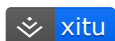


[译] Coding Interview University 一套完整的学习手册帮助自己准备 Google 的面试


- 原文地址: [Coding Interview University](#)

- 原文作者: [John Washam](#)
- 译文出自: [掘金翻译计划](#)
- 译者: [aleen42](#), [Newton](#), [bobmayuze](#), [Jaeger](#), [sqrthree](#)



这是?

这是我为了从 web 开发者（自学、非计算机科学学位）蜕变至 Google 软件工程师所制定的计划，其内容历时数月。

 白板上编程 ———— 来自 HBO 频道的剧集，“硅谷”

这一长列表是从 **Google** 的指导笔记 中萃取出来并进行扩展。因此，有些事情你必须去了解一下。我在列表的底部添加了一些额外项，用于解决面试中可能会出现的问题。这些额外项大部分是来自于 Steve Yegge 的“[得到在 Google 工作的机会](#)”。而在 Google 指导笔记的逐字间，它们有时也会被反映出来。

目录

- [这是?](#)
- [为何要用到它?](#)
- [如何使用它](#)
- [拥有一名 Googler 的心态](#)
- [我得到了工作吗?](#)
- [跟着我的脚步](#)
- [不要妄自菲薄](#)
- [关于 Google](#)
- [相关视频资源](#)
- [面试过程 & 通用的面试准备](#)
- [为你的面试选择一种语言](#)
- [在你开始之前](#)
- [你所看不到的](#)
- [日常计划](#)
- [必备知识](#)
- [算法复杂度 / Big-O / 渐进分析法](#)
- [数据结构](#)
 - [数组 \(Arrays\)](#)
 - [链表 \(Linked Lists\)](#)
 - [堆栈 \(Stack\)](#)
 - [队列 \(Queue\)](#)

- 哈希表 (Hash table)
- 更多的知识
 - 二分查找 (Binary search)
 - 按位运算 (Bitwise operations)
- 树 (Trees)
 - 树 —— 笔记 & 背景
 - 二叉查找树 (Binary search trees) : BSTs
 - 堆 (Heap) / 优先级队列 (Priority Queue) / 二叉堆 (Binary Heap)
 - 字典树 (Tries)
 - 平衡查找树 (Balanced search trees)
 - N 叉树 (K 叉树、M 叉树)
- 排序
- 图 (Graphs)
- 更多知识
 - 递归
 - 动态规划
 - 组合 & 概率
 - NP, NP-完全和近似算法
 - 缓存
 - 进程和线程
 - 系统设计、可伸缩性、数据处理
 - 论文
 - 测试
 - 调度
 - 实现系统例程
 - 字符串搜索和操作
- 终面
- 书籍
- 编码练习和挑战
- 当你临近面试时
- 你的简历
- 当面试来临的时候
- 问面试官的问题
- 当你获得了梦想的职位

----- 下面的内容是可选的 -----

- 附加的学习
 - Unicode
 - 字节顺序
 - Emacs and vi(m)
 - Unix 命令行工具
 - 信息资源 (视频)
 - 奇偶校验位 & 汉明码 (视频)
 - 系统熵值 (系统复杂度)
 - 密码学
 - 压缩

- 网络 (视频)
- 计算机安全
- 释放缓存
- 并行/并发编程
- 设计模式
- 信息传输, 序列化, 和队列化的系统
- 快速傅里叶变换
- 布隆过滤器
- van Emde Boas 树
- 更深入的数据结构
- 跳表
- 网络流
- 不相交集 & 联合查找
- 快速处理数学
- 树堆 (Treap)
- 线性规划
- 几何: 凸包 (Geometry, Convex hull)
- 离散数学
- 机器学习
- Go 语言
- 一些主题的额外内容
- 视频系列
- 计算机科学课程

为何要用到它？

我一直都是遵循该计划去准备 Google 的面试。自 1997 年以来，我一直从事于 web 程序的构建、服务器的构建及创业型公司的创办。对于只有着一个经济学学位，而不是计算机科学学位（CS degree）的我来说，在职业生涯中所取得的都非常成功。然而，我想在 Google 工作，并进入大型系统中，真正地去理解计算机系统、算法效率、数据结构性能、低级别编程语言及其工作原理。可一项都不了解的我，怎么会被 Google 所应聘呢？

当我创建该项目时，我从一个堆栈到一个堆都不了解。那时的我，完全不了解 Big-O、树，或如何去遍历一个图。如果非要我去编写一个排序算法的话，我只能说我所写的肯定是很糟糕。一直以来，我所用的任何数据结构都是内建于编程语言当中。至于它们在背后是如何运作，对此我一概不清楚。此外，以前的我并不需要内存进行管理，最多就只是在一个正在执行的进程抛出了“内存不足”的错误后，采取一些权变措施。而在我的编程生活中，也甚少使用到多维数组，可关联数组却成千上万。而且，从一开始到现在，我都还未曾自己实现过数据结构。

就是这样的我，在经过该学习计划后，已然对被 Google 所雇佣充满信心。这是一个漫长的计划，以至于花费了我数月的时间。若您早已熟悉大部分的知识，那么也许能节省大量的时间。

如何使用它

下面所有的东西都只是一个概述。因此，你需要由上而下逐一地去处理它。

在学习过程中，我是使用 GitHub 特殊的语法特性 markdown flavor 去检查计划的进展，包括使用任务列表。

- ☒ 创建一个新的分支，以使得你可以像这样去检查计划的进展。直接往方括号中填写一个字符 x 即可：
[x]

更多关于 [Github-flavored markdown](#) 的详情

拥有一名 Googler 的心态

把一个（或两个）印有“[future Googler](#)”的图案打印出来，并用你誓要成功的眼神盯着它。



我得到了工作吗？

我还没去应聘。

因为我离完成学习（完成该疯狂的计划列表）还需要数天的时间，并打算在下周开始用一整天的时间，以编程的方式去解决问题。当然，这将会持续数周的时间。然后，我才通过使用在二月份所得到的一个介绍资格，去正式应聘 Google（没错，是二月份时就得到的）。

感谢 JP 的这次介绍。

跟着我的脚步

目前我仍在该计划的执行过程中，如果你想跟随我脚步去学习的话，可以登进我在 [GoogleyAsHeck.com](#) 上所写的博客。

下面是我的联系方式：

- Twitter: [@googleyasheck](#)
- Twitter: [@StartupNextDoor](#)
- Google+: [+Googleyasheck](#)
- LinkedIn: [johnawasham](#)



不要妄自菲薄

- Google 的工程师都是才智过人的。但是，就算是工作在 Google 的他们，仍然会因为觉得自己不够聪明而感到一种不安。
- [天才程序员的神话](#)

关于 Google

- ☐ 面向学生 —— [Google 的职业生涯：技术开发指导](#)
- ☐ Google 检索的原理：
 - ☐ [Google 检索的发展史（视频）](#)
 - ☐ [Google 检索的原理 —— 故事篇](#)
 - ☐ [Google 检索的原理](#)
 - ☐ [Google 检索的原理 —— Matt Cutts（视频）](#)

- ☐ [Google](#) 是如何改善其检索算法（视频）
- ☐ 系列文章：
 - ☐ [Google](#) 检索是如何处理移动设备
 - ☐ [Google](#) 为了寻找大众需求的秘密研究
 - ☐ [Google](#) 检索将成为你的下一个大脑
 - ☐ [Demis Hassabis](#) 的心灵直白
- ☐ 书籍：[Google](#) 公司是如何运作的
- ☐ 由 [Google](#) 通告所制作 —— 2016年10月（视频）

