推、迭代等一部分逻辑。最多再加上并发、多线程等等等等，但后者也已经是充分“库″化。即便是 FP 也差不多是回调函数的另类表达版本。会 Objective-C 的肯定可以更好地掌握 Swift. 但是，会编译器、知道 LLVM 架构或者离散数学、JS 编程、库编程掌握得特别好信手拈来的人，肯定能更快拾起 SWIFT。所有的这些语言的 new features，早在多年前就有人提出的了，只是需要包装、整合和广告一下。新语言基本上只会对想要入门的人有区别。深入了解之后，区别远远没有那么大，因为不管语言

怎么变化，你想要达到的运行效果、想象力、可维护性才是核心。只要你是编程大牛，那么你就是编程大牛，跟语言无关。”

6.3

REPL:（Readevalprint loop）

6.7

workaround 变通

6.11 SCIP Book Received!

《西门庆和李瓶儿之间，算是爱情么？》——张佳玮

“爱情这个词，很大程度上，是被符号化了。谈此二字，必然与精神洁癖挂钩。开初必须得一见钟情，发展到情好意洽，自然而然；中间自然可以有肉欲成分，但必须是美好的；倘若有过利益算计、贪花好色、钩心斗角，这爱情就跌了档次。这种心态的后果之一，就是许多人都有初恋情结，仿佛初恋才是美好的，而非初恋就不干不净。宝黛算是爱情，贾琏和熙凤就明显差了一级。在这里面，爱情是分等级的：皮肤滥淫，就是比发乎情止乎礼要低。其实呢，有点儿变相的仪式化情结。或者说，抒情欲。

世上有种病，姑且叫做爱情仪式化强迫症好了。得这症候的人，假设他们在单身状态，一定会对友絮叨、对己自苦，话痨似的重复着他们多么渴望一份爱情。这份爱情久久得不到，于是他们会自嘲，会刻薄，会在 K 厅里一遍遍嚎同一首情歌。可是等他们有了伴侣，情况并不会变得更好。无论伴侣看似多么完美，他们都会情不自禁的念叨：自己曾经有过一个多么好的梦中情人，却如参商分隔，不能相聚；如今的感情当然也还不错啦，但是，当年的梦中情人才是最纯真最美好的时节……好了，他们这么絮叨着，终于分手了，于是他们又癫狂了。感叹失去的恋人是世上最美丽的对象，深悔自己当初是如何不懂的珍惜，满脸的蓦然回首幡然醒悟。最后，命运都烦透了，发给他们一个合适的伴侣，终于也稳定下来了，终于还有机会见到曾经的旧爱，倾诉当年的心曲，如此终于了无遗憾了吧？不。他们还是没法高兴。他们会感叹时光流逝带走了爱情，感叹旧爱已经被岁月

雕出了皱纹，觉得最美丽的时光都过去了，最纯真的爱情不复存在了——虽然他们也许都没有经历过像样的爱情故事。当然，不妨碍他们在此期间，伴随着各类情绪的抒发方式，比如情歌，比如爱情题材小说，比如写日记，比如做一些电视剧里的人会做的事。

当然，这种症候也有高级的表达形式。归有光《项脊轩志》末尾写“庭有枇杷树，吾妻死之年所手植也，今已亭亭如盖矣”，令人读之泫然，但事实是：第一任妻子魏氏在他三十岁那年过世后，归先生续娶了当地望族之女王氏。到四十二岁时，陪着新欢，悼悼旧爱，自己当然挺好，我们外人看去，多少有些感觉怪怪的。但归先生有才情，就过去了。大多数人，没这么好的文采，表现出来，就是愿意把一分爱说成十分，越苦恋越能满足他们的传奇欲望。所以现实生活中，经常有这样的人：他们会把爱情故事大肆夸张，每个细节都说得像电影——实际上，他们可能真是按照电影结构来改编自己爱情故事的，总觉得自己所经历的状态，离幻想出来的爱情还差许多。福楼拜写《包法利夫人》：爱玛明明拥有平凡温和的婚姻，却自觉婚姻不幸，把自己拟代成浪漫小说女主角，然后就忧世伤生起来。在一个爱情故事里，痛苦而悱恻的爱情比沉着而温暖的爱情更有传奇性，更悲剧，也更瑰丽。

如果一个人单纯沉湎于这份幻想中，当然无妨：各人有自己做梦的自由，在生活里扮演自己幻想出来的角色也是各人乐意。但比较吓人的是，这样的人如果和他人有了段爱情，通常会不小心伤害他人。在得到爱情之前，他们会自动代入，把追求对象当成公主，把自己想成骑士，竭尽所能的追求；而当爱情到手后，他们会心不在焉，继续幻想更符合自己心境的爱情故事。这里有一个奇妙的矛盾：感情和相处有技巧，是需要培养和训练的；但爱情仪式化强迫症总会认定爱情只有第一份才是真诚的，此后都不够纯净，类似于此。

世上的植物，并不都产在温室里。灵芝可以出自山崖，莲花可以出自污泥。没有一段爱情，可以这么恰好——命运也不是导演，不可能给每一段爱情，都设一个 MV 式的剧情，给你抹好妆容、打好灯光，让你来邂逅与一见钟情。实际上，大多数一见钟情只是凑巧，你会觉得那是命运的安排，也许只是因为那天的天气、你的心情、对面伴侣的容貌，恰好凑得好罢了。

老舍先生说过句话，大意是，穷人没法算计爱情，情种都生在大富之家。实际上可以换种表述，通常认可的情种，都生在书香门第，因为他们

有文化，善于修饰和营造美丽的爱情；但荷尔蒙冲动者、土豪、色狼、混球，其实也是有资格谈谈论爱情的。还是说回《金瓶梅》里，李瓶儿死后，西门庆嚎啕大哭，跳得有三尺高，叫的都是些“我的没救的姐姐，有仁义好性儿的姐姐！你怎的闪了我去了？宁可教我西门庆死了罢。我也不久活于世了，平白活着做甚么！”这些言行举止，如今看来，简直滑稽，还带丑角色彩；但设身处地，一个土豪出身如西门庆者，又不是文人雅士，势必无法出口成章。他也只能用这种狼狈的号哭，表达内心的情感。你也许无法同意，但如果去掉爱情仪式化的色彩，去掉西门庆和李瓶儿开始的偷情经历、金钱来往、皮肤滥淫，考虑他们的朝夕相处、天伦之乐，于他们而言，虽然开始得不那么光明正大，但后来就是寻常夫妻，就是爱情。

这话可能有些政治不正确，但爱情只是果，而非因。人们相爱，最初的因，可以千奇百怪：比如虚荣心，比如生理向慕，比如利益计算。但古话说日久生情，诚不我欺。时间长久、相处融洽之后，自然有血肉连心的爱情存在。你可以嫌这些爱情来路不正、不够纯洁、没有从一开始就走精神的高端的路线，但这毕竟是爱情。实际上，大多数泥淖里长出的、满带瑕疵的爱情，往往比浮空凌云的爱情，还要真诚一些——因为没有那么多的自我暗示和仪式感想像。”

此文读来有醍醐灌顶之感，我啊有时候就是自我感动过头了。

SCIP

The book arrived in good condition and I shall get ready to roll!

6.12 开阔的视野和心静的感觉

handy 方便的

Chrome Browser History

今晚偶然发现 Chrome 原来可以把不同 signed-in device 上的 history 汇总，真是很方便，让我惊喜。

心静放可读书

这是我之后一个长期需要做好的事情。“追二兔不得一兔”，有些事该淡淡地放，不然不可能专注起来。

全局之眼光

有了这个，方才能把事情理顺一步步完成；有了这个，方才能面对 distraction 而不惑。可以试多问问：Is this crystal clear for me?

6.15 马刺夺冠

cast 可理解为“阵容”

2014 NBA Champion ——San Antonio Spurs!

“马刺一直在变，自上而下，每个细节，只有一点是不变的：他们学习，而且改变，试图成为一个更好的团队。打好一场比赛是一个晚上的事，打好一个赛季是一整年的事，而保持十余年，不断学习、适应、默契与改变，对有些人来说太难，所以，马刺这个团队，也许只适合某一类人入驻。他们愿意把生命投进这么件事情里，领钱，打球，学习，和团队一起旅行，训练和比赛。马刺比赛里，最动人的部分是这样的：暂停结束，他们一个挨一个或坐或站在技术台边上，偶尔想起什么似的，讨论几句。有时邓肯和吉诺比利会摆开两手，连比带划跟队友们解释，解释完了，邓肯拍拍队友的头。你就知道，下一回合，马刺会打出一套很流畅的“挡、切、传、投”套路，行云流水。”

“我没想过有生之年，还能亲眼看见圣安东尼奥马刺，这些老家伙，回到总决赛。这里没想过的意思是，我已经不去想这件事，就像想一想这件事，也是给他们增加负担似的。但还有一层是：如果可以选择，“马刺这批人在一起多打五年”或者“他们拿个冠军然后退役”，我会选前一个。这里当然有点悖论：你打球当然是为了赢球，为了冠军。但这支团队如此难得，你会希望他们一直这样下去。当然，这世上没有人会不老，没有人能抵抗时间。如上所述，马刺也一直在变。但在偏执和宽泛、学习和守旧之间，你依然有机会选择，学习一些新东西，融入一种新生活，然后成为一支好的团队。”

我很珍惜他们。依然是那句话：马刺就像你看了许久的一部温馨家庭肥皂剧。你总不忍心看到结尾。你希望每个人长生不老，在其中来来去去。把职业体育胜利、利益、金钱的暴风关在门外。

比如，格林的父亲蹲过牢，莱纳德的父亲，邓肯的父亲，都早早过世了。而今天是父亲节。看着他们在板凳上抱成一团、彼此拍打，看见帕克在邓肯怀里小鸟依人时，你会觉得，这一切实在太美好了——你会觉得他们因为夺冠而快乐，比他们夺冠本身还要让人开心。

冠军颁完，是巴黎时间凌晨，天将放亮。我坐在楼梯角的一个蒲团上，哭了一小会儿，接了两个国际长途，听到电话对面也是不说话，只是哭。我明白这种感受。我在微信上跟于老师说 Now I can die in peace，于老师说 no,we。

魔术师 1991 年说他理解乔丹为什么夺冠后大哭，说他自己在 1980 年跟天勾拿新秀首冠时觉得冠军多轻松啊，直到 1987 年自己带队夺冠了才知道多艰难。对我而言，1999-2007，马刺夺冠不太容易，但也不难。甚至已经到了“只要鲨鱼不行了其他都好办”之感。但自那以来，七年之久。多少次希望已经丧尽，希尔和梅森来了又走，麦克戴斯被辜负，布莱尔和杰弗森让人燃起过希望又消失了。2012 和 2013 那两次简直像是最后的火焰，你不知道命运的窗是不是就这样关上了。

某种程度上，2008、2011 和 2014 这三个冠军，有其相似处。凯尔特人和小牛，KG、雷、皮尔斯、德克、基德们都已经被命运玩弄过许多次了，而马刺，上一次冠军已经太久，时代已经断层了。另一方面，这三支老头儿球队，都是完美的团队——他们都为了冠军牺牲了一切，以至于 2008 年和 2014 年，最后一场前，选总决赛 MVP 都那么困难。

上一次马刺夺冠时，我还不到 24 岁。从那以来，每过一年，我回头望，都更深一点明白当时马刺的不易，也明白现在的马刺多么值得珍惜。但这种感觉总是多少会出现：“如果能出现一支，真正让我们明白这一切多么可贵的冠军，多好啊。”（以上 By 张公子）

6.17 到达 Mountain View

6.18

googol means 10^100
so googolplex means 10^(10^100)

6.20

operator operand
percolate 滤，渗透
cascade 倾泻，小瀑布

July

7.9

快速掌握一个语言最常用的 50%

现在的开发工作要求我们能够快速掌握一门语言。一般来说应对这种挑战有两种态度：其一，粗粗看看语法，就撸起袖子开干，边查 Google 边学习；其二是花很多时间完整地把整个语言学习一遍，做到胸有成竹，然后再开始做实际工作。然而这两种方法都有弊病。第二种方法的问题当然很明显，不仅浪费了时间，偏离了目标，而且学习效率不高。因为没有实际问题驱动的语言学习通常是不牢固不深入的。有的人学着学着成了语言专家，反而忘了自己原本是要解决问题来的。第一种路子也有问题，在对于这种语言的脾气秉性还没有了解的情况下大刀阔斧地拼凑代码，写出来的东西肯定不入流。说穿新鞋走老路，新瓶装旧酒，那都是小问题，真正严重的是这样的程序员可以在短时间内堆积大量充满缺陷的垃圾代码。由于通常开发阶段的测试完备程度有限，这些垃圾代码往往能通过这个阶段，从而潜伏下来，在后期成为整个项目的毒瘤，反反复复让后来的维护者陷入西西弗斯困境。

实际上语言学习有一定规律可循，对于已经掌握一门语言的开发者来说，对于一般的语言，完全可以以最快的速度，在几天至一周之内掌握其最常用的 50%，而且保证路子基本正宗，没有出偏的弊病。其实真正写程序不怕完全不会，最怕一知半解的去攒解决方案。因为你完全不会，就自然会去认真查书学习，如果学习能力好的话，写出来的代码质量不会差。而一知半解，自己动手土法炼钢，那搞出来的基本上都是废铜烂铁。比如错误处理和序列化，很多人不去了解“正路子”，而是凭借自己的一知半解去攒野路子，这是最危险的。因此，即使时间再紧张，这些内容也是必须

首先完整了解一遍的。掌握这些内容之后进入实际开发，即使有问题，也基本不会伤及项目大体。而开发者本人则可以安步当车，慢慢在实践中提高自己。

以下列出一个学习提纲，主要针对的是有经验的人，初学者不合适。这个提纲只能用于一般的庸俗编程语言学习，目前在流行编程语言排行榜上排前 20 的基本上都是庸俗语言。如果你要学的是 LISP 之类非庸俗语言，或是某个软件中的二次开发语言，这里的建议未必合适。还是那句话，仅供参考。

1. 首先了解该语言的基本数据类型，基本语法和主要语言构造，主要数学运算符和 print 函数的使用，达到能够写谭浩强程序设计书课后数学习题的程度；

2. 其次掌握数组和其他集合类的使用，有基础的话可以理解一下泛型，如果理解不了也问题不大，后面可以补；

3. 简单字符串处理。所谓简单，就是 Regex 和 Parser 以下的内容，什么查找替换，截断去字串之类的。不过这个阶段有一个难点，就是字符编码问题。如果理解不了，可以先跳过，否则的话最好在这时候把这个问题搞定，免留后患；

4. 基本面向对象或者函数式编程的特征，无非是什么继承、多态、Lambda 函数之类的，如果有经验的话很快就明白了；

5. 异常、错误处理、断言、日志和调试支持，对单元测试的支持。你不一定要用 TDD，但是在这个时候应该掌握在这个语言里做 TDD 的基本技能；

6. 程序代码和可执行代码的组织机制，运行时模块加载、符号查找机制，这是初学时的一个难点，因为大部分书都不太注意介绍这个极为重要的内容；

7. 基本输入输出和文件处理，输入输出流类的组织，这通常是比较繁琐的一部分，可以提纲挈领学一下，搞清楚概念，用到的时候查就是了。到这个阶段可以写大部分控制台应用了；

8. 该语言如何进行 callback 方法调用，如何支持事件驱动编程模型。在现代编程环境下，这个问题是涉及开发思想的一个核心问题，几乎每种语言在这里都会用足功夫，.NET 的 delegate，Java 的 anonymous inner class，Java 7 的 closure，C++OX 的 tr1::function/bind，五花八门。如果能彻底理解这个问题，不但程序就不至于写得太走样，而且对该语言的设

计思路也能有比较好的认识；

9. 如果有必要，可在这时研究 regex 和 XML 处理问题，如无必要可跳过；

10. 序列化和反序列化，掌握一下缺省的机制就可以了；

11. 如果必要，可了解一下线程、并发和异步调用机制，主要是为了读懂别人的代码，如果自己要写这类代码，必须专门花时间严肃认真系统地学习，严禁半桶水上阵；

12. 动态编程，反射和元数据编程，数据和程序之间的相互转化机制，运行时编译和执行的机制，有抱负的开发者在这块可以多下些功夫，能够使你对语言的认识高出一个层面；

13. 如果有必要，可研究一下该语言对于泛型的支持，不必花太多时间，只要能使用现成的泛型集合和泛型函数就可以了，可在以后闲暇时抽时间系统学习。需要注意的是，泛型技术跟多线程技术一样，用不好就成为万恶之源，必须系统学习，谨慎使用，否则不如不学不用；

14. 如果还有时间，最好咨询一下有经验的人，看看这个语言较常用的特色 features 是什么，如果之前没学过，应当补一下。比如 Ruby 的 block interator, Java 的 dynamic proxy，C# 3 的 LINQ 和 extension method。没时间的话，我认为也可以边做边学，没有大问题。

15. 有必要的话，在工作的闲暇时间，可以着重考察两个问题，第一，这个语言有哪些惯用法和模式，第二，这个语言的编译/解释执行机制。

至此语言的基本部分就可以说掌握了，之后是做数据库、网络还是做图形，可以根据具体需求去搞，找相应的成熟框架或库，边做边学，加深理解。对于一个庸俗语言，我自己把上面的内容走一遍大概要花 2-3 周时间，不能算很快，但也耽误不了太多事情，毕竟不是每个月都学新语言。掌握了以上的内容，就给练武术打好了基本功，虽然不见得有多优秀，但是肯定是根正苗红，将来不必绕大弯子。就算是临时使用的语言，把上面这个提纲精简一下，只看蓝色重体字的部分，大致能在几天到一周内搞定，不算是太耗时，而且写出来的代码不会太不靠谱。

以上提纲未设及内存模型。对于 C/C++，这个问题很重要，要放在显著位置来考虑，但对于其他语言，这个问题被透明化了，除非你要做 hardcore 项目，否则不必太关注。

7.10 四张明信片一起收到！

tripe 牛肚，牛百叶
offal 杂肉

7.15 日本学生棒球宪章，搬家完毕

我们的棒球作为日本学生棒球，以学生自觉为基础。如果遗忘了教导学生的任务我们的棒球就无法成立。勤奋和纪律一直与我们同在，对于懒惰和放纵我们必须要在心里加强警戒。棒球作为运动本身就有它自身存在的意义和价值。但是，不要让它只停留在学生棒球上，要通过比赛，领会棒球光明磊落的精神。还要养成面对成功不骄傲，面对失败也不屈服的明朗强韧的精神。要锻炼出能应付任何艰难困苦的强健体魄。这正是引导我们的棒球理念。抱着这个理念，我们在此制定此宪章。

7.16 进化方向

搬家 Notes

- 新家真的是新开始，好好把握
- 书太多了，好好整理，可带回家的年底带回家，早些弄些哪些书到时候需要卖掉或送人，哪些一直会带在身旁
- 新家可以多做饭练练厨艺了：）
- 衣服的整理也好好想想，最简单的肯定需要弄一个衣柜类似物来

进化方向

- 找回安静读书思索的感觉
- Computing 方面，由 R 到 Python，同时通过 Scheme 持续不断的学习编程。其他需要 pick up 的有 SAS 和 SQL。培养好的搜索能力，利用好这个互联网时代的好资源。
- 想做一个好的 Hiker！

- 学习摄影，陶冶对美的欣赏和鉴别力
- 多折腾，多动手。比如编程、摄影、车等等。

7.17

capacity in which acting

7.18

caddy tray 托盘支架

A caddy tray review

Pros: -Works on my Toshiba S50 (with very slight modification). -No problems with connectivity and works as expected. -HDD activity LED

Cons: I personally have zero problems with the caddy itself (I love it) but some people might find the following troublesome... -Lack of proper instructions and OEM-like packaging. -Installation requires a thin precision screwdriver. -Some models may need very very slight modding (like mine; see below). -Drives on it are not boot-able (see below) -"Slow" (again, see below)

You'll see a lot of reviews of this caddy (Silverstone and other brands) in which people complain that the drives on it are slow and cannot be booted from.

-For starters, any drive on this caddy will only run on SATA revision 1 - 1.5 Gbit/s. I repeat, ***ONLY SATA rev. 1!!!*** This is a huge bottleneck for all SSDs and should not be used on such drives. Why only SATA 1 - 1.5 Gbit/s? It's because the port on your laptop's ODD was designed to, you know, run optical drives. SATA 3 will be unnecessary for such drives.

-The reason why it can't be booted from is that most laptop BIOS only recognize the ODD port specifically just for optical drives. Instead of displaying the secondary drive on my BIOS, it simply said "ODD - Not Present" (tested both FAT32 and NTFS drives). This may not apply to

laptops with unlocked BIOS. This isn't a major downside as you should only use this caddy for secondary drives.

-The best option is to put your SSD on your primary drive bay (which will almost always SATA 2 or SATA 3 nowadays) to leverage its speeds and put a secondary hard drive (preferably a 5400 RPM drive; 7200 RPM drives will run very hot as most laptops don't have ventilation on optical drive bays) on the caddy.

-As mentioned before, I did cut of a piece of plastic (there's two of them) to get my drive cover on the caddy. I only had to remove one but it seems like it is by design as the pieces were thinned out as if they were meant to be removed for some laptop models.

-One last note, make sure your laptop optical drive's height is ~9.5 mm as there's another version for older laptops which contain ODDs with the height of ~12.7 mm.

7.20 diskpart

windows 里的 administrative tools 里有很多好用的，比如 disk management

diskpart 里的 clean 命令可以用来清空整个 disk

Q: 如何在 windows 或 ubuntu 下给 ubunta 的 root 分区增加空间？

7.21 Install Laptop Memory

Bonanza: 财源

Q: 购买 Memory 时有哪些重要参数需要注意？

i7 models comparison(i7 4510U vs 4700MQ)

一些注意的参数：Number of Cores and Threads; Operating Frequency; On Chip L2 and L3 cache; Thermal Design Power; Sockets(Like for my Lenovo y410p's i7 4700MQ, it supports Socket G3, which can be used for CPU upgrade).

7.22

convertible pants

7.24 爱好自己，才能爱好别人

venue 演出场所，犯罪现场

gravita 气场

用了 Newegg 的 offer 后才发现有 Tigerdirect 更好的 offer。事前调查不够。。

短评《中国男人外表配不上中国女人》

真的不是相貌的问题，而是修养、气场和自信的问题。

我爸是清瘦严肃型的书生，我先生是阳光壮实的大男孩，两人的气质完全不是一个风格，而且五官长相都只能说是中等略微偏上，个子都不高（170 出头），但一点儿也不妨碍我在我爸教我品酒或者带我去博物馆对着展品侃侃而谈、抑或是我先生给我开车拎行李讲最新的科学发现的时候让我觉得他们都简直帅爆了。反之，每当在街上看到那些五官不难看衣服不便宜个子不矮但却散发着难以阻挡的猥琐气息的男人的时候我心里就会莫名地烦躁：TMD 你挺胸抬头收收你的啤酒肚，学会眼光直视别人再带上一个坦诚端正的笑容真的有这么难么？

我在国外每次上沟通/谈判技巧/演说课的时候教练都会提到一个叫做 gravita 的词，简而言之就是“气场”或者“吸引力”。我个人认为 gravita 对于男性和女性都很重要，但男性缺乏 gravita，后果更加严重：因为女性没有气场人家还能说个柔弱羞涩（虽然作为职场白领，女性主义者，我个人并不欣赏这类气质的女生），男性没有气场就直接被无视或者划入丑角一类了。所幸气场完全是可以后天训练培养的：保持恰当的身材（不一定要很健美）和衣着（不需要名贵），抬头挺胸，目光直视他人，用坚定温和的语调阐述观点，而且任何时候，都要保持对于独特的“自我”的自信，与外在的财富、地位和名誉无关。

爱好自己，才能爱好别人

想清楚了，我现在唯一值得好的事就是尽力提升自己，其他都是扯淡。完成这三年的承诺，该向前走就向前走。

7.25 佘头 is coming!

radiator 散热器；冷却器

For Surface Pro 3:

Had Microsoft made hardware of this quality ten years ago, Apple wouldn't have taken off.

7.27

graphite 石墨，碳纤维

7.29 长歌当哭笑红尘

Zoom in and out in Emacs

Ctrl-x Ctrl-+(or -)

长歌当哭笑红尘

记不清是上大学来第三次还是第四次看胡金铨老先生版的《笑傲江湖》。这一次彻彻底底地感动得一塌糊涂。尤其是当曲洋割断船缆，与挚友刘正风消逝在熊熊大火中的那一段，让人看着，亦只想对酒当歌不知东方之既白，甚至长歌当哭，泪涕并下。　　在这部片子前，叫卖中国风的某些质感男和把大红灯笼当内裤来炫的谋子们可以滚蛋了。嘲讽与苍凉并具，冷灰与热血交融，浑然天成，当之无愧的武侠电影巅峰。

在百度百科中找到有关《笑傲江湖》的一段文字，应该是金庸先生写的后记：　　“令狐冲当情意紧缠在岳灵珊身上之时，是不得自由的。只有到了青纱帐外的大路上，他和盈盈同处大车之中，对岳灵珊的痴情终于消失了，他才得到心灵上的解脱。本书结束时，盈盈伸手扣住令狐冲的手腕，叹道：“想不到我任盈盈竟也终身和一只大马猴锁在一起，再也不分开

了。”盈盈的爱情得到圆满，她是心满意足的，令狐冲的自由却又被锁住了。或许，只有在仪琳的片面爱情之中，他的个性才极少受到拘束。人生在世，充分圆满的自由根本是不能的。解脱一切欲望而得以大彻大悟，不是常人之所能。那些热衷于权力的人，受到心中权力欲的驱策，身不由己，去做许许多多违背自己良心的事，其实都是很可怜的。” 普希金说，没有幸福，只有平静和自由。我一直不懂。 滚滚红尘，勃勃野心，蝇营狗苟，谨小慎微，怎抵得过沧海一笑，醉里狂歌。 一觉醒，继续清醒，不觉晓。

7.31

Sirius 天狼星

August

8.4

Linux Learning

- Some of the common terms used in Linux are: Kernel, Distribution, Boot loader, Service, Filesystem, X Window system, desktop environment, and command line.
- A full Linux distribution consists of the kernel plus a number of other software tools for file-related operations, user management, and software package management.
- All Linux filesystem names are case-sensitive, so /boot, /Boot, and /BOOT represent three different directories (or folders). Many distributions distinguish between core utilities needed for proper system operation and other programs, and place the latter in directories under /usr (think "user").

Boot Process

1. Starting an x86-based Linux system involves a number of steps. When the computer is powered on, the Basic Input/Output System (BIOS) initializes the hardware, including the screen and keyboard, and tests the main memory. This process is also called POST (Power On Self Test). The BIOS software is stored on a ROM chip on the motherboard. After this, the remainder of the boot process is completely controlled by the operating system.

2. Once the POST is completed, the system control passes from the BIOS to the boot loader. The boot loader is usually stored on one of the hard disks in the system, either in the boot sector (for traditional BIOS/MBR systems) or the EFI partition (for more recent (Unified) Extensible Firmware Interface or EFI/UEFI systems). Up to this stage, the machine does not access any mass storage media. Thereafter, information on the date, time, and the most important peripherals are loaded from a small, battery-powered memory store called the CMOS values (after a technology used for the battery - which allows the system to keep track of the date and time even when it is powered off).

3. A number of boot loaders exist for Linux; the most common ones are GRUB (for GRand Unified Boot loader) and ISOLINUX (for booting from removable media). Most Linux boot loaders can present a user interface for choosing alternative options for booting Linux, and even other operating systems that might be installed. When booting Linux, the boot loader is responsible for loading the kernel image and the initial RAM disk (which contains some critical files and device drivers needed to start the system) into memory.

8.6

cordovan 科尔多瓦革，高级皮革
bearing 方位 (Use compass to set a bearing)

8.9 Summer and Winter Milky Way

Our home galaxy, the Milky Way is shaped like a disk, 100 thousands light-years in diameter. The Solar System in inside the disc but located at the rim of the Milky Way Galaxy, some 25,000 light-years off from the center. In northern hemisphere the summer nights is the time we look in the direction of the center of our Galaxy, and the brightest region of the

Milky Way is facing us (right). Then in winter we see the opposite direction and the Milky Way is only a dim patch of light (left).

秋月受伤

今天大家在 Acadia 里骑行，从 bubble pond 的 carriage road 出发一路南行往 Seal Harbor 而去，在穿过 Gatehouse 后不久，因道路状况突然变坏，LYF，冯龙，秋月三人先后遇险，其中秋月直接从 bike 上摔到地上，手上开了个大口子，鼻子和脸也有轻微的擦伤。站起来后还一度昏倒在她姐的怀里。我和 LYF 决定骑回扒车处把车开过来送她去医院（后来看这个决定不是最好的），留冯龙等人在原地应急。我们判断错了遇险的位置且以为那附近的路是单行，所以只好把车停到附近再由我骑车找到她们，等我找到冯龙他们前后大约已有 50 分钟，得知已有好心人叫了救护车，把秋月和她姐送到了 Bar Harbor 的医院。后来大家陆续赶到医院，在 ER 门外等了一个多小时秋月终于缝完针出来了。

经历今天的事，我有这么三点值得去想去做：

1. 如果今后再遇到类似的事情我可以怎么更好的应急呢？
2. 我该把扒车技术练的更好一些才行
3. Health Insurance 要办好，相关内容要多做了解

8.13

sane 理智正常的
exploitation 开发；利用；剥削
ethics 伦理，道德规范
zest 兴趣，热情；风味，滋味；风趣
indubitable 无可置疑的
preoccupation　偏见，成见
ennui 无聊，厌倦

8.15

羚羊挂角：羚羊夜宿，掛角於樹，腳不著地，以避禍患。舊時多比喻詩的意境超脫。

以意逆志：用自己的想法去揣度別人的心思。

“自我塑形，如張愛玲所謂“蒼涼的手勢”，美學風格之謂也。雄渾，蒼涼，沉郁，飄逸，綺麗，閑淡……。自我塑形與身份認同之關係不須論證。可言者，中國詩壇乃自我塑形之巨大場域，又因身份認同衍為種種類型。自我塑形關乎美學趣味，近乎文學生成之核心。雖然，若無轉身向上路，終當死於句下。陶淵明之平淡，李白之仙，王維之佛，杜甫之圣，庶幾乎證果。若太白，自我塑形之跡歷歷，至極則，乃謂“羅帷舒卷，似有人開。明月直入，無心可猜”。萬古長空，一朝風月，意會且不得，豈可妄加揣測？此可謂羚羊掛角。”——藤师

8.18 解决 Chrome 标签乱码问题

How to buff a car? Buff 有擦亮之意

Chrome Bookmark Issue Solved

系统缺少相应字体库支持，用 sudo apt-get install ttf-wqy-microhei ttf-wqy-zenhei xfonts-wqy 命令解决。

8.21

pervasive 普遍的，弥漫的，遍布的
perpetrator 行凶者，作恶者
consensual 一致同意的，同感的
defamation 诽谤，中伤
vent 表达，发泄
retaliation 报复，反击

8.22 BibTex

How to set up the bibtex style with \bibliographystyle{}?

What's diff between **prsty, abbrv, alpha, plain and unsrt?**

8.23 Git & GitHub

生命如此短暂，为什么总要将青春浪费在不断的选择之中呢？罚你，回头阅读心理学家施瓦茨（Barry Schwartz）的 TED 演讲：选择之困惑——为何多即是少，1 百遍啊 1 百遍。请记住施瓦茨的演讲要点：

更多的选择不代表更多的自由；更多的选择导致决策的延迟和降低的满意感；快乐之秘诀，在于降低自己的期望值。

How to use GitHub for beginners?

1. 每个项目作者一般都会在 README 文档中有一项 'contribute', 这里面会说明你应该怎样贡献代码或者其它东西. 另外, 为一个项目做贡献不一定要直接贡献代码才算, 可以检查项目的文档错误, 或者在对整个项目有了解的情况下, 给项目作者提 'feature request'

2. 用 github 来学习编程, 这个我确实还没有发现应该怎么好好利用, 我觉得最主要的原因就是这里面所有的代码都是可以直接获取到的, 而且带有代码作者的提交记录, 如果感兴趣的话, 你可以一个 commit 一个 commit 的查看作者写成整个项目的过程, 这个应该会对初学者有很大的帮助吧, 可能会对为什么整个项目是现在这个架构或形式有一定的了解.

我用 github 的时间也不是很长, 大约一年. 对 github 的认识就是, 它就是一个大宝库, 想要的任何东西几乎都可以在 github 上找到.

我使用 github 是从关注一些人开始的. 通过这些人的 star fork follow 活动, 我就可以知道一些我没有 follow 过的人, 然后我再去看这些我没有看过的人的项目, 然后我又发现了一些有趣的项目, 我觉得这个作者对我很有帮助, 所以我 follow 一下, 不断的通过这种 follow 项目作者的行为, 我知道了非常多的人. 同时也找到了非常多有意思的项目.

每天看这些人的 star fork follow 活动, 你真的可以知道现在的技术趋势是朝向什么方向发展的, 或者这一段时间什么项目最火.

如果对某个项目有兴趣的话, 可以直接 *clone* 下来, 看一看代码的整体是什么样的, 不一定完全看懂, 但是看多了, 就知道大牛们大体都是怎么写代码的. 对自己的好处不言而谕.

我使用 github 和别人协作的机会没有多少, 基本都是我自己在用, 存放自己的一些项目. 再一个作用就是存放自己的配置文件了, 到重装系统或者到其它的机器上, 直接 *clone* 一下, 自己的配置就全部回来了. 还有一个作用就是托管博客了, github 的 jekyll 很好用, 也很省心.

On Writing

In writing. Don't use adjectives which merely tell us how you want us to feel about the thing you are describing. I mean, instead of telling us a thing was "terrible," describe it so that we'll be terrified. Don't say it was "delightful"; make us say "delightful" when we've read the description. You see, all those words (horrifying, wonderful, hideous, exquisite) are only like saying to your readers, "Please will you do my job for me."