相关视频资源

部分视频只能通过[Coursera](#)、[Edx](#) 或 [Lynda.com class](#) 上注册登录才能观看。这些视频被称为网络公开课程（MOOC）。即便是免费观看，部分课程可能会由于不在时间段内而无法获取。因此，你需要多等待几个月。

很感谢您能帮我把网络公开课程的视频链接转换成公开的视频源，以代替那些在线课程的视频。此外，一些大学的讲座视频也是我所青睐的。

面试过程 & 通用的面试准备

- ☐ 视频：
 - ☐ 如何在 [Google](#) 工作 —— 考生指导课程（视频）
 - ☐ [Google](#) 招聘者所分享的技术面试小窍门（视频）
 - ☐ 如何在 [Google](#) 工作：技术型简历的准备（视频）
- ☐ 文章：
 - ☐ 三步成为 [Googler](#)
 - ☐ 得到在 [Google](#) 的工作机会
 - 所有他所提及的事情都列在了下面
 - ☐ ([早已过期](#)) 如何得到 [Google](#) 的一份工作，面试题，应聘过程 ☐ 电话面试的问题
- ☐ 附加的（虽然 [Google](#) 不建议，但我还是添加在此）：
 - ☐ [ABC](#)：永远都要去编程（[Always Be Coding](#)）
 - ☐ 四步成为 [Google](#) 里一名没有学位的员工
 - ☐ 共享白板（[Whiteboarding](#)）
 - ☐ [Google](#) 是如何看待应聘、管理和公司文化
 - ☐ 程序开发面试中有效的白板（[Whiteboarding](#)）
 - ☐ 震撼开发类面试 第一集：
 - ☐ [Gayle L McDowell](#) —— 震撼开发类面试（视频）
 - ☐ 震撼开发类面试 —— 作者 [Gayle Laakmann McDowell](#)（视频）
 - ☐ 如何在世界四强企业中获得一份工作：
 - ☐ “如何在世界四强企业中获得一份工作 —— [Amazon](#)、[Facebook](#)、[Google](#) 和 [Microsoft](#)”（视频）
 - ☐ 面试 [Google](#) 失败

为你的面试选择一种语言

在这，我就以下话题写一篇短文——[重点：为在 Google 的面试选择一种语言](#)

在大多数公司的面试当中，你可以在编程这一环节，使用一种自己用起来较为舒适的语言去完成编程。但在 Google，你只有三种固定的选择：

- C++
- Java
- Python

有时你也可以使用下面两种，但需要事先查阅说明。因为，说明中会有警告：

- JavaScript
- Ruby

你需要对你所选择的语言感到非常舒适且足够了解。

更多关于语言选择的阅读：

- <http://www.byte-by-byte.com/choose-the-right-language-for-your-coding-interview/>
- <http://blog.codingforinterviews.com/best-programming-language-jobs/>
- <https://www.quora.com/What-is-the-best-language-to-program-in-for-an-in-person-Google-interview>

[在此查看相关语言的资源](#)

由于，我正在学习C、C++ 和 Python。因此，在下面你会看到部分关于它们的学习资料。相关书籍请看文章的底部。

在你开始之前

该列表已经持续更新了很长的一段时间，所以，我们的确很容易会对其失去控制。

这里列出了一些我所犯过的错误，希望您不要重蹈覆辙。

1. 你不可能把所有的东西都记住

就算我查看了数小时的视频，并记录了大量的笔记。几个月后的我，仍然会忘却其中大部分的东西。所以，我翻阅了我的笔记，并将可回顾的东西制作成抽认卡（flashcard）（请往下看）

2. 使用抽认卡

为了解决善忘的问题，我制作了一些关于抽认卡的页面，用于添加两种抽认卡：正常的及带有代码的。每种卡都会有不同的格式设计。

而且，我还以移动设备为先去设计这些网页，以使得在任何地方的我，都能通过我的手机及平板去回顾知识。

你也可以免费制作属于你自己的抽认卡网站：

- [抽认卡页面的代码仓库](#)
- [我的抽认卡数据库](#)：有一点需要记住的是，我做事有点过头，以至于把卡片都覆盖到所有的东西上。从汇编语言和 Python 的细枝末节，乃至到机器学习和统计都被覆盖到卡片上。而这种做法，对于 Google 的要求来说，却是多余。

在抽认卡上做笔记：若你第一次发现你知道问题的答案时，先不要急着把其标注成“已懂”。你需要做的，是去查看一下是否有同样的抽认卡，并在你真正懂得如何解决问题之前，多问自己几次。重复地问答可帮助您深刻记住该知识点。

3. 回顾，回顾，回顾

我留有一组 ASCII 码表、OSI 堆栈、Big-O 记号及更多的小抄纸，以便在空余的时候可以学习。

每编程半个小时就要休息一下，并去回顾你的抽认卡。

4. 专注

在学习的过程中，往往会有许多令人分心的事占据着我们宝贵的时间。因此，专注和集中注意力是非常困难的。

你所看不到的

由于，这个巨大的列表一开始是作为我个人从 Google 面试指导笔记所形成的一个事件处理列表。因此，有一些我熟悉且普遍的技术在此都未被谈及到：

- SQL
- Javascript
- HTML、CSS 和其他前端技术

日常计划

部分问题可能会花费一天的时间去学习，而部分则会花费多天。当然，有些学习并不需要我们懂得如何实现。

因此，每一天我都会在下面所列出的列表中选择一项，并查看相关的视频。然后，使用以下的一种语言去实现：

C — 使用结构体和函数，该函数会接受一个结构体指针 * 及其他数据作为参数。

C++ — 不使用内建的数据类型。

C++ — 使用内建的数据类型，如使用 STL 的 `std::list` 来作为链表。

Python — 使用内建的数据类型（为了持续练习 Python），并编写一些测试去保证自己代码的正确性。有时，只需要使用断言函数 `assert()` 即可。

此外，你也可以使用 Java 或其他语言。以上只是我的个人偏好而已。

为何要在这些语言上分别实现一次？

因为可以练习，练习，练习，直至我厌倦它，并完美地实现出来。（若有部分边缘条件没想到时，我会用书写的形式记录下来并去记忆）

因为可以在纯原生的条件下工作（不需垃圾回收机制的帮助下，分配/释放内存（除了 Python））

因为可以利用上内建的数据类型，以使得我拥有在现实中使用内建工具的经验（在生产环境中，我不会去实现自己的链表）

就算我没有时间去每一项都这么做，但我也会尽我所能的。

在这里，你可以查看到我的代码：

- [C](#)
- [C++](#)
- [Python](#)

你不需要记住每一个算法的内部原理。

在一个白板上写代码，而不要直接在计算机上编写。在测试完部分简单的输入后，到计算机上再测试一遍。

必备知识

- ☐ 计算机是如何处理一段程序：
 - ☐ [CPU 是如何执行代码（视频）](#)
 - ☐ [机器码指令（视频）](#)
- ☐ 编译器
 - ☐ [编译器是如何在 ~1 分钟内工作（视频）](#)
 - ☐ [Harvard CS50 —— 编译器（视频）](#)
 - ☐ [C++（视频）](#)
 - ☐ [掌握编译器的优化（C++）（视频）](#)
- ☐ 浮点数是如何存储的：
 - ☐ [简单的 8-bit：浮点数的表达形式 —— 1（视频 —— 在计算上有一个错误 —— 详情请查看视频的介绍）](#)
 - ☐ [32 bit：IEEE754 32-bit 浮点二进制（视频）](#)