Letter to Joan Lancaster, 26 June 1956 – C.S. Lewis, Letters to Children

YiHui Xie's blog is interesting~

- 千秋邈矣独留我百战归来再读书
- 深情似海，问相逢初度，是何年纪？依约而今还记取，不是前生夙世。放学花前，题诗石上，春水园亭里。逢君一笑，人间无此欢喜。无奈苍狗看云，红羊数劫，惘惘休提起。客气渐多真气少，汩没心灵何已。千古声名，百年担负，事事违初意。心头阁住，儿时那种情味。

8.24 油断一秒，怪我一生

标题即是佳段子。

8.25

“极客不是一种身份，而是一种态度。在我眼里，这个词是中性的，极客不代表一个人有多牛，而是他的钻研态度、好奇心以及对新技术的识别和接受能力。有些很牛很聪明的人，未必能把聪明才智转化为生产力（请勿对号入座）。张丹这本书给大家提供了一条通向 R 的极客之路，但这绝对不是终点。技术人士容易沉迷于技术，就像科学研究人士迷信某一种科学一样，唉，我就是这样浑身负能量。希望读者通过这本书能感受到作者探索的乐趣，保持开放心态，积极学习，然后找到适合自己的极客理想（以及女朋友！相信我，后者会让前者更快实现）。写序似乎应该说点鼓励的话吧，我没写过也不清楚贵圈的规矩，那么就引用麦太的话好了：从前有一位小朋友他很努力学习，后来他发财了。“ ——谢益辉

8.26 R 如何进阶？

你有没有爱好，比如自然科学、社会科学等领域？

你有没有投资性需求，比如权益类投资，固定资产，一般财务性事务？

你有没有给他人有偿或无偿提供技术性支持的打算，比如工作环境、生活交际、兴趣团队？

确定一个需求，给定时间和可投入的精力目标，制定可能的技术部分计划和项目流程，在纸面上构思并能够通过一般的可行性评估后，就可以在前面的那个目标约束下展开此次探索工具软件之旅了（避免付诸实践才发现好高骛远）。前提是了解相关软件的基础。——体验通过恰当的工具，细致的计划，解决看似复杂的问题从而带来的乐趣和发现。

这个思路的优点在于，即使部分计划遇到瓶颈也可以绕过先去完成其他部分，最后积累一定经验后，在来解决曾经的技术瓶颈。

ps: 事物源自理解，而不应执着于知道。——网友留言一则

8.27 何谓美的程序

谈“测试驱动的开发”(王垠）

现在的很多公司，包括 Google 和我现在的公司 Coverity，都喜欢一种“测试驱动的开发”（test-driven development）。它的原理是，在写程序的

时候同时写上自动化的“单元测试”（unit test）。在代码修改之后，这些测试可以批量的被运行，这样就可以避免不应该出现的错误。

这不是一个坏主意。我在 Kent 的编译器课程上也使用了很多测试。它们在编译器的开发中是不可缺少的。编译器是一种极其精密的程序，微小的改动都可能带来重大的错误。所以编译器的项目一般都含有大量的测试。

然而测试的构建，应该是在程序主体已经成形的情况下才能进行。如果程序属于创造性的设计，主体并未成形，过早的加入测试反而会大幅度的降低开发效率。所以当我给 Google 开发 Python 静态分析的时候，我几乎没有使用任何测试。虽然组里的成员催我写测试，但是我却知道那只会降低我的开发效率，因为这个程序在几个星期的过程中，被我推翻重来了好几次。要是我一开头就写上测试，这些测试就会碍手碍脚，阻碍我大幅度的修改代码。

测试的另一个副作用是，它让很多人对测试有一种盲目的依赖心理。改了程序之后，把测试跑一遍没出错，就以为自己的代码是正确的。可是测试其实并不能保证代码的正确，即使完全“覆盖”了也是一样。覆盖只是说你的代码被测试碰到过了，可是它在什么条件下碰到的却没法判断。如果实际的条件跟测试时的条件不同，那么实际运行中仍然会出问题。测试的条件往往是“组合爆炸”的数量级，所以你不可能测试所有的情况。唯一能可靠的方法是使用严密的“逻辑推理”，证明它的正确。

当然我并不是让你用 ACL2 或者 Coq 这样的定理证明软件。虽然它们的逻辑非常严密，但是用它们来证明复杂的软件系统，需要顶尖的程序员和大量的时间。即使如此，由于理论的限制，程序的正确性有可能根本无法证明。所以我这里说的“逻辑推理”，只是局部的，人力的，基本的逻辑推理。

很多人写程序只是凭现象来判断，而不能精密的分析程序的逻辑，所以他们修改程序经常“治标不治本”。如果程序出问题了，他们的办法是看看哪里错了，也不怎么理解，就改一下让它不再出错，最多再把所有测试跑一遍。或者再加上一些新的测试，以保证这个地方下次不再出问题。

这种做法的结果是，程序里出现大量的“特殊情况”和“创可贴”。把一个“虫子”按下去，另一个虫子又冒出来。忙活来忙活去，最后仍然不能让程序满足“所有情况”。其实能够“满足所有情况”的程序，往往比能够“满足特殊情况”的程序简单很多。这是一个很奇怪的事情：能做的事越多，代码量却越少。也许这就叫做程序的“美”，它跟数学的“美”其实

是一回事。

美的程序不可能从修修补补中来。它必须完美的把握住事物的本质，否则就会有许许多多无法修补的特例。其实程序员跟画家差不多，画家如果一天到头蹲在家里，肯定什么好东西也画不出来。程序员也一样，蹲在家里面对电脑，其实很难写出什么好的代码。你必须出去观察事物，寻找“灵感”，而不只是写代码。在修改代码的时候，你必须用“心灵之眼”看见代码背后所表达的事物。这也是为什么很多高明的程序员不怎么用调试器（*debugger*）的原因。他们只是用眼睛看着代码，然后闭上眼，脑海里浮现出其中信息的流动，所以他们经常一动手就能改到正确的地方。

8.29 临别赠言

“仍记入学所会之诸友，君乃第一人也。为人平和，谈笑风生，宇间若有豪迈之气，其情犹新。于是一见如故，悠悠四载，置腹推心，更见君之器广量博，优雅从容，志存高远，又兼谈吐风趣，博闻广记，与人为善，诸同窗多受益颇丰，遂引以为良师益友。

时年宇内常有大事接踵，地动西南，金融危机，时局动荡，不一而足。由是同君论内外之事，继而思辨于人生、哲学，推而广之，亦不乏摩擦之火花与光芒。此情此景，如新茶熟酒，历历在目。

此回离校，君将赴海外，再见之日，遥遥无期。素知君有凌云志，然慎当谨记吾辈之缺陷，身体力行奋发向上以正其身，多方观察慎思祥辨以明其心，虽处其中然不迷，虽置其外而不茫，尽所能修为自身。世事莫难于提炼自身，吾人坚信，汝之道路上，绝无敌手。

言语及此，又生离别之情。今日一别，何日见兮；思君之情，不可胜兮。然信东流入海终有时，群雄聚首定有日，他日见君，“定不负相思意”！”

——桃子

“博学，审问，慎思，明辨，笃行。

凝练的十字，谱写出最为辉煌的人生。试问世之大贤，谁不如此？

最好的哲学，就是简约中思与行的精华。相识几年，虽然短暂，你却成了我最知心的朋友之一。

此行千里，你远涉重洋，我无佳言相赠，但望你不虚此行，以十字之哲，伴你左右，成就非凡的人生！

远在大洋彼岸的你，我时刻都惦记在心。
人生如棋，闲暇之时，我们共读藤泽秀行！
望有佳期，与君再聚！”
——佘头

8.30

windbreaker 防风衣
suntan 防晒

September

9.1

prowess 高超技艺
foil 挫败

9.2

aster 紫菀

9.4 Joel on software

extortion 勒索，敲诈
quarry 采石场；菱形；vt 费力地找

Do you have a bug database?

I don't care what you say. If you are developing code, even on a team of one, without an organized database listing all known bugs in the code, you are going to ship low quality code. Lots of programmers think they can hold the bug list in their heads. Nonsense. I can't remember more than two or three bugs at a time, and the next morning, or in the rush of shipping, they are forgotten. You absolutely have to keep track of bugs formally.

Bug databases can be complicated or simple. A minimal useful bug database must include the following data for every bug:

complete steps to reproduce the bug expected behavior observed (buggy) behavior who it's assigned to whether it has been fixed or not If the complexity of bug tracking software is the only thing stopping you from tracking your bugs, just make a simple 5 column table with these crucial fields and start using it.

For more on bug tracking, read Painless Bug Tracking.

Do you fix bugs before writing new code?

The very first version of Microsoft Word for Windows was considered a "death march" project. It took forever. It kept slipping. The whole team was working ridiculous hours, the project was delayed again, and again, and again, and the stress was incredible. When the dang thing finally shipped, years late, Microsoft sent the whole team off to Cancun for a vacation, then sat down for some serious soul-searching.

What they realized was that the project managers had been so insistent on keeping to the "schedule" that programmers simply rushed through the coding process, writing extremely bad code, because the bug fixing phase was not a part of the formal schedule. There was no attempt to keep the bug-count down. Quite the opposite. The story goes that one programmer, who had to write the code to calculate the height of a line of text, simply wrote "return 12;" and waited for the bug report to come in about how his function is not always correct. The schedule was merely a checklist of features waiting to be turned into bugs. In the post-mortem, this was referred to as "infinite defects methodology".

To correct the problem, Microsoft universally adopted something called a "zero defects methodology". Many of the programmers in the company giggled, since it sounded like management thought they could reduce the bug count by executive fiat. Actually, "zero defects" meant that at any given time, the highest priority is to eliminate bugs before writing any new code. Here's why.

In general, the longer you wait before fixing a bug, the costlier (in time and money) it is to fix.

For example, when you make a typo or syntax error that the compiler

catches, fixing it is basically trivial.

When you have a bug in your code that you see the first time you try to run it, you will be able to fix it in no time at all, because all the code is still fresh in your mind.

If you find a bug in some code that you wrote a few days ago, it will take you a while to hunt it down, but when you reread the code you wrote, you'll remember everything and you'll be able to fix the bug in a reasonable amount of time.

But if you find a bug in code that you wrote a few months ago, you'll probably have forgotten a lot of things about that code, and it's much harder to fix. By that time you may be fixing somebody else's code, and they may be in Aruba on vacation, in which case, fixing the bug is like science: you have to be slow, methodical, and meticulous, and you can't be sure how long it will take to discover the cure.

And if you find a bug in code that has already shipped, you're going to incur incredible expense getting it fixed.

That's one reason to fix bugs right away: because it takes less time. There's another reason, which relates to the fact that it's easier to predict how long it will take to write new code than to fix an existing bug. For example, if I asked you to predict how long it would take to write the code to sort a list, you could give me a pretty good estimate. But if I asked you how to predict how long it would take to fix that bug where your code doesn't work if Internet Explorer 5.5 is installed, you can't even guess, because you don't know (by definition) what's causing the bug. It could take 3 days to track it down, or it could take 2 minutes.

What this means is that if you have a schedule with a lot of bugs remaining to be fixed, the schedule is unreliable. But if you've fixed all the known bugs, and all that's left is new code, then your schedule will be stunningly more accurate.

Another great thing about keeping the bug count at zero is that you can respond much faster to competition. Some programmers think of this as keeping the product ready to ship at all times. Then if your competitor introduces a killer new feature that is stealing your customers, you can

implement just that feature and ship on the spot, without having to fix a large number of accumulated bugs.

9.5 SICP

Python switch for the MIT 6.001

"Costanza asked Sussman why MIT had switched away from Scheme for their introductory programming course, 6.001. This was a gem. He said that the reason that happened was because engineering in 1980 was not what it was in the mid-90s or in 2000. In 1980, good programmers spent a lot of time thinking, and then produced spare code that they thought should work. Code ran close to the metal, even Scheme —it was understandable all the way down. Like a resistor, where you could read the bands and know the power rating and the tolerance and the resistance and V=IR and that' s all there was to know. 6.001 had been conceived to teach engineers how to take small parts that they understood entirely and use simple techniques to compose them into larger things that do what you want.

But programming now isn' t so much like that, said Sussman. Nowadays you muck around with incomprehensible or nonexistent man pages for software you don' t know who wrote. You have to do basic science on your libraries to see how they work, trying out different inputs and seeing how the code reacts. This is a fundamentally different job, and it needed a different course.

So the good thing about the new 6.001 was that it was robot-centered —you had to program a little robot to move around. And robots are not like resistors, behaving according to ideal functions. Wheels slip, the environment changes, etc —you have to build in robustness to the system, in a different way than the one SICP discusses.

And why Python, then? Well, said Sussman, it probably just had a library already implemented for the robotics interface, that was all."

A comment

"I don't agree. Coming with heavy assembly, C, C++ background Scheme was a train crash for me. I think Scheme is a beautiful tiny language with 1 page language spec, yet powerful enough to do even OOP. If you teach it right it is the best language you can teach to an average Joe and makes him a "very good" programmer in any language he might encounter in the future. Within a few weeks your average Joe starts developing his own algorithms within 2 months he shuffles objects around, in 6 months starts writing his own compiler. This is way faster learning curve than C/C++. Yet once you master it you become a very good programmer, the rest is just about learning the sytax rules of any language they propose you. I still do C/C++ for living (mostly on realtime) but even now for mocking up a small algorithm I use Scheme. Python inherits a lot from Scheme except "the consistent" but a bit hard-on-the-eyes syntax of Scheme. Consistency is important for a starter because with his one page spec knowledge he can decode any expression he might encounter, without diving deep in to the thick language manual (e.g. ANSI C spec and any later C/C++ specs). One good thing on Python's account perhaps is that it has lots of open source support libraries so for anything which requires interaction with the "real world" (including GUI, Web, custom hardware) is easier. I think I would teach Scheme first to teach about programming, then switch to Python (which is relatively easy), for real world then C/C++/Java is just something you pickup when you need it."

9.6

A pencil set review

When I first started out doing art I didn't care so much about quality. I just wanted pencils that did the trick. Having more experience under my belt I see that quality does matter to a certain degree; and this kit fails to impress me in that area.

The pencils included are cheaply made, which makes them a pain to

sharpen. Too much energy is spent trying to sharpen them that you'll forget that you're working on a drawing.

The white eraser is a bit too gummy like, a few uses and I already feel the structure beginning to weaken...

The sharpener is OK. But there are better ones (See my list below after review)

The only thing I liked about this kit were the charcoal sticks. I like to sharpen mine with sandpaper and these do produce nice dark tones.

This product receives two stars because it really isn't the kind of quality you should be seeking out in your practices. In the end the flaws just add aggravation which can easily be remedied by buying a better quality.

If you want to REALLY get a good start at art (or good quality for continuing art), then I'd suggest the following:

(Pencils)

You can't go wrong with either Faber-castell or Staedtler. I myself use the Staedtler Mars Lumograph. You can find these in kits, containing various degrees (A good range for the beginner would generally be 2H, HB, 2B, 4B, 6B, 7B and 8B. Pencils ending in H are harder and thus produce lighter tones while pencils ending in B are softer and produce darker tones.) What will also work are the Ticonderoga pencils, which come in packs of 12.

(Erasers)

I recommend the Staedtler White eraser (which is fairly cheap. Mine cost me about 2$ for a fairly good-sized stick), but Pentel also works just as well. As for kneaded erasers, these are usually used to eraser thicker, heavier material like charcoal, but I recommend either Sanford or Prismacolor. Do not use any pink erasers for art, They leave a nasty pinkish residue on your paper and don't erase as good as the white (Infact, the only thing I ever use pink erasers for now is erasing stuff off of walls and desks.)

(Paper)

You'll see terms such as 15lb or 60lb. These indicate the thickness of the paper, with standard copy paper being 20lb and bristol pads coming in weights around 90 to 300 lb. Higher weights are for mediums such as

painting, charcoal, ink pencil, etc., Each paper also has its own texture, with some more coarse or more smooth than others.

Hot pressed paper means that it has a smooth texture which is ideal for drawing detail and drawing with graphite and pen.

Cold-pressed paper on the other hand has a coarse texture to it, which means that it is more absorbent and allows for layer build up, and so this is ideal for painting.

You'll want to choose texture based on which medium you'll be using. Acid-free paper is preferred as a high level of acid will tend to ruin your artwork with time. I'd advise not to buy expensive paper if you're just starting out and just stick with printer paper. It's cheap and you'll be making a lot of practice sketches to burn through your supply. Strathmore is a great choice. They make paper for all kinds of mediums.

(Sharpeners) You'll want one similar to the one included in this package, but bought from an art/ office supply store. The one included in the package was not as good as the metal one I picked up from Michael's for about two dollars. But if you REALLY want to get serious with it, I'd spend a bit more and get the KUM long-point sharpener. I use it almost exclusively and it is the best sharpener I've ever used. You can buy them here on Amazon or at any other art supply website (I've yet to see them sold in a store, unfortunately.) But it's a bit more expensive than your average sharpener (Mine was $10, shipping included). But it's well worth it, considering the fact that you save more wood and get to the point easier and faster. (Staedtler also sells a sharpener that is well-received here on Amazon. I've never tried it but it has glowing reviews)

Overall, don't get this kit; infact, don't get a "Kit," at all. Do a bit of research and get supplies separately. It's a bit more painstaking but in the end you're getting better quality products. You may save time and cash going for this product, but you'll be doing yourself, or that aspiring artist you know, a great disservice by giving them a poorly-made product.

9.9

plenary 全体的（会议）；完全的；绝对的

C and C++

C 语言属于结构化编程语言，其核心思想是将软件分解为一组数据描述与一组函数，因此用 C 语言写软件，归根到底就是写一系列的函数；C++ 属于面向对象语言，其核心思想是将软件分解为一组对象，通过对象间的交互来实现软件功能，并且还提供了模板等进一步抽象的手段，因此用 C++ 语言写软件，归根到底就是要写一系列的类来建模运行时对象的交互行为。

很多年过去了，尽管现在我也常常用 C++ 写程序，但我通常不建议初学者从 C++ 开始学习编程。我会建议他们先从 C 开始。C 非常简单，但是却非常强大，可以编写任何程序。重要的是，这会给你带来许多成就感。这非常重要，因为它能够激励你不断去探索，继续在程序设计的海洋里遨游下去。

一、小测试，你准备好学习 C 语言了吗？

- 你是否听说过二进制数，他们是如何进行运算的？（基本的数的进制知识）
- 你能说出一个最小的计算机系统由哪些部分构成吗？（基本的计算机结构知识）
- 计算机是如何存储图像的？（数字化原理）

上面这三个问题，能回答的同学举手。如果你举手了，那么我们进入下阶段的讨论，否则，我将告诉你，你现在最重要的事情，是赶紧找一本《计算机科学导论》把预备知识补充好。

因为——

C 语言说到底是一门以内存为中心的编程语言，你能不能学懂它，其实很大程度上不是取决于你智商高低，而是你是否拥有扎实的计算机结构、存储、运算原理方面的知识!!

在这里，我想强调，任何新知识的学习都是有一定的前提条件的。C 语言学习的前提条件就是，对计算机系统要有一个整体的，科学的基础认识。脱离了这个基础认识，一切都很困难。

所以，有些同学学不会 C 语言，不是因为他们智商不行，而是因为他们没有准备好。

我花时间写这篇文章来给大家讲 C 语言的学习，当然是希望大家真正的学会、学懂 C 语言，并能够真正感觉到它的用处，所以很抱歉我不会像培训机构那样，告诉你“零基础”就能够开始学。但如果你真的按照我说的去做，那么至少从现在起，你确实是开始有点“学院派”的认真了。

在这篇文章的最后，我推荐了一本我曾经读过的《计算机科学导论》。这本书可以说是我的启蒙书，我读过，真心觉得好，所以推荐给需要的同学。

好了，接下来我们进入 C 语言的学习过程。

二、学 C 语言到底学些什么——「语法」和「函数库」

C 语言学习的关键，是要先搞明白，学 C 语言到底是在学些什么？

我这么一问，有的同学就要翻开课本，指着目录说，我知道我知道，有变量，数据类型，循环语句，函数，哦哦哦，还有指针等等！

很遗憾，如果你以为学习 C 语言就是学习这些东西，那你得赶紧纠正一下看法，否则接下来的学习会困难重重。因为我曾经就是这么掉到坑里的，这种狭隘的观点让我浪费了大量的时间去学习琐碎的细节，又让我迟迟不能接触到更重要的知识，结果是眼高手低——知道许多别人不知道的无用知识，但是又写不出什么真正像样的程序来。

所谓「语法」，就是入门教材里最着重讲解的内容。也就是那些所谓的变量、数据类型、分支判断、循环、函数、指针等等。

这些内容比较枯燥，但是好消息是这些内容并不难，都是一些格式化的东西。只要你多练习，就会自动的刻在你的脑子里，成为一种下意识的习惯。

但是「语法」本身其实没什么用。因为它只是一种格式规范，你学得再好，也不能引导你写出厉害的程序。因为在软件设计中，实际上最核心的部分还是在于其「函数库」部分。

什么是「函数库」?

简单来说，函数库就是别人编写好的 C 函数，直接提供给你用，你只

要调用里面的函数，就能实现一定的功能。例如 printf() 函数，你肯定知道，只要调用这个函数，你就能够在那个黑糊糊的窗口里显示一段文字。你并不明白 printf() 的内部工作原理，但是你知道你只要按照说明去调用，就能够实现对应的功能。

这就是函数库——别人写好的，打包送到你面前，你可以自由调用来做各种各样的事情的函数集合。

我来说几个来自函数库的函数，例如 CreateWindowEx() 函数可以用来创建一个窗体（这个函数由微软提供），例如 GaussianBlur() 函数可以用来对一个图像进行高斯模糊处理（这个函数由 OpenCV 提供），再比如 evhttp_new() 函数可以创建一个 HTTP 服务程序（这个函数由 libevent 提供）……

其他还有什么库函数？太多了。从控制网络通信，到截取视频画面并分析其中的人脸位置，到加解密本地文件，甚至包括微信收发消息，抓取淘宝商品信息……库函数的数量和有用程度远远超乎大家的想象。

诶？你从来没听说过还有这些东西？课本里也没提到？

那是因为你看的是入门教材，着重讲解语法，顺带提到了少量的 C 语言自带的库函数而已。事实上 C 语言包含的库函数本身就有不少，但是更多更强大的还是许多第三方库函数，例如我上面提到的这些。

重点是在于，我想告诉你，库函数才是你学习 C 语言并将其应用于实际的关键!!

学会调用别人的库函数，甚至写出自己的库函数，都是极其重要的。因为一个函数，本质上就是一个功能单位。你拥有的基础设施越多，你的发挥空间越大。道理就是这么简单。这就是我需要向大家强调的第二个观点，要想写出实用的 C 程序，一大关键就是研究并学会使用各种库函数。

你看见隔壁王二写了个三维程序能让一个彩色的立方体在空中旋转？快找 OpenGL 库函数来用。什么？刘大宝写了个网络软件能在局域网里聊天？快找 Socket 库来用。

看见了吗？库函数有多重要？

但是，函数库的学习并不是孤立的。许多库函数背后需要一定的领域知识支撑。同样如我第一个观点所述，需要预备知识。学懂一个函数库，代表的不仅仅是明白如何调用那么简单，而更反映了我们对一个特定领域——网络、数字图像、密码学、操作系统等的认识。

三、基本的学习过程与一般规律

如我前面所述，C 语言学习主要是以语法入门，然后到函数库。再具体一些来说，包括以下阶段：

- 学习基础语法
- 学习简单的少数几个 C 语言自带的函数
- 学习一些程序设计的基础知识（数据结构，算法）
- 学习更多更强大的 C 语言自带的函数
- 学习一些特定应用领域的基本理论知识（操作系统，数据库，网络，图像……）
- 应用上一阶段学习的到知识进一步学习其他人提供的函数库（网络处理、操作系统管理、图像、密码学等等）

对于本科阶段的同学来说，着重学好 1-4，有选择的学习一下 5-6，做一些小作品出来，就已经非常非常不错了。这就是一般的学习规律。说得很简短，但是要做到真的很不容易。

9.11 Stack and Tail Call

desiderata: something eagerly needed
tinker 修补
paradigm 范式，范例
indefinite 无限的；不定的
denote 指代；预示
predicate 断言；谓语
accessor 存取器

Call Stack，Tail Call & Tail Call Elimination

9.12 Mac Shortcuts

Mac Shortcuts

In Finder

Cm+T Cm+N Cm+Q Cm+I(get information) Cm+M(minimize window) Cm+O(open selected item) Cm+Sht+U(open utility folder)

For Screenshot

Cm+Shf+3 and Cm+Sht+4

Navigate

Ct+f2(focus on menus bar)
Ct+f3(focus on dock)
Cm+alt+D show/hide dock

9.13

怎么成为一个优秀的程序员，而不是一个优秀的码农？(知乎）

优秀的程序员会告诉你打根基的重要性，会劝你在厚积薄发前要隐忍。优秀的码农会告诉你学啥底层、啥啥啥一拖就好了，学了 python 还要啥自行车啊，数据结构排序函数二分搜索这不都内置了吗？工作中永远用不到，学算法有啥用啊？成为高手有很多种方法汇编是个屁啊？

+++ 基础的分割线 +++

列举几个我认为比较重要的根基并附入门书编程语言，《程序设计语言 -实践之路》《concepts of programming languages》计算机通用知识，《csapp》算法、数据结构，《算法导论》程序设计、结构，没有书推荐软件工程，这个词大家理解不同，我以为，《人月》《代码大全》《the pragmatic programmer》《sicp》、讲测试讲重构的都是软件工程，其实上面设计模式也是软件工程，哈哈

这些书，初时读来感觉全无作用，而且要读多次才能体会其中意味，所以叫它根基也是十分合适，你根基越深才能爬得越高嘛。

+++ 方向的分割线 +++

啥是优秀程序员？记者和很多网民说熊猫烧香作者是高手公司里你出什么 bug 他都能告诉你原因用什么软件有问题他都能回答你的你就觉得是高手有人说徒手做产品的全栈才是高手各语言的作者都是高手有不写代码，扔出一个 restful 论文的还有人说高德纳是神，他如果是神，那他那些代码一定是在考验我们，嗯。。。。上面这些的确都算是高手，我琢磨着前两年被开掉的 moto 公司员工里肯定也有做功能机的高手和写廉价板驱动的高手

你想自己选自己的方向还是被人忽悠方向？我的想法是自己都尝试玩玩，然后做自己喜欢的方向。当然，程序员的生态金字塔是上面做工具、基础设施给下面人用来给普通人编程序，所以你选方向可以参考一下这个金字塔模型

+++ 爬坑的分割线 +++

方向定下，然后就是做事了，一大误区就是【追求最好的东西】，于是非得弄清楚：php 是最好的语言吗？OpenGL 比 directx 差吗？程序员要先学数学吗？最好的 c 语言书是谭浩强写的吗？放屁要先脱裤子吗？linux 发行版那么多该选哪个？某大牛说 IDE 不如编辑器听说黑客都是用记事本写程序的 C# 是升调记号应该读 csharp 而 java 不应念 [加 wa（轻声）] 。。。

如果你是一个 *nix 世界的玩家的话，你应该知道有一个 jargon 来上面的毛病，叫 yak-shaving，我以前提过几次 yak-shaving，但是很多人看不懂，它的字面意思是 Any seemingly pointless activity which is actually necessary to solve a problem which solves a problem which, several levels of recursion later, solves the real problem you're working on. 但一般都引申其意使用它，我这里举例一下：你本来要打开软件写一个 helloworld，软件提示你升级，你点了升级，提示你 xx 库不够新，然后你更新 xx 库，提示你要升级 yy 驱动，然后你升级 yy 驱动，系统告诉你要编译这个驱动，你必须下载 s.f 版本的编译器和库，你更新编译器，系统说 s.f 版本编译器必须在 e.n 系统上运行，然后你就升级系统了，几个小时过去，你发现系统升级导致了几个软件损坏，然后你更新那些软件，去找解决问题的方法，不知不觉到了半夜，你累成了狗，却发现问题还有一大堆，而 helloworld 也没写成。。。。

这些问题我都遇到过，我的建议是挑一本大牛说的书就是了，看会了

其他也会了。当然，如果你不幸不认识大牛（都上知乎了只要会搜索这种事情不可能发生），或者单纯好奇——就像我当年那样的话，那就每种都试试，不过有的答案你自己知道就好，像是编辑器 emacs 比 vim 更好，写程序 ide 比编辑器更好这种话，你是不应该乱说出来的。对了，像是不同范式的编程语言、不同的开发环境是应该尝试体验一下的，不过这种建议书上都写了，我这里说显得有些废话了。

所以说，不能被无关的东西弄偏了目标，要专注，坚持。等你学深了一门语言，就算是学另一们其他范式的语言也不会太难，你学会了 opengl，dx 也就看看就能写了。

你看看武侠小说里，段誉就是一个傻逼，仗着有时发出有时发不出的脉冲波和绕圈圈就能快跑直线的 bug 技就加上一门佛学能独步武林最终迎娶了白富美，出任了 CEO，走向了人生巅峰，乔峰永远是一招降龙十八掌，更夸张。而慕容复文武双才，基本精通天下武学，每天读书 4 时辰练功 4 时辰，论用功谁能和他比啊。。。可到了 30 多岁还是一事无成，pk 连段誉这种新手都搞不过，最后被人抢了老婆，就是因为方向不对，而且太不专注了。所以求多不如求专，深度到了，再花 20% 的时间去扩展一下广度即可。

+++ 重要的分割线 +++

以上东西你都做好了，要花个 2 年时间的样子，对于学生来说，如果你有一个好的学校背景，人生可以就此扬帆起步了但这不是终点，俗话说人靠衣装，美靠包装。包装是门学问，这里的包装不是让你西装领带亮皮鞋黑丝套裙白衬衫整一个营销狗 hr 的造型，是说专业技能上的包装。

为什么这是最重要的部分呢？因为别人一般不和你说这么多，尤其是懂得包装的人，更不会传你这些不传之秘啦。。。。。

包装自己的第一步是提高实力没有实力的包装那是空中楼阁，只能靠每天日常搞外包的忽悠架构大数据云计算过活，明眼人也能看出来，所以纯属作秀，没有意义。在某个领域（编译器、虚拟机、开发架构、前端。。。）成为专家（专家的定义嘛，，我的理解是能在简历里写精通）后，包装的实力就算具备了。

包装的第二步是定位提到美国会有一大堆人跳出来说是人类的希望民主的大救星，提到 google 就是最纯的技术公司不作恶、软件业的翘楚开发界的标杆、心美人美白莲花。。。哪怕你列举 google 卖假药、恶意打压 yelp、挟持 web 标准等等等等这些事，他们也会说百度更差（咦这不 5 角

钱常用的语句吗怎么被民主进步人士盗取了？。。。），可我根本没提到百度好吗。。。这全是包装造成的，所以包装的巨大威力，以此可窥之。

google 是搜索引擎，百度难道也说自己是搜索引擎？不，百度说自己是最懂中文的搜索引擎。。。哈哈哈，你别笑，这的确很管用，就像 google 说自己不作恶是好女孩一样。

程序员的包装定位，无非稀缺和独特这两点。物以稀为贵，稀缺就是要做到不可替代，这很好理解，比如你知道世界第一高峰是珠穆朗玛峰，如果没看过禅师精选集你很难知道第二高峰是乔戈里峰，但文青特别偏好乞力马扎罗山，不爬不跟你结婚，为啥？独特性嘛。。。我不跟你比高，我和你比文化底蕴，于是就赢了。

由于你有实力，所以你应该尽量把自己的实力包装成稀缺属性，你是专家嘛。。然后你实力多，应该把独特的实力包装出来，避免和他人共同曝光，以免被人压在身下。

我说一个我朋友包装自己的故事，他进公司接手了一个项目，已经是被隔壁组开发了 3，4 个月的一个软件，其实这软件 2 个月也能做好的，但是隔壁人忙而且也不上心，不熟悉这个技术，加上又不是自己的老板，所以做事拖沓了那么久才做好。他接手后一刻没闲，晚上带回家也做，做到半夜，10 天做成，然后整个公司的人都知道他的名字了，他也就立稳脚跟了。这个包装的主题是，技术实力强，开发速度快。

包装的第三步是推广推广就是让人知道你做得好，强化你的个人品牌，可以用博客、知乎、mailing-list、github 等，通过写文章、参加线下聚会演讲、回答问题、帮助他人等方法。这个用好了是门学问，用不好就是装逼，不展开了。。。记住不要匿名，匿名你基本啥也得不到

有人明明技术实力强，但是由于没有包装好，或被埋没了才能，或被贴上了各种不应该有的标签，或被人偏见看待。比如赵劼的 c# 和 vczh 的微软标签有人明明实力一般，但是善于鼓动小白，包装得好，所以有一批忠实粉丝，这种人我都不太敢说名字了，呵呵。。所以优秀的程序员应该善用包装啊

+++ 结尾的分割线 +++

上面说了那么多，但是坚持做来需要不少推动力，有人能考上清华，但是也能在大学堕落，有人能取得成绩，但是也会固步自封。。要想优秀，得有巨大的推动力，你为啥想成为优秀的程序员？你的推动力是什么？

好比，我们说，嫖娼是有巨大道德压力、金钱压力和风险的事情，为

啥知乎上那么多嫖客乐此不疲并努力给自己洗脑合理化这件事呢？因为在他们的眼中嫖娼是最有趣的事情、是不吃饭不睡觉也不能不做的事情、是不做就活着没劲的事，有了这种死也要死在床上的精神，还怕什么呢？

这就是推动力

如何评价电视剧《历史转折中的邓小平》？(知乎）

最后，大家千万不要太过沉迷这种“宏大叙事”性的东西，可能有时给你启发催你奋进，但不少时候仅仅是一剂兴奋剂，不能提高你的眼界，也不能增加你的学识，不能给你带来足够的经验和知识。真正关注那些有内涵、有实质的事情，为学求务实、做事求尽心，不天天沉迷一些虚幻的“宏大叙事”，做好自己的一份工。如有机会，你可能用才华和智慧做一个大梦，如果没有或是命运不甚公平，慢慢来，每天有所进益，往往就足够宽慰己心了。一代人有一代人的宿命和任务，慢慢来、不要停，你能更好的。切换到知乎频道，少抖机灵，多出走心的、务实的答案，比什么都强。

9.14 Game Over, Game Start

9.15 It turns out the game is not over, and might be in a new stage

9.22 变与不变

How to write function inside a function(like in R)? Any caveat?

9.23

amenable 易控制的

9.24 Diffusion direction in MRI(核磁共振成像）

vowel 元音

ensemble 全体，总效果

modulo 以……为模

- "Patience, Perseverance and Diligence always work."
- "It isn't that they can't see the solution. It is that they can't see the problem." (G. K. Chesterton)
- "Nothing would be done at all if a man waited till he could do it so well that no one could find fault with it." (J. H. Newman)
- " Common sense is the layer of prejudice laid down in the mind prior to the age of eighteen." (A. Einstein)

What is voxel ?