算法复杂度 / Big-O / 渐进分析法

- 并不需要实现
- ☐ [Harvard CS50 —— 渐进表示（视频）](#)
- ☐ [Big O 记号（通用快速教程）（视频）](#)
- ☐ [Big O 记号（以及 Omega 和 Theta）—— 最佳数学解释（视频）](#)
- ☐ Skiena 算法：
 - [视频](#)
 - [幻灯片](#)
- ☐ [对于算法复杂度分析的一次详细介绍](#)
- ☐ [增长阶数（Orders of Growth）（视频）](#)
- ☐ [渐进性（Asymptotics）（视频）](#)
- ☐ [UC Berkeley Big O（视频）](#)
- ☐ [UC Berkeley Big Omega（视频）](#)

- ☐ [平摊分析法 \(Amortized Analysis\)](#) (视频)
- ☐ [举证“Big O”](#) (视频)
- ☐ 高级编程 (包括递归关系和主定理):
 - [计算性复杂度: 第一部](#)
 - [计算性复杂度: 第二部](#)
- ☐ [速查表 \(Cheat sheet\)](#)

如果部分课程过于学术性, 你可直接跳到文章底部, 去查看离散数学的视频以获取相关背景知识。

数据结构

- 数组 (Arrays)
 - 实现一个可自动调整大小的动态数组。
 - ☐ 介绍:
 - [数组 \(视频\)](#)
 - [数组的基础知识 \(视频\)](#)
 - [多维数组 \(视频\)](#)
 - [动态数组 \(视频\)](#)
 - [不规则数组 \(视频\)](#)
 - [调整数组的大小 \(视频\)](#)
 - ☐ 实现一个动态数组 (可自动调整大小的可变数组):
 - ☐ 练习使用数组和指针去编码, 并且指针是通过计算去跳转而不是使用索引
 - ☐ 通过分配内存来新建一个原生数据类型数组
 - 可以使用 `int` 类型的数组, 但不能使用其语法特性
 - 从大小为16或更大的数 (使用2的倍数 —— 16、32、64、128) 开始编写
 - ☐ `size()` —— 数组元素的个数
 - ☐ `capacity()` —— 可容纳元素的个数
 - ☐ `is_empty()`
 - ☐ `at(index)` —— 返回对应索引的元素, 且若索引越界则愤然报错
 - ☐ `push(item)`
 - ☐ `insert(index, item)` —— 在指定索引中插入元素, 并把后面的元素依次后移
 - ☐ `prepend(item)` —— 可以使用上面的 `insert` 函数, 传参 `index` 为 0
 - ☐ `pop()` —— 删除在数组末端的元素, 并返回其值
 - ☐ `delete(index)` —— 删除指定索引的元素, 并把后面的元素依次前移
 - ☐ `remove(item)` —— 删除指定值的元素, 并返回其索引 (即使有多个元素)
 - ☐ `find(item)` —— 寻找指定值的元素并返回其中第一个出现的元素其索引, 若未找到则返回 -1
 - ☐ `resize(new_capacity)` // 私有函数
 - 若数组的大小到达其容积, 则变大一倍
 - 获取元素后, 若数组大小为其容积的1/4, 则缩小一半
 - ☐ 时间复杂度
 - 在数组末端增加/删除、定位、更新元素, 只允许占 $O(1)$ 的时间复杂度 (平摊 (amortized) 去分配内存以获取更多空间)
 - 在数组任何地方插入/移除元素, 只允许 $O(n)$ 的时间复杂度

- ☐ 空间复杂度
 - 因为在内存中分配的空间邻近，所以有助于提高性能
 - 空间需求 = （大于或等于 n 的数组容积）* 元素的大小。即便空间需求为 $2n$ ，其空间复杂度仍然是 $O(n)$
- 链表（Linked Lists）
 - ☐ 介绍：
 - ☐ 单向链表（视频）
 - ☐ CS 61B —— 链表（视频）
 - ☐ C 代码（视频）
 - 并非看完整个视频，只需要看关于节点结果和内存分配那一部分即可
 - ☐ 链表 vs 数组：
 - 基本链表 Vs 数组（视频）
 - 在现实中，链表 Vs 数组（视频）
 - ☐ 为什么你需要避免使用链表（视频）
 - ☐ 的确：你需要关于“指向指针的指针”的相关知识：（因为当你传递一个指针到一个函数时，该函数可能会改变指针所指向的地址）该页只是为了让你了解“指向指针的指针”这一概念。但我并不推荐这种链式遍历的风格。因为，这种风格的代码，其可读性和可维护性太低。
 - [指向指针的指针](#)
 - ☐ 实现（我实现了使用尾指针以及没有使用尾指针这两种情况）：
 - ☐ size() —— 返回链表中数据元素的个数
 - ☐ empty() —— 若链表为空则返回一个布尔值 true
 - ☐ value_at(index) —— 返回第 n 个元素的值（从0开始计算）
 - ☐ push_front(value) —— 添加元素到链表的首部
 - ☐ pop_front() —— 删除首部元素并返回其值
 - ☐ push_back(value) —— 添加元素到链表的尾部
 - ☐ pop_back() —— 删除尾部元素并返回其值
 - ☐ front() —— 返回首部元素的值
 - ☐ back() —— 返回尾部元素的值
 - ☐ insert(index, value) —— 插入值到指定的索引，并把当前索引的元素指向到新的元素
 - ☐ erase(index) —— 删除指定索引的节点
 - ☐ value_n_from_end(n) —— 返回倒数第 n 个节点的值
 - ☐ reverse() —— 逆序链表
 - ☐ remove_value(value) —— 删除链表中指定值的第一个元素
 - ☐ 双向链表
 - [介绍（视频）](#)
 - 并不需要实现
- 堆栈（Stack）
 - ☐ [堆栈（视频）](#)
 - ☐ [使用堆栈 —— 后进先出（视频）](#)
 - ☐ 可以不实现，因为使用数组来实现并不重要
- 队列（Queue）
 - ☐ [使用队列 —— 先进先出（视频）](#)

- ☐ [队列（视频）](#)
- ☐ [原型队列/先进先出（FIFO）](#)
- ☐ [优先级队列（视频）](#)
- ☐ 使用含有尾部指针的链表来实现:
 - `enqueue(value)` —— 在尾部添加值
 - `dequeue()` —— 删除最早添加的元素并返回其值（首部元素）
 - `empty()`
- ☐ 使用固定大小的数组实现:
 - `enqueue(value)` —— 在可容的情况下添加元素到尾部
 - `dequeue()` —— 删除最早添加的元素并返回其值
 - `empty()`
 - `full()`
- ☐ 花销:
 - 在糟糕的实现情况下，使用链表所实现的队列，其入列和出列的时间复杂度将会是 $O(n)$ 。因为，你需要找到下一个元素，以致循环整个队列
 - `enqueue`: $O(1)$ （平摊（amortized）、链表和数组 [探测（probing）]）
 - `dequeue`: $O(1)$ （链表和数组）
 - `empty`: $O(1)$ （链表和数组）
- 哈希表（Hash table）
 - ☐ 视频:
 - ☐ [链式哈希表（视频）](#)
 - ☐ [Table Doubling 和 Karp-Rabin（视频）](#)
 - ☐ [Open Addressing 和密码型哈希（Cryptographic Hashing）（视频）](#)
 - ☐ [PyCon 2010: The Mighty Dictionary（视频）](#)
 - ☐（进阶）[随机取样（Randomization）：全域哈希（Universal Hashing）& 完美哈希（Perfect Hashing）（视频）](#)
 - ☐（进阶）[完美哈希（Perfect hashing）（视频）](#)
 - ☐ 在线课程:
 - ☐ [哈希函数的掌握（视频）](#)
 - ☐ [使用哈希表（视频）](#)
 - ☐ [哈希表的支持（视频）](#)
 - ☐ [哈希表的语言支持（视频）](#)
 - ☐ [基本哈希表（视频）](#)
 - ☐ [数据结构（视频）](#)
 - ☐ [电话簿问题（Phone Book Problem）（视频）](#)
 - ☐ 分布式哈希表:
 - [Dropbox 中的瞬时上传及存储优化（视频）](#)
 - [分布式哈希表（视频）](#)
 - ☐ 使用线性探测的数组去实现
 - `hash(k, m)` —— m 是哈希表的大小
 - `add(key, value)` —— 如果 `key` 已存在则更新值
 - `exists(key)`