MRI

人体内含有非常丰富的水，不同的组织，水的含量也各不相同，如果能够探测到这些水的分布信息，就能够绘制出一幅比较完整的人体内部结构图像，核磁共振成像技术就是通过识别水分子中氢原子信号的分布来推测水分子在人体内的分布，进而探测人体内部结构的技术。

临床试验

`http://zh.wikipedia.org/wiki/%E8%87%A8%E5%BA%8A%E8%A9%A6%E9%A9%97`

实验组与对照组

对照组一方面可以更加有力地证明某一事物形成的原因是实验者假定的，而不是其他的。另一方面，对照组可以证明某一事物的作用确实是实验者假定的，而不是其他甚至相反的。

注意事项：

- 要尽可能消除无关变量，即让所有要形成对比的变量（称作“实验变量”）之外的变量都尽可能地减少。比如，在证明“吸烟会增大得肺癌的几率”的实验或者调查中，如果一个人群为吸烟的官员，另一个人群为不吸烟的核废料处理厂工人，那么这个实验显然是有问题的，

因为这增加了职业（准确的讲，为核辐射的受照剂量）这个重要的无关变量。

- 对比要鲜明，易于观察。
- 要考虑实验中的种种现实因素之制约，要具有可行性。

9.25 What is Structured Editor?

AST: abstract syntax tree

9.26 Vermont Plan

enviable 令人羡慕的
imputation 归罪，非难
gondola 狭长小船
maroon 褐红色

Programming via API

我建议在学习 c 语言的过程中，把系统 API 什么的也加进去一起学吧，教人写个带界面的程序比教人成天面对控制台程序，效果可能会好很多，因为这让人觉得学习 c 语言能做事情。

1. 我是在学会了 windows 编程的时候，才对编程产生了很强的兴趣的，说说我当时写的一些小程序，简单但比较好玩，适合培养编程兴趣（代码量依次增大）：点击，打开一个对话框，问你是猪不？然后告诉他，说不是的会自动关机，让他选吧。要是他点是，就嘲笑他，点不是，就关机吧。

2. 桌面下雪程序，在冬天大家都期待下雪的时候，做个简单的下雪程序，用到 windows 几个基本的 api 就行了，把这个程序发给大家，不懂的人会觉得非常神奇。我曾经发给几个 mm，她们貌似都非常喜欢。

3. 桌面贪吃蛇，就是用桌面图标玩贪吃蛇，当时听说的时候，觉得太有创意了，网上曾风靡一时啊，但是原理其实很简单，如果你把这个演示给你的同学看，他们会把你视为偶像的。

4. 然后呢，对 qq 有兴趣的，去看看早期的 qq 是怎么写的吧，我记得有一篇《qq 是怎样练成的》，还有源代码。改编一下，补充内容，甚至可以去参加学校的软件比赛去了，哈哈。

写几个好玩的程序以后，相信你就知道该如何学习编程了，要多给自己找乐趣。

9.28 浮星槎

往事北定中原日 序

公元 755 年，安史乱起，“渔阳鼙鼓动地来，惊破羽衣霓裳曲”。但是从后人的眼光看，渔阳鼙鼓，惊破得不仅仅是唐玄宗杨贵妃的美梦, 也不仅仅是大唐的国运。尘埃落地后，一去不返的，还有中国人的尚武精神。

安史之乱前八百年，平民百姓自愿参军，以图出人头地；安史之乱后八百年，当兵之人形同奴隶，几无出头之日；安史之乱前八百年，壮士投笔从戎；安史之乱后八百年，书生范进中举；安史之乱前八百年，李陵以五千步兵对抗匈奴单于数万主力骑兵，转战千里，杀伤过万；安史之乱后八百年，南明，大顺，大西百万军队在几万八旗军队前一溃万里；安史之乱前八百年，卫青霍去病横扫朔漠，封侯拜将；安史之乱后八百年，熊廷弼袁崇焕力抗后金，或被枭首或被凌迟、、、、安史之乱前后中国人的精神面貌是如此不同，几乎难以让人相信是同一个民族。　　渔阳鼙鼓，敲响的就是这种民族面貌剧烈转变的序曲。而宋朝是这个转变定型的关键时期。宋惩晚唐五代藩镇之祸，校枉过正，抑武崇文，武人地位每况愈下。两汉大将军（或大司马等等最高军事长官）往往凌驾宰相之上；唐朝的传统是出将入相；而宋朝最高军事首长－枢密使比宰相地位低，即使这样，也很少由武人担任；与之伴随的是社会上尚武精神的萎缩。一个例子就是，北宋官员还会骑马，不屑坐轿；而南宋的大部分官员不会骑马，只能坐轿。尚武精神的消失，在和平时期影响不大。但是，当外族入侵，整个民族在生死存亡间挣扎的时候，对于统治者来说，是重新唤起民族的尚武精神，恢复中原，廓清漠北？还是继续以文制武，奴颜婢膝，苟延残喘？

摆在南宋面前的，就是这样一个问题。南宋一百五十年间，从高宗到度宗，从秦桧到史弥远，从李纲到文天祥，从岳飞到张世杰，无数国贼巨蠹，仁人志士，都在如磐风雨中作出选择；从绍兴和议到鄂州议和，从岳飞北伐到端平入洛，诸多庙堂大计，战略构想，都在惊涛骇浪中付诸

实施。善与恶，美与丑，睿智与愚蠢，勇敢与怯懦，都在那个时代的金戈铁马声中凝聚，凸现，穿越时空，扑面而来，震撼每一个翻阅这段历史的人。 “长忆观潮，满郭人争江上望。 来疑江海尽成空，万面鼓声中。 弄潮儿向涛头立，手把红旗旗不湿。 别来几向梦中看，梦觉尚心寒”。 这首宋初诗人潘阆的「酒泉子」魔幻般得预言了这段历史给我的感觉。「王师北定中原日」这个系列，就是要讲叙这种感觉；讲叙这段历史中最惊心动魄的片断：徽宗宣和、钦宗靖康、高宗建炎年间的海上之盟，孝宗隆兴年间的隆兴和战，宁宗开禧年间的开禧北伐和理宗端平年间的端平入洛；讲叙这段历史中傲然立于时代涛头的英雄：李纲，宗泽，岳飞，马扩，孝宗，虞允文，李显忠，魏胜，毕在遇，孟珙，余玠、、、。 有人可能会问，这些有什么意义呢？当年拼得你死我活的契丹，女真和汉族，今天不都融为一体了吗？关于这个问题，金庸在「袁崇焕评传」中回答过。大意是英雄的功业不免随着时代褪色，但是英雄气概的绝代风华却永远不会泯灭，正象战国的纷争在今天已经没有多大意义，而荆柯，信陵君这些人物却超越了时空一样。透过历史的迷雾，瞻仰那些光彩夺目的英雄，正是我写这个系列的一个目的。但是，更主要的是，我希望通过这个系列，对我们的民族精神做些力所能及的反思，因为，那个时代的经验教训，同样适用于今天我们这个融合了五胡（鲜卑、匈奴、羌、氐、羯）、回纥、沙陀、契丹、女真、满族等等昔日不共戴天之仇的新的民族。 cosmicstring/浮星槎于公元 2002 年五月十五日 前言之后记 就在写这篇前言的时候，我突然产生了一个猜测，那就是，赵构在下定决心杀岳飞的时候，或许想到了刘裕。对于赵构来说，刘裕的存在，就足以构成除掉岳飞的理由了。刘裕象岳飞一样崛起于寒素，象岳飞一样战无不胜，象岳飞一样料机如神。他统帅北府精兵，灭慕容氏南燕，谯氏西蜀，姚氏后秦，把东晋疆域，推进到黄河北岸，在当时就已经被崔浩评为百年间天下第一。但是，英雄的结局截然不同。刘裕在北伐后建立刘宋，是为宋武帝。而岳飞，却在北伐后被杀。英雄际遇的天壤之别，让我对东晋南北朝和南宋的差别格外好奇。 东晋南朝的经济，人口远远比不上后来的南宋；五胡乱华开始时士大夫精神的萎靡，却远过南宋。然而，五胡乱起，汉民族的血性被奇迹般得激发。东晋的大臣，从祖逖，谢安到庾亮，桓温，无不枕戈待旦，伺机恢复，终于由刘裕把当时汉民族的战斗力发挥到了光辉的巅峰。其后南朝宋齐梁陈，不断北伐，或胜或败，不但象赵构那样卖国求荣的皇帝没有出现过，两宋朝的岁币在当时更是闻

所未闻。而南宋，有着无数光彩绝伦的英雄，有着无与伦比的经济，有着庞大的人口，有着灿烂的文化，为什么却始终不能再现刘裕的辉煌呢？这是个很有意义的话题。关于东晋南北朝，已经有几部很好的作品，如「参合陂－慕容垂」（作者 mesh），「元嘉三十年」（cinason）等等。但是，关于刘裕，关于桓温，触及的人不多，我计划在可预见的将来写写他们的故事。关于桓温的，就叫「师老瀛上」，关于刘裕的，就叫「气吞万里」。

到今天，我只完成了一篇「隆兴和战」，所以我在这儿说这么多，其实只是在描述一个梦想。可是我想，让这个梦想来打发悠悠岁月，也是人生一大快吧。　　cosmicstring/浮星槎于公元 2002 年五月十五日　　截至 2004 年二月，初稿完成的，是《隆兴和战》、《开禧北伐》、《何意百炼钢》，完成一半左右的是《师老瀛上》，正在写的是《何必桑干方是远》

大勇

所谓豪杰之士者，必有过人之节。人情有所不能忍者，匹夫见辱，拔剑而起，挺身而斗，此不足为勇也。天下有大勇者，卒然临之而不惊，无故加之而不怒。此其所挟持者甚大，而其志甚远也。

9.30 功不唐捐 & 殊途同归

回头看学生时代，最大的弯路就是怕走弯路、想不走弯路。纠结该学什么语言、该研究哪个方向、该做项目还是啃算法，生怕一失足成千古恨，踏上一条不归路。

很久之后才发现，与其纠结选择，不如找个点坚持下去。好比爬山，你在山脚下纠结该从哪条路上去，而实际上，每一条都能通往山顶，每一条都不会是笔直平坦的。你怕错过另一条路的风景踟蹰不前，却不知道只要登上山顶就可以一览众山小。

如果一定要说个经验教诲，那就是尽可能多地写代码、读源码、读文档。

有两个词，一个叫做功不唐捐，一个叫做殊途同归。

October

10.3 tex file compilation in different OS

Tex file compilation in Mac/Ubuntu/Windows

Try to produce a lecture slide which need to include the statrep package for generating SAS output. On Mac it's fine but on Ubuntu 14.04 and Windows 8 it failed. Need to find out what's going on here.

10.4 Arrived at Albany, going to see VT tomorrow!

How the leave change?

Science says chilly days and growing nights precipitated by the earth's angle relative to the sun, convince cells in leaf stem bases that it is time for a change. The cells begin to die. The dying cells form a wall preventing nutrients from reaching the leaf. As this happens the green pigmentation of leaves, chlorophyll, begins to break down unmasking the yellows and oranges that are present all year in the leaf.

10.10 白萝卜为什么辣

白萝卜为什么辣

主要因为芥子油，味道非常刺激，遇到高温就挥发所以含芥子油的植物在煮熟后就不辣了。

10.14 Optimal Bayes Rule

anneal: making harden steel soft and removing brittleness.
tilting at windmills: fighting imagary enemies
kale: 甘蓝菜

Q: When will a Bayes Rule be optimal and serve as a benchmark?

10.15 R 语言, Try RegexBuddy at sometime

overhaul 彻底检查
impeachment 弹劾，控告
parchment 羊皮纸，上等纸
flesh 肉，肉体；果肉

Successfully enlarge root disk space on Ubuntu via Gpart booting from USB

R 指南（知乎）

学 R 主要在于 5 点三阶段：

第一阶段有一点：

基础的文件操作 (read.*, write.*)、数据结构知识，认识什么是数据框 (data.frame)、列表 (list)、矩阵 (matrix)、向量 (vector)，如何提取 (包括 which, [] 等)、置换 (t, matrix 等)、删除 (-, which 等)、运算 (+, -, *, / , %%, %/% 等)、转换 (as.*)、修改 (edit, fix 等) 数据（包括单个数、行、列、表、变量），安装包、调用包以及 session 的保存。完成这一阶段，你就大致能像 excel 里处理数据一样了。

第二阶段有三点： 1、学习统计。这是贯穿整个 R 学习的最重要的一部，很多时候你并不是不知道在哪里找，怎么使用某个函数的参数，更多的时候你是不知道某个统计方法的原理，所代表的意义甚至不知道该用什么方法。所以学习统计学知识往往才是学习 R 的关键，之后找函数、怎么用其实都是傻瓜式的，并不需要你从头编写算法。这部分内容页要结合每个人要做的事做 2、批量处理。由于 R 和 matlab 一样，注重的是批量处

理，而且 R 之中的循环往往效率极低，所以在 R 之中如果你发现你要使用双层循环的时候，就要想想了，有没有批量处理的方法。a、首先，几乎所有的 R 里的运算符和自带的函数都是可以批量处理的。比如向量 a+ 向量 b 是指每个元素按照 index 相加，所以就没必要 for 一下了；b、其次，R 自带有的 apply 族函数 (因为是一系列以 apply 结尾的函数，所以称为 apply 族)，split，以及 aggregate 函数。这三类就是 R 自带的批量处理的利器，学好这三类函数，基本就可以完成绝大部分的数据批量处理了。c、然后就是 reshape2 包以及 plyr 包了，这是批量处理的两个利器，reshape 主要是整形，plyr 包基本提供了一套整理数据的理念，学好这两个包，批量处理将事半功倍。d、在实际过程中，一些 for 还是无法避免的。这时候就要考虑用别的语言来处理这部分事情了。比较常用的方法就是用别的语言批量生成 R 的代码，还有就是直接用 R 调用别的语言处理的结果或者用别的语言调用 R 的处理结果。3、绘图系统。总结而言，我们可以把 R 的绘图系统分成四个：Graphics，lattice，ggplot2 以及 grid。最好学习顺序也是按照这个来。a、自带的绘图系统。这套系统可以完成最基本的事情，其操作也类似于 matlab，可以看做是分步骤命令参数式绘图，基本就是将一系列作图看做一步步的命令，每一句都干一件事，然后通过参数调整其中的某个元素的大小、位置、颜色。b、lattice。绘图逻辑也同上。只是加了分组绘图、facet 的功能，这些都很实用，其目的就是讲自带函数中需要大量预处理以及多步绘图的命令用一行命令代替。上手也非常简单。c、ggplot2. 这是经典的 R 绘图包，绘图哲学是图层式的，理解成一个一个图层的覆盖。这个绘图系统能做很多事，而且其自带主题也相当漂亮。有一定的学习难度。以下就是我用 ggplot 画的图

d、grid。grid 绘图系统算是最基元的绘图命令，很多指令都是从画圆、直线、矩形开始的，这算是 R 里最好理解但也是最复杂的绘图系统。适合想入深坑的人士学习，如果要自由创造一些新的图形，或者编写绘图包，这是必学的绘图系统。另一个值得说的就是 grid 中也有专门用来整理拼图的指令，这个对于有一些论文拼图需求的人来说还是学学比较好。

如果你完成了以上两个阶段，你已经可以在工作学习中完成绝大部分的工作。但如果你是知识的创造者，或者是个程序员，或者是要实践自己的算法、理论、统计方法、绘图方法，或者亦或是你只是脑抽了，那就要进入第三阶段的学习。这部分包括，*C* 语言掌握与精通、*R* 语言调试、改进、编写包、写一个地道的帮助文档、推销自己的想法。这一阶段完成了，

你也就是一个 R 语言的大牛了。少年到处是你可以施展拳脚的地方。

最后是学习资源的问题，总结如下：

1、课程类。@uhuruqingcheng 已经介绍的 coursera 上的课程的确很适合入门。2、书籍类。建议入门用 R 语言实战 (豆瓣)，然后想要快速指南式的了解 R 语言的统计应用也可以看复杂数据统计方法 (豆瓣) 。统计学的内容按照自己的需要自行补充，在这里就补推荐了，推荐了也是和 R 没啥关系的。绘图系统推荐两本书就够用了：ggplot2 (豆瓣) 和 R Graphics Cookbook (豆瓣) 这两本也都有中文版可以买到。如果这部分都已经学得不错了，其实你就不需要书了，直接看 R 的帮助文档吧，help() 或者? XXX 都可以。在此建议用 RStudio，可以帮助你很快的查看帮助，编写 script、断点调试等等。需要补充的是 springer 出了一系列叫 Use R! - Springer 的书，一直有更新，也是免费获取的，大家可以自己下载。3、网站类。一个是博客，以上已有推荐。二是问答类的网站。Stack Overflow 和 SegmentFault 都可以尝试一下，不过后者有一种山寨感。三是 R 的 journal。题主可以自行在 google 里搜索 R journal 第一个就是。四是包和函数的搜索网站 Search all R packages and function manuals 。这个网站提供 R 里所有包和函数的搜索系统。其贴心之处还有 (1) 提供分类功能;(2) 提供下载的排名，每个包的下载时间线。

此外，*其实 R 语言是一门轻编程重统计的语言，所以题主完全不需要担心自己的编程基础。直接做几个小项目，你会很快上手，千万不要从教材第一页读到最后一页，那种效率极低，且容易半途而废。*

R 的优缺点（知乎私言）

就一个 R 和 SAS 精通程度几乎相同，在两种环境下都做过比较大（moderately big, ~xx Gb）的数据分析，与其他语言环境嵌合（SQL, Perl, etc...）使用过的个人（统计专业人员，非编程专业人员）的感受而言：

R 的优点：1. 免费... 开源... （这是最重要的一点好不好，也是 SAS 流行于公司，R 流行于研究机构和大学的最主要原因）2. 是专门为统计和数据分析开发的语言，各种功能和函数琳琅满目，其中成熟稳定的一抓一把 3. 语言简单易学。虽与 C 语言之类的程序设计语言已差别很大（比如语言结构相对松散，使用变量前不需明确正式定义变量类型等等），但仍保留了程序设计语言的基础逻辑与自然的语言风格。我这样说可能让人听得云里雾里，但是如果你对 SAS 或者 SPSS 有一点点了解，就会明白我的意

思了... 4. 小... 安装程序只有 50Mb 左右，比起某些死贵且 3 个 G 的付费软件真的是超级迷你小巧玲珑... 因为体积轻便，运行起来系统负担也小。5. 同各种 OS 的兼容性好。我两台本本一台 Windows，一台 Linux，都用得很顺手。相比之下，你有见过人在 Mac 上用 SAS 吗... 这人是要多么的想不开... =. = 6. 因为用的人越来越多，又是开源，有很多配套的“插件”为其锦上添花。比如 xtable 里有一个函数可以直接将 R 里的表格导出为TEX格式；另有 RStudio 的插件让你可以在同一个环境里写 TEX跑 R 并可在你的 TEX文件中插入你的 R 代码，多么的贤良淑德... （这个插件我没用过，不过我同学一天到晚在用）7. 有 R GUI 和 RStudio 两种风格供君选择，说实话我觉得这两种风格已经涵括了大多数人的使用偏好... 8. 已经提过了开源，还想再强调一下。各种包和函数的透明性极好，这使得对函数的调整和改良变得非常便利。只需要把源码调出来，自己稍微修改一下就可以了。这种事情放在任何其他统计软件里都近乎奢望。9. 如果你做 Bayesian，用 R 你有 OpenBUGS, WinBUGS, JAGS 等各种成熟活泼的包裹，很多语言又简单又附带各种预设的 plot，你只需调用即可；还可以自己写 MCMC。如果你用 SAS/SPSS/Stata，你可以... @@? = =bbb 10. 漂亮又灵活的图，大家也都已经讲过了。原本不是什么特别突出的长处（有则好，没也没啥），不过现在数据可视化越来越热，也就一跃成为主要优点了。

说说缺点：1. 对大文本（text data）处理极差... 或者说 data management 本就不是 R 的强项。*SAS 于 R 的最大优势之一可能就在于它兼顾了数据分析和数据管理。在 SAS 里对数据进行各种复杂操作都相对容易，只需要简单的 DATA STEP（必要时结合 PROC SQL）即可完成；在 R 里可就真的是千辛万苦... 虽然也有相应的 aggregate, merge 之类的函数，但是大都不太好用。这也是为什么大家常常把数据（尤其是数据大时）在别的环境下整好/分割好再喂给 R。*人家术业有专攻，数据管理真是有些难为它了。2. 内存管理和平行处理（parallel processing/programming）都为人诟病。数据小时没有感觉，数据大了就各种报错... =. = 3. package 的可靠性问题。我第一门完全使用 R 做作业的课是门统计课，教授已经六十多岁，见过各种统计软件的出生发展和湮没。*她同我们说到 R 时第一句话就是 Never use a package before you understand the manual and confirm the validity of the functions. 也就是包裹虽然好，使用需谨慎。主要原因还是在于开源。不常用的 package 一定要搞清楚函数的用法和核实过输出，不*

然真的不推荐使用。我个人也是倾向非常用函数尽量自己写，至少错了也容易 *debug*... 4. 不得不提的 package 的版本问题。就算你确认了包裹的可靠性并熟知了各个变量要怎么用，还是可能掉入潜在的陷阱=. = 讲个真事：去年工作的时候一个项目是使用 11 年某项目的一个贝叶斯模型分析新的数据。当年写代码的人因为相信末日说两年前就已经辞职环游世界去了，于是我只好独自研读他的代码。第一步，很自然的，就是重复当年的分析结果。这时发现当年他用的一个 package 和现在的 R 已经不兼容，于是就下载了这个 package 的最新版本。结果有一个简单的 credible interval 怎么都重复不出来... 怎么怎么都重复不出来... 我都快绝望了。最后经各种推理验证，发现这个区别源自于新旧版本的函数内部在对数据排序之后对 NaN 的不同处理... 而这个小小的修改未在任何地方留下任何文字记录。所以怎么说呢... 很多时候还是写自己的程序靠谱哇... 5. 当你跑比较大的 simulation，对效率有要求的时候，有时还是不得不用 C，这可能是 10 小时和 10 分钟的差别，毫不夸张。6. 想不出来乐 =w=

大致就是这样。

最后，因为不是学计算机出身，文中与编程语言和系统相关的措辞可能不准确或有误用，请程序员们多包涵 =)

R 包安装问题

无法在 ubuntu 14.04 上安装 devtools 包所需要的 Rcurl 包, 在 mac 上能成功安装，不过依然无法在 devtools 基础上安装 github 上的 slidify 包。

R 包推荐

分析与建模：

Matrix 包：先进的稀疏矩阵处理，不了解稀疏矩阵概念的时候内存占用和运行速度都不忍直视。

Reshape2/ddply：数据处理不用愁。*apply 系列：比 for 更好用的函数，其中 tapply 远不如 lapply 流行，但是实用程度不在其下。实际上 lapply 有没有变快得看各人的实现，因为虽然 lapply 调用了 C 实现，但是它还是要回头调用在 R 里用户定义的函数才能做计算，这个函数速度如何才是关键。

compiler 包：即使代码里有 for 也可以加速。

foreach：通用的并行接口，跨平台多功能。

lubridate：处理时间日期格式不求人。

gbm：效果和 randomForest 相近，但是占用内存很少很幸福，而且支持多核 CrossValidation 运算。stats::optim()：做优化的最傻瓜选择。不信看这个三行 R 代码做出 SVM：http://weibo.com/1459604443/A3x1VtIQn，不懂牛顿法也没关系。

报告与可视化：

knitr/slidify：knitr 是 @ 谢益辉的代表作。做报告、幻灯片 so easy，但是 slidify 的作者不喜欢写文档，所以很头疼。

shiny：用 R 生成 Web App，后端强劲接口统一。例如：https://hetong.shinyapps.io/imgsvd 。

recharts：在 R 中方便快捷地生成可交互图形，再也不用从 R 换到 js 了。

其他：

devtools::install_github()：脱离 CRAN 强权统治，Github 让世界更美好。

base::match()：很多情况下比 which，is.element 不知高到哪里去了。

utils::read.table()：设置 nrows 能提前分配内存，设置 comment.char="" 与 colClasses 更能加快读入。OpenBLAS 库：虽然不是 R 包，但是多核 CPU 上对矩阵运算的加速效果实在是太方便明显了，而且 Ubuntu 上安装方便，并不需要重新编译 R。定义启动项：如果对 stringsAsFactors 永远默认为 TRUE 深痛恶觉，可以修改 Rprofile.site 文件，加上每次启动都自动运行的命令。@ 任坤在评论中提到：定义启动项比较危险，不注意的话会使得代码的可移植性出现问题哦，放到别人电脑上一运行发现各种 factor。升级 R 包：R 的版本更迭之后，可以把老 R 包复制到新版本的 library 目录下，然后运行 update.packages(checkBuilt=TRUE, ask=FALSE) ，这是官方的提示，放在 FAQ 里，不知道有多少人留意了：R for Windows FAQ

任坤：

reshape2 横向、纵向做数据变换，例如把纵向堆叠在数据库中的证券行情数据转换成一个按照不同证券代码横向排列，按照时间纵向排列收盘价的数据表 stringr 方便地用正则表达式做批量字符串操作，可做检测、匹配、替换、计数等等 lubridate 方便地做日期/时间操作，各种标准化时间和时区的处理 plyr 轻松地在 vector, list, data.frame 之间做分组变换，实现拆分、变换、合并的操作 dplyr 轻松地处理 data.frame, data.table

以及多种数据库为基础的数据，实现选择、变换、分组等等，速度很快 RODBC 连接 ODBC 数据库接口 RSQLite 连接轻量级 SQLite 数据库连接 jsonlite 读写 json 文件 yaml 读写 yaml 文件，实现灵活的程序外部配置 Rcpp, Rcpp11 写 C++03/11 代码直接编译后给 R 调用，大幅提升算法性能 data.table 快速处理较大数据表 ggplot2 高级绘图，一套统一的语法实现复杂图像组合绘制 zoo 时间序列数据的预处理，比如滚动平均等等 rmarkdown 用 Markdown 写文档并可方便地运行 R 代码与绘图 knitr 自动文档生成 devtools 扩展包开发必备，在线安装托管的扩展包，检查扩展包是否符合 CRAN 标准等等 testthat 扩展包自动测试 pipeR 自己写的高性能、低损耗、分工明确的管道操作（pipeline operator）扩展包，使得数据变换流程化

Elrond 课程补记（剑气箫心）

去年参加了公司搞的数据分析师培训（Elrond），有些简单的收获和想法一直想记一下，本来也是该去年完成的，拖到现在算是补上。

另一种视角

一路学数学过来，更喜欢也更习惯的方式是解决“难”的问题。比如证明数学定理，或想出严格时空复杂度下的算法问题。但实际工作中，很多时候连方法是否有效都是件不容易验证的事，更不要说完美的解答了。所以不能总是期待有一个完备的定理结论或是一个漂亮的解析解在等着你。但是换一种思路，在充满不确定性的概率世界里，进行不完全信息下的决策反而是更常见的情形。也许这种实际情况下答案的不“标准”会让习惯了数学完美的人稍感失望和不顺手，但挑战性绝不亚于那些诸如 ACM 竞赛的试题。

既然是不完全信息，这个时候跳出“标准”答案的束缚，从多个视角来尽最大可能达到自己的目的就是件很考验人的事。课上讲的经济学视角、运筹视角，包括联想到之前 Nullpointer 在分析数据时借用热力学模型和语言学模型的例子，这些往往可以让陷入思维定式的自己柳暗花明。这时，数据也好，方法论也好，会更好地帮助你完成决策。

Make things happen

记得我那年校招专场时，胖子说过对算法工程师的几点要求，中间有一点就是“你是一个工程师，最终要 make things happen”。这次帮忙搭建最后竞赛用的应用对这点也挺有感触的。之前一直对前端的东西很排斥，也很无知，css 都不知道的那种。虽然原来写过几行简单的 webpy，但总觉得麻烦。这次从头开始照着 Bootstrap 和 DAE 的文档一点点搭一个小应用出来收获还不少。至少整个流程走下来，虽说东西土到掉渣自己都不忍直视，但好歹不像原来那样排斥了；而且现在也能写写 service，学学 Flask 什么的。

寒砚之前也说过，让别人理解你的想法，特别是让 PM 了解算法能做什么，公式的效果一般，而一个好的 demo 则很有说服力。虽说对最近很火的“全栈工程师”并不感冒，但多懂一些，多少可以避免因技术短板而盲人摸象。

再说训练

前几天看到有人转赵行德的一条豆瓣广播：“传播负能量：你缺的根本不是什么鼓励和赞扬，你那臭水平犯的低级错误根本没法通过鼓励和赞扬改正过来。你缺的是训练！严格的重复训练！可惜高中毕业之后，就没人花心思来训练你了，就更别提严格了。”

回想下自己真正掌握得好的那些东西，其实都少不了训练。本科学 *PDE* 时老师也说过，最后真正做问题的时候，能用起来的都是那些原来花了大量时间练习的东西；而那些只为应付考试而学的方法定理，你自己都想不起来。

这次最后的竞赛也是个例子。直接用工具会掩盖很多问题，有些收获还是要真正自己动手写代码实现才能得到的。至于那些结果比我好的，其背后尝试过的方法、思考的情况、得到的经验，可都比那简单的数值结果的差距多多了。

说到底，还是欠功夫。学上了那么多年，冰冻三尺非一日之寒的老道理反到快忘了。

阅读数学书（知乎）

个人认为，高效的阅读数学书的正确办法是去读一本最适合你当前水平的书。一个人阅读针对同一个主题的不同的数学书，效率（以及效果）可能有天壤之别。这是因为，这些书可能：1，对于同一个主题讲述的侧重点不同；2，对读者预备知识的要求不同；3，内容深浅，难易程度不同；4，写作风格不同。

事实上，我的导师，法国大学数学教授，也经常会遇到读不懂的文章或者看不懂的书。这是因为这些文献超出了他所具备的知识体系所能（容易）接纳的范围。学习任何知识都需要循序渐进。哪怕是天才，只要他不遵循这个原则就一定会事倍功半。

有的书，语言直观，通俗易懂，示例丰富，绕过繁琐的技术细节而重视思路，方法，原理，这种书就比较适合初学者；有的书比较抽象追求详尽，严密，以及一般化，这种书就适合进阶级的读者并且可作为常备的参考书；有的书意在介绍最新的学术成果，这类就只适合专业的研究人员。

所以，看书之前，要先弄清楚自己当前掌握哪些预备知识，以及希望通过学习达到的目标。选书的时候，可以先咨询一下相关资深人士的意见（网上有很多学霸介绍各种书的优缺点）；此外，任何书的序章里都会有介绍该书的目标人群。

本人经验之谈：选到一本适合自己的书等于成功了一半。

10.16

phonograph 留声机
sloppy 草率，懒散
slant 倾斜
binding 莫非就是绑定的来源？

Faces and Fonts

What is face and what's the relationship with these two concepts?

Emacs can display text in several different styles, called faces. Each face can specify various face attributes, such as the font, height, weight, slant, foreground and background color, and underlining or overlining. To

see what faces are currently defined, and what they look like, type M-x list-faces-display. *With a prefix argument, this prompts for a regular expression, and displays only faces with names matching that regular expression.*

It's possible for a given face to look different in different frames. For instance, some text terminals do not support all face attributes, particularly font, height, and width, and some support a limited range of colors. In addition, most Emacs faces are defined so that their attributes are different on light and dark frame backgrounds, for reasons of legibility. By default, Emacs automatically chooses which set of face attributes to display on each frame, based on the frame's current background color. However, you can override this by giving the variable frame-background-mode a non-nil value. A value of dark makes Emacs treat all frames as if they have a dark background, whereas a value of light makes it treat all frames as if they have a light background.

To specify colors for a face, you can either use a color name or an RGB triplet. To view a list of color names in emacs, type M-x list-colors-display.

To change the foreground and background color temporary, use M-x set-face-foreground(background).

Go to line & go to column

M-g c, M-g g, M-g Tab

10.17 Org mode is WONDERFUL!!!

a grain of salt 打折扣，不可尽信
blur sth out 使模糊

About Text Editors(From quora)

I had a CompSci mentor in 1980 who used to write all of his code on paper, then punched it in with punch cards. He had an amazingly efficient system and it was a marvel to watch how he kept snippets, code review

information, theoretical sidenotes, etc. all on small pieces of paper written in what had to be the most perfect tiny handwritten "font" I've ever seen. He was 10 times faster and his results were 10 times more accurate than anybody else I knew.

When people start talking about "which editor is better" and "how much more productive I am in X", I always think about that guy with his Bic accountant fine point pen, realising that people can become amazingly proficient with even the most archaic tools. *Our brains are marvels of adaptation and invention.*

So, if you are proficient in your editor, don't give the editor so much credit. You are probably the proficient one, and you have adapted to your editor, good or bad. Likewise, don't assume that another editor is better than another unless you have used both for long periods of time.... many years at least.

I like Emacs for the same reason I like the piano. My knowledge and learning will take me far. I used both 30 years ago. I will use both 30 years from now if I am still around. *Such reliable tools help you stop thinking about tools, and start creating the art or engineering that is your gift to the rest of us.*

Don't get distracted by shiny objects. Find good tools, invest your time in them, learn then and use them forever.

Virtualbox problem on Ubuntu

Cannot load SAS Univ Edition in Virtualbox on Ubuntu.

Remove virtualbox, I use sudo apt-get remove virtualbox-*, to remove global conf file, I use sudo apt-get purge virtualbox-*.

Some Emacs Key

C-q(means quote in emacs)M-=; C-x =; C-j; C-o(insert a line); C-x C-o(delete many consecutive lines); M-g g(go to line); M-g TAB(go to column); M-g c(go to char); C-v; M-v.

Ways to Repeat Emacs Command

Arguments: Data provided to a function or operation. We can give any emacs command a *numeric argument*.

- M-5 C-n(move down 5 lines); M-5 0 C-n(move down 50 lines). To insert five "0", use M-5 C-u 0.
- *C-u(universal-argument) alone has a special meaning "four times": it multiplies the argument for the next command by four. Example: C-u C-u C-f, C-u C-u C-n, C-u -5 C-n, C-u C-u C-o, C-u C-k. Comparison: C-u 6 4 a; C-u 6 4 C-u 1.*
- If the command you want to repeat prompts for input or use the numeric argument in another way, the previous methods won't work. Instead, try C-x z z z .....

Minibuffer

- C-a C-k kills entire argument
- Much like other emacs window, we can use C-x o to switch windows
- SPC can complete up to one word from the minibuffer text before point. Not available for arguments that often include spaces, like file names. ? is used to display a list of completions.
- M-p, M-n, M-r regexp, M-s regexp can be used to search Minibuffer history.
- C-x ESC ESC can repeat a recent minibuffer command from the command history.

Mark Region

- C- SPC or C-@ for setting the mark at point and activate it. C-x C-x, do the previous then move point where the mark used to be.

- M-@ set mark after end of next word, does not move point. C-M-@ set mark after end of following balanced expression, does not move point. M-h mark current paragraph. C-M-h mark current defun. C-x C-p mark page. C-x h mark buffer.

- C-x C-l, C-x C-u for convert case. C-u C-/ undo changes within region. M-% replace text within region. C-x TAB for indent region. M-$ for check spelling within region.

- C-SPC C-SPC for setting the mark, putting it onto ther mark ring without activating it. C-u C-SPC for moving point to where the mark *WAS*, and restore the mark from the ring of former marks.

Killing and Moving Text

- Note the difference between Killing and Deleting in Emacs.

- M-\ delete spaces and tabs around point. M-SPC does similar but leaves one space. C-x C-o delete blank lines around current line. M-^ join two lines by deleting the intervening newline, along with indentation following it.

- C-S-backspace for killing an entire line at once, including newline.

- C-w kill region, M-w copy region into kill ring, M-d kill next word, M-DEL kill one word backwards, C-x DEL kill back to beginning of sentence, M-k kill to the end of sentence, C-M-k kill the following balanced expression, *M-z char* kill through the next occurrence of *char*.

- C-y yank the last kill into buffer, at point. M-y replace the text just yanked with an earlier batch of killed text. C-y leave the cursor at the end of the inserted text, sets the mark at the beginning, C-u C-y does the reverse. C-u 4 C-y yank the fourth most recent kill.

Multple Cursor in emacs/sublime text 2

First install the multiple-cursor mode, then use C-> and C-< (etc) to add cursors. In ST2, use Com+D to choose the next ocurrence.

Revisiting Org mode

This time I find it's really easy to export an org file into HTML, LATEX and PDF format, which makes life wonderful!

10.18 Creating symbolic link is useful

peripheral 周边，外围设备
opt 选择，挑选 option 的来源

Setting up scheme on Mac

To use in terminal:

sudo ln -s /Applications/MIT\:GNU\ Scheme.app/Contents/Resources /usr/local/lib/mit-scheme-x86-64

sudo ln -s /usr/local/lib/mit-scheme-x86-64/mit-scheme /usr/bin/scheme

To use in emacs:

set up via M-x customize-group scheme

Enable visiting Racket in terminal on Mac(Essense like the above)

sudo ln -s /Applications/Racket\ v6.1/bin /usr/local/lib/racket-x86-64

sudo ln -s /usr/local/lib/racket-x86-64/racket /usr/bin/racket

Then install the *racket-mode* in emacs then I can run racket interpreter inside emacs!

Linux File System

"On a UNIX system, everything is a file; if something is not a file, it is a process."

This statement is true because there are special files that are more than just files (named pipes and sockets, for instance), but to keep things simple, saying that everything is a file is an acceptable generalization. A Linux system, just like UNIX, makes no difference between a file and a directory, since a directory is just a file containing names of other files. Programs, services, texts, images, and so forth, are all files. Input and output devices, and generally all devices, are considered to be files, according to the system.

In order to manage all those files in an orderly fashion, man likes to think of them in an ordered tree-like structure on the hard disk, as we know from MS-DOS (Disk Operating System) for instance. The large branches contain more branches, and the branches at the end contain the tree's leaves or normal files. For now we will use this image of the tree, but we will find out later why this is not a fully accurate image.

Partitions and Mounting point

All partitions are attached to the system via a mount point. The mount point defines the place of a particular data set in the file system. Usually, all partitions are connected through the root partition. On this partition, which is indicated with the slash (/), directories are created. These empty directories will be the starting point of the partitions that are attached to them.

Some commands: mount, df, ls -i(means inode)

File System in Reality

For most users and for most common system administration tasks, it is enough to accept that files and directories are ordered in a tree-like structure. The computer, however, doesn't understand a thing about trees or tree-structures.

Every partition has its own file system. By imagining all those file

systems together, we can form an idea of the tree-structure of the entire system, but it is not as simple as that. In a file system, a file is represented by an inode, a kind of serial number containing information about the actual data that makes up the file: to whom this file belongs, and where is it located on the hard disk.

Every partition has its own set of inodes; throughout a system with multiple partitions, files with the same inode number can exist.

Each inode describes a data structure on the hard disk, storing the properties of a file, including the physical location of the file data. When a hard disk is initialized to accept data storage, usually during the initial system installation process or when adding extra disks to an existing system, a fixed number of inodes per partition is created. This number will be the maximum amount of files, of all types (including directories, special files, links etc.) that can exist at the same time on the partition. We typically count on having 1 inode per 2 to 8 kilobytes of storage.

At the time a new file is created, it gets a free inode. In that inode is the following information:

- Owner and group owner of the file.
- File type (regular, directory, ...)
- Permissions on the file Section 3.4.1
- Date and time of creation, last read and change.
- Date and time this information has been changed in the inode.
- Number of links to this file (see later in this chapter).
- File size
- An address defining the actual location of the file data.

The only information not included in an inode, is the file name and directory. These are stored in the special directory files. By comparing file names and inode numbers, the system can make up a tree-structure that the user understands. Users can display inode numbers using the -i option to ls. The inodes have their own separate space on the disk.

The PATH

When you want the system to execute a command, you almost never have to give the full path to that command. For example, we know that the ls command is in the /bin directory (check with which -a ls), yet we don't have to enter the command /bin/ls for the computer to list the content of the current directory.

The PATH environment variable takes care of this. This variable lists those directories in the system where executable files can be found, and thus saves the user a lot of typing and memorizing locations of commands. So the path naturally contains a lot of directories containing bin somewhere in their names, as the user below demonstrates. The echo command is used to display the content ("$") of the variable PATH:

echo $PATH

/opt/local/bin:/usr/X11R6/bin:/usr/bin:/usr/sbin/:/bin

In this example, the directories /opt/local/bin, /usr/X11R6/bin, /usr/bin, /usr/sbin and /bin are subsequently searched for the required program. *As soon as a match is found, the search is stopped, even if not every directory in the path has been searched. This can lead to strange situations.* In the first example below, the user knows there is a program called sendsms to send an SMS message, and another user on the same system can use it, but she can't. The difference is in the configuration of the PATH variable

Absolute and Relative Paths

A path, which is the way you need to follow in the tree structure to reach a given file, can be described as starting from the trunk of the tree (the / or root directory). In that case, the path starts with a slash and is called an absolute path, since there can be no mistake: only one file on the system can comply.

In the other case, the path doesn't start with a slash and confusion is possible between ~/bin/wc (in the user's home directory) and bin/wc in /usr, from the previous example. Paths that don't start with a slash are always relative.

In relative paths we also use the . and .. indications for the current and the parent directory. A couple of practical examples:

- When you want to compile source code, the installation documentation often instructs you to run the command ./configure, which runs the configure program located in the current directory (that came with the new code), as opposed to running another configure program elsewhere on the system.
- In HTML files, relative paths are often used to make a set of pages easily movable to another place:

<img alt="Garden with trees" src="../images/garden.jpg">

Note

The File system Chapter in Garrels' book *Intro to Linux* is really good!

What is a shell?

When I was looking for an appropriate explanation on the concept of a shell, it gave me more trouble than I expected. All kinds of definitions are available, ranging from the simple comparison that "the shell is the steering wheel of the car", to the vague definition in the Bash manual which says that "bash is an sh-compatible command language interpreter," or an even more obscure expression, "a shell manages the interaction between the system and its users". A shell is much more than that.