- `get(key)`
- `remove(key)`

更多的知识

- 二分查找 (Binary search)
 - [二分查找 \(视频\)](#)
 - [二分查找 \(视频\)](#)
 - [详情](#)
 - [实现](#):
 - 二分查找 (在一个已排序好的整型数组中查找)
 - 迭代式二分查找
- 按位运算 (Bitwise operations)
 - [Bits 速查表](#)
 - 你需要知道大量2的幂数值 (从 2^1 到 2^{16} 及 2^{32})
 - [好好理解位操作符的含义: `&`、`|`、`^`、`~`、`>>`、`<<`](#)
 - [字码 \(words\)](#)
 - [好的介绍: 位操作 \(视频\)](#)
 - [C 语言编程教程 2-10: 按位运算 \(视频\)](#)
 - [位操作](#)
 - [按位运算](#)
 - [Bithacks](#)
 - [位元抚弄者 \(The Bit Twiddler\)](#)
 - [交互式位元抚弄者 \(The Bit Twiddler Interactive\)](#)
 - [一补数和补码](#)
 - [二进制: 利 & 弊 \(为什么我们要使用补码\) \(视频\)](#)
 - [一补数 \(1s Complement\)](#)
 - [补码 \(2s Complement\)](#)
 - [计算置位 \(Set Bits\)](#)
 - [计算一个字节中置位 \(Set Bits\) 的四种方式 \(视频\)](#)
 - [计算比特位](#)
 - [如何在一个 32 位的整型中计算置位 \(Set Bits\) 的数量](#)
 - [四舍五入2的幂数](#):
 - [四舍五入到2的下一幂数](#)
 - [交换值](#):
 - [交换 \(Swap\)](#)
 - [绝对值](#):
 - [绝对整型 \(Absolute Integer\)](#)

树 (Trees)

- 树 —— 笔记 & 背景
 - [系列: 基本树 \(视频\)](#)
 - [系列: 树 \(视频\)](#)

- 基本的树形结构
- 遍历
- 操作算法
- BFS（广度优先检索，breadth-first search）
 - [MIT（视频）](#)
 - 层序遍历（使用队列的 BFS 算法）
 - 时间复杂度： $O(n)$
 - 空间复杂度：
 - 最好情况： $O(1)$
 - 最坏情况： $O(n/2)=O(n)$
- DFS（深度优先检索，depth-first search）
 - [MIT（视频）](#)
 - 笔记：
 - 时间复杂度： $O(n)$
 - 空间复杂度：
 - 最好情况： $O(\log n)$ - 树的平均高度
 - 最坏情况： $O(n)$
 - 中序遍历（DFS：左、节点本身、右）
 - 后序遍历（DFS：左、右、节点本身）
 - 先序遍历（DFS：节点本身、左、右）
- 二叉查找树（Binary search trees）：BSTs
 - ☐ [二叉查找树概览（视频）](#)
 - ☐ [系列（视频）](#)
 - 从符号表开始到 BST 程序
 - ☐ [介绍（视频）](#)
 - ☐ [MIT（视频）](#)
 - C/C++:
 - ☐ [二叉查找树 —— 在 C/C++ 中实现（视频）](#)
 - ☐ [BST 的实现 —— 在堆栈和堆中的内存分配（视频）](#)
 - ☐ [在二叉查找树中找到最小和最大的元素（视频）](#)
 - ☐ [寻找二叉树的高度（视频）](#)
 - ☐ [二叉树的遍历 —— 广度优先和深度优先策略（视频）](#)
 - ☐ [二叉树：层序遍历（视频）](#)
 - ☐ [二叉树的遍历：先序、中序、后序（视频）](#)
 - ☐ [判断一棵二叉树是否为二叉查找树（视频）](#)
 - ☐ [从二叉查找树中删除一个节点（视频）](#)
 - ☐ [二叉查找树中序遍历的后继者（视频）](#)
 - ☐ 实现：
 - ☐ insert // 往树上插值
 - ☐ get_node_count // 查找树上的节点数
 - ☐ print_values // 从小到大打印树中节点的值
 - ☐ delete_tree
 - ☐ is_in_tree // 如果值存在于树中则返回 true
 - ☐ get_height // 返回节点所在的高度（如果只有一个节点，那么高度则为1）
 - ☐ get_min // 返回树上的最小值

- [get_max](#) // 返回树上的最大值
- [is_binary_search_tree](#)
- [delete_value](#)
- [get_successor](#) // 返回给定值的后继者，若没有则返回-1

- 堆（Heap） / 优先级队列（Priority Queue） / 二叉堆（Binary Heap）

- 可视化是一棵树，但通常是以线性的形式存储（数组、链表）
- [堆](#)
- [介绍（视频）](#)
- [无知的实现（视频）](#)
- [二叉树（视频）](#)
- [关于树高的讨论（视频）](#)
- [基本操作（视频）](#)
- [完全二叉树（视频）](#)
- [伪代码（视频）](#)
- [堆排序 —— 跳到起点（视频）](#)
- [堆排序（视频）](#)
- [构建一个堆（视频）](#)
- [MIT：堆与堆排序（视频）](#)
- [CS 61B Lecture 24：优先级队列（视频）](#)
- [构建线性时间复杂度的堆（大顶堆）](#)
- [实现一个大顶堆：](#)
 - [insert](#)
 - [sift_up](#) —— 用于插入元素
 - [get_max](#) —— 返回最大值但不移除元素
 - [get_size\(\)](#) —— 返回存储的元素数量
 - [is_empty\(\)](#) —— 若堆为空则返回 true
 - [extract_max](#) —— 返回最大值并移除
 - [sift_down](#) —— 用于获取最大值元素
 - [remove\(i\)](#) —— 删除指定索引的元素
 - [heapify](#) —— 构建堆，用于堆排序
 - [heap_sort\(\)](#) —— 拿到一个未排序的数组，然后使用大顶堆进行就地排序
 - 注意：若用小顶堆可节省操作，但导致空间复杂度加倍。（无法做到就地）

- 字典树（Tries）

- 需要注意的是，字典树各式各样。有些有前缀，而有些则没有。有些使用字符串而不使用比特位来追踪路径。
- 阅读代码，但不实现。
- [数据结构笔记及编程技术](#)
- [短课程视频：](#)
 - [对字典树的介绍（视频）](#)
 - [字典树的性能（视频）](#)
 - [实现一棵字典树（视频）](#)
- [字典树：一个被忽略的数据结构](#)
- [高级编程 —— 使用字典树](#)