A shell can best be compared with a way of talking to the computer, a language. Most users do know that other language, the point-and-click language of the desktop. But in that language the computer is leading the conversation, while the user has the passive role of picking tasks from the ones presented. *It is very difficult for a programmer to include all options and possible uses of a command in the GUI-format. Thus, GUIs are almost always less capable than the command or commands that form the backend.*

The shell, on the other hand, is an advanced way of communicating with the system, because it allows for two-way conversation and taking

initiative. Both partners in the communication are equal, so new ideas can be tested. The shell allows the user to handle a system in a very flexible way. An additional asset is that the shell allows for task automation.

Finding Files

Example:

- ls dirname/*/*/*[2-3]
- find . -size +5000k

Useful Commands:

- which
- find <path> -name <searchstring>; This can be interpreted as "Look in all files and subdirectories contained in a given path, and print the names of the files containing the search string in their name" (not in their content).
- find . -name "*.tmp" -exec rm {} \ ; It is best to first test without the -exec option that the correct files are selected. This command will call on rm as many times as a file answering the requirements is found. In the worst case, this might be thousands or millions of times. This is quite a load on your system. *A more realistic way of working would be the use of a pipe (|) and the xargs tool with rm as an argument. This way, the rm command is only called when the command line is full, instead of for every file.*
- tail -10 .bash_history

Notes:

- locate is a symbolic link to the slocate program in some systems, check via: ls -l /usr/bin/locate
- Characters that have a special meaning to the shell have to be escaped. The escape character in Bash is backslash, as in most shells.

Links

- Hard link: Associate two or more file names with the same inode. Hard links share the same data blocks on the hard disk, while they continue to behave as independent files. There is an immediate disadvantage: hard links can't span partitions, because inode numbers are only unique within a given partition.
- Soft link or symbolic link (or for short: symlink): a small file that is a pointer to another file. A symbolic link contains the path to the target file instead of a physical location on the hard disk. Since inodes are not used in this system, soft links can span across partitions.
- Creating Symbolic link: ln -s targetfile linkname
- Symbolic links are always very small files, *while hard links have the same size as the original file*(? Why).
- The application of symbolic links is widespread. They are often used to save disk space, to make a copy of a file in order to satisfy installation requirements of a new program that expects the file to be in another location, they are used to fix scripts that suddenly have to run in a new environment and can generally save a lot of work. A system admin may decide to move the home directories of the users to a new location, disk2 for instance, but if he wants everything to work like before, like the /etc/passwd file, with a minimum of effort he will create a symlink from /home to the new location /disk2/home.

File Security

- First understand Access rights. Use chmod to change it, eg: chmod u+rwx,go-rwx hello
- id for basic user info, newgrp for accessing another group, umask for setting default file permission, chown is similar

10.19 不记则思不起；What is abstraction barrier?

perspicuous: 明白易懂的
tenuous: 薄的，细的

Peter Norvig and John McCarthy

Lisp 最最最优美的特性是什么？弱类型？FP？GC？交互式 REPL？NO！Lisp 最重要的特性便是“代码即数据”。代码可以生成代码，编译时和运行时并没有绝对的区分，这才是 Lisp 的杀手锏。作为历史上第二个高级语言，Lisp 的特性被后来的很多语言所借鉴。Java 和 Python 学艺不精，前者只学得 GC，后者虽多了一个 REPL，但依旧不得其要领。Ruby 甚好，所说也是基于 C 系的语法，但是已经深得 Lisp 要领了。Ruby 的 metaprogramming 能力之强大，直接催生了 RoR 的诞生，风靡全球。那么回到 Lisp，曾经有过这么一个故事：

在 ILC 2002 大会上前 Lisp 大神，当今的 Python 倡导者 Peter Norvig，由于某些原因，做一个类似于马丁路德在梵蒂冈宣扬新教的主题演讲，因为他在演讲中大胆地声称 Python 就是一种 Lisp。

讲完后进入提问环节，出乎我意料的是，Peter 点了我过道另一侧，靠上面几排座位的一个老头，他衣着皱褶，在演讲刚开始的时候踱步进来，然后就靠在了那个座位上面。

这老头满头凌乱的白发，邋遢的白胡须，像是从旅行团中落下的游客，已经完全迷路了，闲逛到这里来歇歇脚，随便看看我们都在这里干什么。我的第一个念头是，他会因为我们的奇怪的话题感到相当失望；接着，我意识到这位老头的年纪，想到斯坦福就在附近，而且我想那人也在斯坦福——难道他是……

“嗨，John，有什么问题？”Peter 说。

虽然这只是 10 个字左右的问题，我不会假装自己记住了 Lisp 之父约翰麦卡锡说的每一个字。他在问 Python 程序能不能像处理数据一样，优雅地处理 Python 代码。

“不行。John, Python 做不到。”Peter 就回答了这一句，然后静静地等待，准备接受教授的质疑，但老人没有再说什么了。此时，无语已胜千言。

RMQ(Range Minimum Query) and ST(Segmented Tree) algorithms

`http://hihocoder.com/contest/hiho16/problem/1`

小 Hi 和小 Ho 在美国旅行了相当长的一段时间之后，终于准备要回国啦！而在回国之前，他们准备去超市采购一些当地特产——比如汉堡（大雾）之类的回国。

但等到了超市之后，小 Hi 和小 Ho 发现者超市拥有的商品种类实在太多了——他们实在看不过来了！于是小 Hi 决定向小 Ho 委派一个任务：假设整个货架上从左到右拜访了 N 种商品，并且依次标号为 1 到 N，每次小 Hi 都给出一段区间 [L, R]，小 Ho 要做的是选出标号在这个区间内的所有商品重量最轻的一种，并且告诉小 Hi 这个商品的重量，于是他们就可以毫不费劲的买上一大堆东西了——多么可悲的选择困难症患者。

（虽然说每次给出的区间仍然要小 Hi 来进行决定——但是小 Hi 最终机智的选择了使用随机数生成这些区间！但是为什么小 Hi 不直接使用随机数生成购物清单呢？——问那么多做什么！）

提示一：二分法是宇宙至强之法！（真的么？）

提示二：线段树不也是二分法么？

输入

每个测试点（输入文件）有且仅有一组测试数据。

每组测试数据的第 1 行为一个整数 N，意义如前文所述。

每组测试数据的第 2 行为 N 个整数，分别描述每种商品的重量，其中第 i 个整数表示标号为 i 的商品的重量 weight_i。

每组测试数据的第 3 行为一个整数 Q，表示小 Hi 总共询问的次数。

每组测试数据的第 N+4~N+Q+3 行，每行分别描述一个询问，其中第 N+i+3 行为两个整数 Li, Ri，表示小 Hi 询问的一个区间 [Li, Ri]。

对于 100% 的数据，满足 N<=10^6，Q<=10^6, 1<=Li<=Ri<=N，0<weight_i<=10^4。

输出

对于每组测试数据，对于每个小 Hi 的询问，按照在输入中出现的顺序，各输出一行，表示查询的结果：标号在区间 [Li, Ri] 中的所有商品中重量最轻的商品的重量。

Junsong Li's Blog

一：“编程只是一种技能，提高技能还需要个人素质的支撑，包括毅力，决心，远见等等。学习其它学科不能直接作用于这项技能，但能大大提升个人的这些素质，而这些素质的提高常常能直接作用于技能的提升，并且，它们远远比技能的提升对于人的一生重要很多。就从 CS 这个领域来说，能编程但个人素质不高的大有人在（可以在“一亩三分地”上看到很多不学无术找到工作就振臂高呼的），但个人素质很高的却不太可能编程不在行。”

二：“不记则思不起” 在来到美国前的一年，我又读了一遍《射雕英雄传》。在第三遍读的时候，我对郭靖背九阴真经，而后在丐帮大会上看到北斗七星悟到北斗天罡阵法的片段念念不忘。如果当初不把九阴真经背下来的话，郭靖怎么又能感悟得了呢？

后来读王阳明全集，对于知行合一的大话和空话之前是听了一遍又一遍。“看似高深，也就那么一回事吧”。但有一天我突然意识到，知行合一并不是要同时发生，也不可能同时发生，像郭靖，他一样是做到了知行合一，只不过是先知而后行，然后做到了知行合一。

知行合一并没有告诉人们怎么去学习，只是告诉人们理论实践要相互强化，相互印证。但哪里是个开始呢？如果我今天要开始学习 design，那从理论开始呢还是实践开始呢？

其实这个问题不难，如果我们思考人类的优势，我们会发现，后一代人做的比前一代人多是因为我们都站在巨人的肩膀上。我们有千年传承下来的知识体系，不需要从零开始造轮子。学习一个东西的过程，应该从知道基本概念出发，然后动手实践强化概念。

我们其实也可以说先行后知是不得已的事情，先知后行才是学习的模式。

这里的想法可能让人疑惑。因为我们现在大多数人大多数时间都是在遵循先知后行的学习模式，貌似见效不高啊！的确大家都在用，但有多少人是真正有行动力的呢？缺少主动性的人在“知”这一阶段很容易就蒙混过去（“我来看看这个。嗯，不过就是这样”），到了“行”的时候，需要的时候拿不出来，碰到了困难就退却了（“尼玛，这个怎么破”）。

我只需要举一个例子就知道了。*SICP* 这本书，对于很大一部分人学 *CS* 的都耳熟能详（我指书名），大家都在博客里面吹牛逼这么书是多么让

他们大看眼界毁三观。但实际上写过前三章练习题的人都算少，根本不用说做完全部五章。你看，大家读了前几章，刚知道一点点什么是抽象，就合上书了，觉得练习不需要做。真是这样吗？其实大家不做的原因不过是因为难罢了，因为前面几章简单的题大家都在做。这本书的习题也就只有第一章和第二章的简单。

想清楚了这点之后，我们就有了做事的原则。我们读文史哲，常常讲究背诵。背诵是一件很痛苦的事情，但中国的历史和文学，恐怕都只能是通过“常读诵习之”的方法来理解和感悟，从而获得“知”。CS 的人找工作，常常谈到刷算法题（特别是 leetcode）。跟风刷 leetcode 是一件很冒险的事情。因为我们要“学习”算法，所以我们应该遵守先知后行的原则，更有意义的应该是找一本口碑好的算法书，看完一章之后，从头到尾刷一遍后面的练习题。学习 *design*，了解了原理之后要马上自己动手，光线，阴影，布局，自己需要亲自尝试才能为之后的知识打下基础。

英语学习则更体现出先知后行。我们在新闻报纸上看到生词，除了猜测它的意思之外，应该动手查阅词典了解意思，这个才是“知”的过程，如果一直放任不管，即使这个单词出现了几十次，恐怕也难得理解。

先知而后行常常都是一个让人痛苦的过程。在学习理论的过程中不知道怎么去实战，认为自己只知道理论的东西毫无意义。急于求成的人往往就直接开始动手，但往往重复造了轮子又浪费了时间，还说，这个自己要从头做才记得牢。但别忘了，人一旦不反复使用知识，哪怕是自己的发明，也会忘记得一干二净（还记得为什么大家编程要加注释吗？哪怕这些代码是自己的发明，遗忘它是必然的）。

三： 我又看了一遍乔布斯的访谈。这次又发现了一些新的东西。

他说，在一般的职业中，比如出租车司机，最好的和中等的，差别不过（百分之）十或者二十。但是在程序员上，最好的比起中等的来说，差别是在百分之五十到一百。A player 渴望和其它的 A player 合作，一旦他们合作过，他们再也不想和 B Player 以及 C player 合作了。我一直在考虑如何成为一名 A player。在我面试 Jane Street 的时候，我发现了他们喜欢考我认为比较有趣的东西；在面试 Google 的时候，我发现他们喜欢考“你能给出几种解法”。工业界认为的 A player 应该有所长，有的方面能钻得深；也要知识面广，其它的方面也都懂。如何成为 *A Player* 是一个很难的回答。但我认为一个 *A player* 的成长过程，必须有好奇心和行动力在。有

了对 *science* 的好奇心，但凡是和计算机相关的问题，都是可以研究和探讨的问题；有了行动力才找得到相关的答案。有了这两点，一切都会自然而然的发生。但大学四年为成绩紧紧束缚住了自己，已经丢掉了一大半的好奇心；剩下一些勉强存活但也不成气候。当我意识到自己的好奇心死掉的时候，内心微微悲哀。

四： 如果你和我一样，喜欢写作，你很快就会发现，这是我们自己搞翻译，所以并不要求“信达雅”。我们按照我们的想法，磨练自己的文字，然后展现出去，表达我们的思想，就这样。我们不做 translate，做 paraphrase，所以不会枯燥。

另外对于兴趣，我实在不知道说什么。兴趣在大学中和正义在现代社会中一样，在留有余地的情况下常常像气球一样被压力压得越来越来小。很少有人为了自己的兴趣而不顾一切，我想这也是为什么《老男孩》这一类以忏悔为题材的电影能看得人热泪盈眶的原因吧。[题外话，所以我觉得，能找到这样一些人非常非常不容易，他们也大多像稀有动物一样集聚在一起。从学校的角度上说，名校效益只会破坏这样一种氛围（如果这种氛围是在学校里面的话），因为奋力挤进去的人并不为这氛围而去，并且在往里挤的时候失掉了时间和精力，没有这样一种特性，好比是浓糖水里面参入了平淡无味的白开水一样。]

而因为太底层而不喜欢，我的确觉得奇怪。没有学过多少 CPP 的人每天都在拿着 C 当 CPP 用，用的也全是指针，链表。在这里，我们将很少用到汇编（除非我们重写文档开头的部分），也全是和指针内存打交道，为什么同样的东西却持有不同的态度？那只能归咎于一种恐惧感了。

无论如何，如果你正在犹豫，那我觉得你正需要刺激一下。比如这个一直在刺激着我的话：

“我认为，在计算机科学中保持计算中的趣味性是特别重要的事情。这一学科在起步时饱含着趣味性。当然，那些付钱的客户们时常觉得受了骗。一段时间之后，我们开始严肃地看待他们的抱怨。我们开始感觉到，自己真的像是负起成功地、无差错地，完美地使用这些机器的责任。我不认为我们可以做到这些。我认为我们的责任是去拓展这一领域，将其发展到新的方向，并在自己的家里保持趣味性。我希望计算机科学的领域绝不要丧失其趣味意识，最重要的是，我希望我们不要变成传道士，不要认为你是兜售圣经的人，世界上这种人已经足够多了。你所知道有关计算的东

西，其他人也都能学到。绝不要以为似乎成功计算的钥匙就掌握在你的手里。你所掌握的，也是我认为并希望的，也就是智慧：那种看到这一机器比你第一次站在它面前时能做得更多的能力，这样你才能将它向前推进。”—Alan J. Perlis

也因为我一直想保持一种趣味性，我一直在克服各种困难做一些自己喜欢的事情，常常在考试前几天还会继续学习 Emacs，写写 shell 脚本和其他 Linux 技术。后来在 SICP 上看到这句话的时候，更是不顾课程考试等等大胆地从头开始学 scheme，写 shell 脚本，写博客。我希望有更多的人能真正地在“拓展这一领域“的时候，保持趣味性，哪怕必要的时候牺牲一些东西。

五：时间分配的两种形式 盲目地做时间规划是没有用的。一个人不知道自己在一个小时内到底能干多少事情，却妄图按照排序的方式把任务安排到时间槽里面，然后定个一个小时或者两个小时在某个任务上。这根本就解决不了自己在众多任务面前手慌脚乱的处境。

时间分配有两种形式：一种按照任务优先级，标注 ABCD，然后做完 A 做 B，做完 B 做 C；另一种是直接安排几点到几点做什么事情。其他的一些方法都是这两种方法上的一些方法上的 trick，比如把时间划作几个二十分钟，然后每个二十分钟全力去做事情等等。

我们为什么要花时间来分配时间？像我这样在学校读书的人做时间分配，无非就是求心里踏实呗—分配好了时间，做事情就可以有条不紊地完成。

如果想分配完时间就开始着手开始安心地按照规律干活，第一种方法完全不适合我。因为我每天可能有很多门的作业都要开始做，他们根本没有优先级。即使在我激烈的思想斗争之后我把数学学习安排为 A，也有一堆理由来导致焦虑感，最主要的一个是，如果做不完数学，这一天不就耗这了？明天还要交其他的作业呢！

第一种方法的灵活性带来了对不确定性的恐惧。

第二种方法还算有效，因为它是固定时间的，我能够在头一天就为第二天的各种事情安排好时间。但同样的，即使给数学学习安排了早上八点到十点，第二天需要提交，但如果八点到十点做不完，整个计划都会有所改变。这样产生的焦虑感其实更为严重。

有趣的是，在整理笔袋中碎纸片的时候，我看到了一些在大二时候做

的当日计划。有的上面列了一些 ABCD 标记的条目，表示优先级；有的上面直接写了几点到几点做什么东西。那时我还会把做完的任务划掉，以此来减弱一点自己的焦虑感。

它们有用吗？有用。我当时常常陷入用第一种还是第二种的方式来安排这一天的苦恼中。有时候列计划是因为胡乱忙了一上午，想列一下计划看看剩下的这一天还可以用来干多少事情。可以想像，他们的用处常常是作为一种心理安慰剂，因为我根本不知道我一个小时能够做多少事情。通过这些小纸条，我此刻可以清楚地看到那时候每天划掉的任务是多么少……

怎样的时间，怎样的我们 那如何安排时间呢？从高中第一次接触《how to control your time and life》开始，我就在尝试把我的生活安排得规规矩矩。但无论是怎样安排，最终达到的目标总是和我的目的背道而驰。看着时间表上的一个个任务，特别是在当前任务踟蹰不前的时候，只凭添了一些对下一时刻的恐惧—要延误了，要延误了，妈的，今天又毁了。

所有的这些恐惧都来源于我们对自己的不了解。我们根本就不了解这一个小时我们都能干什么，不了解明天还会有多少时间来给我们使用。这些问题，即使分配好了时间，依然是解决不掉的，因为我们的生活由自己和其他人组成。我们安排好了我们的时间，其他人可不这样。

最好的规划方式，是从记录时间而不是规划时间开始。因为我们的生物性，我们早就已经在某些时刻养成了某些习惯，我们只需要做一个忠实的记录者—今天花了哪些时间做哪些事情，然后定期作总结，之后的规划就会自然而然的发生。这种神奇的自然性好像我们在收拾书桌的时候会把书目按照大小或者类型归类一样。在一段时间之后我们就会发现，原来我们之前对时间的掌控能力原来这么弱。

或许一个小时就看了两页研究生的数学课本，半个小时就看了十几页科幻小说，更不能容忍的是原来今天用在正事上的时间只有四个小时，这样的情况已经持续了一个星期了，以前居然信誓旦旦地要发奋学习十四个小时……如果是以前凭空来安排进度，一本数学课本 300 页，我准备在这个假期里面自学一下它，每天不要安排太多，就安排 10 页吧。

你看，人对自己的认识也是有代价的，它需要时间。从小到大我们都在不断地认识自己，从认识到自己有某种能力到高估自己的能力用不了多长时间，但从高估自己到再认识自己却有很长一段路要走。如果你问一个

人，你在假期可以每天用多少时间来学习？他的回答一定是会大于 4 甚至于 6 小时，为什么是那么一些时间呢，猜的呗，太少了面子也挂不住。对自己的不了解造成了我们从高一直到大学，放假就往书包里疯狂地塞书，然后开学又原封不动地把书背到学校来；也造成了我们在平时制作计划的时候，常常感觉计划赶不上变化，最终放弃的结局。

不要急，慢慢来。

六：由一个序言谈起 “我认为，在计算机科学中保持计算中的趣味性是特别重要的事情。这一学科在起步时饱含着趣味性。当然，那些付钱的客户们时常觉得受了骗。一段时间之后，我们开始严肃地看待他们的抱怨。我们开始感觉到，自己真的像是负起成功地、无差错地，完美地使用这些机器的责任。我不认为我们可以做到这些。我认为我们的责任是去拓展这一领域，将其发展到新的方向，并在自己的家里保持趣味性。我希望计算机科学的领域绝不要丧失其趣味意识，最重要的是，我希望我们不要变成传道士，不要认为你是兜售圣经的人，世界上这种人已经足够多了。你所知道有关计算的东西，其他人也都能学到。绝不要以为似乎成功计算的钥匙就掌握在你的手里。你所掌握的，也是我认为并希望的，也就是智慧：那种看到这一机器比你第一次站在它面前时能做得更多的能力，这样你才能将它向前推进。”—Alan J. Perlis

偶然在 TP 书架上看到了《计算机程序的构造和解释》。然后看到了这个很像题词一样的东西。

“传教士”一个词很有意味，这趣味并非来自传教士本身，而是”想做一名传教士“背后的动机。在这个越来越不关注个体的社会中，为标榜而传教恐怕是最大的动机了吧。”传教士“一词我听得不多，第一次是在看 Linux 与 Windows 大战的时候看到的，人们提到 Linux 的传教士；第二次是在学习 Emacs 的时候看到的，指的是 Emacs 的信徒。宣扬 Linux 的蠢事情我也干过，但现在干得少了：）至于 Emacs，我就从来没宣扬过，因为学习曲线太曲折了，估计我的同学不愿意上这种当。花很多时间为了用 Emacs 来学习 Emacs，特别是 Windows 用户，实在有点亏：）我在默默忍受了一段 Emacs 的痛苦学习后，现在好多了。

另外一个词是“推进”，看到这个词，我先想到了方向。计算机学科的迷人处应该在于提高效率了吧，比如 MIT 讲“算法导论”那“老头儿”说他喜欢 skating，喜欢速度带来的快感，速度即效率嘛。一直以来学的的

东西，无论是 Vi、Emacs、还是各种语言，那是他人的智慧，我们利用一下，产生了新的“智慧”。于是有了各种配置文件，一代又一代的人从 (autoload ···.) 开始学起，以便创造出更高效的工具，我在有一点上似乎忘记了一个重点 高效工具的目的何在，若没有目的，去提高效率的意义又何在。这样看来，对 Emacs 各种功能的扩充（上网，听音乐）却显得略有多余了，也就是反对 Emacs 的人所提到的 Emacs 不符合‘K.I.S.S.’精神。

在上 Linux 课的时候，老师说 MIT 的计算机系第一门计算机语言课是讲授 Lisp 语言。我还查到了卡内基梅隆的计算机语言课直接讲授 java，国内除 C 之外没有例外了吧。Eric 说如果想做黑客，他建议从 Python 开始学起，C 的确很强大，但不适合入门学习，很多底层的东西是要靠自己去实现的。但讲了 C 之后有个好处，好比英语考试，直接朝着 GRE 的目标去的话，现在也不至于在痛苦地背单词了（笑）。

在网上无意中看到了有人对《计算机程序的构造和解释》做出的评价，大意是，按照自己的十几年的编程经验，这本书对他的启发大大出乎他的意料。我刚开始看这本书，内容的确有些出乎意料。因为这本书不仅和程序有关，还和计算有关，还没有讲到循环的时候已经涉及到牛顿迭代法了。我一度把“在计算机科学中保持计算中的趣味性是特别重要的事情”看成了“在计算机科学中保持计算机的趣味性是特别重要的事情”。因为我每接触一门语言，必然只从语法，构成讲起，然后把语言介绍完之后就结束了。而《计算机程序的构造和解释》不是以语言为目标，而是能力。

首先，我们希望建立起一种看法：一个计算机语言并不仅仅是让计算机去执行操作的一种方式，更重要的，它是一种表述有关方法学的思想的新颖的形式化媒介。因此，程序必须写得能够供人们阅读，偶尔地去供计算机执行。其次，我们相信，在这一层次的课程里，最基本的材料并不是特定程序设计语言的语法，不是有效计算某种功能的巧妙算法，也不是算法的数学分析或者计算的本质基础，而是一些能够用于控制大型软件系统的智力复杂性的技术。

我们的目标是，使完成了这一科目的学生能对程序设计的风格要素和审美观有一种很好的感觉。他们应该掌握了控制大型系统中的复杂性的主要技术。他们应该能够去读 50 页长的程序，只要该程序是以一种值得模仿的形式写出来的。他们应该知道在什么时候哪些东西不需要去读，哪些东西不需要去理解。他们应该很有把握地去修改一个程序，同时又能保持原来作者的精神和风格。

我开始慢慢觉得，计算机科学与技术专业，除了那些迷人的技术，更多的应该是和数字打交道的科学，只有这样才能真正地做到“你所掌握的，也是我认为并希望的，也就是智慧：那种看到这一机器比你第一次站在它面前时能做得更多的能力，这样你才能将它向前推进。”在现在这个阶段，无论是学习什么语言，应该是以改进思维模式为目标，而不应该简单地以项目需要来决定学还是不学。为了求 a + abs(b)，我们可以在 C 中用两个 if 来做到，下面的代码用了不一样的方式

(define (a-plus-abs-b a b) ((if (> b 0) + -) a b)) 思维的不一致性就是这样体现的。

我曾经想过要多多的学习计算机语言，想了解各种语言的特性。但当我真正想要用这些语言来实践的时候，却不知道可以用它来做什么。学习语言没有问题，有人提出应该给自己定个每年学习一种语言的计划，我看是合适的，但要清楚学习语言的目的，不为学语言而学习，应该以改变思维，培养能力为目标来学习。这里的能力应该如作者所言：

他们应该能够去读 *50* 页长的程序，只要该程序是以一种值得模仿的形式写出来的。他们应该知道在什么时候哪些东西不需要去读，哪些东西不需要去理解。他们应该很有把握地去修改一个程序，同时又能保持原来作者的精神和风格。

这样一种能力是难得的，就从我的学习过程来看，我的同龄人中鲜有这样的能力。

一场奇妙的学习旅程（Perry)

这是一本很有趣的书，任何对编程真正感兴趣的人都应该看看。它讲了程序结构的很多方面，但始终围绕着一个主题，那就是从各个层次上来减少计算的复杂度。这和我读过的另外几本书核心是一样的，只是维度不同。比如《代码大全》厚厚的一本书讲的也是管理复杂度（http://book.douban.com/review/4981648/），但是它针对的是软件工程这门工作该从哪些方面来提高生产效率，减少沟通和维护的成本，比如变量名该怎样起，函数该多长，注释该怎么写。而这本书的出发点是各种编程问题，重点放在该如何分层以减少程序的复杂度，有点像是《Head First 设计模式》，讨论的也是如何封装变化，针对接口编程等等，不过《SICP》里的问题难得多而已，类似算法一样很费脑筋。 这本书的全称是《计算机程序的构造和解释》，从字面上就已经把本书的主题交代清楚

了，构造指的是程序的结构和构造的方式，解释指的是程序的执行和编译。这本书总共分五章，前三章讨论构造，后两章讨论解释。第一章讨论的是过程（函数）的结构，用了各种数学问题做例子，如计算斐波那契数列，分析用树形递归和线性迭代产生的计算过程有何不同，对时间和空间的开销的区别等等，后面提到了一个找零钱的问题，花了我很长时间才做出线性迭代的解法（http://goo.gl/ViGzPf），顺便学习了下动态规划。这章后面讨论了一下高阶函数，如何将一些通用的过程模式封装起来，解决各种类似的数学问题，有点像《设计模式》里的模板方法模式。

第二章讲的是数据抽象，主要是如何用两种简单的过程（构造过程和选择过程）来自定义一些数据类型，再针对这些数据类型定义一些独特的计算，最后将这些计算组合起来以解决更复杂的问题。这章的实例是重点，尤其是图形语言，你可以看到数据分层的魅力，如何在较高的层次讨论问题而不用操心底层的细节。你也可以见识到序对，(cons a b)，这种 scheme 里最简单的数据类型，是多么的强大，能组合成各种复杂的数据结构，如表、数、集合等等。第二章最后讨论了数据导向的程序设计和消息传递，算是为第三章的面向对象做铺垫。第三章讲的是程序的状态。程序的执行会改变各种变量的状态，我们可以通过定义内部变量来将一些状态封装起来，再加上一些修改局部变量的操作，能大大提高程序的内聚性，减少模块间的耦合。这样的设计加上第二章末尾介绍的消息传递，就实现了面向对象的主要功能。修改局部状态依赖于求值的环境模型。每个过程执行时都依赖于一个环境，从这个环境中可以得到其使用的变量名和函数名的值。通常定义在全局环境中的过程绑定的是全局环境，但是内部定义的过程绑定的则是一个新的环境，它继承了全局环境，并增加了定义时存在的其他变量到新的环境中。这一节对于理解 JS 闭包里的域的问题很有帮助。要实现这样的环境模型，我们需要一种可以修改的表结构，而这都可以通过序对的改变函数实现，这让我再一次见识到了序对这一数据类型的威力。引入状态的改变后就会带来并发的问题，为此本章介绍了锁的机制和相关的问题。本章最后介绍的是流的概念，将状态看作是一个连续的表来解决并发的问题，这节里最棒的内容是关于延时求值及无穷流的运用，很有想象力。第四章讲的是求值器。求值器本身也是一个程序，它执行各种表达式，给出反馈，然后等待你输入下一个指令。本章开头先给出了一个简单的 Lisp 求值器，支持几种简单的语法，最后一小节将语法的解析和分析分离很有用处，能大大提升求值器的效率，避免重复分析。

第二节讲的是惰性求值，只获取当前需要用到的参数的值，对于剩下的参数或表里的其它项目则不用管，也是改善效率的好方法。本章后面两节则是两个实例，各介绍一种新的语言，并使用 Scheme 来实现它的求值器。第三节的非确定性计算很少见，不过将它用于自然语言的分析很有意思，可以罗列出所有合理的主谓宾分解。第四节则是一个基本的数据库查询语言，支持逻辑关系。这两个实例给我的印象就是按照它们的思路一步步看下来会觉得它们的设计很完美，数据结构很合理，程序的分层页很清晰，但是你自己很难做出这样的设计，尤其是定义出那么合理的数据结构。

第五章讲的更底层，使用一些寄存器机器的基本操作来实现第四章里开头介绍的求值器。第四章是用一个 Scheme 程序来模拟 Scheme 求值器，使用了很多 Scheme 原生支持的功能（比如序对），也忽略了很多细节，第五章则去掉了这些依赖，理论上说看完这章你就可以用 C 来写一个简单的 Scheme 求值器和编译器了。这章开头先介绍了一些寄存器的基本操作和应用实例，其实就是一些汇编语言，让你对机器指令的执行有一些了解。接着又用 Scheme 来实现这个汇编语言的编译器，相当于用高阶语言实现机器语言。这两节都可以看作第四章的延续，即怎样用 Scheme 实现其他的语言。第三节才开始触到核心，用寄存器来实现序对，并用垃圾收集来回收内存。我以前一直以为垃圾收集是由 JAVA 发明的，其实 Lisp 早就使用了这个技术。第四节和第五节是本章的重点，用一个堆栈和七个寄存器来实现 Scheme 的求值器和编译器。编译的好处是能直接生成机器指令，提升效率，但是不够灵活，不好修改。所以理想的 Scheme 编程就是先用求值器调试好程序，再用编译器生成机器指令。本章最后的两个练习题就是让你用 C 来实现 Scheme 的求值器和编译器，看到这里的同学是不是都有点跃跃欲试呢：）　　我觉得看这本书就像是一趟启迪程序员心智的旅程，充满乐趣。推荐每个爱好编程的程序员都来看看。

抽象能力

这本书提到的很多次的一个词就是 abstraction：对于函数进行抽象，对于数据进行抽象，这种抽象能力其实时非常重要的。　　阅读代码时的抽象　　在学好编程之前总是对于所有函数的所有实现都感兴趣，碰到一个大型的项目就恨不得将所有函数都弄明白，但是这种方法其实很不明智，在开发大型项目时其实每一个人都不可能懂得程序的每一个部分，Linus 可能懂得 Linux 源码中的每个函数吗？所以在阅读源码时最好是将程序分为

多个模块，模块之间如何互相调用以及模块之间的关系都不重要，你只需理解该模块是干什么的，然后在调用该模块的时候只要考虑接口，不要想着模块是怎么实现。就像编写递归程序时一样，只要递归程序考虑到了所有的情况那么该程序就是对的，不需要把自己当成编译器去跟踪递归的每次调用，那样只会累死人。　　编程过程中的抽象　　在本书的视频教程中总是提到 Wishful thinking，具体意思就是在实现一个小函数之前不考虑如何进行实现，而只考虑它的功能和接口，这样思考大型项目的框架时才能解放自己，否则在考虑框架的时候就去考虑实现只会让你的思维陷入泥潭。　　所有看过本书视频教程的人都应该对第 7 讲印象深刻：教授根本不告诉学生如何实现一个具体的结构实现，也不告诉学生这个程序到底如何调用，只说了每个函数的功能就结束了，我觉得形容听这种课的感觉就像教授说的那样“itchy”，如果我们能忍住去探讨具体实现的痒，那么构建大型项目也就易如反掌了。

Other comments about SICP from douban

一：贯穿计算机科学的重要思想：

（1）计算机只能够解决离散有穷域问题，所以尝试将连续问题离散化求解

（2）递归与栈是等价的，没有栈就实现不了递归，而递归必须利用栈结构存储信息。只是递归隐式的或显式应用栈。

（3）循环和递归能力是等同的。事实上只要有了递归，就可以用线性递归代替循环结构。而有了循环，再利用栈结构，就可以模拟递归。

（4）自顶向下思想和自底向上思想贯穿在计算机科学的软硬件上的方方面面，如算法设计中的递归与递推、硬件设计的分模块、软件工程组件技术。

（5）事件驱动和时间驱动是两种基本的计算机模拟思想，应用极其广泛，例如在消息传递响应和操作系统设计，如时间分片，中断机制等。

（6）多数计算机问题都可以通过设计中间层解决。这一观点体现在软硬件设计中，如 MMU、函数接口、面向对象等

（7）计算机的软硬件是等同的，软件能干的硬件就可以，硬件能做的，软件就可以模拟。

二： 该书从计算机语言的本质讲起，通过 Lisp 语言作为工具，一步一步的将这些本质展开从而引出了计算机语言和程序设计中的关键概念。最后，使用这些概念和 Lisp 语言编写了一个虚拟机以及在其上运行的 Lisp 语言解释器和编译器。虽然本书是以 Lisp 语言作为描述概念的桥梁，但许多关键概念都可以应用到其他语言的实践中。 前几章告诉我们：一个程序员应该铭记在心的： 计算机语言其实只有三点： *The primitive*、 *The means of combination*、 *The means of abstraction* 程序员其实只需要一点：*Wishful Thinking*

中间稍前几章告诉我们： 在 Lambda Calculus 中也可以看到的 Substitution Model Procedure 和 Process 的关系：递归的 Procedure 仍然可以产生迭代的 Process 函数作为 First Class 时候：使用高阶函数以及把函数作为数据来处理 中间稍后几章告诉我们：状态和时间的概念 没有状态的函数描述的是真理世界，而有状态的函数则将它与时间联系（也就毁灭了真理）。 状态打破 Substitution Model：引入 Environment Model 为什么需要状态和如何控制状态：Lexical Closure、Stream 和 Lazy evaluation 最后几章告诉我们： Apply 是将 Procedure 和 Arguments 转化为 Environment 和 Expression Eval 是将 Environment 和 Expression 转化为 Procedure 和 Arguments Eval/Apply 循环展示了计算机语言的真正结构和解析方法。 该部分还引出了更加深入的 Denotaional semantics 的内容，用来证明递归能够最终结束。有兴趣的可以好好研究研究这个 Denotaional semantics。 最后的最后告诉我们： 将程序与机器操作相联系的方法：寄存器机器、数据通路图和控制序列 通过实现解释器来解释语言、再用语言实现编译器去编译语言、以及将解释器和编译器相互交融在一起、构成在线编程的环境。看了这部分其实才能理解 Lisp 语言真正强大的地方。

三： Lisp 只支持一种复杂数据结构：列表（list），用列表可以很容易构造出更复杂的数据结构，如：集合，hash 表，树等等。

本书大概在一半篇幅之后才提出“赋值”的概念，这和通常的命令式编程语言是截然不同的。“赋值”的引入，颠覆了 Lisp 的替换计算模型，并且带来很多其他问题，如：对象状态维护/并发。书中对并发的讨论几乎上升到哲学的高度，什么是并发？什么是时间？很少有计算机书籍能给人

这种感觉。为了解决“赋值”引入的问题，Lisp 提出“流”的概念，把一个对象在各个时间点的状态看成一个“流”，显然，当你把这个流看作一个对象时，这个对象就是一个不变的对象，从而巧妙的绕过并发的问题。

10.20 [], {},(),<>

Chevron: 臂章（军人佩戴以示军衔和军兵种的）;<> 符号可叫 chevrons（另一种叫法是 angle brackets, 区别于 [] brackets，还有 {}braces/curly brackets)

Tranposing in Emacs

C-t for char, M-t for word, C-M-t for balanced expression, C-x C-t for lines.

Epilogue of “Learn Python The Hard Way”

以下是《Learn Python The Hard Way, 2nd Edition》这本书的尾声部分。

看完了这本书，你决定继续做编程。也许它能成为你的一个职业，也许它能成为你的一项爱好。但你需要一些指导，确保自己不会走错了道路，或帮助你从这个新业余爱好中得到最大的乐趣。

我做了很久的编程。久的你都想象不出来，久的都让我苦恼。就在我写这本书的时候，我大概懂 20 种编程语言，而且我可以用一天或长点儿用一周的时间学会一种新语言——要依这种语言有多奇怪而定。但这最终成为了我的苦恼，它们已经不能再吸引我的兴趣。我并不是说这些语言没有意思，或告诉你你会觉得它们很枯燥。只是想说在我的职业旅程走到现在，我已不再对语言有兴趣。

经过这么多年的学习经历，我发现语言本身并不重要，重要的是你如何用它们。事实上，我一直知道这个道理，但我总是被语言吸引走，周期性的忘记这个道理。现在我不再忘记了，你也应该这样。

你会什么语言、你用什么语言，这并不重要。不要被围绕在编程语言周围的各种宗教宣传迷惑，那些只会遮蔽你的眼睛，让你看不出这些语言只是一种让你做有趣的事情的工具而已。这才是它们的真正属性。

编程作为一种智力活动，它是唯一的一种能让你创造出交互式艺术作品的艺术形式。你创造出来人们可以操作的软件，你是在间接的和人们交互。没有任何其它艺术形式有如此的交互性。电影是单向的向观众传输信息。绘画是静态的。而软件程序却是双向动态的。

编程只能算是一项一般有趣的工作。它可以成为一个不错的职业，但如果你既想多挣钱又要干的高兴，不如去开一家快餐馆。如果你把编程当做一种秘密武器在其它行业里使用，也许会有更好的效果。

科技界科技公司里会编程的人多如牛毛，没人会在意他们。而在生物界，医药界，政府，社会学界，物理界，历史界和数学界，如果你有这种技能，你能做出令人瞩目的事情。

当然，所有的这些话都是没有意义的。如果通过这本书，你喜欢上了编程，你应该尽你最大的努力，通过它来改善你的生活。去探索这神奇的精彩的智力活动，也只有近 50 年来的人有机会从事这种职业。如果你喜欢它，就尽情的热爱它吧。

Latex

- \stackrel{top}{bot}: stack something above something else; used in math mode
- \includegraphics[bb=30 25 45 50,clip]{pic.eps}: bb option is used to shrink the white margin

10.21

tally: 测量，计数
suffix 后缀

10.22 Hypertext，Compile Emacs 24.4 on Ubuntu

snigger 暗笑
cascade 倾泻；小瀑布

chock 止动器；ramp （装车或上下飞机的）活动梯这两者都是自换机油需要的

module 模块，组件

a great handicap to cool decision-making 感情用事是决策大忌

balkanize 使巴尔干化, 使割据

snappiness 漂亮，时髦

snappy 敏捷的；漂亮的，时髦的；爽快的

slick 光滑的；熟练的，灵巧的(learn this meaning through brace expasion of the shell)

readiness 战备

inkling 头绪；暗示，迹象 eg:"Most people *have an inkling of what the load averages mean*"

&: ampersand character 与字符；连字符

ubiquitous: 无所不在的

谭嗣（si）同对梁启超："不有行者，无以图将来；不有死者，无以招后起。君行其难，我行其易。""程婴杵臼 (chu jiu)，月照西乡，吾与足下分任之。"

曲线拟合

"我个人最喜欢的问题是，随便画一条曲线，问应该用什么方程拟合。调整参数会对曲线形状产生怎样的影响？"

Update Emacs 24.3 to 24.4 on Ubuntu

Took me some time, learned a little about how to use tar, /.configure(and how the conf file looks like), and make. After I installed 24.4, initially I found an issues that the startup buffer is always named 24 in my /home/sadapple/emacs 24.4/ folder(the folder use to compile 24.4) no matter how I modify the .emacs.desktop file in ~/.emacs.d/. Then I figured out I'm openning emacs from the laucher and if I directly open emacs from the usr/local/bin, then it seems fine. So I remove the emacs laucher and the old emacs in /usr/bin/(version 23 and 24.3 there, and purge emacs23 in command line, I can not try emacs24, so emacs 24.3 might still not fully re-

moved, but *HOW TO?*), then add a new emacs laucher using the command "sudo gnome-desktop-item-edit /usr/share/applications/ –create-new".

Create Laucher for Emacs on Ubuntu

sudo apt-get install –no-install-recommends gnome-panel
sudo gnome-desktop-item-edit /usr/share/applications/ –create-new

load-path in emacs

The variable 'load-path' lists all the directories where Emacs should look for Elisp files. *The first file found is used, therefore the order of the directories is relevant.*

New Features of Emacs 24.4

- rectangle-mark-mode 【Ctrl+x Space】
- Emacs 24.4 has a nice prettify-symbols-mode. In lisp modes, it displays lambda as "". Note that it only DISPLAYS the function named "lambda" as , it doesn't really change the string. To set it globally, use (global-prettify-symbols-mode 1).
- call toggle-frame-fullscreen 【F11】to go to full screen(*Fun and Great for me!*)
- emacs 24.4's image-mode has several new features. *But still having problem openning large jpg files from my digital camera.*
- Support for menus in text terminals
- Emacs can now change window sizes in units of pixels(*What does it mean?*)

Search and Highlighting in Emacs

While in isearch:

- 【Ctrl+s】 jump to next occurrence.

- 【Ctrl+r】 jump to previous occurrence.
- 【Ctrl+g】 exit and place cursor at original position.
- 【Enter 】 exit and place cursor at current position.
- 【Ctrl+w】to select more strings to the right of cursor.
- Type 【Alt+s c】to toggle search case-sensitivity.
- Type 【Alt+s i】to toggle search in invisible text.
- Type 【Alt+s r】to toggle regular-expression mode.
- Type 【Alt+s w】to toggle word mode.
- Type 【Alt+s _】to toggle symbol mode.
- Type 【Alt+s Space】to toggle whitespace matching.

Not in isearch:

- 【Alt+s .】, 【Alt+s _】, 【Alt+s w】
- 【Alt+s o】list the lines matching a regex
- 【Alt+s h .】highlight-symbol-at-point
- 【Alt+s h f】hi-lock-find-patterns
- 【Alt+s h l】highlight-lines-matching-regexp
- 【Alt+s h p】highlight-phrase
- 【Alt+s h r】highlight-regexp
- 【Alt+s h u】unhighlight-regexp

Emacs Image-Mode

Use dired to open a image in your photos folder:

- Press 【n】(image-next-file) to view next image.
- Press 【p】(image-previous-file) to view previous image.

When viewing a gif animation:

- press 【f】(image-next-frame) for next frame.
- press 【b】(image-previous-frame) for previous frame
- Press 【F】(image-goto-frame) to go to a specific frame.
- 【a +】(image-increase-speed)
- 【a -】(image-decrease-speed)
- 【a 0】(image-reset-speed)
- 【a r】(image-reverse-speed)

Call *describe-mode* 【*F1 m*】 to see all commands.

Hypertext and Hyperlink

超文本是一种用户接口范式，用以显示文本及与文本相关的内容。现时超文本普遍以电子文档的方式存在，其中的文字包含有可以链接到其他字段或者文档的超文本链接，允许从当前阅读位置直接切换到超文本链接所指向的文字。超文本的格式有很多，目前最常使用的是 HTML（超文本标记语言）及 RTF（富文本格式）。我们日常浏览的网页都属于超文本。

Memory Usage diff in top command and System Monitor

Linux is borrowing unused memory for disk caching. This makes it looks like you are low on memory, but you are not!

Both you and Linux agree that memory taken by applications is "used", while memory that isn't used for anything is "free". But what do you call memory that is both used for something and available for applications?

You would call that "free", but Linux calls it "used".

This "something" is what top and free calls "buffers" and "cached". Since your and Linux's terminology differs, you think you are low on ram when you're not.

To change sort methods in top(Bash shell), use Shift+F.

Use *pstree* to see processes in tree-pattern.

10.23

什么是“重造轮子”（Reinvent the Wheel)？

什么是重复造轮子？它为何会出现呢？

现代计算机技术的发展，是建立在无数先贤的集体智慧之上的。编程语言的发展也经历了若干个阶段，现代的编程语言已经变得越来越强大，每种编程语言都提供了大量的资源类库可供使用，让程序员可以通过很简练的语言，就能实现很复杂的功能。

我依然记得大学上编程课时，老师给我们举的一个例子：一个简单的按钮，在“可见即所得”的编程环境下（比如 Delphi），通过简单的拖拽就能把这个组件拖到面板上，生成一个可以点击的按钮；而在老师自己上大学那阵子，要实现这样一个功能，则要从汇编语言写起，还要自己绘制按钮的图形界面。做这么一个小按钮，可能就要花上半天或一天的时间。而现在，这一大堆代码，全部封装到了一个简单的拖拽动作上，编程变得简单了。

可如果在编码的程序员，不喜欢这个按钮怎么办？那他可能就要自己写一个，从底层开始写起，然后绘制按钮的图形界面，最后让这个按钮变得可以点击。这就是“重复造轮子”，这个程序员花了数倍的时间，做了一个已经有的功能。

大多数情况下，自己实现一个一样的功能，是因为已经有的这个功能不好用。比如上例中，这个按钮太丑了，或者程序员想让按钮的点击动作发生变化什么的，总之已有的按钮功能无法满足需求，所以要自己重新造一个出来。在这种情况下，可以看做是一种创新，是对现有“轮子”的有益补充。

而“重复造轮子”之所以被诟病，往往是因为以下这些原因：

1. 程序员相轻。

别人写的东西总是没自己写的好，没自己写的顺手。如果代码风格不好话，维护起来也很麻烦—- 这是一个看起来似乎很充分的理由。所以程序员都喜欢自己写一套东西，哪怕这个东西有开源的类库使用。我见过不少程序员经常把别人写的代码批成一坨 shit。

2. 因为版权问题。

商业产品毋庸置疑，开源产品也是需要遵守 License 的（我在《互联网怎么赚钱—五》中解释过）。因为受到这样的客观限制，而只能选择自己重复造个轮子，这也是没办法的事情。

3. 因为开发进度和沟通问题。

比如一个公司经常有多个项目组研发不同的产品。在研发过程中可能会涉及到同样的组件，如果一个项目组实现过了，另一个项目组理论上来说就没有必要重复造轮子，完全可以代码复用。

但这只是理论情况，很多时候，因为要赶项目进度，所以做出来的东西是“能用就好”，而代码复用或多或少都会带来一些额外的工作量，以及后续可能会带来的维护工作量。而程序员又是很不善于沟通的，所以经常是脾气一上来就懒得再说，自己写了。

如果公司有一个好的技术负责人，能从一定程度上推动代码复用，降低项目组之间重复造轮子的成本。据说在苹果公司内部，乔帮主要求每个团队做出来的东西都要开放 API 接口，并把文档公布出来，其他部门可以直接根据文档复用这些功能，减少“重复造轮子”，这是一种比较可取的做法。

4. 因为一些“战略”判断。

比如我的老东家，认为以后不可能拿着 Hadoop 去和亚马逊和 Google 竞争，所以重复造轮子搞了个类似 Hadoop 的飞天系统；认为不可能拿着 Android 去打手机市场，所以把 Android 的核心组件之一 JVM 替换掉了。

我个人认为这样做不值得。从今天的互联网行业来看，有很多新兴公司通过包装 Hadoop，包装 OpenStack，为 Android 定制 ROM，都取得了不错的成绩。如果老东家在成立的第一天就把精力集中在业务上，而不是重复造轮子的技术上，也许在云主机、大数据、手机三个领域早就独领风骚了，也不会有小米什么事。我依然记得当时云手机发布后（当时已经是推迟了半年发布，因为研发难度超预估），小米很紧张，被迫提前发布。

不过我现在说这些都是站着说话不腰疼，因为站在老东家高管的位置，对于成功的定义已经不是一家小小的创业公司和新兴业务能满足胃口的了。顶着那么大的压力，造出了那么大的声势，兄弟公司在一边也等着看笑话，

谁去坐那个位置都是如坐针毡，送我我都不要。

最后，对于创业公司，我有一个建议：尽可能多采用开源技术，拥抱开源社区，把精力放在业务上，怎么快怎么来。至于自己研发个框架、研发个类库，那是大公司才玩得起的奢侈的事情，如非特别必要，远离之。

Org mode note

C-c C-t toggle TODO, C-c C-s attach date, C-c [add an TODO item to agenda.

C-c C-n 下个标题 C-c C-p 上个标题 C-c C-f 下个同级的标题 C-c C-b 上个同级的标题 *C-c C-u 回到上层标题*

Tables:

- C-c C-c 调整表格，不移动光标
- TAB 调整表格，将光标移到下一个区域，必要时新建一行
- S-TAB 调整表格，将光标移到上一个区域
- RET 调整表格，将光标移到下一行，必要时会新建一行
- M-LEFT/RIGHT 左/右移当前列
- M-S-LEFT 删除当前行
- M-S-RIGHT 在光标位置左边添加一列
- M-UP/DOWN 上/下移当前行
- M-S-UP 删除当前行
- M-S-DOWN 在当前行上面添加一行。如果有前缀，则在下面添加一行
- C-c - 在当前行下面添加一个水平线。如果带前缀，则在上面添加一行水平线
- C-c RET 在当前行下面添加一个水平线。并将光标移动到下一行
- C-c ^ 将表排序。当前位置所在的列作为排序的依据。排序在距当前位置最近的两个水平线之间的行（或者整个表）中进行

Creating link: Try visiting another file—for example, your emacs initialization file. Then hit *C-c l to call org-store-link*. You'll see a message that a link was stored to the current location in the file you're visiting.

Then switch back to your org-mode buffer and paste the link using *C-c C-l to call org-insert-link*. (You may need to press the arrow keys to scroll through and find the link you just recorded.)

Org 支持的链接格式包括文件、网页、新闻组、BBDB 数据库项、IRC 会话和记录。外部链接是 URL 格式的定位器。以识别符开头，后面跟着一个冒号，冒号后面不能有空格。

Tags: C-c C-q 为当前标题输入标签。Org 模式既支持补全，也支持单键接口来设置标签，见下文。回车之后，标签会被插入，并放到第 org-tags-column 列。如果用前缀 C-u，会把当前缓冲区中的所有标签都对齐到那一列，这看起来很酷。

Emacs help

- “M-x describe-variable”, 快捷键”C-h v”, 查看变量的文档
- “M-x describe-function”, 快捷键”C-h f”, 查看命令的文档
- “M-x describe-key”, 快捷键“C-h k”, 查看快捷键的文档

10.24 RTFM could save time, at least for today's Gnus Tour

shoot oneself in the foot 搬起石头砸自己的脚
expiry 终止，满期
evangelize 传福音

Gmail via Gnus

Remove the “Inbox”tag by deleting from the “INBOX”folder with B DEL. Use B c to assign tags.

Question 4.1 When I enter a group, all read messages are gone. How to view them again?

Answer If you enter the group by saying 'RET' in group buffer with point over the group, only unread and ticked messages are loaded. Say 'C-u RET' instead to load all available messages. If you want only the e.g. 300 newest say 'C-u 300 RET'.

Question 4.2 How to tell Gnus to show an important message every time I enter a group, even when it's read?

Answer You can tick important messages. To do this hit 'u' while point is in summary buffer over the message. When you want to remove the mark, hit either 'd' (this deletes the tick mark and set's unread mark) or 'M c' (which deletes all marks for the message).

Question 4.3 How to view the headers of a message?

Answer Say 't' to show all headers, one more 't' hides them again.

IMAP & SMTP

首次在 Ubuntu 上 compile 中文

用的是 XeCJK 包，CTEX 包还是不行

解释器（Yin Wang)

Yin Wang 那篇《怎样写一个解释器》写的好！读完后我对环境、闭包、模式匹配、lambda calculus 这些理解的更清楚了一些，文中还有些没弄明白的地方（主要在实现解释器的那部分代码），以后再返回去看。

怎样写一个解释器

这是一篇解释器的入门教程。虽然我试图从最基本的原理讲起，尽量让这篇文章不依赖于其它知识，但是这篇教程并不是针对编程的入门知识，

所以我假设你已经学会了最基本的 Scheme 和函数式编程。我不是很推崇函数式编程，但它里面确实包含了很重要的一些方法。如果你完全不了解这些，可以读一下 SICP 的第一，二章（或者接下去读 The Little Schemer)。当然你也可以继续读这篇文章，有不懂的地方再去查资料。我在这里也会讲递归和模式匹配的原理。如果你已经了解这些东西，这里的内容也许可以加深你的理解。

解释器是一种简单却又深奥的东西，以至于好多人都不会写，或者自认为会写却又不真正的会写。在这个领域里有一些历史遗留下来的误解，以至于很少有人真正的知道如何写出正确的解释器。很多“语言专家”或者“逻辑学家”的解释器代码里面有各种各样的错误，却又以谬传谬，搞得无比复杂。这误解的渊源之深，真是一言难尽。

你必须从最简单的语言开始，逐步增加语言的复杂度，才能构造出正确的解释器。这篇文章就是告诉你如何写出一个最简单的语言 (lambda calculus) 的解释器，并且带有基本的的算术功能，可以作为一个高级计算器来使用。

一般的程序语言课程往往从语法分析 (parsing) 开始，折腾 lex 和 yacc 等麻烦却不中用的工具，解决一些根本不需要存在的问题。Parsing 的作用其实只是把字符串解码成程序的语法树（AST）结构。麻烦好久得到了 AST 之后，真正的困难才开始！而很多人在写完 parser 之后就已经倒下了。鉴于这个原因，这里我用所谓的“S-expression”来表示程序的语法树（AST）结构。S-expression （加上 Lisp 对它发自“本能”的处理能力）让我们可以直接跳过 parse 的步骤，进入关键的主题：语义 (semantics)。

这里用的 Scheme 实现是 Racket。为了让程序简洁，我使用了 Racket 的模式匹配（pattern matching)。我对 Racket 没有特别的好感。但它安装比较方便，而且是免费的。如果你用其它的 Scheme 实现的话，恐怕要自己做一些调整。

解释器是什么 首先我们来谈一下解释器是什么。说白了解释器跟计算器差不多。它们都接受一个“表达式”，输出一个“结果”。比如，得到'(+ 1 2) 之后就输出 3。不过解释器的表达式要比计算器的表达式复杂一些。解释器接受的表达式叫做“程序”，而不只是简单的算术表达式。从本质上讲，每个程序都是一台机器的“描述”，而解释器就是在“模拟”这台机器的运转，也就是在进行“计算”。所以从某种意义上讲，解释器就是计算的

本质。当然，不同的解释器就会带来不同的计算。

需要注意的是，我们的解释器接受的参数是一个表达式的“数据结构”，而不是一个字符串。这里我们用一种叫“S-expression”的数据结构来表示表达式。比如表达式'(+ 1 2) 里面的内容是三个符号：'+, '1 和'2，而不是字符串“(+ 1 2)”。从结构化的数据里面提取信息很方便，而从字符串里提取信息很麻烦，而且容易出错。

从广义上讲，解释器是一个通用的概念。计算器实际上是解释器的一种形式，只不过它处理的语言比程序的解释器简单很多。也许你会发现，CPU 和人脑，从本质上来讲也是解释器，因为解释器的本质实际上是“任何用于处理语言的机器”。

递归定义 (recursive definition) 解释器一般都是“递归程序”。之所以是递归的原因，在于它处理的数据结构（程序）本身是“递归定义”的结构。算术表达式就是一个这样的结构，比如：'(* (+ 1 2) (* (- 9 6) 4))。每一个表达式里面可以含有子表达式，子表达式里面还可以有子表达式，如此无穷无尽的嵌套。看似很复杂，其实它的定义不过是：

“算术表达式”有两种形式：

一个数一个'(op e1 e2) 这样的结构（其中 e1 和 e2 是两个“算术表达式”）看出来哪里在“递归”了吗？我们本来在定义“算术表达式”这个概念，而它的定义里面用到了“算术表达式”这个概念本身！这就构造了一个“回路”，让我们可以生成任意深度的表达式。

很多其它的数据，包括自然数，都是可以用递归来定义的。比如常见的对自然数的定义是：

“自然数”有两种形式：

零某个“自然数”的后继看到了吗？“自然数”的定义里面出现了它自己！这就是为什么我们有无穷多个自然数。

所以可以说递归是无所不在的，甚至有人说递归就是自然界的终极原理。递归的数据总是需要递归的程序来处理。虽然递归有时候表现为另外的形式，比如循环 (loop)，但是“递归”这个概念比“循环”更广泛一些。有很多递归程序不能用循环来表达，比如我们今天要写的解释器就是一个递归程序，它就不能用循环来表达。所以写出正确的递归程序，对于设计任何系统都是至关重要的。其实递归的概念不限于程序设计。在数学证明里面有个概念叫“归纳法”(induction)，比如“数学归纳法”

(mathematical induction)。其实归纳法跟递归完全是一回事。

我们今天的解释器就是一个递归程序。它接受一个表达式，递归的调用它自己来处理各个子表达式，然后把各个递归的结果组合在一起，形成最后的结果。这有点像二叉树遍历，只不过我们的数据结构（程序）比二叉树复杂一些。

模式匹配和递归：一个简单的计算器 既然计算器是一种最简单的解释器，那么我们为何不从计算器开始写？下面就是一个计算器，它可以计算四则运算的表达式。这些表达式可以任意的嵌套，比如'(* (+ 1 2) (+ 3 4))。我想从这个简单的例子来讲一下模式匹配 (pattern matching) 和递归 (recursion) 的原理。

下面就是这个计算器的代码。它接受一个表达式，输出一个数字作为结果，正如上一节所示。

(define calc (lambda (exp) (match exp ; 匹配表达式的两种情况 [(? number? x) x] ; 是数字，直接返回 [`(,op ,e1 ,e2) ; 匹配并且提取出操作符 op 和两个操作数 e1, e2 (let ([v1 (calc e1)] ; 递归调用 calc 自己，得到 e1 的值 [v2 (calc e2)]) ; 递归调用 calc 自己，得到 e2 的值 (match op ; 分支：处理操作符 op 的 4 种情况 ['+ (+ v1 v2)] ; 如果是加号，输出结果为 (+ v1 v2) ['- (- v1 v2)] ; 如果是减号，乘号，除号，相似的处理 ['* (* v1 v2)] ['/ (/ v1 v2)]))])))) 这里的 match 语句是一个模式匹配。它的形式是这样：

(match exp [模式结果] [模式结果] ... ...) 它根据表达式 exp 的“结构”来进行“分支”操作。每一个分支由两部分组成，左边的是一个“模式”，右边的是一个结果。左边的模式在匹配之后可能会绑定一些变量，它们可以在右边的表达式里面使用。

一般说来，数据的“定义”有多少种情况，用来处理它的“模式”就有多少情况。比如算术表达式有两种情况，数字或者 (op e1 e2)。所以用来处理它的 match 语句就有两种模式。“你所有的情况，我都能处理”，这就是“穷举法”。穷举的思想非常重要，你漏掉的任何一种情况，都非常有可能带来麻烦。所谓的“数学归纳法”，就是这种穷举法在自然数的递归定义上面的表现。因为你穷举了所有的自然数可能被构造的两种形式，所以你能确保定理对“任意自然数”成立。

那么模式是如何工作的呢？比如'(,op ,e1 ,e2) 就是一个模式（pattern），它被用来匹配输入的 exp。模式匹配基本的原理就是匹配与它“结构相同”

的数据。比如，如果 exp 是'(+ 1 2)，那么'(,op ,e1 ,e2) 就会把 op 绑定到'+，把 e1 绑定到'1，把 e2 绑定到'2。这是因为它们结构相同：

'(,op ,e1 ,e2) '(+ 1 2) 说白了，模式就是一个可以含有“名字”（像 op, e1 和 e2）的“数据结构”，像'(,op ,e1 ,e2)。我们拿这个带有名字的结构去“匹配”实际的数据（像'(+ 1 2))。当它们一一对应之后，这些名字就自动被绑定到实际数据里相应位置的值。模式里面不但可以含有名字，也可以含有具体的数据。比如你可以构造一个模式'(,op ,e1 42)，用来匹配第二个操作数固定为 42 的那些表达式。

看见左边的模式，你就像直接“看见”了输入数据的形态，然后对里面的元素进行操作。它可以让我们一次性的“拆散”(destruct) 数据结构，把各个部件（域）的值绑定到多个变量，而不需要使用多个访问函数。所以模式匹配是非常直观的编程方式，值得每种语言借鉴。很多函数式语言里都有类似的功能，比如 ML 和 Haskell。

注意这里 e1 和 e2 里面的操作数还不是值，它们是表达式。我们递归的调用 interp1 自己，分别得到 e1 和 e2 的值 v1 和 v2。它们应该是数字。

你注意到我们在什么地方使用了递归吗？如果你再看一下“算术表达式”的定义：

“算术表达式”有两种形式：

一个数一个'(op e1 e2) 这样的结构（其中 e1 和 e2 是两个“算术表达式”）你就会发现这个定义里面“递归”的地方就是 e1 和 e2，所以 calc 在 e1 和 e2 上面递归的调用自己。如果你在数据定义的每个递归处都进行递归，那么你的递归程序就会穷举所有的情况。

之后，我们根据操作符 op 的不同，对这两个值 v1 和 v2 分别进行操作。如果 op 是加号'+，我们就调用 Scheme 的加法操作，作用于 v1 和 v2，并且返回运算所得的值。如果是减号，乘号，除号，我们也进行相应的操作，返回它们的值。

所以你就可以得到如下的测试结果：

(calc '(+ 1 2)) ;; => 3

(calc '(* 2 3)) ;; => 6

(calc '(* (+ 1 2) (+ 3 4))) ;; => 21 一个计算器就是这么简单。你可以试试这些例子，然后自己再做一些新的例子。

什么是 lambda calculus？ 现在让我们过渡到一种更强大的语言：lambda calculus。它虽然名字看起来很吓人，但是其实非常简单。它的三个元素分别是是：变量，函数，调用。用传统的表达法，它们看起来就是：

变量：x 函数：x.t 调用：t1 t2 每个程序语言里面都有这三个元素，只不过具体的语法不同，所以你其实每天都在使用 lambda calculus。用 Scheme 作为例子，这三个元素看起来就像：

变量：x 函数：(lambda (x) e) 调用：(e1 e2) 一般的程序语言还有很多其它的结构，可是这三个元素却是缺一不可的。所以构建解释器的最关键步骤就是把这三个东西搞清楚。构造任何一个语言的解释器一般都是从这三个元素开始，在确保它们完全正确之后才慢慢加入其它的元素。

有一个很简单的思维方式可以让你直接看到这三元素的本质。记得我说过，每个程序都是一个“机器的描述”吗？所以每个 lambda calculus 的表达式也是一个机器的描述。这种机器跟电子线路非常相似。lambda calculus 的程序和机器有这样的一一对应关系：一个变量就是一根导线。一个函数就是某种电子器件的“样板”，有它自己的输入和输出端子，自己的逻辑。一个调用都是在设计中插入一个电子器件的“实例”，把它的输入端子连接到某些已有的导线，这些导线被叫做“参数”。所以一个 lambda calculus 的解释器实际上就是一个电子线路的模拟器。所以如果你听说有些芯片公司开始用类似 Haskell 的语言（比如 Bluespec System Verilog）来设计硬件，也就不奇怪了。

需要注意的是，跟一般语言不同，lambda calculus 的函数只有一个参数。这其实不是一个严重的限制，因为 lambda calculus 的函数可以被作为值传递 (这叫 first-class function)，所以你可以用嵌套的函数定义来表示两个以上参数的函数。比如，(lambda (x) (lambda (y) y)) 就可以表示一个两个参数的函数，它返回第二个参数。不过当它被调用的时候，你需要两层调用，就像这样：

(((lambda (x) (lambda (y) y)) 1) 2) ;; => 2 虽然看起来丑一点，但是它让我们的解释器达到终极的简单。简单对于设计程序语言的人是至关重要的。一开头就追求复杂的设计，往往导致一堆纠缠不清的问题。

lambda calculus 不同于普通语言的另外一个特点就是它没有数字等基本的数据类型，所以你不能直接用 lambda calculus 来计算像 (+ 1 2) 这样的表达式。但是有意思的是，数字却可以被 lambda calculus 的三个基本元素“编码”(encoding) 出来。这种编码可以用来表示自然数，布尔类

型，pair，list，以至于所有的数据结构。它还可以表示 if 条件语句等复杂的语法结构。常见的一种这样的编码叫做 Church encoding。所以 lambda calculus 其实可以产生出几乎所有程序语言的功能。中国的古话“三生万物”，也许就是这个意思。

求值顺序，call-by-name, call-by-value 当解释一个程序的时候，我们可以有好几种不同的“求值顺序”(evaluation order)。这有点像遍历二叉树有好几种不同的顺序一样（中序，前序，后序）。只不过这里的顺序更加复杂一些。比如下面的程序：

((lambda (x) (* x x)) (+ 1 2)) 我们可以先执行最外层的调用，把 (+ 1 2) 传递进入函数，得到 (* (+ 1 2) (+ 1 2))。所以求值顺序是：

((lambda (x) (* x x)) (+ 1 2)) => (* (+ 1 2) (+ 1 2)) => (* 3 (+ 1 2)) => (* 3 3) => 9 但是我们也可以先算出 (+ 1 2) 的结果，然后再把它传进这个函数。所以求值顺序是：

((lambda (x) (* x x)) (+ 1 2)) => ((lambda (x) (* x x)) 3) => (* 3 3) => 9 我们把第一种方式叫做 call-by-name (CBN)，因为它把参数的“名字”（也就是表达式自己）传进函数。我们把第二种方式叫做 call-by-value (CBV)，因为它先把参数的名字进行解释，得到它们的“值”之后，才把它们传进函数。

这两种解释方式的效率是不一样的。从上面的例子，你可以看出 CBN 比 CBV 多出了一步。为什么呢？因为函数 (lambda (x) (* x x)) 里面有两个 x，所以 (+ 1 2) 被传进函数的时候被复制了一份。之后我们需要对它的每一拷贝都进行一次解释，所以 (+ 1 2) 被计算了两次！

鉴于这个原因，几乎所有的程序语言都采用 CBV，而不是 CBN。CBV 常常被叫做“strict”或者“applicative order”。虽然 CBN 效率低下，与它等价的一种顺序 call-by-need 却没有这个问题。call-by-need 的基本原理是对 CBN 中被拷贝的表达式进行“共享”和“记忆”。当一个表达式的一个拷贝被计算过了之后，其它的拷贝自动得到它的值，从而避免重复求值。call-by-need 也叫“lazy evaluation”，它是 Haskell 语言所用的语义。

求值顺序不只停留于 call-by-name, call-by-value, call-by-need。人们还设计了很多种其它的求值顺序，虽然它们大部分都不能像 call-by-value 和 call-by-need 这么实用。

完整的 lambda calculus 解释器 下面是我们今天要完成的解释器，它只有 39 行（不包括空行和注释）。你可以先留意一下各个部分的注释，它们标注各个部件的名称，并且有少许讲解。这个解释器实现的是 CBV 顺序的 lambda calculus，外加基本的算术。加入基本算术的原因是为了可以让初学者写出比较有趣一点的程序，不至于一开头就被迫去学 Church encoding。

;;

;;; 以下三个定义 env0, ent-env, lookup 是对环境（environment）的基本操作：

;; 空环境 (define env0 '())

;; 扩展。对环境 env 进行扩展，把 x 映射到 v，得到一个新的环境 (define ext-env (lambda (x v env) (cons `(,x . ,v) env)))

;; 查找。在环境中 env 中查找 x 的值 (define lookup (lambda (x env) (let ([p (assq x env)]) (cond [(not p) x] [else (cdr p)]))))

;; 闭包的数据结构定义，包含一个函数定义 f 和它定义时所在的环境 (struct Closure (f env))

;; 解释器的递归定义（接受两个参数，表达式 exp 和环境 env）;; 共 5 种情况（变量，函数，调用，数字，算术表达式）(define interp1 (lambda (exp env) (match exp ; 模式匹配 exp 的以下情况（分支）[(? symbol? x) (lookup x env)] ; 变量 [(? number? x) x] ; 数字 [`(lambda (,x) ,e) ; 函数 (Closure exp env)] [`(,e1 ,e2) ; 调用 (let ([v1 (interp1 e1 env)] [v2 (interp1 e2 env)]) (match v1 [(Closure `(lambda (,x) ,e) env1) (interp1 e (ext-env x v2 env1))]))] [`(,op ,e1 ,e2) ; 算术表达式 (let ([v1 (interp1 e1 env)] [v2 (interp1 e2 env)]) (match op ['+ (+ v1 v2)] ['- (- v1 v2)] ['* (* v1 v2)] ['/ (/ v1 v2)]))])))

;; 解释器的“用户界面”函数。它把 interp1 包装起来，掩盖第二个参数，初始值为 env0 (define interp (lambda (exp) (interp1 exp env0)))

;;

测试例子 这里有一些测试的例子。你最好先玩一下再继续往下看，或者自己写一些新的例子。学习程序的最好办法就是玩弄这个程序，给它一些输入，观察它的行为。有时候这比任何语言的描述都要直观和清晰。

(interp '(+ 1 2)) ;; => 3

(interp '(* 2 3)) ;; => 6

(interp '(* 2 (+ 3 4))) ;; => 14

(interp '(* (+ 1 2) (+ 3 4))) ;; => 21

(interp '(((lambda (x) (lambda (y) (* x y))) 2) 3)) ;; => 6

(interp '((lambda (x) (* 2 x)) 3)) ;; => 6

(interp '((lambda (y) (((lambda (y) (lambda (x) (* y 2))) 3) 0)) 4)) ;; => 6

;; (interp '(1 2)) ;; => match: no matching clause for 1 在接下来的几节，我们来看看这个解释器里主要的分支（match）表达式的各种情况。

对基本算术操作的解释 算术操作在解释器里是最简单也是最“基础”的东西，因为它们不能再被细分为更小的元素了。所以在接触函数，调用等复杂的结构之前，我们来看一看对算术操作的处理。以下就是这个解释器里处理基本算术的部分，它是 interp1 的最后一个分支。

(match exp ... ... [`(,op ,e1 ,e2) (let ([v1 (interp1 e1 env)] ; 递归调用 interp1 自己，得到 e1 的值 [v2 (interp1 e2 env)]) ; 递归调用 interp1 自己，得到 e2 的值 (match op ; 分支：处理操作符 op 的 4 种情况 ['+ (+ v1 v2)] ; 如果是加号，输出结果为 (+ v1 v2) ['- (- v1 v2)] ; 如果是减号，乘号，除号，相似的处理 ['* (* v1 v2)] ['/ (/ v1 v2)]))]) 你可以看到它几乎跟刚才写的计算器一模一样，不过现在 interp1 的调用多了一个参数 env 而已。这个 env 是什么，我们下面很快就讲。

变量和函数 我想用两个小节来简单介绍一下变量，函数和环境。稍后的几节我们再来看它们是如何实现的。

变量 (variable) 的产生是数学史上的最大突破之一。因为变量可以被绑定到不同的值，从而使得函数的实现成为可能。比如数学函数 f(x) = x*2，其中 x 是一个变量，它把输入的值传递到函数的主体 x*2 里面。如果没有变量，函数就不可能实现。

对变量的最基本的操作是对它的“绑定”（binding）和“取值”（evaluate）。什么是绑定呢？拿上面的函数 f(x) 作为例子吧。当 x 等于 1 的时候，f(x) 的值是 2，而当 x 等于 2 的时候，f(x) 的值是 4。在上面的句子里，我们对 x 进行了两次绑定。第一次 x 被绑定到了 1，第二次被绑定到了 2。你可以把“绑定”理解成这样一个动作，就像当你把插头插进电源插座的那一瞬间。插头的插脚就是 f(x) 里面的那个 x，而 x*2 里面的 x，则是电线

的另外一端。所以当你把插头插进插座，电流就通过这根电线到达另外一端。如果电线导电性能良好，两头的电压应该几乎相等。有点跑题了……反正只要记住一点：绑定就是插进插座的那个“动作”。

那么“取值”呢？再想一下前面的例子，当我们用伏特表测电线另外一端的电压的时候，我们就是在对这个变量进行取值。有时候这种取值的过程不是那么明显，比如电流如果驱动了风扇的电动机。虽然电线的另外一头没有显示电压，其实电流已经作用于电动机的输入端子，进入线圈。所以你也可以说其实是电动机在对变量进行取值。

环境 我们的解释器是一个挺笨的程序，它只能一步一步的做事情。比如，当它需要求 f(1) 的值的时候，它做以下两步操作：1) 把 x 绑定到 1; 2) 进入 f 的函数体对 x*2 进行求值。这就像一个人做出这两个动作：1) 把插头插进插座，2) 走到电线的另外一头测量它的电压，并且把结果乘以 2。在第一步和第二步之间，我们如何记住 x 的值呢？它必须被传递到那个用来处理函数体的递归解释器里面。这就是为什么我们需要“环境”，也就是 interp1 的第二个参数 env。

环境记录变量的值，并且把它们传递到它们的“可见区域”，用术语说就叫做“作用域”(scope)。通常作用域是整个函数体，但是有一个例外，就是当函数体内有嵌套的函数定义的时候，内部的那个函数如果有同样的参数名，那么外层的参数名就会被“屏蔽”（shadow）掉。这样内部的函数体就看不到外层的参数了，只看到它自己的。比如 (lambda (x) (lambda (x) (* x 2)))，里面的那个 x 看到的就是内层函数的 x，而不是外层的。