- ☐ [标准教程（现实中的用例）（视频）](#)
- ☐ [MIT，高阶数据结构，使用字符串追踪路径（可事半功倍）](#)
- 平衡查找树（Balanced search trees）
 - 掌握至少一种平衡查找树（并懂得如何实现）：
 - “在各种平衡查找树当中，AVL 树和2-3树已经成为了过去，而红黑树（red-black trees）看似变得越来越受人青睐。这种令人特别感兴趣的数据结构，亦称伸展树（splay tree）。它可以自我管理，且会使用轮换来移除任何访问过根节点的 key。”—— Skiena
 - 因此，在各种各样的平衡查找树当中，我选择了伸展树来实现。虽然，通过我的阅读，我发现在 Google 的面试中并不会被要求实现一棵平衡查找树。但是，为了胜人一筹，我们还是应该看看如何去实现。在阅读了大量关于红黑树的代码后，我才发现伸展树的实现确实会使得各方面更为高效。
 - 伸展树：插入、查找、删除函数的实现，而如果你最终实现了红黑树，那么请尝试一下：
 - 跳过删除函数，直接实现搜索和插入功能
 - 我希望能阅读到更多关于 B 树的资料，因为它也被广泛地应用到大型的数据库当中。
 - ☐ [自平衡二叉查找树](#)
 - ☐ **AVL 树**
 - 实际中：我能告诉你的是，该种树并无太多的用途，但我能看到有用的地方在哪里：AVL 树是另一种平衡查找树结构。其可支持时间复杂度为 $O(\log n)$ 的查询、插入及删除。它比红黑树严格意义上更为平衡，从而导致插入和删除更慢，但遍历却更快。正因如此，才彰显其结构的魅力。只需要构建一次，就可以在不重新构造的情况下读取，适合于实现诸如语言字典（或程序字典，如一个汇编程序或解释程序的操作码）。
 - ☐ [MIT AVL 树 / AVL 树的排序（视频）](#)
 - ☐ [AVL 树（视频）](#)
 - ☐ [AVL 树的实现（视频）](#)
 - ☐ [分离与合并](#)
 - ☐ **伸展树**
 - 实际中：伸展树一般用于缓存、内存分配者、路由器、垃圾回收者、数据压缩、ropes（字符串的一种替代品，用于存储长串的文本字符）、Windows NT（虚拟内存、网络及文件系统）等的实现。
 - ☐ [CS 61B：伸展树（Splay trees）（视频）](#)
 - ☐ [MIT 教程：伸展树（Splay trees）：](#)
 - 该教程会过于学术，但请观看到最后的10分钟以确保掌握。
 - [视频](#)
 - ☐ **2-3查找树**
 - 实际中：2-3树的元素插入非常快速，但却有着查询慢的代价（因为相比较 AVL 树来说，其高度更高）。
 - 你会很少用到2-3树。这是因为，其实现过程中涉及到不同类型的节点。因此，人们更多地会选择红黑树。

- [2-3树的直感与定义（视频）](#)
- [2-3树的二元观点](#)
- [2-3树（学生叙述）（视频）](#)
- [2-3-4树 \(亦称2-4树\)](#)
 - 实际中：对于每一棵2-4树，都有着对应的红黑树来存储同样顺序的数据元素。在2-4树上进行插入及删除操作等同于在红黑树上进行颜色翻转及轮换。这使得2-4树成为一种用于掌握红黑树背后逻辑的重要工具。这就是为什么许多算法引导文章都会在介绍红黑树之前，先介绍2-4树，尽管2-4树在实际中并不经常使用。
 - [CS 61B Lecture 26: 平衡查找树（视频）](#)
 - [自底向上的2-4树（视频）](#)
 - [自顶向下的2-4树（视频）](#)
- [B 树](#)
 - 有趣的是：为啥叫 B 仍然是一个神秘。因为 B 可代表波音（Boeing）、平衡（Balanced）或 Bayer（联合创造者）
 - 实际中：B 树会被广泛适用于数据库中，而现代大多数的文件系统都会使用到这种树（或变种）。除了运用在数据库中，B 树也会被用于文件系统以快速访问一个文件的任意块。但存在着一个基本的问题，那就是如何将文件块 i 转换成一个硬盘块（或一个柱面-磁头-扇区）上的地址。
 - [B 树](#)
 - [B 树的介绍（视频）](#)
 - [B 树的定义及其插入操作（视频）](#)
 - [B 树的删除操作（视频）](#)
 - [MIT 6.851 —— 内存层次模块（Memory Hierarchy Models）（视频）](#)
 - 覆盖有高速缓存参数无关型（cache-oblivious）B 树和非常有趣的数据结构
 - 头37分钟讲述的很专业，或许可以跳过（B 指块的大小、即缓存行的大小）
- [红黑树](#)
 - 实际中：红黑树提供了在最坏情况下插入操作、删除操作和查找操作的时间保证。这些时间值的保障不仅对时间敏感型应用有用，例如实时应用，还对在其他数据结构中块的构建非常有用，而这些数据结构都提供了最坏情况下的保障；例如，许多用于计算几何学的数据结构都可以基于红黑树，而目前 Linux 系统所采用的完全公平调度器（the Completely Fair Scheduler）也使用到了该种树。在 Java 8 中，红黑树也被用于存储哈希列表集合中相同的数据，而不是使用链表及哈希码。
 - [Aduni —— 算法 —— 课程4（该链接直接跳到开始部分）（视频）](#)
 - [Aduni —— 算法 —— 课程5（视频）](#)
 - [黑树（Black Tree）](#)
 - [二分查找及红黑树的介绍](#)
- N 叉树（K 叉树、M 叉树）
 - 注意：N 或 K 指的是分支系数（即树的最大分支数）：
 - 二叉树是一种分支系数为2的树
 - 2-3树是一种分支系数为3的树
 - [K 叉树](#)

排序 (Sorting)

- ☐ 笔记:
 - 实现各种排序 & 知道每种排序的最坏、最好和平均的复杂度分别是什么场景:
 - 不要用冒泡排序 - 大多数情况下效率感人 - 时间复杂度 $O(n^2)$, 除非 $n \leq 16$
 - ☐ 排序算法的稳定性 ("快排是稳定的么?")
 - [排序算法的稳定性](#)
 - [排序算法的稳定性](#)
 - [排序算法的稳定性](#)
 - [排序算法的稳定性](#)
 - [排序算法 - 稳定性](#)
 - ☐ 哪种排序算法可以用链表? 哪种用数组? 哪种两者都可?
 - 并不推荐对一个链表排序, 但归并排序是可行的.
 - [链表的归并排序](#)
- 关于堆排序, 请查看前文堆的数据结构部分。堆排序很强大, 不过是非稳定排序。
- ☐ [冒泡排序 \(video\)](#)
- ☐ [冒泡排序分析 \(video\)](#)
- ☐ [插入排序 & 归并排序 \(video\)](#)
- ☐ [插入排序 \(video\)](#)
- ☐ [归并排序 \(video\)](#)
- ☐ [快排 \(video\)](#)
- ☐ [选择排序 \(video\)](#)
- ☐ 斯坦福大学关于排序算法的视频:
 - ☐ [课程 15 | 编程抽象 \(video\)](#)
 - ☐ [课程 16 | 编程抽象 \(video\)](#)
- ☐ Shai Simonson 视频, [Aduni.org](#):
 - ☐ [算法 - 排序 - 第二讲 \(video\)](#)
 - ☐ [算法 - 排序2 - 第三讲 \(video\)](#)
- ☐ Steven Skiena 关于排序的视频:
 - ☐ [课程从 26:46 开始 \(video\)](#)
 - ☐ [课程从 27:40 开始 \(video\)](#)
 - ☐ [课程从 35:00 开始 \(video\)](#)
 - ☐ [课程从 23:50 开始 \(video\)](#)
- ☐ 加州大学伯克利分校 (UC Berkeley) 大学课程:
 - ☐ [CS 61B 课程 29: 排序 I \(video\)](#)
 - ☐ [CS 61B 课程 30: 排序 II \(video\)](#)