在我们的解释器里，用于处理环境的主要部件如下：

;; 空环境 (define env0 '())

;; 对环境 env 进行扩展，把 x 映射到 v (define ext-env (lambda (x v env) (cons `(,x . ,v) env)))

;; 取值。在环境中 env 中查找 x 的值 (define lookup (lambda (x env) (let ([p (assq x env)]) (cond [(not p) x] [else (cdr p)])))) 这里我们用的是 Scheme 的 association list 来表示环境。Association list 看起来像这个样子：((x . 1) (y . 2) (z . 5))。也就是一个两元组（pair）的链表，左边的元素是 key，右边的元素是 value。写的直观一点就是：

((x . 1) (y . 2) (z . 5)) 查表操作就是从头到尾搜索，如果左边的 key 是要找的变量，就返回整个 pair。简单吧？

ext-env 扩展一个环境。比如，如果原来的环境是 ((y . 2) (z . 5)) 那么 (ext-env x 1 ((y . 2) (z . 5)))，就会得到 ((x . 1) (y . 2) (z . 5))。也就是把 (x . 1) 放到最前面去。值得注意的一点是，环境被扩展以后其实是形成了一个新的环境，原来的环境并没有被“改变”。比如上面红色的部分就是原来的数据结构，只不过它被放到另一个更大的结构里面了。这叫做“函数式数据结构”。这个性质在我们的解释器里是至关重要的，因为当我们扩展了一个环境之后，其它部分的代码仍然可以原封不动的访问扩展前的那个旧的环境。当我们讲到调用的时候也许你就会发现这个性质的用处。

你也可以用另外的，更高效的数据结构（比如 splay tree）来表示环境。你甚至可以用函数来表示环境。唯一的要求就是，它是变量到值的“映射”(map)。你把 x 映射到 1，待会儿查询 x 的值，它应该仍然是 1，而不会消失掉或者别的值。也就是说，这几个函数要满足这样的一种“界面约定”：如果 e 是 (ext-env 'x 1 env) 返回的环境，那么 (lookup 'x e) 应该返回 1。只要满足这样的界面约定的函数都可以被叫做 ext-env 和 lookup，以至于可以它们用来完全替代这里的函数而不会导致其它代码的修改。这叫做“抽象”，也就是“面向对象语言”的精髓所在。

对变量的解释 了解了变量，函数和环境，让我们来看看解释器对变量的操作，也就是 interp1 的 match 的第一种情况。它非常简单，就是在环境中查找变量的值。这里的 (? symbol? x) 是一个特殊的模式，它使用 Scheme 函数 symbol? 来判断输入是否匹配，如果是的就把它绑定到 x，查找它的值，然后返回这个值。

[(? symbol? x) (lookup x env)] 注意由于我们的解释器是递归的，所以这个值也许会被返回到更高层的表达式，比如 (* x 2)。

对数字的解释 对数字的解释也很简单。由于在 Scheme 里面名字'2 就是数字 2（我认为这是 Scheme 设计上的一个小错误），所以我们不需要对数字的名字做特殊的处理，把它们原封不动的返回。

[(? number? x) x]

对函数的解释 对函数的解释是一个比较难说清楚的问题。由于函数体内也许会含有外层函数的参数，比如 (lambda (y) (lambda (x) (* y 2))) 里面的 y 是外层函数的参数，却出现在内层函数定义中。如果内层函数被作为值返回，那么 (* y 2) 就会跑到 y 的作用域以外。所以我们必须把函数做成

“闭包”(closure)。闭包是一种特殊的数据结构，它由两个元素组成：函数的定义和当前的环境。所以我们对 (lambda (x) e) 这样一个函数的解释就是这样：

[‘(lambda (,x) ,e) (Closure exp env)] 注意这里的 exp 就是 ‘(lambda (,x) ,e) 自己。我们只是把它包装了一下，把它与当前的环境一起放到一个数据结构 (闭包) 里，并不进行任何复杂的运算。这里我们的闭包用的是一个 Racket 的 struct 结构，也就是一个记录类型 (record)。你也可以用其它形式来表示闭包，比如有些解释器教程提倡用函数来表示闭包。其实用什么形式都无所谓，只要能存储 exp 和 env 的值。我比较喜欢使用 struct，因为它的界面简单清晰。

为什么需要保存当前的环境呢？因为当这个函数被作为一个值返回的时候，我们必须记住里面的外层函数的参数的绑定。比如，(lambda (y) (lambda (x) (* y 2)))。当它被作用于 1 之后，我们会得到内层的函数 (lambda (x) (* y 2))。当这个函数被经过一阵周折之后再被调用的时候，y 应该等于几呢？正确的做法应该是等于 1。这种把外层参数的值记录在内层函数的闭包里的做法，叫做“lexical scoping”或者“static scoping”。

如果你不做闭包，而是把函数体直接返回，那么在 (lambda (x) (* y 2)) 被调用的位置，你可能会另外找到一个 y，从而使用它的值。在调用的时候“动态”解析变量的做法，叫做“dynamic scoping”。事实证明 dynamic scoping 的做法是严重错误的，它导致了早期语言里面出现的各种很难发现的 bug。很多早期的语言是 dynamic scoping，就是因为它们只保存了函数的代码，而没有保存它定义处的环境。这样要简单一些，但是带来太多的麻烦。早期的 Lisp，现在的 Emacs Lisp 和 TEX 就是使用 dynamic scoping 的语言。

为了演示 lexical scoping 和 dynamic scoping 的区别。你可以在我们的解释器里执行以下代码：

(interp ’((lambda (y) (((lambda (y) (lambda (x) (* y 2))) 3) 0)) 4)) 其中红色的部分就是上面提到的例子。在这里，(* y 2) 里的 y，其实是最里面的那个 (lambda (y) ...) 里的。当红色部分被作用于 3 之后。(lambda (x) (* y 2)) 被作为一个值返回。然后它被作用于 0（x 被绑定到 0，被忽略），所以 (* y 2) 应该等于 6。但是如果我们的解释器是 dynamic scoping，那么最后的结果就会等于 8。这是因为最外层的 y 开头被绑定到了 4，而 dynamic scoping 没有记住内层的 y 的值，所以使用了外层那个 y 的值。

为什么 Lexical scoping 更好呢？你可以从很简单的直觉来理解。当你构造一个“内部函数”的时候，如果它引用了外面的变量，比如这个例子里的 y，那么从外层的 y 到这个函数的内部，出现了一条“信道”（channel）。你可以把这个内部函数想象成一个电路元件，它的内部有一个节点 y 连接到一根从外部来的电线 y。当这个元件被返回，就像这个元件被挖出来送到别的地方去用。但是在它被使用的地方（调用），这个 y 节点应该从哪里得到输入呢？显然你不应该使用调用处的某个 y，因为这个 y 和之前的那个 y，虽然都叫 y，却不是“同一个 y”，也就是同名异义。它们甚至可以代表不同的类型的东西。所以这个 y 应该仍然连接原来的那根 y 电线。当这个内部元件移动的时候，就像这跟电线被无限的延长，但是它始终连接到原来的节点。

对函数调用的解释 好，我们终于到了最后的关头，函数调用。函数调用都是 (e1 e2) 这样的形式，所以我们需要先分别求出 e1 和 e2 的值。这跟基本运算的时候需要先求出两个操作数的值相似。

函数调用就像把一个电器的插头插进插座，使它开始运转。比如，当 (lambda (x) (* x 2)) 被作用于 1 时，我们把 x 绑定到 1，然后解释它的函数体 (* x 2)。但是这里有一个问题，如果函数体内有未绑定的变量，它应该取什么值呢？从上面闭包的讨论，你已经知道了，其实操作数 e1 被求值之后应该是一个闭包，所以它的里面应该有未绑定变量的值。所以，我们就把这个闭包中保存的环境 (env1) 取出来，扩展它，把 x 绑定到 v2，然后用这个扩展后的环境来解释函数体。

所以函数调用的代码如下：

[‘(,e1 ,e2) (let ([v1 (interp1 e1 env)] [v2 (interp1 e2 env)]) (match v1 [(Closure ‘(lambda (,x) ,e) env1) ; 用模式匹配的方式取出闭包里的各个子结构 (interp1 e (ext-env x v2 env1))] ; 在闭包的环境中把 x 绑定到 v2，解释函数体))] 你可能会奇怪，那么解释器的环境 env 难道这里就不用了吗？是的。我们通过 env 来计算 e1 和 e2 的值，是因为 e1 和 e2 里面的变量存在于“当前环境”。我们把 e1 里面的环境 env1 取出来用于计算函数体，是因为函数体并不是在当前环境定义的，它的代码在别的地方。如果我们用 env 来解释函数体，那就成了 dynamic scoping。

实验：你可以把 (interp1 e (ext-env x v2 env1)) 里面的 env1 改成 env，再试试我们之前讨论过的代码，它的输出就会是 8：

(interp '((lambda (y) (((lambda (y) (lambda (x) (* y 2))) 3) 0)) 4)) 另外在这里我们也看到环境用“函数式数据结构”表示的好处。闭包被调用时它的环境被扩展，但是这并不会影响原来的那个环境，我们得到的是一个新的环境。所以当函数调用返回之后，函数的参数绑定就自动“注销”了。如果你用一个非函数式的数据结构，在绑定参数时不生成新的环境，而是对已有环境进行赋值，那么这个赋值操作就会永久性的改变原来环境的内容。所以你在函数返回之后必须删除参数的绑定。这样不但麻烦，而且在复杂的情况下几乎不可能有效的控制。每一次当我使用赋值操作来修改环境，最后都会出现意想不到的麻烦。所以在写解释器，编译器的时候，我都只使用函数式数据结构来表示环境。

下一步 在懂得了这里讲述的基本的解释器构造之后，下一步可以做什么呢？其实从这个基本的解释器原型，你可以进一步发展出很多内容，比如：

在这个解释器里加一些构造，比如递归和状态，你就可以得到一个完整的程序语言的解释器，比如 Scheme 或者 Python。

对这个解释器进行“抽象”，你就可以对程序进行类型推导。感兴趣的话可以参考我实现的这个 Hindley-Milner 系统，或者 Python 类型推导。

对这个解释器进行一些改变，就可以得到一个非常强大的 online partial evaluator，可以用于编译器优化。

另外需要指出的是，学会这个解释器并不等于理解了程序语言的理论。所以在学会了这些之后，还是要看一些语义学的书。

什么是语义学

很多人问我如何在掌握基本的程序语言技能之后进入“语义学”的学习。现在我就简单介绍一下什么是“语义”，然后推荐一本入门的书。这里我说的“语义”主要是针对程序语言，不过自然语言里的语义，其实本质上也是一样的。

一个程序的“语义”通常是由另一个程序决定的，这另一个程序叫做“解释器”(interpreter)。程序只是一个数据结构，通常表示为语法树(abstract syntax tree) 或者指令序列。这个数据结构本身其实没有意义，是解释器让它产生了意义。对同一个程序可以有不同的解释，就像上面这幅图，对画面元素的不同解释，可以看到不同的内容（少女或者老妇）。

解释器接受一个“程序”(program)，输出一个“值”(value)。用图形

的方法表示，解释器看起来就像一个箭头：程序 ===> 值。这个所谓的“值”可以具有非常广泛的含义。它可能是一个整数，一个字符串，也有可能是更加奇妙的东西。

其实解释器不止存在于计算机中，它是一个很广泛的概念。其中好些你可能还没有意识到。写 Python 程序，需要 Python 解释器，它的输入是 Python 代码，输出是一个 Python 里面的数据，比如 42 或者“foo”。CPU 其实也是一个解释器，它的输入是以二进制表示的机器指令，输出是一些电信号。人脑也是一个解释器，它的输入是图像或者声音，输出是神经元之间产生的“概念”。如果你了解类型推导系统 (type inference)，就会发现类型推导的过程也是一个解释器，它的输入是一个程序，输出是一个“类型”。类型也是一种值，不过它是一种抽象的值。比如，42 对应的类型是 int，我们说 42 被抽象为 int。

所以“语义学”，基本上就是研究各种解释器。解释器的原理其实很简单，但是结构非常精巧微妙，如果你从复杂的语言入手，恐怕永远也学不会。最好的起步方式是写一个基本的 lambda calculus 的解释器。lambda calculus 只有三种元素，却可以表达所有程序语言的复杂结构。

专门讲语义的书很少，现在推荐一本我觉得深入浅出的：《Programming Languages and Lambda Calculi》。只需要看完前半部分（Part I 和 II，100 来页）就可以了。这书好在什么地方呢？它是从非常简单的布尔表达式（而不是 lambda calculus）开始讲解什么是递归定义，什么是解释，什么是 Church-Rosser，什么是上下文 (evaluation context)。在让你理解了这种简单语言的语义，有了足够的信心之后，才告诉你更多的东西。比如 lambda calculus 和 CEK，SECD 等抽象机 (abstract machine)。理解了这些概念之后，你就会发现所有的程序语言都可以比较容易的理解了。

10.28

好的习惯和平常心

如题。CFA 尽量每天能看三四个小时，research 和 programming 相关的习题也尽量形成练习的习惯。

10.30

conversant 熟悉的；精通的
prudent 精明的；稳健的
plagiarism 剽窃，抄袭
hitherto 迄今
aspect 方面
shoddy 假冒的，劣质的
helix 螺旋结构 double helix nature of DNA
hit and miss 碰运气

Virtual Box Problem Solved

For some time, if I updated my ubuntu, then virtual box might fail to load the SAS university edision image. Today I solved this problem by following the hint given by virtual box(run "sudo /etc/init.d/vboxdrv setup" in bash).

10.31 Departmental Retreat

Departmental Retreat

今天的 retreat 进行的不错，很多 phd 学生都对课程还有我们系的未来发展提了很多建议（我分在 department culture 组，提了一个希望系里 professor 多介绍他们的 research 的建议），从这点上说，一二三年的 Phd 在课程和氛围上真是比我这第五年的幸运很多。不过，读 Phd 的难易和苦乐终归于个人内心深处的 motivation，这点我在 12 年底前做的很不够，开始追冬冬后慢慢变好，在今年尤其有大的进步，过去的无法改变，我只能做好现在，形成一种努力发奋的习惯，这样往后一年啊一定更进一步：）

BibTEX的使用方法

当你用 LATEX来写文档，在管理参考文献时，你可能会用到 bibtex，也许你会嫌麻烦，会选择用

\begin{thebibliography}{10}

\bibitem xxxx

\bibitem xxxx

\end{thebibliography}

的方式来处理参考文献，然后 \cite{} 来引用。

但我要说的是，如果你只是偶尔用下参考文献（一次管理，一次使用），那么就不需要去用 bibtex 来管理参考文献了，如果经常使用，还是选择用 bibtex 来管理你的参考文献比较多（一次管理，终身使用）

bibtex 是什么，这些就不多介绍了，很多 lnote 之类的文档都介绍了，本文只介绍怎么用 bibtex。

在使用时，一般会接触到两个文件，.bib 和.bst 两个文件，.bib 就是你的参考文献数据文件.bst 一般由期刊或者杂志提供, bst 设定了参考文献出现在文末的方式，例如设置排序方式，设置作者名按缩写的方式还是不缩写，标题要不要大写，这些琐碎的事情，这些不用管太多。我们维护 bib 文件即可。

先看下.bib 文件是什么样子的：

@ARTICLE{dbhat,

author = {D. Bhat and S. Nayar},

title = {Ordinal measures for image correspondence},

journal = {IEEE Transactions on Pattern Analysis and Machine Intelligence},

year = {1998}, volume = {20(4)},

pages = {415-423},

owner = {flyskymlf},

timestamp = {02} }

@CONFERENCE{hampapur,

author = {Hampapur, A. and Bolle, R.},

title = {Comparison of sequence matching techniques for video copy detection},

booktitle = {In Conference on Storage and Retrieval for Media Databases},

year = {2002},

pages = {194-201},

owner = {flyskymlf},

timestamp = {15} }

大概都是由这些一个个的这样的标签组成，每个表示一个文献，全部按照这样写即可。

@ARTICLE @CONFERENCE 这些指定了文献的类型，article 是期刊文章，conference 是会议文章，可以从名称上分辨。类型有下面几种（来自 http://amath.colorado.edu/documentation/LaTeX/reference/faq/bibstyles.html）：

@article An article from a journal or magazine

@book A book with an explicit publisher

@booklet A work that is printed and bound, but without a named publisher or sponsoring institution

@conference The same as inproceedings

@inbook A part of a book, which may be a chapter (or section or whatever) and/or a range of pages

@incollection A part of a book having its own title

@inproceedings An article in a conference proceedings

@manual Technical documentation

@mastersthesis A Master's thesis

@misc Use this type when nothing else fits

@phdthesis A PhD thesis

@proceedings The proceedings of a conference

@techreport A report published by a school or other institution, usually numbered within a series

@unpublished A document having an author and title, but not formally published

@collection Not a standard entry type. Use proceedings instead.

@patent Not a standard entry type.

不翻译了，很简单的文字。

—-

紧跟在 @ARTICLE{之后的文字就是这个文献的标签 id 了，这个在一个 bib 文件中需要是唯一的，因为我们在 tex 中用 \cite{} 引用时需要用这个 id 来引用，当然需要唯一的值（也不用怕，如果写重复了，bibtex 在编译时不会给你通过的，你可以再修改）。再后面的标签的说明也很简单了，author 就是作者阿，title 就是标题阿，这个不说了，大家一看肯定就都明

白了。

PS: 有直观的 GUI 的界面可以去管理，推荐使用 JabRef 来管理，我就用的这个，很方便，很快捷。jabref 跨平台的，基于 java 开发的一个东东，而且是免费的。ubuntu 下直接 sudo apt-get install jabref 即可，其他版本 linux 的自行处理，我没用过，win 下也有版本，安装下就行。

编写好这些之后，保存，然后在你的 tex 文档中加入下面两行：

\bibliographystyle{bst 文件 xx.bst}

\bibliography{你的 bib 文件名 xx.bib}

加入位置就是文章的正文之后，\end{document} 之前，在正文中引用时，采用 \cite{id} 的方式来引用,id 就是上文中所说的唯一的 id。PS: 如果没有 bst 文件时，那么就需要自己来用默认提供的几个 bst 模板了：

—(来源：http://zzg34b.w3.c361.com/package/reference.htm)

plain，按字母的顺序排列，比较次序为作者、年度和标题

unsrt，样式同 plain，只是按照引用的先后排序

alpha，用作者名首字母 + 年份后两位作标号，以字母顺序排序

abbrv，类似 plain，将月份全拼改为缩写，更显紧凑

ieeetr，国际电气电子工程师协会期刊样式

acm，美国计算机学会期刊样式

siam，美国工业和应用数学学会期刊样式

apalike，美国心理学学会期刊样式

——

还有很多，这里不列举了，这里有所有的类型。一般简单的用 plain 或者 alpha 就行了，ieeetr 这些是针对特定的会议来使用。上面也说了，一些期刊之类的也会提供他们自己的 bst 文件的，去相关网站下载即可。

tex 和 bib 都设置好之后，就可以来编译了，编译分 4 步：1. latex xx.tex 编译 tex 文件，没错误的话会生成 aux 文件，aux 文件包含了引用这些的信息。2. bibtex xx.aux 根据 aux 文件中记录的信息来检测 bib 文件中的相关文献，此时也会检测 bib 中相关的书写有无错误。有错会提示，上面说的修改错误的时候就在这个时候来检测。没有错误的话会生成 bbl 文件，你可以打开 bbl 文件看下就明白了，bbl 里面其实就是本文最刚开头说的直接用

\begin{thebibliography}{10}

\bibitem xxxx

\bibitem xxxx

\end{thebibliography} 的这种形势。

3. latex xx.tex 再次编译，把参考文献编译进文档中。4. latex xx.tex 三次编译，把交叉引用编译正确。

完毕，可以打开 dvi 看下效果了！~！~ 如果嫌麻烦，可以写个脚本或者一次执行多个命令。我一般就把所有的写成一行来用：latex my.tex; bibtex my.aux; latex my.tex; latex my.tex; dvipdf; acroread my.pdf;PS：shell 脚本我用的还不熟，不献丑了 ^_^

November

11.1

schmaltzy 软绵绵；过分伤感的

11.4 对某些缺点较劲，不如一开始就不碰

敌戒柳宗元

皆知敌之仇，而不知为益之尤；皆知敌之害，而不知为利之大。秦有六国，兢兢以强；六国既除，訑訑（yi 移）乃亡。晋败楚鄢（yan 烟），范文为患；厉之不图，举国造怨。孟孙恶（wu 务）臧，孟死臧恤，“药石去矣，吾亡无日”。智能知之，犹卒以危，矧（shen 审）今之人，曾不是思。敌存而惧，敌去而舞，废备自盈，祇益为愈。敌存灭祸，敌去召过。有能知此，道大名播。惩病克寿，矜 *(jin 今, 自夸)* 壮死暴；纵欲不戒，匪愚伊耄（*mao 冒, 年老昏乱*）。我作戒诗，思者无咎。

11.6

程序的编译和解释

取决于你怎么解读这个问题。

从数学的角度，或者从科学的角度来说，王垠的说法是正确的，但这个正确的答案对解答题主的疑问并无帮助。

从工程的角度来说，我们讲究的是一个问题的『实用价值』，把这个问题上升到哲学意义上其实用处不大。

传统意义上的所谓编译与解释，区别在于代码是在什么时候被翻译成目标 CPU 的指令。——虽然这种解释从科学上说不通（科学上，王垠的那个定义更准确），但这却是一直以来大家更认可的更约定俗成的定义。

对 C 语言或者其他编译型语言来说，编译生成了目标文件，而这个目标文件是针对特定的 CPU 体系的，为 ARM 生成的目标文件，不能被用于 MIPS 的 CPU。这段代码在编译过程中就已经被翻译成了目标 CPU 指令，所以，如果这个程序需要在另外一种 CPU 上面运行，这个代码就必须重新编译。

对于各种非编译型语言（例如 python/java）来说，同样也可能存在某种编译过程，但他们编译生成的通常是一种『平台无关』的中间代码，这种代码一般不是针对特定的 CPU 平台，他们是在运行过程中才被翻译成目标 CPU 指令的，因而，在 ARM CPU 上能执行，换到 MIPS 也能执行，换到 X86 也能执行，不需要重新对源代码进行编译。

至于为什么会有虚拟机的存在？这个答案也很简单了，因为那些非编译型语言生成的并不是目标平台的代码，而是某种中间代码。而能够运行这种中间代码的机器并不广泛存在，所以我们在每个不同的平台中用软件模拟出这个假想平台的虚拟机，这个虚拟机执行这种中间代码，而虚拟机负责把代码转换成最终的目标平台上的指令。

– 简单总结：

1. 编译型语言在编译过程中生成目标平台的指令，解释型语言在运行过程中才生成目标平台的指令。

2. 虚拟机的任务是在运行过程中将中间代码翻译成目标平台的指令。

拥抱太阳的月亮

- 君子不怨天，也不怨人；农夫不怨田地，乐工不怨乐器。问题在于自己身上，而不是对象身上。

- 奸臣的路简单，忠臣的路很难，什么才是真的是为世子的路才是应该想的吧。

- 我觉得人虽然没有贵贱，但人格是有贵贱的。虽然不知道小姐丢了多少钱，但能和今天这孩子内心的伤痛比拟吗？

- 活在山上的僧人，因为贪念月色，甚至把月色盛载在水瓶中，回到寺里才最终有所领悟。把水瓶空出来的话，天上的月光已不在，贪念的月色也不能随心盛载。把无知的小女虚像装在心中，又有何用？
- 棋子不悔，落下的棋子便不能再反悔。你也是这么教我的，人心也一样，一次给的心是不容易改变的。
- 既然殿下问的话，小女斗胆回答：是一两。对于饥肠辘辘的百姓，没有比一两更有价值的。拥有万两的富翁，虽然不知一两的珍贵，但一无所有的穷人，却深切体会到了一两的珍贵。对于平穷的百姓，主上殿下就是犹如一两的迫切和珍贵。请您一定要为所有百姓，进行公平的选定。
- 想领悟世界的道理而读书。虽是一介巫女，不是因为个人利益而读书，而是想成为帮助别人的人而读书。还有，读了圣人们的话，我已经知道了领悟世界道理的喜悦，无法停止！即使是移动天地万物的季节，也要休息片刻，才能继续转动大地，让百花盛开。何况是要治理国家的主上，是更加需要休息的。

On Habits

Elements of habit

Cue, Routine, Reward, Cravings

How habits are formed

Cue, Routine and Reward forms a circle for a habit. Cravings are what drives habits. If you know how a spark a craving, this would make creating a new habit easier.

However, these cravings don't have complete authority over us. There are mechanisms that can help us ignore the temptations. But to overpower the habit, we must recognise which craving is driving the behavior.

How to change a habit?

Genuine change requires work and self-understanding of the cravings driving the behaviors. Changing any habit requires determination. No one

will quit smoking simply because they sketch a habit loop.

However, by understanding habits' mechanisms, we gain insights that make new behaviors easier to grasp. Changes are accomplished because people examine the cues, cravings and rewards that drive their behaviors and then find ways to replace their self-destructive routines with healthier alternatives, even if they aren't fully aware of what they are doing at the time. Understanding the cues and cravings driving your habits won't make them suddenly disappear, but it will give you a way to plan how to change the pattern.

Golden Rule of Habit Change *You can't extinguish a bad habit, you can only change it.*

How it works:

Use the same cue. Provide the same reward. Change the routine.

Comments For a habit to stay changed, people must believve change is possible. And most often, that belief only emerges with the help of a group.

If you want to change a habit, you must find an alternative routine, and your odds of success go up dramatically when you commit to changing as part of a group. Belief is essensial, and it grows out of a communal experience, even if that community is only as large as two people.

11.7

程序语言的比较（知乎赵磊）

因为某些语言代表了一种编程范式，讨论这些语言其实是讨论这些分析问题方式的差异。

有人说用什么语言不重要，算法才是关键。一般两种人会说这种话，一种是见过的东西太少，世界都没看全，就别谈世界观了；另外一种是什么都接触过，从极高的抽象层次看到了它们的相似。

编程语言很像是给解决计算问题的方法进行了正交分解 (像那些无关痛痒的语法糖肯定不是正交基咯)，虽然对图灵完备的语言而言，可计算的问题的范围都是一致的，但是看世界的角度完全不一样，我认为这些理解

世界的不同方式之间是有高下的。至于有一些没什么技术含量的语言之争，笑笑就算了，嘴仗而已。

没有思想高度的语言之争毫无意义，其他地方同样适用.....

11.8 Interstellar

影评（知乎）

这部电影能看出什么东西，完全取决于个人的经历和知识。人生阅历更丰富，对电影深处的共鸣越深；科学知识越丰富，越能感受到令人震撼的美。

剧情方面我就不多说了，前后照应，伏笔，诺兰的威力大家自己看 ~

科幻方面简单说说。高票答案中 @ 胡晓已经把理论知识解释得很详细了，感谢 ~ 对于一般人来说，看点在：一，磅礴壮丽的太空背景，深深感受到人类渺小；二，巧妙地在三维世界中构造出高维度的世界（看过三体的同学肯定都想象过高维度世界是怎么样的" 给我一块二向箔“XD)，如此脑洞大开的设计让人嘴巴都张大了 ~ 通过引力波，在那个可以穿梭时光的空间里向过去传递信息真是像三点五维的上帝视角了！

重点说说人性方面。深刻地触碰到了人性中的三个困境：离别与执着，恐惧与探索，爱与无能为力。

执着与离别是人生中最难放下的事情。特别是当你有得选择，可以选择不离开，但为了某一个执着的原因，不得不理性地离开时，那种撕裂的痛，是无法言表的。知道这一离开就可能是永远，却仍要催促自己狠下心迈开步子；又在路上无数次想调转了车头，只要说服自己放下了那一点执着的追求，就可以回到以前，和心爱人继续在一起。这种纠结的斗争一直持续到飞机起飞，才逼迫自己面对现实。当库柏狠心把东西扔进车里，当墨菲三人看着汽车扬起的灰土蔓延到地平线时，送行者是无助望着远去，而旅人想大声吼叫却发不出声，只剩下潸潸的泪水。（留学党们，游子们，看哭了吗？XDDDDDD) 而好容易让自己接受了此次的离别，期待着下一次见面，却突然被告知之前的离别即使永别，再也无法见面，这才是痛彻心扉，只后悔见最后一面时没有再依偎得久一点，猛烈一点，浓重一点，好让一切更感觉留得下来一点。库珀看到视频里墨菲质问布兰德是否早已知道不可回去，陷入疯狂，不再留恋，奔向黑洞；而墨菲绝望得哪也不去也不逃跑，相信老爸会回来，只求解开那个方程，虽说是剧情安排：-|）支

撑自己的信念垮了，人很容易会崩溃自杀的 ~（90 年代初期，以北大教授解万英为代表的一批左派自杀）

恐惧。最强大的恐惧来自于深深的未知。从古到今人类为了克服对未知的恐惧而探索，促成了人类的进步。黑人科学家（忘了名字了...) 刚上太空时，告诉库柏他害怕，因为这薄薄的几毫米铝后面就是几百万公里什么也没有；第一个全是水的星球，细思极恐，望不到头的水，一个又一个的浪，永远不知道到底有什么；相信看了三体的同学，潜过水的同学，望着天空，黑夜和大海时都会有这种体会。

最后是爱。爸爸对女儿爱之深，因为这是除了妈妈以外，这世界上唯一以血脉相连，无条件的爱。(关于这个台词里有一段经典的话，可是我忘了...) 所以当库柏要走了，再不能保护墨菲；当他在水星球上一晃二十三年回来，看到儿子因为自己没有消息而失望消沉却无法回话；看到自己的父亲去世却无法送行；在黑洞中看到从前的自己要离开墨菲却无法阻止；被救了以后看到自己的女儿老去却无法战胜时间... 相信有过爱，却无能为力的感受的朋友，看到都会深深地受到触动。

总之很有深度，观赏性强，引人共鸣，脑洞大开，就算 500 块钱都值得一看的。

11.9 Vim, evil mode & vimium(an extension in chrome)

obsolete 老式的，过时的
endurance 忍耐力
ammonia-saturated 氨水（气）饱和
jettison 丢弃
slingshot 弹弓，弹射
tesseract 超立方体

烤鸡翅

1. 鸡翅洗净，加入生抽、生粉、白糖、盐腌制约 2 小时。
2. 烤箱 200ř 预热 10 分钟。
3. 把鸡翅摆烤盘上，在鸡翅表面刷蜂蜜，放入烤箱 220ř 烤 20 分钟。

4. 取出鸡翅翻面，刷蜂蜜，继续放进烤箱，200ř 烤 10 分钟就 OK。

Travel Inside a Blackhole

gravitational lensing

Vim, evil mode & vimium

I learned some vim basic from coolshell post and set up the evil mode in emacs. What's more interesting is there is an vim extension available in chrome so that I could use some vim shortcuts to navigate the chrome browser, which is fun and efficient! I'll continue to use emacs for most of the editing work while at the same time am going to practise vim keybind more often with vimium and evil mode.

11.10 心中的宇宙！

C 语言 for 语句

for (initial; continue situation; iterator) { // bla bla bla } 第一部分是初始语句，即给一个或多个变量，声明一个值。第二部分是终止表达式，即当这个表达式的值为 true 时，循环节里面的代码再跑一遍；否则就终止循环节，执行循环节外面的代码。第三部分是遍历语句，即每次循环时跑一遍这里的语句。

11.11 *Three Body Problem, Jekyll & Git*

etched in one's memory
vigilant 警惕的
ARC(ARE):advance reading copy, 试行本
dusk jacket 书皮
hiccup 连续地打嗝；暂时性的小问题 eg. the only hiccup I met
cougar 有熟女的意思　mather 为男性的对应词
excerpt 引用；摘要

《千与千寻》(知乎刘鹏程）

那么整个过程中，有多少时间是无脸男自己的思想，有多少是受青蛙这个社会底层思想的影响腻？宫崎骏没有明说，我也只是猜测。是不是宫崎骏想要借这个角色来表达一个游离于社会之外的人，一个对于社会是透明人的身份所有者。这被社会某些美好事物吸引之后，有了欲望，这个欲望本身没有恶意，是淡淡的，是美好的。但是一旦看到社会中主导欲望的那些东西的威力，被欲望左右和引导之后产生了一种可怕的东西腻？无脸男那一段邪恶的表现是社会属性玷污了纯粹的吸引，只有放弃这些才能回归个人属性的美好么？人性本身就是来源于动物性并超越于动物性，人性中是有动物性的，那么无脸男在被占有欲主导之后变得像个凶残的动物，但是变回自己之后，即使不会说话，但是却有令人温暖的人性。

总之，被美好事物吸引是无罪而美丽的，但是当这种想法升级为占有欲和社会规则结合到一起可能就产生了破坏性。

说完无脸男再说说千寻的不能回头。很多答案都说了自己的想法。但是这个情节其实是有很多类似的典故的，而且东西方的传说和典故里都有这样的故事。比如说我所知道最早的类似传说是希腊神话里奥菲斯（Orpheus）去冥界救妻子的故事。回头的后果是马上就被自己带来地狱的妻子瞬间被拉回冥界，再没有拯救的机会。

也就是我们小时候都看过的白银圣斗士天琴座的原型。

奥菲斯不能回头是因为冥王为了守护冥界的规则，即使他愿意为了奥菲斯美妙的琴声而破例，但是其实他利用人性的弱点设计了这样一个陷阱。因为他妻子离开冥界的一路上是没有脚步声的。但是人只相信自己的感官可能捕捉的到的东西，越是离人间近，奥菲斯越是会不放心。他会害怕冥王欺骗他，他会怕所有的努力是一场空。所以他必然禁不住回头去看。只要他看一眼，冥王就既不违背他的承诺，也保住了冥界的规则。

在汤屋的世界也是一样，汤婆婆是不希望千寻顺利离开的，因为千寻如果能自由，那么汤屋里的其它人也会以千寻为案例要求自由。所以千寻要通过找出自己父母的选择，同时白龙是不能和千寻一起走的。留着这样一个情感上的牵绊。千寻很可能会回头看一眼，这样汤婆婆就可能名正言顺的把之前的一起承诺收回。

而且千寻属于过汤屋的世界，即使名字找回了，契约也废掉了。但是她生命中的一部分是属于汤屋的，在彻底离开之前，她和汤屋有着千丝万缕的联系，所以一旦回头，后果不堪设想。就像戒烟越是接近成功的人，

越是不能闻见烟味一样。

宫崎骏设计这样的结尾，我猜不仅仅是想说悲情往事不能回首，人要往前看这么简单的道理。其实还暗含了人是不应该用现在这一秒这一刻能看到的东西来证明自己曾经的努力的，应该相信所有的努力成果都在你前行的路上，越是时时刻刻都想确认一下自己得到了什么，离开了什么，越是断了继续往前的路。

Three Body Problem english edition received

My comment on amazon

I've read the Chinese edition for the first volume before and I feel that it is really the best Chinese science fiction I've ever encountered. I ordered this book one month ago and it finally arrived today. It is well printed while I do expect they could include some photo illustrations like the front cover. I first read the author's postscript for the English edition near the end of the book. Here are some quotes:

1. In this book, a man named "humanity" confronts a disaster, and everything he demonstrates in the face of existence and annihilation undoubtedly has sources in the reality that I experienced. The wonder of science fiction is that it can, when given certain hypothetical world settings, turn what in our reality is evil and dark into what is righteous and bright, and vice versa. This book and its two sequels try to do just that, but no matter how reality is twisted by imagination, it ultimately remains there.

2. I think it should be precisely the opposite: Let's turn the kindness we show toward the stars to members of the human race on Earth and build up the trust and understanding between the different peoples and civilizations that make up humanity. But for the universe outside the solar system, we should be ever vigilant, and be ready to attribute the worst of intentions to any Others that might exist space. For a fragile civilization like ours, this is without a doubt the most responsible path.

Well, at last, I hope all of you who's going to pick up this book enjoy it~

Installing Jekyll & rvm on Ubuntu

\curl -sSL https://get.rvm.io | bash -s stable –rails

The \curl portion uses the curl web grabbing utility to grab a script file from the rvm website. The backslash that leads the command ensures that we are using the regular curl command and not any altered, aliased version.

The -s flag indicates that the utility should operate in silent mode, the -S flag overrides some of this to allow curl to output errors if it fails. The -L flag tells the utility to follow redirects.

The script is then piped directly to bash for processing. The -s flag indicates that the input is coming from standard in. We then specify that we want the latest stable version of rvm, and that we also want to install the latest stable Rails version, which will pull in the associated Ruby.

Following a long installation procedure, all you need to do is source the rvm scripts by typing:

source ~/.rvm/scripts/rvm

You should now have a full Ruby on Rails environment configured.

Switch between Ruby versions via rvm

If you need to install specific versions of Ruby for your application, you can do so with rvm like this:

rvm install ruby_version

After the installation, we can list the available Ruby versions we have installed by typing:

rvm list

We can switch between the Ruby versions by typing:

rvm use ruby_version

Jekyll basic

First create a blog folder via “jekyll new [folder name]”

Then cd to that folder, then run “jekyll serve”.