- ☐ [CS 61B 课程 32: 排序 III \(video\)](#)
 - ☐ [CS 61B 课程 33: 排序 V \(video\)](#)
- ☐ - 归并排序:
 - ☐ [使用外部数组](#)
 - ☐ [对原数组直接排序](#)
- ☐ - 快速排序:
 - ☐ [实现](#)
 - ☐ [实现](#)
- ☐ 实现:
 - ☐ 归并: 平均和最差情况的时间复杂度为 $O(n \log n)$ 。
 - ☐ 快排: 平均时间复杂度为 $O(n \log n)$ 。
 - ☐ 选择排序和插入排序的最坏、平均时间复杂度都是 $O(n^2)$ 。
 - ☐ 关于堆排序, 请查看前文堆的数据结构部分。
- ☐ 有兴趣的话, 还有一些补充 - 但并不是必须的:
 - ☐ [基数排序](#)
 - ☐ [基数排序 \(video\)](#)
 - ☐ [基数排序, 计数排序 \(线性时间内\) \(video\)](#)
 - ☐ [随机算法: 矩阵相乘, 快排, Freivalds' 算法 \(video\)](#)
 - ☐ [线性时间内的排序 \(video\)](#)

图 (Graphs)

图论能解决计算机科学里的很多问题, 所以这一节会比较长, 像树和排序的部分一样。

- Yegge 的笔记:
 - ☐ 有 3 种基本方式在内存里表示一个图:
 - ☐ 对象和指针
 - ☐ 矩阵
 - ☐ 邻接表
 - ☐ 熟悉以上每一种图的表示法, 并了解各自的优缺点
 - ☐ 宽度优先搜索和深度优先搜索 - 知道它们的计算复杂度和设计上的权衡以及如何用代码实现它们
 - ☐ 遇到一个问题时, 首先尝试基于图的解决方案, 如果没有再去尝试其他的。
- ☐ Skiena 教授的课程 - 很不错的介绍:
 - ☐ [CSE373 2012 - 课程 11 - 图的数据结构 \(video\)](#)
 - ☐ [CSE373 2012 - 课程 12 - 广度优先搜索 \(video\)](#)
 - ☐ [CSE373 2012 - 课程 13 - 图的算法 \(video\)](#)
 - ☐ [CSE373 2012 - 课程 14 - 图的算法 \(1\) \(video\)](#)
 - ☐ [CSE373 2012 - 课程 15 - 图的算法 \(2\) \(video\)](#)
 - ☐ [CSE373 2012 - 课程 16 - 图的算法 \(3\) \(video\)](#)

- ☐ 图 (复习和其他):
 - ☐ 6.006 单源最短路径问题 (video)
 - ☐ 6.006 Dijkstra 算法 (video)
 - ☐ 6.006 Bellman-Ford 算法(video)
 - ☐ 6.006 Dijkstra 效率优化 (video)
 - ☐ Aduni: 图的算法 I - 拓扑排序, 最小生成树, Prim 算法 - 第六课 (video)
 - ☐ Aduni: 图的算法 II - 深度优先搜索, 广度优先搜索, Kruskal 算法, 并查集数据结构 - 第七课 (video)
 - ☐ Aduni: 图的算法 III: 最短路径 - 第八课 (video)
 - ☐ Aduni: 图的算法. IV: 几何算法介绍 - 第九课 (video)
 - ☐ CS 61B 2014 (从 58:09 开始) (video)
 - ☐ CS 61B 2014: 加权图 (video)
 - ☐ 贪心算法: 最小生成树 (video)
 - ☐ 图的算法之强连通分量 Kosaraju 算法 (video)
- 完整的 Coursera 课程:
 - ☐ 图的算法 (video)
- Yegge: 如果有机会, 可以试试研究更酷炫的算法:
 - ☐ Dijkstra 算法 - 上文 - 6.006
 - ☐ A* 算法
 - ☐ A* 算法
 - ☐ A* 寻路教程 (video)
 - ☐ A* 寻路 (E01: 算法解释) (video)
- 我会实现:
 - ☐ DFS 邻接表 (递归)
 - ☐ DFS 邻接表 (栈迭代)
 - ☐ DFS 邻接矩阵 (递归)
 - ☐ DFS 邻接矩阵 (栈迭代)
 - ☐ BFS 邻接表
 - ☐ BFS 邻接矩阵
 - ☐ 单源最短路径问题 (Dijkstra)
 - ☐ 最小生成树
 - 基于 DFS 的算法 (根据上文 Aduni 的视频):
 - ☐ 检查环 (我们会先检查是否有环存在以便做拓扑排序)
 - ☐ 拓扑排序
 - ☐ 计算图中的连通分支
 - ☐ 列出强连通分量
 - ☐ 检查双向图

可以从 Skiena 的书（参考下面的书推荐小节）和面试书籍中学习更多关于图的实践。

更多知识

- 递归（Recursion）

- ☐ Stanford 大学关于递归 & 回溯的课程:
 - ☐ [课程 8 | 抽象编程 \(video\)](#)
 - ☐ [课程 9 | 抽象编程 \(video\)](#)
 - ☐ [课程 10 | 抽象编程 \(video\)](#)
 - ☐ [课程 11 | 抽象编程 \(video\)](#)
- 什么时候适合使用
- 尾递归会更好么?
 - ☐ [什么是尾递归以及为什么它如此糟糕?](#)
 - ☐ [尾递归 \(video\)](#)
- 动态规划 (Dynamic Programming)
 - 注意：动态规划是门极为重要的技术，尽管其并未被 Google 提供的准备手册提及，但你可能会对寻求最佳解的方式有点疑问，所以我将其列入这份表单。
 - 这一部分会有点困难，每个可以用动态规划解决的问题都必须先定义出递推关系，要推导出来可能会有点棘手。
 - 我建议先阅读和学习足够多的动态规划的例子，以便对解决 DP 问题的一般模式有个扎实的理解。
 - ☐ 视频:
 - Skiena 的视频可能会有点难跟上，有时候他用白板写的字会比较小，难看清楚。
 - ☐ [Skiena: CSE373 2012 - 课程 19 - 动态规划介绍 \(video\)](#)
 - ☐ [Skiena: CSE373 2012 - 课程 20 - 编辑距离 \(video\)](#)
 - ☐ [Skiena: CSE373 2012 - 课程 21 - 动态规划举例 \(video\)](#)
 - ☐ [Skiena: CSE373 2012 - 课程 22 - 动态规划应用 \(video\)](#)
 - ☐ [Simonson: 动态规划 0 \(starts at 59:18\) \(video\)](#)
 - ☐ [Simonson: 动态规划 I - 课程 11 \(video\)](#)
 - ☐ [Simonson: 动态规划 II - 课程 12 \(video\)](#)
 - ☐ 单独的 DP 问题 (每一个视频都很短): [动态规划 \(video\)](#)
 - ☐ Yale 课程笔记:
 - ☐ [动态规划](#)
 - ☐ Coursera 课程:
 - ☐ [RNA 二级结构问题 \(video\)](#)
 - ☐ [动态规划算法 \(video\)](#)
 - ☐ [DP 算法描述 \(video\)](#)
 - ☐ [DP 算法的运行时间 \(video\)](#)
 - ☐ [DP vs 递归实现 \(video\)](#)
 - ☐ [全局成对序列排列 \(video\)](#)
 - ☐ [本地成对序列排列 \(video\)](#)
- 组合 (Combinatorics) (n 中选 k 个) & 概率 (Probability)
 - ☐ [数据技巧: 如何找出阶乘、排列和组合\(选择\) \(video\)](#)
 - ☐ [来点学校的東西: 概率 \(video\)](#)