At last open the browser, run "http://localhost:4000" in the address line since Jekyll server listen at port number 4000 .

Include images eg …which is shown in the screenshot below:

```
![My helpful screenshot]({{ site.url }}/assets/screenshot.jpg)
```

Linking to a pdf eg …you can [get the PDF]({{ site.url }}/assets/mydoc.pdf) directly.

Git Notes

commit 提交

从本地到远程 要关联一个远程库，使用命令 git remote add origin git@server-name:path/repo-name.git；

关联后，使用命令 git push -u origin master 第一次推送 master 分支的所有内容；

此后，每次本地提交后，只要有必要，就可以使用命令 git push origin master 推送最新修改；

分布式版本系统的最大好处之一是在本地工作完全不需要考虑远程库的存在，也就是有没有联网都可以正常工作，而 SVN 在没有联网的时候是拒绝干活的！当有网络的时候，再把本地提交推送一下就完成了同步，真是太方便了！

从远程到本地 要克隆一个仓库，首先必须知道仓库的地址，然后使用 git clone 命令克隆。

Git 支持多种协议，包括 https，但通过 ssh 支持的原生 git 协议速度最快。

版本回退 git log 或者 git log –pretty=oneline 可以给出 commit id。

返回上一个版本，用 git reset –hard HEAD^，往上返回一百个用 HEAD~100 替换 HEAD^。

回到未来某个版本，用 git reset –hard [commit id]，commit id 不必写全。如果忘记 commit id，用 git reflog 来寻找，此命令记录了每一次命令。

工作区和暂存区 理解这两个概念很重要

撤销修改 命令 git checkout – readme.txt 意思就是，把 readme.txt 文件在工作区的修改全部撤销，这里有两种情况：

一种是 readme.txt 自修改后还没有被放到暂存区，现在，撤销修改就回到和版本库一模一样的状态；

一种是 readme.txt 已经添加到暂存区后，又作了修改，现在，撤销修改就回到添加到暂存区后的状态。

总之，就是让这个文件回到最近一次 git commit 或 git add 时的状态。

用命令 git reset HEAD file 可以把暂存区的修改撤销掉（unstage），重新放回工作区.git reset 命令既可以回退版本，也可以把暂存区的修改回退到工作区。当我们用 HEAD 时，表示最新的版本。

现在，假设你不但改错了东西，还从暂存区提交到了版本库，怎么办呢？还记得版本回退一节吗？可以回退到上一个版本。不过，这是有条件的，就是你还没有把自己的本地版本库推送到远程。还记得 *Git* 是分布式版本控制系统吗？我们后面会讲到远程版本库，一旦你把“*stupid boss*”提交推送到远程版本库，你就真的惨了……

删除文件 一般情况下，你通常直接在文件管理器中把没用的文件删了，或者用 rm 命令删了。这个时候，Git 知道你删除了文件，因此，工作区和版本库就不一致了，git status 命令会立刻告诉你哪些文件被删除了。现在你有两个选择，一是确实要从版本库中删除该文件，那就用命令 git rm 删掉，并且 git commit。另一种情况是删错了，因为版本库里还有，这时用 git checkout –file 可以轻松恢复。*git checkout* 其实是用版本库里的版本替换工作区的版本，无论工作区是修改还是删除，都可以“一键还原”。总之，命令 git rm 用于删除一个文件。如果一个文件已经被提交到版本库，那么你永远不用担心误删，但是要小心，你只能恢复文件到最新版本，你会丢失最近一次提交后你修改的内容。

创建与合并分支 git checkout -b dev 相当于以下两条命令 git branch dev ＋ git checkout dev。用、git branch 命令可以查看当前分支，当前分支前面会标一个 * 号。切换分支用 git checkout。而 git merge 命令用于合并指定分支到当前分支。删除分支用 git branch -d [branch name]

11.12 tr command for encrypting

breakeven 收支平衡点
shutdown 倒闭

Removing Deleted Files from your Git Working Directory

"git add -u" This is the same as "git add –update". *Use this when you don't want to commit new changes yet*, but want to move deleted files to your index for the next commit.

"git add -A" This is the same as using "git add –all". This is like "git add -u," however it matches files in your working directory, and your index. That means that it will add new files, as well as updating modified content in your index, and remove files that are no longer in the working directory.

tr command in linux

I learned this encrypting command in Linghua Zhang's about.markdown file and I've created my own version—"echo abcdefgchidijkldgfmanl | tr abcdefghijklmn chuanlist@gm.o". I did this by brute force. So my question would be *is there a automatic way to generate a encrypting scheme given an arbitary text?*

Creating my own blog based on Linghua Zhang's template

One question is *how to modify the bottom entry "fork me" in the main page*. I tried to modify the default.html in layout folder then push the change to github, but github return an error message "The variable '{{' in '_layouts/default.html' was not properly closed with '}}'".

Tips for Linux

Basics

- Learn basic Bash. Actually, read the whole bash man page; it's pretty easy to follow and not that long. Alternate shells can be nice, but bash

is powerful and always available (learning mainly zsh or tcsh restricts you in many situations).

- Learn vim. There's really no competition for random Linux editing (even if you use Emacs or Eclipse most of the time).
- *Know ssh, and the basics of passwordless authentication, via ssh-agent, ssh-add*, etc.
- Be familiar with bash job management: &, Ctrl-Z, Ctrl-C, jobs, fg, bg, kill, etc.
- Basic file management: ls and ls -l (in particular, learn what every column in "ls -l" means), less, head, tail and tail -f, ln and ln -s (learn the differences and advantages of hard versus soft links), *chown, chmod*, du (for a quick summary of disk usage: du -sk *), *df, mount*.
- Basic network management: ip or ifconfig, dig.
- *Know regular expressions well, and the various flags to grep/egrep. The -o, -A, and -B options are worth knowing*.
- Learn to use apt-get or yum (depending on distro) to find and install packages.

Everyday use

- *In bash, use Ctrl-R to search through command history.*
- In bash, use Ctrl-W to kill the last word, and Ctrl-U to kill the line. See man readline for default keybindings in bash. There are a lot. For example *Alt-. cycles through prevous arguments, and Alt-* expands a glob*.
- To go back to the previous working directory: cd -
- If you are halfway through typing a command but change your mind, hit Alt-# to add a # at the beginning and enter it as a comment (or use Ctrl-A, #, enter). You can then return to it later via command history.

- Use xargs (or parallel). It's very powerful. Note you can control how many items execute per line (-L) as well as parallelism (-P). If you're not sure if it'll do the right thing, use xargs echo first. Also, -I{} is handy. Examples: find . -name *.py | xargs grep some_function cat hosts | xargs -I{} ssh root@{} hostname
- pstree -p is a helpful display of the process tree.
- Use pgrep and pkill to find or signal processes by name (-f is helpful).
- Know the various signals you can send processes. For example, to suspend a process, use kill -STOP [pid]. For the full list, see man 7 signal
- Use nohup or disown if you want a background process to keep running forever.
- Check what processes are listening via netstat -lntp. See also lsof.
- In bash scripts, use set -x for debugging output. Use set -e to abort on errors. Consider using set -o pipefail as well, to be strict about errors (though this topic is a bit subtle). For more involved scripts, also use trap.
- In bash scripts, subshells (written with parentheses) are convenient ways to group commands. A common example is to temporarily move to a different working directory, e.g. # do something in current dir (cd /some/other/dir; other-command) # continue in original dir
- In bash, note there are lots of kinds of variable expansion. Checking a variable exists: ${name:?error message}. For example, if a bash script requires a single argument, just write input_file=${1:?usage: $0 input_file}. Arithmetic expansion: i=$(((i + 1) % 5)). Sequences: {1..10}. Trimming of strings: ${var%suffix} and ${var#prefix}. For example if var=foo.pdf, then echo ${var%.pdf}.txt prints "foo.txt".
- The output of a command can be treated like a file via <(some command). For example, compare local /etc/hosts with a remote one: diff /etc/hosts <(ssh somehost cat /etc/hosts)

Manage bash history

- insert “export HISTSIZE=32768” into ~/.bashrc, this modifies the history size
- grab a command long ago: history | grep "keyword"|grep "keyword2". Then type !+”id”
- insert the command into shell instead of execute it: shopt -s histverify
- short name for history command: alias h=history

11.13

Dangerous Linux Command

- rm * -rf
- git push -f

11.14 《收获与播种》

Coder 的成长

用三年时间成长为顶级 Coder 是有可能的。

不信？排名第二的答案是酱紫说的，

快速成为顶级 *Coder*，你需要的是「自律」和「坚持」。请注意，我们说的是「*Coder*」，而不是「*Programmer*」或「*Software Engineer*」。对于 *Coder*，我们是有客观评价标准的，那就是参加「编程比赛」。

楼主研究了一下所谓的「编程比赛」或「算法大赛」：

除了国内的各种程序设计大赛，如百度之星、有道难题，更推荐参加国外的编程比赛网站，如「topcoder」、「codeforeces」。这些网站上每个月都有比赛，只要你有一台能上网的电脑就可以免费参加，和全球的 Coder 竞争，赢得了比赛还有美金拿哦。

简单来说，你至少需要做到以下几点：

- 进入 topCoder 的练习室，每天花 4-6 个小时练习编程。

- 制定一个计划，比如：每天解决 10 个初级问题，每周搞定 3 个中级问题、1 个高级问题。
- 如果在某个问题上困住了，马上去论坛求助。要知道，问问题体现了你的思维逻辑，问出好问题也是需要练习的。试着每周在问答网站 stack over flow 上问一个问题吧！
- 不要闭门造车，把你的代码展示给别人看，他们提出的意见会让你事半功倍，推荐免费网站 coder review。
- 学会看别人的代码，看代码其实比自己写代码更难，但是高手是可以轻易调用别人的代码的，推荐 *github*，可以轻松地找到海量的开源代码。
- 建议采取一些外部的措施克服拖延症，比如开通一个博客、或微博，每天在上面记下你今天做过的一件重要的事情，保证每天晚上都带着巨大的满足感入睡。

如果坚持做到以上这些，你完全成为一名顶级 Coder。

需要注意的是，编程比赛关注的是算法能力；但是，要成为一名 *Programmer* 或 *Software Engineer*，你还需要项目经验。

正如 Facebook 喜欢雇佣所谓的「Full stack programmer」，就是一个人从设计、到交互、html、css、javascript、server、sql、架构，以及数据统计都能做。成为 Full Stack Programmer 最好的方式就是不断做个人项目。

参加 Hackathons 或 game jams 就是锻炼项目能力的好方法。

在规定的时间内，一群开发者分享、讨论、组队、分工协作，用创意思维完成项目任务。无论是制作一个游戏，还是网站，在整个项目的进程中，你会不断经历发现问题、解决问题、获得经验的过程，从而保证在实际工作中也能轻松解决问题。

总之，

楼主以为，入行时间并不是衡量人才能力的绝对标准，碌碌无为的「老人」在各行业都不鲜见。在 3 年之内能否成长为一名优秀的 Coder 或 Programmer，其实取决于你对所做事情的理解程度。

对所做事情理解的越深，你就会做的越好。

成为一名优秀的程序员和成为其他行业的高手一样，都需要不停地学习、练习、反省和总结。这不仅是最初 3 年的要求，而是贯穿整个职业生涯的要义。

所以，初级程序员想要获得快速成长，一定要拥有不怕麻烦的责任心，和不满足于现状的上进心。

《收获与播种》

- 格洛腾迪克曾经这样说过，一个人从来就不应该试着去证明那些几乎不显然的东西。这句话意思不是说大家在选择研究的问题时不要有抱负。而是，“如果你看不出你正在工作的问题不是几乎显然的话，那么你还不到研究它的时候，”加州大学伯克莱分校的 Arthur Ogus 如此解释：“在这个方向再做些准备吧。而这就是他研究数学的方式，每样东西都应该如此自然，它看上去是完全直接的。”很多数学家会选择一个描述清晰的问题来敲打它，这种方式格洛腾迪克很不喜欢。在《收获与播种》一段广为人知的段落里，他将这种方式比喻成拿着锤子和凿子去敲核桃。他自己宁愿将核桃放在水里将壳泡软，或者将它放在阳光和雨下，等待核桃自然爆裂的恰当时机（第 552 — 553 页）。“因此格洛腾迪克所做的很多事情就象是事情的自然面貌一样，因为它看上去是自己长出来的，”Ogus 注意到。

- 尽管格洛腾迪克拥有伟大的技术能力，这一直都是第二位的；这只是他执行他的更大的观点的方式而已。众所周知，他证明了某些结果和发展了某些工具，但他最大的遗产是创立了数学的一个新的观点。

- 最终来说，格洛腾迪克在数学上的成就的源泉是某种相当谦卑的东西：他对他所研究的数学对象的爱。

- 在 1960 年代，哈佛大学的 Barry Mazur 和他妻子访问过高等科学研究所（IHES）。尽管那时候格洛腾迪克已经有了自己的家庭和房子，他仍然在 Mazur 居住的大楼里保留了一间公寓，并且常常在那里工作到深夜。由于公寓的钥匙不能开外面的门，而这道门到晚上 11 点的时候就锁上了，在巴黎度过一个晚上后回到大楼就会有困难。但是“我记得我们从来没有遇到过麻烦，”Mazur 回忆道。“我们会乘末班火车回来，百分之百的确信格洛腾迪克还在工作，而他的书桌靠着窗。我们会扔点石子到他窗户上，他就会来为我们开门。”

- 这种格洛腾迪克在一间斯巴达式的公寓里工作到深夜的略显孤独的形象刻划了 1960 年代他的生活的一个方面。那个时候他不停地研究数学。他得和同事们讨论问题，指导学生们的学习，做讲座，和法国外的数学家们保持广泛联系，还得去撰写看上去没有尽头的 EGA 和 SGA。毫不夸张地说他单枪匹马地领导了世界范围内代数几何里一个巨大而蓬勃发展的部分。他在数学外似乎没有多少爱好；同事们说他从来不看报纸。就是在数学家中间，他们习惯于诚实而且高度投入对待工作，格洛腾迪克也是一个异类。“整整十年里格洛腾迪克一周七天，一天十二个小时研究代数几何的基础，”他的 IHES 同事 David Ruelle 注意到。

- *In the work of discovery, this intense attention, this ardent solicitude(关心，渴望), are an essential force, just like the warmth of the sun for the obscure gestation of seeds covered in nourishing soil, and for their humble and miraculous blossoming in the light of day.*

- 尽管格洛腾迪克从一个非常一般化的观点来研究问题，他并不是为了一般化而这样做的，而是因为他可以采用一般化观点而成果丰硕。“这种研究方式在那些天赋稍缺的人手里只会导致大多少人所谓的毫无意义的一般化，”Katz 评价说，“而他不知何故却知道应该去思考哪样的一般问题。”格洛腾迪克一直是寻找最恰好的一般情形，它正好能够提供正确的杠杆作用来领悟问题。“一次接一次地，他看上去就有一个诀窍，（在研究问题时）去掉恰当多的东西，而留存下来的不是特殊情况，也不是真空，”得克萨斯大学奥斯汀分校的 John Tate 评论道，“它如同行云流水，不带累赘。它就是恰如其分的好。

- 如果说数学里有什么东西让我比对别的东西更着迷的话（毫无疑问，总有些让我着迷的），它既不是“数”也不是“大小”，而是型。在一千零一张通过其型来展示给我的面孔中，让我比其他更着迷的而且会继续让我着迷下去的，就是那隐藏在数学对象下的结构。(If there is one thing in mathematics that fascinates me more than anything else(and doubtless always has), it is the neither “number” nor “size”, but always *form*. And among the thousand-and-one faces whereby form chooses to reveal itself to us, the one that fascinates me more than any other and continues to fascinate me, is the *structure* hidden

in mathematical things.)

- 在格洛腾迪克哈佛的讲座上，曼福德发现到抽象化的跃进相当惊险。有一次他询问格洛腾迪克某个引理如何证明，结果得到一个高度抽象的论证作为回复。曼福德开始时不相信如此抽象的论证能够证明如此具体的引理。“于是我走开了，将它想了好几天，结果我意识到它是完全正确的。”曼福德回忆道,“他比我见到的任何人都更具有这种能力，去完成一个绝对令人吃惊的飞跃到某个在度上更抽象的东西上去…他一直都在寻找某种方法来叙述一个问题，看上去很明显地将所有的东西都从问题里抛开，这样你会认为里面什么都没有了。然而还有些东西留了下来，而他能够在这看上去的真空里发现真正的结构。”

- 在《收获与播种》第一卷里，格洛腾迪克对他的工作作了一个解释性的概括，意在让非数学家能够理解（第 25 — 48 页）。在那儿他写道，从最根本上来讲，他的工作是寻找两个世界的统一：“算术世界，其中（所谓的）‘空间’没有连续性的概念，和连续物体的世界，其中的‘空间’在恰当的条件下，可以用分析学家的方法来理解”。韦依猜想如此让人渴望正是因为它们提供了此种统一的线索。胜于直接尝试解决韦依猜想，格洛腾迪克大大地推广了它们的整个内涵。这样做可以让他感知更大的结构，这些猜想所凭依于此结构，却只能给它提供惊鸿一瞥。

- 基本上说，概型是代数簇概念的一个推广。给定一组素特征有限域，一个概型就可以产生一组代数簇，而每一个都有它自己与众不同的几何结构。“这些具有不同特征的不同代数簇构成的组可以想象为一个‘由代数簇组成的无限扇面的扇子’（每个特征构成一个扇面），”格洛腾迪克写道.“‘概型’就是这样的魔术扇子，就孺扇子连接很多不同的‘分支’一样，它连接着所有可能特征的‘化身’或‘转世’。”到概型的推广则可以让大家在一个统一方法下，研究一个代数簇所有的不同“化身”。在格洛腾迪克之前，“我认为大家都不真正相信能够这样做，”迈克－阿廷评论说，“这太激进了。没有人有勇气哪怕去想象这个方法可能行，甚至可能在完全一般的情况下都行。这个想法真的太出色了。”

- 从 19 世纪意大利数学家 Enrico Betti 的远见开始，同调和它的对偶上同调那时候已经发展成为研究拓扑空间的工具。基本上说，上同调

理论提供一些不变量，这些不变量可以认为是衡量空间的这个或那个方面的‘准尺’。由韦依猜想隐含着的洞察力所激发的巨大期望就是拓扑空间的上同调方法可以适用于簇与概型。这个期望在很大程度上由格洛腾迪克及其合作者的工作实现了。“就象夜以继日一样将这些上同调技巧带到代数几何中”，曼福德注意到。“它完全颠覆了这个领域。这就象傅立叶分析之前和之后的分析学。你一旦知道傅立叶分析的技巧，突然间你看一个函数的时候就有了完全深厚的洞察力。这和上同调很类似。”

- John Tate 回忆道，“很多数学家都相当孩子气，有时不通世务，不过格洛腾迪克犹有甚之。他看上去就那么无辜——不工于心计，不伪装自己，也不惺惺作态。他想问题的时候相当清晰，解释问题的时候非常有耐心，没有自觉比别人高明的意思。他没有被任何文明、权力或者高人一等的作风所污染。”Karin Tate 回忆说格洛腾迪克乐于享受快乐，他很有魅力，并喜欢开怀大笑。但他也可以变得很极端，用非黑即白的眼光来看待问题，容不得半点灰色地带。另外他很诚实: “你和他在一起的时候总知道他要说的是什么，”她说，“他不假装任何事情。他总是很直接。”

- 格洛腾迪克不是由于韦依猜想很有名、或者由于别人认为它们很难而对韦依猜想感兴趣的。事实上，他并不是靠对困难问题的挑战来推动自己。他感兴趣的问题，是那些看上去会指向更大而又隐藏着的结构。“他目标在于发现和创造问题的自然栖息之家，”Deligne 注意到，“这个部分是他感兴趣的，尤甚于解决问题。”

- “此时格洛腾迪克走了出来，说道：‘不，黎曼－洛赫定理不是一个关于簇的定理，而是一个关于簇间态射的定理’，”普林斯顿大学的尼克莱斯－卡兹说，“这是一个根本性的新观点…整个定理的陈述完全改变了。”范畴论的基本哲学，也就是大家应该更加注意的是对象间的箭头（态射），而不是对象自身，才刚刚开始在数学上取得一点影响。“格洛腾迪克所做的事情就是将这种哲学应用到数学上很困难的一个论题上去，”波莱尔说，“这真的很符合范畴和函子的精神，不过人们从没有想过在如此困难的论题上使用它…如果人们已经知道这个陈述，并且明白它在说什么，可能别的某个人可以证明这个陈述。不过单单这个陈述本身就已经领先别的任何人 10 年时间。”

Less is More

- 人们问高得纳，“你怎么这么牛？”他答：“我学编程的时候，一天能摸 5 分钟计算机就不错了。想让程序跑起来，就必须写得没有错误。所以编程就像在石头上雕刻一样，必须小心翼翼。我就是这样学编程的。”
- 现在的手机游戏相对于以前的电脑游戏，有剧情的越来越少，打发时间的越来越多。现在的网上社交相对于以前的聊天室和论坛，交流的越来越少，炫耀自我的越来越多。现在的程序员相对于以前的，Google 的越来越多，思考的越来越少，写书的越来越多，看书的越来越少。这时代究竟是进步了还是退步了？

window 8.1 wireless reset for Qiuyue

In cmd, enter “netsh winsock reset”

11.15 毛驴（罗丹）回忆高二高三

你在高三是怎样大幅度提高成绩的？（毛驴知乎）

先想想自己有没有在学习。如果你学习没有纪律，即你学习没有讲究的方式，没有合理的时间安排和计划，不会把学过的知识弄清楚并合理复述，不会把自己思考的逻辑推向连贯和严密，那我可以认为你是不在学习的。教室里天天坐着那么多读书的人，当中学习的人真没多少，学生对自己处于什么状态很难有正确的了解。

你对你自己有充分的了解没有？你成绩差的原因是不是因为真的不够聪明，还是你知识基础差，还是缺乏解题的经验技巧导致做不完试卷？搞清楚自己的问题在哪里很重要。

聪明第一看你能不能借助已有的知识和技巧来解决问题，比如利用数学上常见的拆借项化简运算，或者根据问题来举出相符的例子进而运用数学归纳法，这一步可以通过训练来完成。进一步是能避免经验思考而陷入常识陷阱，这对避免解题陷阱和理解一些高阶的知识很重要，例如说生物上的概率问题。再进一步是指有足够创造力，能不依靠已有的知识和技巧来创造新的解决问题的思路和方法，这基本上就是天才所为了，有些人就是聪明到不学习也能自己发现解决问题的方法。

如果你做不到第三点没关系，难题在数学里会占据 20~30 的分值，理综里基本没有。就算这些难题其实也都是一些高阶技巧的组合，有方法可循。第二点因为高阶的知识基本不在考纲里，而避免解题陷阱可以通过多做题来解决。这个请你在做题和考试后多总结，专门记个笔记本时刻核对。第一点也可以通过训练完成，多多总结解题方法，并且在自己练习的时候回顾下自己的方法看看有没有可以运用的。不过如果你现在还没有形成良好的学习方式和纪律的话，200 天对你来说时间做到以上要求会很紧张。多寻求帮助，但是不要依赖这些帮助。

如果知识基础差请不要放松自己的学习，多看教科书。保证自己没有遗漏掉基础知识。辅导书基本上不提供解题方法以外的东西。解题方法对提高成绩是很有帮助的，比如说选择填空题不需要你把答案精确求出，可以采用代入法之类的技巧。我高三花了一段时间来强化我的解题速度，确保如果出现 03 年一样的难度时，解决难题有足够的时间。你在对大部分题目游刃有余时可以考虑去强化这一点，就算做不出难题也有时间检查答题卡。

一点个人的例子，高二的时候因为成绩太差老爹老妈给我换了一个班，发现坐在我前面的女生学习好又漂亮，于是起早贪黑 6 点起床看书 1 点睡觉，过了一个学期就年级第一了。当时我睡觉的双人大床另外一半堆满做过的卷子，看过的书和写过的笔记。

11.16

Update the OSX to Yosemite

除了 update 时候“苹果重新定义了两分钟”这个问题外，目前其他感觉都不错 (Speed, UI, spotlight, etc)。

另外发现原来 OSX 在文本编辑时候使用了 emacs 的一些 key binding，比如 C+a, C+e, C+k，还有就是知道了把光标移到屏幕四角对应四种操作。

11.17 Vim Notes 2

Vim Notes 2

Vim distinguishes between screen-lines (those shown on the monitor) and real lines (those ended with a new-line).

So here the most important commands

- 0 ⋯first column of the line
- ^ ⋯first non-blank character of the line
- w ⋯jump to next word
- W ⋯jump to next word, ignore punctuation
- e ⋯jump to word-end
- E ⋯jump to word-end, ignore punctuation
- b ⋯jump to word-beginning
- B ⋯jump to word-beginning, ignore punctuation
- ge ⋯jump to previous word-ending
- gE ⋯jump to previous word-ending, ignore punctuation
- g_ ⋯jump to last non-blank character of the line
- $ ⋯jump to the last character of the line

If you remember just a few of them, you' ll get very quickly from A to B! *Another important fact is, that these commands give the range for other commands.*

Inserting text is pretty simple in Vim, just type i and start typing. But Vim offers quite sophisticated text-editing commands.

- d ⋯delete the characters from the cursor position up the position given by the next command (for example d$ deletes all character from the current cursor position up to the last column of the line).

- c ⋯change the character from the cursor position up to the position indicated by the next command.
- x ⋯delete the character under the cursor.
- X ⋯delete the character before the cursor (Backspace).
- y ⋯copy the characters from the current cursor position up to the position indicated by the next command.
- p ⋯paste previous deleted or yanked (copied) text after the current cursor position.
- P ⋯paste previous deleted or yanked (copied) text before the current cursor position.
- r ⋯replace the current character with the newly typed one.
- s ⋯substitute the text from the current cursor position up to the position given by the next command with the newly typed one.
- . ⋯repeat the last insertion or editing command (x,d,p⋯).
- Doubling d, c or y operates on the whole line, for example yy copies the whole line.

Please note, many commands are much more powerful than I describe them here. For example you can specify a buffer into some text is yanked. Typing "ayy copies the current line into register a, pasting the contents of register a is done by "ap. *Vim remembers the last few yanks and deletions in automatic registers, to show the contents of the registers type :registers, you can also use them to paste some older text.*

Visual Block

Using the visual block-mode it' s possible to insert characters on each line of the selection easily.

Suppose you have selected a rectangle (using Ctrl-v), you can insert text in front of it by typing I (switch to insert mode) and inserting your

text. As soon as you leave the insert mode, the text will be added to all the other selected lines. Use A to enter text after the selection.

Another useful feature is to substitute the whole block with a new text. For that matter select a block and type s, Vim enters the insert mode and you can type. After you leave the insert mode, Vim inserts the text in the remaining lines.

If you' d like to append some text at the end of some lines, use Ctrl-v$ and select the lines. The difference between the former variant is, that the $ explicitly says "end of line" whereas a selection with Ctrl-v operates on the columns, ignoring the text.

Using Ctrl-v:

This is a testNEWLY INSERTED This is a NEWLY INSERTED This is NEWLY INSERTED

Using Ctrl-v$:

This is a testNEWLY INSERTED This is aNEWLY INSERTED This isNEWLY INSERTED

Text-objects

Vim commands operate on text-objects these are characters, words, characters delimited by parentheses, sentences and so on.

For me the most important one is the inner word: iw. To select the current word, just type viw (v for selection mode, and iw for the inner word), similar for deletion: diw.

The difference between inner-word/block and a-word/block etc is that the inner variant selects only the contents like the characters of the word (no blank afterwards) or the contents of the parentheses but not the parentheses. The a-variant selects the parentheses or a blank after a word too.

- iw ⋯inner word
- aw ⋯a word
- iW ⋯inner WORD
- aW ⋯a WORD

- is ⋯inner sentence
- as ⋯a sentence
- ip ⋯inner paragraph
- ap ⋯a paragraph
- i(or i) ⋯inner block
- a(or a) ⋯a block
- i< or i> ⋯inner block
- a< or i> ⋯a block
- i{ or i} ⋯inner block
- a{ or a} ⋯a block
- i" ⋯inner block
- a" ⋯a block
- i‘ ⋯inner block
- a‘ ⋯a block

Here a quick visualisation of the commands the color and the [ ] mark the selected text:

- iw This is a [test] sentence.
- aw This is a [test]sentence.
- iW This is a [⋯test⋯] sentence.
- aW This is a [⋯test⋯]sentence.
- is ⋯sentence. [This is a sentence.] This⋯
- as ⋯sentence. [This is a sentence.]This⋯
- i(or i) 1 * ([2 + 3])

- a(or a) 1 * [(2 + 3)]
- i< or i> The <[tag]>
- a< or i> The [<tag>]
- i{ or i} some {[code block]}
- a{ or a} some [{ code block }]
- i" The "[best]"
- a" The[“best”]
- i‘ The ‘[best]‘
- a‘ The[‘best‘]

Try them out and remember the ones you need regularly (in my case iw and i() they are the real time-savers!

External commands

In Vim it’ s easy to include the output of external commands or to filter the whole line or just a part through an external filter.

To issue an external command type :!command, the output will be shown and that’ s it.

To filter the text through an external command type :!sort %.

To insert the output of the external command in the current file type :r!command (for example :r!which ls).

Search for “filter” for more information :h filter.

Searching and Replacing

Searching in Vim is very easy. Type / in the command mode and insert the term you search, and Vim will search the file (in forward direction) for the term. Use ? for the backward direction. Using n or N you can repeat the search in the same or opposite direction.

If the option “incsearch” is set, Vim immediately jumps to the matching text when you enter something. If “hlsearch” is set, it highlights all matches. To remove the highlight type :nohl.

Replacing something isn’t very hard too, but you should have a good understanding of regular expressions.

To substitute a regular expression with some other text, type :%s/old/new/gc this command takes the whole file %, and substitutes s the word "old@ with “new” and looks for more than one occurrence within one line g and asks if it really should replace the shown one c.

To replace some text only in a selected area, select the area, and type :s/old/new/g. This should look like :’<,’>s/old/new/g in the command line. You’ll understand ’< and ’> after the “Marks” section.

Completion

While you are typing, it’s pretty common to use the same word over and over again. Using Ctrl-p Vim searches the currently typed text backwards for a word starting with the same characters as already typed. Ctrl-x Ctrl-l completes the whole line.

If you’re not sure how to type some word and you’ve enabled spell-checking (:set spell), you can type Ctrl-x Ctrl-k to do a dictionary lookup for the already typed characters. Vim’s completion system has much improved during the last major update (Vim 7.0).

Note the completion commands *work only in the insert mode*, they have other meanings in the command mode!

Macros

Vim allows to replay some commands using . (a dot). For more than one command use macros.

You can start macro-recording using q and one of {0-9a-zA-Z}, so for example qq records the macro to buffer “q”. Hit q when you are finished recording.

Now you can replay the macro at any time using @q.

To be explored in the future

- Marks in Vim
- Tabs, windows, buffers in Vim

11.19 平常心

“我始终认为胜负全在运气，运气好就赢，运气不好就会输。围棋不是想赢就能赢的，外界总是在骚动，我的心如果因为这种骚动而动摇，就会输棋。幸好当时无论对局前，还是对局中，我的心情都很平静。”

11.23 Be Fearless

《坂上之云》网文一篇

“坂上之云”本身意思是“顺着山坡（坂）上升的云”，折射日本在明治维新时期奋发图强，学习追赶西方列强，国力不断增强的情景。

变革时代的日本年轻人为了自己的梦想向前向上，走出自己或长或短的壮丽人生。看这部司马辽太郎的大河剧（幸好不长，不拖沓）可以结合着梁启超的《少年中国说》来印证。

中国的航母今天下水了，我不知道现在中国有多少青少年像秋山兄弟那样“单纯”的关心国家海防的。

小时候看过一部日本电视剧《烈驹》，女主人公让人眼前一亮，仿佛远处的海风直接吹到了自己的脸上，沁入自己的心脾，于是冲动了，要努力走出去看看这个大千世界。当时还有国产辛亥革命题材的《海龙?》、《黄兴》真是让人看后热血沸腾。而今很少能见到此类奋发向上，清新锐利的作品了。多数影视文学作品表现的是迷茫、狂躁、缠绵、揶揄。绿江南的潇潇细雨，渔舟唱晚，暮鼓晨钟；东国的苍浪排空，樱飞如雪；大漠草原上的晴空白云，骏马羊群；冰峰雪山的瑰丽神光，天籁传音；北国长白山的林海雪原，探宝寻珍。这些景象在我的记忆里是那样的真实和完美。

时间如飞流逝，正如织田信长特别喜欢幸若舞《敦盛》的其中一节“人间五十年，跟天下比起来，如梦似幻，人生一度得生，焉有长生不灭者?”。浪花淘尽多少英雄，古今天下事，都付笑谈中。

（自注：蓝字句犹有同感）

秋山真之和秋山好古轶事

- （好古）生活十分简朴。吃饭时一直以腌萝蔔佐膳。真之来家里借住的时候也只有一个碗，自己吃完把碗借给弟弟。不舍得穿袜子。
- 在陆军大学教课时，为了向学生们说明骑兵的特点，竟一拳打破窗户玻璃，满手是血的向学生们说道：“这就是骑兵！”
- 秋山（真之）小时候很调皮，领着邻居小孩玩打仗游戏。学着书里自己制作焰火。由于太顽皮母亲贞拔出刀说“お前も杀して私も死ぬ”（杀了你我也自杀算了），秋山才有所收敛。喜欢画图，游泳和跑步。
- （真之）候补生时期，后辈问他为什么不见他用功成绩还是那么好。秋山回答参考过去考试出的题目，猜测教官的出题心理。
- 喜欢用军服的袖口擦鼻涕，想作战方案时可以几天不洗澡忘我投入。
- 从美国归国途中遭遇赌博骗子。输光的秋山用刀逼骗子把骗去的钱还了出来。

联合舰队解散训示

二十月余征战已成往事，我们联合舰队的使命已经完成，由此解散。但是我们作为海军军人的责任感却不能因此松懈。为了保护胜利果实，为了带给国家一个繁荣昌盛的未来，不论是战是和，海军必须站在帝国与外来冲击之间，必须一致保存它在海上的实力以面对不测。

这种实力不单单是蕴含在军舰与装备之中，它更包含在运用这些器物的无形能力里。当我们明白了百发百中之一炮可敌百发一中之百炮后，我等军人便不得不求武力于形而上。近期海军的胜利极大程度上要归功于我们为了获胜而进行的日常训练。

继往开来，尽管战争暂歇，我们却不可放纵于轻松。想来武人之一生应对连绵不断之战争，不由一时之停战而使其职责有分轻重之理。有事，则发挥武力；无事，则修养之。始终一贯，唯尽其本分而已。

过去的一年半时间里我们从没遇过轻松的任务，我们战风斗浪，敌寒御暑，与顽敌死战以争夺海权；而今回顾，这只是总战略里的一次演习而已，我们于此获得的启发带来的快乐，战争的劳苦与之相比是那么无意义。

如果自称军人却只愿享受和平时的愉悦，那么不论战争机器外表如何光鲜，也不过是沙上之塔，在风暴面前只会轰然倒塌。往昔，神功皇后征服三韩以来，四百多年间韩国曾在我国统治下，结果海军衰落时一朝皆失。又如德川幕府时我国武备松弛，美国仅几艘战舰便让我方无法应对，同样在千岛群岛和库页岛我们也无法进行抵抗俄舰。另一方面，当我们回顾西洋的历史，可以看到 19 世纪初英国海军通过在尼罗河与特拉法尔加获得的胜利不但使得国家安全稳如泰山，保持实力随着世界的进步而提升，更在当前时代保护国家利益并扩张国势。

以上，无论古代今世、不分西洋东洋，尽管一些是政治所导致的，仍然得承认，主要的战果都取决于士兵和时能否备战。我们这些幸存的军人应当牢记战例，并在将来的训练时将获得的经验加入并磨练，更为将来不落于人后的发展而探索。

只有平时遵照圣谕，用心勤勉服役，才能保持饱满的实力以待时局，履行永远保卫国家的使命。神明，仅向那些平素勤加锻炼，无需战斗，便已胜人一筹之人授予胜利之桂冠的同时，对于那些满足於一时之胜利，安于太平之人，将立即予以剥夺。

古训曰：胜者尤须系盔绪。

明治 *38* 年 *12* 月 *21* 日联合舰队司令东乡平八郎

Teach Yourself Programming in Ten Years(Peter Norvig)

- Get interested in programming, and do some because it is fun. Make sure that it keeps being enough fun so that you will be willing to put in your ten years/10,000 hours.

- **Program**. The best kind of learning is learning by doing. To put it more technically, "the maximal level of performance for individuals in a given domain is not attained automatically as a function of extended experience, but the level of performance can be increased even by highly experienced individuals as a result of deliberate efforts to improve." and "the most effective learning requires a well-defined task with an appropriate difficulty level for the particular individual, informative feedback, and opportunities for repetition and corrections of errors."