- ☐ 来点学校的东西: 概率和马尔可夫链 (video)
- ☐ 可汗学院:
 - 课程设置:
 - ☐ 概率理论基础
 - 视频 - 41 (每一个都短小精悍):
 - ☐ 概率解释 (video)
- NP, NP-完全和近似算法
 - 知道最经典的一些 NP 完全问题, 比如旅行商问题和背包问题, 而且能在面试官试图忽悠你的时候识别出他们。
 - 知道 NP 完全是什么意思。
 - ☐ 计算复杂度 (video)
 - ☐ Simonson:
 - ☐ 贪心算法. II & 介绍 NP-完全性 (video)
 - ☐ NP-完全性 II & 归约 (video)
 - ☐ NP-完全性 III (Video)
 - ☐ NP-完全性 IV (video)
 - ☐ Skiena:
 - ☐ CSE373 2012 - 课程 23 - 介绍 NP-完全性 IV (video)
 - ☐ CSE373 2012 - 课程 24 - NP-完全性证明 (video)
 - ☐ CSE373 2012 - 课程 25 - NP-完全性挑战 (video)
 - ☐ 复杂度: P, NP, NP-完全性, 规约 (video)
 - ☐ 复杂度: 近视算法 Algorithms (video)
 - ☐ 复杂度: 固定参数算法 (video)
 - Peter Norvik 讨论旅行商问题的近似最优解:
 - [Jupyter 笔记本](#)
 - 《算法导论》的第 1048 - 1140 页。
- 缓存 (Cache)
 - ☐ LRU 缓存:
 - ☐ LRU 的魔力 (100 Days of Google Dev) (video)
 - ☐ 实现 LRU (video)
 - ☐ LeetCode - 146 LRU Cache (C++) (video)
 - ☐ CPU 缓存:
 - ☐ MIT 6.004 L15: 存储体系 (video)
 - ☐ MIT 6.004 L16: 缓存的问题 (video)
- 进程 (Process) 和线程 (Thread)
 - ☐ 计算机科学 162 - 操作系统 (25 个视频):
 - 视频 1-11 是关于进程和线程
 - 操作系统和系统编程 (video)
 - 进程和线程的区别是什么?
 - 涵盖了:
 - 进程、线程、协程
 - 进程和线程的区别

- 进程
- 线程
- 锁
- 互斥
- 信号量
- 监控
- 他们是如何工作的
- 死锁
- 活锁
- CPU 活动, 中断, 上下文切换
- 现代多核处理器的并发式结构
- 进程资源需要（内存：代码、静态存储器、栈、堆、文件描述符、I/O）
- 线程资源需要（在同一个进程内和其他线程共享以上的资源，但是每个线程都有独立的程序计数器、栈计数器、寄存器和栈）
- Fork 操作是真正的写时复制（只读），直到新的进程写到内存中，才会生成一份新的拷贝。
- 上下文切换
 - 操作系统和底层硬件是如何初始化上下文切换的。
- ☐ [C++ 的线程 \(系列 - 10 个视频\)](#)
- ☐ Python 的协程 (视频):
 - ☐ [线程系列](#)
 - ☐ [Python 线程](#)
 - ☐ [理解 Python 的 GIL \(2010\)](#)
 - [参考](#)
 - ☐ [David Beazley - Python 协程 - PyCon 2015](#)
 - ☐ [Keynote David Beazley - 兴趣主题 \(Python 异步 I/O\)](#)
 - ☐ [Python 中的互斥](#)

系统设计以及可伸缩性，要把软硬件的伸缩性设计的足够好有很多的东西要考虑，所以这是个包含非常多内容和资源的大主题。需要花费相当多的时间在这个主题上。

- 系统设计、可伸缩性、数据处理
 - Yegge 的注意事项:
 - 伸缩性
 - 把大数据集提取为单一值
 - 大数据集转换
 - 处理大量的数据集
 - 系统
 - 特征集
 - 接口
 - 类层次结构
 - 在特定的约束下设计系统
 - 轻量 and 健壮性
 - 权衡和折衷
 - 性能分析和优化
 - ☐ 从这里开始: [HiredInTech: 系统设计](#)

- ☐ 该如何为技术面试里设计方面的问题做准备?
- ☐ 在系统设计面试前必须知道的 8 件事
- ☐ 算法设计
- ☐ 数据库范式 - 1NF, 2NF, 3NF and 4NF (video)
- ☐ 系统设计面试 - 这一部分有很多的资源, 浏览一下我放在下面的文章和例子。
- ☐ 如何在系统设计面试中脱颖而出
- ☐ 每个人都该知道的一些数字
- ☐ 上下文切换操作会耗费多少时间?
- ☐ 跨数据中心的事务 (video)
- ☐ 简明 CAP 理论介绍
- ☐ Paxos 一致性算法:
 - ☐ 时间很短
 - ☐ 用例 和 multi-paxos
 - ☐ 论文
- ☐ 一致性哈希
- ☐ NoSQL 模式
- ☐ OOSE: UML 2.0 系列 (video)
- ☐ OOSE: 使用 UML 和 Java 开发软件 (21 videos):
 - 如果你对 OO 都深刻的理解和实践, 可以跳过这部分。
 - ☐ OOSE: 使用 UML 和 Java 开发软件
- ☐ 面向对象编程的 SOLID 原则:
 - ☐ Bob Martin 面向对象的 SOLID 原则和敏捷设计 (video)
 - ☐ C# SOLID 设计模式 (video)
 - ☐ SOLID 原则 (video)
 - ☐ S - 单一职责原则 | 每个对象的单一职责
 - ☐ 更多
 - ☐ O - 开闭原则 | 生产环境里的对象应该为扩展做准备而不是为更改
 - ☐ 更多
 - ☐ L - 里氏代换原则 | 基类和继承类遵循 'IS A' 原则
 - ☐ 更多
 - ☐ I - 接口隔离原则 | 客户端被迫实现用不到的接口
 - ☐ 5 分钟讲解接口隔离原则 (video)
 - ☐ 更多
 - ☐ D - 依赖反转原则 | 减少对象里的依赖。
 - ☐ 什么是依赖倒置以及它为什么重要
 - ☐ 更多
- ☐ 可伸缩性:
 - ☐ 很棒的概述 (video)
 - ☐ 简短系列:
 - ☐ 克隆
 - ☐ 数据库
 - ☐ 缓存
 - ☐ 异步
 - ☐ 可伸缩的 Web 架构和分布式系统
 - ☐ 错误的分布式系统解释
 - ☐ 实用编程技术