- **Talk with** other programmers; read other programs. This is more important than any book or training course.

- If you want, put in four years at a college (or more at a graduate school). This will give you access to some jobs that require credentials, and it will give you a deeper understanding of the field, but if you don't enjoy school, you can (with some dedication) get similar experience on your own or on the job. In any case, book learning alone won't be enough. "Computer science education cannot make anybody an expert programmer any more than studying brushes and pigment can make somebody an expert painter" says Eric Raymond, author of *The New Hacker's Dictionary.* One of the best programmers I ever hired had only a High School degree; he's produced a lot of great software, has his own news group, and made enough in stock options to buy his own nightclub.

- **Work on projects with** other programmers. Be the best programmer on some projects; be the worst on some others. When you're the best, you get to test your abilities to lead a project, and to inspire others with your vision. When you're the worst, you learn what the masters do, and you learn what they don't like to do (because they make you do it for them).

- **Work on projects after** other programmers. Understand a program written by someone else. See what it takes to understand and fix it when the original programmers are not around. Think about how to design your programs to make it easier for those who will maintain them after you.

- Learn at least a half dozen programming languages. Include one language that emphasizes class abstractions (like Java or C++), one that emphasizes functional abstraction (like Lisp or ML or Haskell), one that supports syntactic abstraction (like Lisp), one that supports declarative specifications (like Prolog or C++ templates), and one that emphasizes parallelism (like Clojure or Go).

- Remember that there is a "computer" in "computer science". Know how long it takes your computer to execute an instruction, fetch a word from memory (with and without a cache miss), read consecutive words from disk, and seek to a new location on disk.

With all that in mind, its questionable how far you can get just by book learning. Before my first child was born, I read all the How To books, and still felt like a clueless novice. 30 Months later, when my second child was due, did I go back to the books for a refresher? No. Instead, I relied on my personal experience, which turned out to be far more useful and reassuring to me than the thousands of pages written by experts.

Fred Brooks, in his essay No Silver Bullet identified a three-part plan for finding great software designers:

1. Systematically identify top designers as early as possible.
2. Assign a career mentor to be responsible for the development of the prospect and carefully keep a career file.
3. Provide opportunities for growing designers to interact and stimulate each other.

This assumes that some people already have the qualities necessary for being a great designer; the job is to properly coax them along. *Alan Perlis put it more succinctly: "Everyone can be taught to sculpt: Michelangelo would have had to be taught how not to. So it is with the great programmers"*. Perlis is saying that the greats have some internal quality that transcends their training. But where does the quality come from? Is it innate? Or do they develop it through diligence? As Auguste Gusteau (the fictional chef in Ratatouille) puts it, "*anyone can cook, but only the fearless can be great.*" I think of it more as willingness to devote a large portion of one's life to deliberative practice. But maybe *fearless* is a way to summarize that. Or, as Gusteau's critic, Anton Ego, says: "Not everyone can become a great artist, but a great artist can come from anywhere."

So go ahead and buy that Java/Ruby/Javascript/PHP book; you'll probably get some use out of it. But you won't change your life, or your

real overall expertise as a programmer in 24 hours or 21 days. How about working hard to continually improve over 24 months? Well, now you're starting to get somewhere...

11.25

柯南 513-514 充满杀意的咖啡香

有点虐心，有点虐心。

11.26 吕世浩讲《史记》

11.30 兄弟生离两行泪，兄弟死别一枝花

自由之魂——严冬冬

- "Now you can fight, now you can lose, now you can choose. That's the measure of a man." - Elton John ///The measure of a man, or of Man, Mankind as something unique?
- 前半年着力培养的是*“死磕精神”/投入度/commitment*，同一种东西的不同叫法。矫枉有点过正，被玉龙当头一棒，之后在上日乌且的大本营，Kyle 跟我聊起*目的因（motivation）*高于投入度的话题：首先要有目的因的支撑，也就是“为什么要去跟这座山死磕”，然后才能够投入，并且投入才有意义。目的因最基本的当然是攀登欲望本身，受到身体状态和一些客观因素的影响。还记得我开篇时的措辞吗？“想登山的时候就去登山”——单在字面上就有另一重意思：不想爬的时候就歇着。
- "To strive, to seek, to find, and never, ever to yield!"

December

12.5

藤师新作《望江南》三首

西湖月，浩蕩出雲中。兩漿煙波入裡海，半池荷叶動秋風。此景與誰同？

東山路，濃霧滿東山。車水馬龍俱不見，霧燈雙閃莫輕關。生死一線間。

林中筑，心遠地自偏。為烤麵包修爐灶，欲知辛苦下農田。耕讀樂無邊。

12.7 CFA 结束，新阶段开始

小结

应该说，自己准备 CFA 还是有用心，不过时间计划的不充分，基本是十一月中旬才开始集中投入时间。在准备的过程中有时候状态好能完成一天 100 页的计划，有时候状态差就只能翻个 30 页（主要是觉得很多东西需要背，很多单词还不知道意思，这读着读着积极性就低了，然后开始玩其他的 = =）。最后没有完成复习计划（原计划是能仔细过一遍而且还有四五天做题，最后一遍都没过完，有三分之一的内容是匆匆看了看上阵的），比较可惜。

经此一役，我会更注意设定计划后时间的安排，有时候不是说你自己觉得一天哪都不去集中时间能有个 12 小时花在读书上就能完成 10 页/hour，一天 120 页的计划的。非常重要的是要考虑到做一件事效率可能

的波动和变化。当然这过程中我也意识到如果集中精力去做一件事情，我的时间能有多少：）关键的还是效率啦！

How to determine the encoding scheme of a file? And in Emacs?

Read the first two bytes of the file.
Corresponding encoding and hex codes are as follow:
unicode Little Endian = "\xFF\xFE"
unicode Big Endian = "\xFE\xFF"
utf8 = "\xEF\xBB"
ASCII = straight to content

Quora 和知乎

考虑以后在 quora 和知乎分配差不多的阅读时间。

12.8 Back to the Regression class

GLM and M-est

Need to figure out what the asyptomtic normality result trying to say in today's lecture(lecture 25).

Job Notes(From Renren)

我的专业里绝大部分都是美国人，一点身为中国人的优势都没有。学院里呢，校友会不是没有，但刻意不让学生接触这些资源；校园网上空位不是没有，但投 N 份简历都不见一个回复；Career Center 不是没有，但只帮你改改简历；招聘会也不是没有，但去的结果你懂的。。。所以在各种得不到资源的情况下找工作，只能剑走偏锋了。我虽然被面的经验很少，但是 informational interview 做过近百个，结果是别人主动给我工作机会。我觉得找工作和写文章一样，都是厚积薄发的过程，所以把一些我一些正面反面的经验总结一下，希望对以后要找工作的人略有帮助。

相信很多人都听到过“找工作靠 networking”的说法。所有前辈和经验帖都在不断强调要 networking，但是没什么人说过到底要怎样 networking，尤其是在本身没有地缘人脉又没校友资源的时候怎么 networking.

更新：看到不少人的评论说比如太冒险，或者太耗时，或者经济代价大（因为有时候我是趁假期飞到其他城市去谈的）云云——统统是为自己不做找借口！我只讲一句关于为什么这个方法最好：因为你可以在开始工作前就了解那个职位、那个公司、那个行业，比同期开始工作的人起点高很多，以后对你职业发展自然有利，况且你还可以选自己喜欢的工作，而不是到了要用 OPT 找工作的时候将就地乱投一通。

1. 预备工作 －在大二下、大三上的时候确定想了解的行业，并加入这个行业的一个专业组织。

很多时候找不到工作是因为一种供求错位。比如我读传媒会理所当然想进广告公司，但广告公司缺的是信息技术和金融的人才，而这两类人才多半从来不考虑申广告公司的职位。所以早一点开始了解行业和人才需求，可以提早准备相关的技能和经验。

加入一个组织的目的是可以从这个组织里开始 networking. 而且以后和专业人士谈的时候，他们多半会问你这个问题：“有没有在哪个组织里啊？”如果你说有，代表你很主动地想进这个行业，有上进心。所谓专业组织，比如营销的 AMA，公共的 PRSSA，广告的 AAAA 和 AAF 等等。

－大三升大四的暑假做一个实习。

其实实习 paid 不 paid 无所谓，公司大小也无所谓，大的知名的当然好，但小的有很多锻炼的机会，而且同事之间的关系也会更近。实习的好处除了可以放在简历上，更重要的时候以后和人 networking 的时候有谈资。因为他们 100% 会问到你做了什么实习这个问题。

2. 如何找人？ 大四上，或者大三下是 networking with professional 的最佳时间。

我试下来最好的办法是 LinkedIn （可能不同行业情况不同，营销公共的在 LinkedIn 上很活跃）. 在“高级搜索”里输想了解的公司和城市（用邮编），然后就会出现一大堆在那个公司工作的人。找 VP 那一层的人。因为——这是我导师告诉我的——CEO 一般比较难约，VP 以下的人太忙于做事了，也没空和你谈。我曾经发邮件给一个 CEO，直到两个月以后他秘书回我说有空可以见面，我都快不记得这事了。

然后记下来名字，去公司主页里找邮箱。一般美国公司有两种邮箱地址，拿我的名字举例，yiqi.chen@ XXX.com OR ychen@ XXX.com 。在 contact us 里能找到一个公关的邮箱地址，照那个人的邮箱格式改就可以了。

然后发邮件给想谈的人。最佳发送时间是周二上午 8 — 9 点，或者下午 2 — 3 点——这个也是我导师告诉我的，因为周一周三一般人忙着回邮件。

邮件要 1. 简洁明了。据说一般工作的人每天收到 200 封邮件，没人有空看大段大段的字。但是目的要说得明确，没人有空猜你想干嘛。

2. 有礼貌。打学生牌的话，一般人还是挺愿意帮忙的。

以下是个我惯用的模板：

Dear XXX

My name is _____ and I am a Senior in (major) at (school). I have passion in (field of study) and career goals in (industry). I saw that you have abundant experiences in (expertise). At this point, it would be very helpful if I could ask you for some advice. I'm very interested in learning more about what might be best to develop this experience.

Would it be possible to speak with you - at your convenience - to ask a few specific questions? My schedule is very flexible.

In the meantime, thank you for being willing to share your thoughts, experience and time.

Best regards,

XXX

有多少人会回呢？差不多我每周发十封，1 — 2 个人回。但我会第二个星期二再 follow up（这词一定要写在标题里！）一下，至少 5 个人回，可以和 3 个人定下时间地点谈。

约在哪里谈？尽量让对方定时间地点。因为你留给他的选择余地越大，越有机会谈。我的经验是，地方大部分在办公室或者公司的会议室，偶尔会在 Starbucks 或者 Panera 喝咖啡。咖啡是对方买的（大家都理解大学生穷）。但如果第一次见面就说吃午饭吃晚饭的话要小心一点。

3. 谈什么？ 首先当然是你对他的个人经历和公司情况都要有了解，最好对行业也有了解——这时候之前的积累就派上用场了。然后准备十个问题。

越宽泛越好，最好不要问太多公司的情况。目的其实不一定是要得到答案，而是要让他多说。因为他说得越多，心理上对你越信任。

比如常见的：

1. How did you get started in (the area)?

2. What do you enjoy most about your job?

3. What significant changes have you seen in the industry in the last 5 years?

4. What do you think would be the coming trends in (field) in five years?

5. People in your position who haven't succeeded, what were some of the reasons that lead to their lack of success? （这个是我导师推荐的，特别 aggressive，我一般不太敢问）

6. When you took this position, what was the biggest surprise you encountered?

同时还要穿插地讲一些自己的情况，至少提到一点个性，一点经历和一点特长，这样他比较容易记住你。

最后一定要问 Can I keep in touch with you/ add you on LinkedIn? Any person you recommend me to talk to? 以及 Can you put me into contact with him/her? 这三个就是 networking 的重点！

一次会面一般半小时左右。那些人平时工作很忙的，所以要尽量事先做足功课，对得起他花这半小时来和你谈。

4. 重中之重！ 聊完一定要马上回家写 email 感谢（或者趁加 LinkedIn 的时候留言，一举两得），同时寄一张手写的感谢卡！有时候我会自己做感谢卡寄。

（以防有人问感谢卡怎么写，这个也是有模式滴～）

Dear XXX

Thank you very much for meeting with me! I really enjoy the conversation with you. 然后写两句对话中的细节，说你如何惊讶，或者之前没意识到，或者恍然大悟云云。加上两句你以后将会怎样怎样（说明他的建议你不但记住了，而且会有行动）。最后 hope you have a wonderful day.

如果那人发邮件给你和他介绍的人，你的第一份新邮件一定要 cc 他。然后在和那个人见过面以后再回邮件感谢他的介绍。

这就是认识的人怎样倍数级增长的方法。

5. 隔一两个月邮件问候一下，汇报下近况什么的，保持联系。

6. 关于为什么高层愿意和你谈 其一，因为人总喜欢有人听自己的经历，这是人之常情。只要你表现出听的兴趣，而且礼貌又得体，肯定印象分飙升。

其二，因为美国人认为懂得提携小辈是作为 leader 的一个很重要的品质。他们希望多认识聪明能干的学生，招这样的下属。

其三，高层也会有前浪死在沙滩上的紧迫感，愿意和年轻人谈表示他们时刻愿意学习新观念新思想，很 open-minded.

7）关于这么一次面谈体现了你什么品质

很主动、有勇气、做事积极、肯学习、会思考、勤奋努力、对这个行业有巨大的热情、有规划有目标。。。于是他不招你招谁呢？

8）最后说怎么样让人主动给你工作

这个是模板：

I hope all is well. Thank you for your thoughtful comments and suggestions concerning my career. Attached you will find my resume, which provides details concerning my background. If you happen to come across any potential job opportunities that you believe might be suitable for me (e.g., internships, part-time or full-time employment), please do not hesitate to let me know about them. I very much appreciate your time.

我曾经收到过一家我没申过的公司给我面试。导师说就是要这种效果 XD

有个理论我很认同：越不费精力的申请，HR 拒你越容易。网申是最不挑战胆量的，海投一般几百封都没什么回音。如果申完打电话跟进一下，效果可能会好一点。像这种面谈，虽然一开始会很怕很紧张，但做多了也就习惯了。不可否认有时候会很僵或者尴尬，这种话不投机、气场不合的情况在和人打交道的时候肯定会碰到的，但千万不要因为一次有挫败感就放弃。大部分谈话还是很愉快的。

祝大家都收到满意的 offer~

Python in Linux

Check installed python modules

Use python-pip:
sudo apt-get install python-pip
pip freeze
Or in python, enter:
help("modules")

Torrenting Software on Linux

Transmission(default on Ubuntu) and qBittorrent

Can not delete a dir with "rm -rf"

Tonight I encounter a problem that the ubuntu fails to delete a dir with "rm -rf", it kepts on returning the error saying the dir is not empty and when I check it in the terminal it turns out that some hidden file .fuse* would keep on regenerating when you try to remove it. Here are some notes I found online:

"Files of the form .fuse_hidden* are created by FUSE filesystems when a file is deleted but still in use somewhere and must still have a directory entry. This is similar to .nfs* files on directories exported over NFS.

Run df -T . to see the type of filesystem that's mounted on the current directory and its mount point. For an external hard disk, chances are that this is an NTFS filesystem mounted through the NTFS-3G driver, which is base on FUSE.

The name is a fake name that the filesystem driver invents for a deleted file. You can't delete the file (or rather, if you create the file, it reappears under another name). You can't delete the directory either, since it isn't empty. You'll need to find what is using this file. The most likely cause of being in use is if it's open by some application. Run lsof /media/mount-point where /media/mount-point is the filesystem mount point and look for an open file in that directory."

Note: lsof means "list open files"

12.10 习惯，另一个视点

"Missing System Settings" problem solved

The following answer worked for me:

"You may have accidentally removed some packages (or some dependencies which caused the package to uninstall). In any case, you may try installing (or re installing) the package ubuntu-desktop. Correct any accidental package uninstallation by:

sudo apt-get install ubuntu-desktop"

习惯按住背着你偷偷逃走的时间 (张佳玮）

康德先生的时间表，很是有名：

五点起床。喝茶，抽烟，备课。

七点到九点上课。

九点到十二点三刻写东西——他著名的三大批判，也就是这会儿折腾出来的。

下午一点到四点午餐，见客人。

四点到五点，著名的"康德出门散步"，镇上的诸位都可以念着他出门的时候，来校准手表时间。

五点到十点，看书。

十点睡觉。

差不多七个小时睡眠。

其他的某些大师们的生活起居，你可以打这儿看见：The Daily Routines of Famous Creative People 巴尔扎克出了名的每天工作 12 小时。但他每天晚上六点到凌晨一点睡觉。七小时。弥尔顿，晚上九点到凌晨四点。七小时。富兰克林，晚上十点到凌晨五点。七小时。卡夫卡凌晨只睡两小时，但他下午睡四小时。值得一提的是：他们大多数生活，都相当规律。

伟大的、异于常人的、被上天眷顾的大师们，也不是一天只睡两小时。当然了：你可以说，莫扎特每天睡五小时。不过，他也只活了 35 岁……鲁迅先生在进入 20 世纪 30 年代后睡眠也很少，只是，他老人家的健康状况，恐怕也不太值得称道……当然，世上总有体质特异之辈，常年缺睡，还能生龙活虎。但人不能总把自己跟特殊例子去比较。所以，时间都去哪儿了呢？

八月在上海时，我有一天突发奇想，做了个掐表试验。四小时里，做什么事，就掐表，记录时间。在我的记忆里，那四个小时，我用来专心致志的看书与写字了。但事后一划拉时间，我发现：真用来看书或写字的时间，大概不到一半；剩下的时间，除了用于掐表计算的零星几分钟，便是：刷网页；看社交网络提醒；和朋友聊微信；喝茶；翻漫画；发“我其实也没什么事情做但就是不想看书或写字”的呆。打个比方：如果把正经做事儿——读书、写东西、写邮件——当作筋肉和骨骼，把其他纾解心情的零散事儿——刷网页、看微信、聊天、打游戏——当作脂肪，那我的时间安排看上去，就是个脂肪率超高的不健康胖子，而且，还是个自以为很精壮的胖子。

相当多数人，大概与我有类似的毛病：时间都花在闲散的漫游里了，而自己犹且不觉。、

当然不能怪人类的感知，这里是印象玩的把戏：人类对时间的印象，很重要的因素之一，是注意力。通常你注意力集中、收集周围信息时，会感到时间变慢；分心，则时间印象会变快。大多数人或者都有类似体验：跑步或看书时，会觉得时间流逝颇慢；跑完步或看书间隙刷一刷社交网络，“怎么半小时就过去啦？”时间是在分心之中流逝的。我们用以闲散游荡的时间，通常远超出我们的想象。最简单的例子：一个人在书架前犹豫“要不要开始看这本书呢”的时间，通常足够你把这本书翻完了。时间就是这样流逝的：犹豫、分心、等候刷新出来的新提醒。实际上，事后你回忆这些浪费掉的时间，不会觉得那很有快感，说不定还大感罪恶，但下一次，你依然会如此选择。这很正常：好逸恶劳、贪吃好色、自由散漫，实在是人类最基本天性。能反其道而行之的人，通常都是用一些自我暗示，扭曲过心理的人。

比如上头所言的：规律生活。

海明威认为规律的生活有利于写作。福楼拜认为规律的生活能让你充满原创天分。类似话语很动人，但也让人觉得有圣徒般的殉道感，“哇好像的确很不错，但很累吧？”实际上，实施起来，其实并没有想象的那么累。

因为人类的身体具有极强的适应性，给予规律的生活，身体就会接受，并在适当的时间，让你进入适当的状态。当然，所谓 21 天可以养成个好习惯也实在有些乐观，伦敦大学的说法是：人类平均要 66 天左右才能养成一个习惯，而且，复杂行为习惯的养成需要更长时间——比如我每天吃一块巧克力可能一天就能养成，每天做 50 个俯卧撑就更花时间些。但一个习惯

养成后，就是养成了。一个隐藏的细节是：人类总觉得自己拥有自由意志，但其实大多数时候，乃是激素的奴隶。生理的规律，会很自然的影响心理。人类又是会自我暗示、给自己找台阶下的动物。比如危桥效应，许多人会自觉接受“我现在状态这么好，都是因为某某好习惯呀”，于是，最初需要的毅力，在后期就习惯成自然了。比如，许多人会感叹“哎呀你能坚持做一些事好久，真的好有毅力”，但当事人一定明白：最初需要的可能是毅力，之后其实是习惯，甚至是这事情本身的快感，在推动着你了。

一个例子：每个人生命里，都有那么一本厚书，开始，你会百般犹豫，千番推诿，而且会找“看不懂”“太闷了”之类的理由，翻两页就算。但熬过最初的开头后，你会进入顺流而下的状态，你会越看越快，欲罢不能，使追读这本书变成生活习惯的一部分，之后，当你不断跟人推荐这本书，念叨“别看开始有些厚，可好看了”时，也常会追悔“我那时候真笨，干嘛要这么拖呢？”

如是，许多时间，其实都浪费在了犹豫不决和散漫之上，而不被当事人所知；许多习惯的养成和自我约束，其实只需要开始时投入一点毅力，不问青红皂白的延续一段时间，然后就会习惯成自然。控制时间和自己，有时并没有想象中那么困难，许多痛苦久了就不再是痛苦了。比如，你只要尝试过两三次“提前解决问题，不再被拖延症纠缠”的快乐，以后就会像个贱骨头一样快乐的追求提前完成，“这样就没有罪恶感了”。你当然可以说这样很扭曲，但身体就是会用快感来反馈这一切的。同样的，压榨时间，开始并不快乐，会闷，会分心，但当你习惯了压榨时间来避免荒废时间的罪恶感，而且尝到一两次延迟满足的快乐后，自然就会欲罢不能了——身体这玩意一开始总会别扭说不要，时间一长，它自己就会乖了。

12.12

小さな恋のうた

広い宇宙の数ある一つ hiroi uchuu no kazoa ru hitotsu
青い地球の広い世界で aoi chikyuu no hiroi sekai de
小さな恋の思いは届く chiisa na koi no omoi wa todoku
小さな岛のあなたのもとへ chiisa na jima no anatanomotoe
あなたと出会い时は流れる anatato deai toki wa nagare ru
思いを込めた手纸もふえる omoi wo kome ta tegami mofueru

いつしか二人互いに响く itsushika futari tagaini hibiku
时に激しく时に切なく toki ni hageshiku toki ni setsuna ku
响きは远く遥か彼方へ hibikiwa tooku haruka kanata e
やさしい歌は世界を変える yasashii uta wa sekai wo kae ru
ほらあなたにとって大事な人ほど hora anatanitode daiji na nin hodo
すぐそばにいるの sugusobaniiruno
ただあなたにだけ届いて欲しい tada anatanidake todoi te hoshii
响け恋の歌 hibike koi no uta
ほらほらほら响け恋の歌 hora hora hora hibike koi no uta
あなたは気づく二人は歩く anatawa kidu ku futari wa aruku
暗い道でも日々照らす月 kurai michi demo hibi tera su gatsu
握りしめた手离すことなく nigiri shimeta te hanasusukotonaku
思いは强く永远誓う omoi wa tsuyoku eien chikau
永远の渊きっと仆は言う eien no fuchi kitto boku wa iu
思い变わらず同じ言叶を omoi kawa razu onaji kodoba wo
それでも足りず soredemo tari zu
涙にかわり喜びになり namida nikawari yorokobi ninari
言叶にできず kodoba nidekizu
ただ抱きしめるただ抱きしめる tada daki shimeru tada daki shimeru
ほらあなたにとって大事な人ほど hora anatanitode daiji na nin hodo
すぐそばにいるの sugusobaniiruno
ただあなたにだけ届いて欲しい tada anatanidake todoi te hoshii
响け恋の歌 hibike koi no uta
ほらほらほら响け恋の歌 hora hora hora hibike koi no uta
梦ならば觉めないで yume naraba same naide
梦ならば觉めないで yume naraba same naide
あなたと过ごした时永远の星となる anatato sugoshita toki eien no hoshi tonaru
ほらあなたにとって大事な人ほど hora anatanitotte daiji na hido hodo
すぐそばにいるの sugusobaniiruno
ただあなたにだけ届いて欲しい tada anatanidake todoi te hoshii
响け恋の歌 hibike koi no uta

ほらあなたにとって大事な人ほど hora anatanitotte daiji na hido hodo

すぐそばにいるの sugusobaniiruno

ただあなたにだけ届いて欲しい tada anatanidake todoi te hoshii

响け恋の歌 hibike koi no uta

ほらほらほら响け恋の歌 hora hora hora hibike koi no uta

中文翻译：

广阔的宇宙当中蓝色的地球的广阔世界小小恋爱的思念传达到在那细小岛屿的你身上和你相遇之时思念的信件与日俱增不知不觉间我们互相影响著时候激烈地时候难过温柔的歌回响到遥远的远方连世界也改变了

对你来说重要的人就是可以一直在身边只是现在我最想送给你的就是唱这恋爱之歌唱这恋爱之歌

你记得在那月亮照射下我们二人漫步的黑暗道上我们紧握著手说要永不分离强烈地发誓永远也要这样就像不变的河川

对於我而言已想不到其他的言词尽管如此也不足够就将它变成眼泪变成喜悦

如果说话会伤害人的话那就改变成拥抱吧改变成拥抱吧

对你来说重要的人就是可以一直在身边只是现在我最想送给你的就是唱这恋爱之歌唱这恋爱之歌

如果梦就此不醒来如果梦就此不醒来能够和你一直一起就像那永恒的星星

对你来说重要的人就是可以一直在身边只是现在我最想送给你的就是唱这恋爱之歌

对你来说重要的人就是可以一直在身边只是现在我最想送给你的就是唱这恋爱之歌唱这恋爱之歌

单元测试 (Unit Testing)

单元测试（又称为模块测试）是针对程序模块（软件设计的最小单位）来进行正确性检验的测试工作。程序单元是应用的最小可测试部件。在过程化编程中，一个单元就是单个程序、函数、过程等；对于面向对象编程，最小单元就是方法，包括基类（超类）、抽象类、或者派生类（子类）中的方法。

局限

测试不可能发现所有的程序错误，单元测试也不例外。按定义，单元测试只测试程序单元自身的功能。因此，它不能发现集成错误、性能问题、或者其他系统级别的问题。单元测试结合其他软件测试活动更为有效。与其它形式的软件测试类似，单元测试只能表明测到的问题，不能表明不存在未测试到的错误。

Debug 中的打印大法

Python 八荣八耻有云，“以打印日志为荣，以单步跟踪为耻”，为何？

答：打印日誌查錯效率是 O(log（n）)，單步跟蹤是 O(n)。况且單步跟蹤還有很多局限性。其实简单说就是打印日志可以二分查找，单步只能一步一步跟踪，所以一个是 O(log n) 一个是 O(n)。

Python 八荣八耻： 以动手实践为荣, 以只看不练为耻; 以打印日志为荣, 以单步跟踪为耻; 以空格缩进为荣, 以制表缩进为耻; 以单元测试为荣, 以人工测试为耻;

以模块复用为荣, 以复制粘贴为耻; 以多态应用为荣, 以分支判断为耻; 以 Pythonic 为荣, 以冗余拖沓为耻; 以总结分享为荣, 以跪地求解为耻。

12.13 正则化到底是什么意思

香港机场到深圳

大巴，乘车地点：香港机场二号客运大楼旅游车总站，电话：852-2261-2566 。时间是 7：00-24：00，到达的口岸有皇岗、深圳湾、沙头角。

船：无须在香港国际机场办理出入境手续或提取行李，於快船的预定出发时间前最少 60 分钟抵达「往深圳」票务柜台；如没有托运行李，只需於 30 分钟前抵达。

机场快线：服务时间由早上 5 时 50 分到凌晨 1 时 15 分，列車每 10 分鐘一班，全线停靠机场、青衣、九龙、香港 4 站，全程最快只需 23 分钟。单程费用为 HK$60-100。机场快线可接驳地铁，再转地铁到罗湖或福田口岸。

机器学习中常常提到的正则化到底是什么意思？

正则化就是对最小化经验误差函数上加约束，这样的约束可以解释为先验知识 (正则化参数等价于对参数引入先验分布)。约束有引导作用，在优化误差函数的时候倾向于选择满足约束的梯度减少的方向，使最终的解倾向于符合先验知识 (如一般的 l-norm 先验，表示原问题更可能是比较简单的，这样的优化倾向于产生参数值量级小的解，一般对应于稀疏参数的平滑解)。同时正则化，解决了逆问题的不适定性，产生的解是存在，唯一同时也依赖于数据的，噪声对不适定的影响就弱，解就不会过拟合，而且如果先验 (正则化) 合适，则解就倾向于是符合真解 (更不会过拟合了)，即使训练集中彼此间不相关的样本数很少。

12.18

"search for program permission denied Rterm" problem solved

The easiest way is to install/update ESS to the newest version. Another solution I found online need some hacking but is more illuminating:

"I just installed R 3.0.0 on a laptop running Windows 7, 64-bit professional, and Emacs 24.3. I am using ESS 12.09-2 (since that is the latest version on the download page, even though the HTML documentation on the download page reports itself as 13.03).

R is *not* installed in the default location (C:\Program Files\R\R-3.0.0) because that makes it more difficult to install new packages. Instead, it is installed in C:\R\R-3.0.0.

My problem: ESS cannot find the default Rterm.exe. It gives an error message: "Searching for program: permission denied, Rterm"

My .emacs file includes the two lines: (setq ess-directory-containing-R "C:") (require 'ess-site)

On the same machine, with the same .emacs file, ESS is able to locate R-2.15.2, which is installed at C:\R\R-2.15.2.

I know how to hack a solution to this problem by editing the ess-site.el file to explicitly define (setq-default inferior-R-program-name "C:\\R\\R-

3.0.0\\bin\\x64\\Rterm.exe") and will use that as a short-term solution. But this is obviously painful to remember to fix whenever I install a new version of R.

What is the correct long-term solution? “

12.20 麻将和数学

麻将 Notes

打麻将分为三个阶段：
初局：是指刚拿到牌及前五巡的阶段；
中局：是指从五巡后到听牌之间的阶段；
末局：则是指从听牌到胡牌或臭庄之间的这一段时间。

1．根据对手出的牌，判断牌型。 最早的 5 张牌很重要，因为在这时候一般都是整理牌型。比如有人打一张 7 条，那么可以判断他手上从 6 条到 9 条都不会有。因为这些条子都可以和 7 条搭在一起。那么可以想象，如果听条子的话，就是 1、4 条、2、5 条、3、6 条。根据对手后来条子打的情况，很快就可以判断出对手需不需要条子，需要什么样的条子。

2．了解习惯 一般了解习惯都是在 7、8 局牌以后。这时候就需要看出对手三人的习惯，根据对手的习惯改变听牌的策略。比如有的人喜欢冒险，总是很容易打出生张；有些人则喜欢保守，宁愿拆牌也不打生张。那么对于保守派，则尽量听熟张。对于冒险派，则无所谓。（因为对手是 3 个人，所以必然会是 2 人 VS1 人或者 3 人同是）

3．学会放弃 用我们的花麻将来说，由于每个人的花不同，造成输给不同的人，输出去的钱财也是不一样的。而本身能赢的钱财也是不同的。那么必然就需要选择。生冷的牌张容易点炮，但不出自己就要把牌拆了，这就是一种选择。而选择的本身应该根据牌局的大小来选择。比如我赢牌就 5 块钱，而点炮则输 15，这时候就要忍耐；反之则可以冒险。

12.22 山东够级

今晚系里大聚餐，来了 14 个人。吃完后到常镭家打牌，第一次接触够级这种玩法，牌运前三把好的一塌糊涂。。

12.23 Floating Numbers Related Issue

Vim Navigation Notes

R 语言精度问题

一个例子：((0.81*0.1)+(0.09*(-0.9)))
另一个：.3-.4+.4；options(digits = 22)；.3-.4+.4
以下是一些 Notes:

- 浮点数的比较用 all.equal 或 identical,all.equal 的精度可以控制到小数点后 7 位，identical 可以控制到 16 位
- 在最后比较的时候可以考虑使用 round
- The function all.equal() compares two objects using a numeric tolerance of .Machine$double.eps ^ 0.5. *If you want much greater accuracy than this you will need to consider error propagation carefully.*
- Read *What Every Computer Scientist Should Know About Floating-Point Arithmetic* at some time

StackOverflow 上的一个回答：

you'll run into floating point issues with == and != in pretty much any other language too. One important aspect of them in R is the vectorization part.

It would be much better to define a new function almostEqual, fuzzyEqual or similar. It is unfortunate that there is no such base function. all.equal isn't very efficient since it handles all kinds of objects and returns a string describing the difference when mostly you just want TRUE or FALSE.

Here's an example of such a function. It's vectorized like ==.

```
almostEqual <- function(x, y, tolerance=1e-8) {
diff <- abs(x - y)
mag <- pmax( abs(x), abs(y) )
ifelse( mag > tolerance, diff/mag <= tolerance, diff <= tolerance)
}
almostEqual(1, c(1+1e-8, 1+2e-8)) # [1] TRUE FALSE
```

It is around 2x faster than all.equal for scalar values, and much faster with vectors.

```
x <- 1
y <- 1+1e-8
system.time(for(i in 1:1e4) almostEqual(x, y)) # 0.44 seconds
system.time(for(i in 1:1e4) all.equal(x, y)) # 0.93 seconds
```

Note For previous code example you need to understand the notion of *relative tolerance* and *abosolute tolerance*.

dput()

dput opens file and deparses the object x into that file. The object name is not written (unlike dump). If x is a function the associated environment is stripped. Hence scoping information can be lost.

Deparsing an object is difficult, and not always possible. With the default control, dput() attempts to deparse in a way that is readable, but for more complex or unusual objects (see dump, not likely to be parsed as identical to the original. Use control = "all" for the most complete deparsing; use control = NULL for the simplest deparsing, not even including attributes.

Note dput is only writing to a low precision. Use save() for writing R objects out exactly.

Loop in a list

By passing a numeric vector to lapply and then using that as an index to extract the elements from the list you want. Sample code:

```
lapply(seq_along(r), function(i)dput(r[i]))
structure(list(a = structure(1:4, unit = "test")), .Names = "a")
structure(list(b = "abc"), .Names = "b")
```

Note seq_along(x) returns a sequence of the same length as x.

12.24 圣诞前夜聚餐

我，冯龙，常镭，侠璐，心言，雅彦，spoon，李承芮

12.29 爬南山＋又一年相聚

昨晚冬冬出乎预料地提议吃饭前先去玩 Mr.X 密室逃脱 (Room 4 黑暗之书)，这个我在美国就想去可一直没去成，所以非常激动。到六点半见了她真的开始玩的时候，因为要抓紧时间，一开始都是跟着她的指示走（她之前玩过一次知道前面几个谜怎么解)，二十分钟就到了拼积木的场景，无奈我太相信她，没有自己开动脑筋思考（这里靠我自己应该能注意到没块文字积木 pattern 的 orientation 是不该变这一点的!)，等到我们拼着拼着觉得始终无法 fit 的时候已只剩十分钟。呼叫求助，店员说“我可以告诉你们 70% 都拼错了，我只有在你们拼出大部分的情况下才能帮你们拼出剩余部分”，真是太坑爹了，而且她们上次 7 个人有三次求助，为神马这次我们两个人只有一次求助机会?！不过，他暗示我们应该按照棺材上的指示拼，于是我们明白这里 orientation 是应该不变的，冬冬一开始试了不太行就把几个积木 pattern 转了转，而后又不记得原来的样子了＝。＝ 到时间走完，我们还终究没能完成。事后总结，主要有两个问题：1． 我没有很主动的思考，其实一旦明白 orientation 不变我觉得我完全能拼出来 2．我们只有两人太少，积木谜题最好在开始房间和墓室里都需要有一人以上，而且人越多需要记忆的文字对应的积木 pattern 越少。就我们俩做起来还是无法多费了很多事。

之后到茂业吃饭，貌似是这么多年第一次就两个人一起吃。边吃烤鱼边聊非常开心（虽然烤鱼我没能吃很多，中午可能吃太多了)，我把 iphone 6+ 送给了她还问了她胎记的事（当年 QQ 签名“愿平安健康”的事情则忘了＝＝)。当时的时间啊你别走那么快多好～

吃完我们一起看电影，徐克导的《智取威虎山》。之前看贾思敏推荐所以让冬冬选了这部看，看完觉得之前的期待高了。这片子确实主旋律了一点，我因为读过原小说，所以在看的时候没有特别入胜的感觉，而且觉得影片加入的柱子母亲的那条线处理的很一般，我看着是觉得可有可无。详细的评价待这两天得空补上～

看完已是十二点多，我们打车先送她回家，到她家门口我把剩下所有的东西都交给了她并招呼她来了个大大的拥抱（原计划是看电影时候找机会牵手的，可她右手实在放的有点距离。。所以下次再把握了，去年到今年这次我已变得更好了，下次也不例外！）。

12.30 回到赣州 + 在电脑上建立 wifi 网络用于手机上网