- [extra: Google Pregel 图形处理](#)
- [Jeff Dean - 在 Google 构建软件系统 \(video\)](#)
- [可伸缩系统架构设计介绍](#)
- [使用 App Engine 和云存储扩展面向全球用户的手机游戏架构实践\(video\)](#)
- [How Google Does Planet-Scale Engineering for Planet-Scale Infra \(video\)](#)
- [算法的重要性](#)
- [分片](#)
- [Facebook 系统规模扩展实践 \(2009\)](#)
- [Facebook 系统规模扩展实践 \(2012\), "为 10 亿用户构建" \(video\)](#)
- [Long Game 工程实践 - Astrid Atkinson Keynote\(video\)](#)
- [30 分钟看完 YouTube 7 年系统扩展经验](#)
 - [video](#)
- [PayPal 如何用 8 台虚拟机扛住 10 亿日交易量系统](#)
- [如何对大数据集去重](#)
- [Etsy 的扩展和工程文化探究 Jon Cowie \(video\)](#)
- [是什么造就了 Amazon 自己的微服务架构](#)
- [压缩还是不压缩，是 Uber 面临的问题](#)
- [异步 I/O Tarantool 队列](#)
- [什么时候应该用近视查询处理？](#)
- [Google 从单数据中心到故障转移, 到本地多宿主架构的演变](#)
- [Spanner](#)
- [Egnyte: 构建和扩展 PB 级分布式系统架构的经验教训](#)
- [机器学习驱动的编程: 新世界的新编程方式](#)
- [日服务数百万请求的图像优化技术](#)
- [Patreon 架构](#)
- [Tinder: 推荐引擎是如何决定下一个你将会看到谁的？](#)
- [现代缓存设计](#)
- [Facebook 实时视频流扩展](#)
- [在 Amazon AWS 上把服务扩展到 1100 万量级的新手教程](#)
- [对延时敏感的应用是否应该使用 Docker？](#)
- [AMP \(Accelerated Mobile Pages\) 的存在是对 Google 的威胁么？](#)
- [360 度解读 Netflix 技术栈](#)
- [延迟无处不在 - 如何搞定它？](#)
- [无服务器架构](#)
- [是什么驱动着 Instagram: 上百个实例、几十种技术](#)
- [Cinchcast 架构 - 每天处理 1500 小时的音频](#)
- [Justin.Tv 实时视频播放架构](#)
- [Playfish's 社交游戏架构 - 每月五千万用户增长](#)
- [猫途鹰架构 - 40 万访客, 200 万动态页面访问, 30TB 数据](#)
- [PlentyOfFish 架构](#)
- [Salesforce 架构 - 如何扛住 13 亿日交易量](#)
- [ESPN's 架构扩展](#)
- [下面『消息、序列化和消息系统』部分的内容会提到什么样的技术能把各种服务整合到一起](#)
- [Twitter:](#)
 - [O'Reilly MySQL CE 2011: Jeremy Cole, "Big and Small Data at @Twitter" \(video\)](#)

- 时间线的扩展
- 更多内容可以查看视频部分的『大规模数据挖掘』视频系列。
- ☐ 系统设计问题练习：下面有一些指导原则，每一个都有相关文档以及在现实中该如何处理。
 - 复习: [HiredInTech 的系统设计](#)
 - [cheat sheet](#)
 - 流程:
 1. 理解问题和范围:
 - 在面试官的帮助下定义用例
 - 提出附加功能的建议
 - 去掉面试官认定范围以外的内容
 - 假定高可用是必须的，而且要作为一个用例
 2. 考虑约束:
 - 问一下每月请求量
 - 问一下每秒请求量 (他们可能会主动提到或者让你算一下)
 - 评估读写所占的百分比
 - 评估的时候牢记 2/8 原则
 - 每秒写多少数据
 - 总的数据存储量要考虑超过 5 年的情况
 - 每秒读多少数据
 3. 抽象设计:
 - 分层 (服务, 数据, 缓存)
 - 基础设施: 负载均衡, 消息
 - 粗略的概括任何驱动整个服务的关键算法
 - 考虑瓶颈并指出解决方案
 - 练习:
 - [设计一个 CDN 网络](#)
 - [设计一个随机唯一 ID 生成系统](#)
 - [设计一个在线多人卡牌游戏](#)
 - [设计一个 key-value 数据库](#)
 - [设计一个函数获取过去某个时间段内前 K 个最高频访问的请求](#)
 - [设计一个图片分享系统](#)
 - [设计一个推荐系统](#)
 - [设计一个短域名生成系统](#)
 - [设计一个缓存系统](#)
- 论文
 - 有 Google 的论文和一些知名的论文。
 - 你很可能实在没时间一篇篇完整的读完他们。我建议可以有选择的读其中一些论文里的核心部分。
 - ☐ [1978: 通信顺序处理](#)
 - [Go 实现](#)
 - [喜欢经典的论文?](#)
 - ☐ [2003: The Google 文件系统](#)
 - 2012 年被 Colossus 取代了
 - ☐ [2004: MapReduce: Simplified Data Processing on Large Clusters](#)
 - 大多被云数据流取代了?

- ☐ [2007: 每个程序员都应该知道的内存知识 \(非常长, 作者建议跳过某些章节来阅读\)](#)
- ☐ [2012: Google 的 Colossus](#)
 - 没有论文
- ☐ [2012: AddressSanitizer: 快速的内存访问检查器:](#)
 - [论文](#)
 - [视频](#)
- ☐ [2013: Spanner: Google 的分布式数据库:](#)
 - [论文](#)
 - [视频](#)
- ☐ [2014: Machine Learning: The High-Interest Credit Card of Technical Debt](#)
- ☐ [2015: Continuous Pipelines at Google](#)
- ☐ [2015: 大规模高可用: 构建 Google Ads 的数据基础设施](#)
- ☐ [2015: TensorFlow: 异构分布式系统上的大规模机器学习](#)
- ☐ [2015: 开发者应该如何搜索代码: 用例学习](#)
- ☐ [2016: Borg, Omega, and Kubernetes](#)

• 测试

- 涵盖了:
 - 单元测试是如何工作的
 - 什么是模拟对象
 - 什么是集成测试
 - 什么是依赖注入
- ☐ [James Bach 讲敏捷软件测试 \(video\)](#)
- ☐ [James Bach 软件测试公开课 \(video\)](#)
- ☐ [Steve Freeman - 测试驱动的开发 \(video\)](#)
 - [slides](#)
- ☐ [测试驱动的开发已死。测试不朽。](#)
- ☐ [测试驱动的开发已死? \(video\)](#)
- ☐ [视频系列 \(152 个\) - 并不都是必须 \(video\)](#)
- ☐ [Python: 测试驱动的 Web 开发](#)
- ☐ 依赖注入:
 - ☐ [视频](#)
 - ☐ [测试之道](#)
- ☐ [如何编写测试](#)

• 调度

- 在操作系统中是如何运作的
- 在操作系统部分的视频里有很多资料

• 实现系统例程

- 理解你使用的系统 API 底层有什么
- 你能自己实现它们么?

• 字符串搜索和操作

- ☐ [文本的搜索模式 \(video\)](#)
 - ☐ Rabin-Karp (videos):
 - [Rabin Karps 算法](#)
 - [预先计算的优化](#)
 - [优化: 实现和分析](#)
 - [Table Doubling, Karp-Rabin](#)
 - [滚动哈希](#)
 - ☐ Knuth-Morris-Pratt (KMP) 算法:
 - [Pratt 算法](#)
 - [教程: Knuth-Morris-Pratt \(KMP\) 字符串匹配算法](#)
 - ☐ Boyer-Moore 字符串搜索算法
 - [Boyer-Moore字符串搜索算法](#)
 - [Boyer-Moore-Horspool 高级字符串搜索算法 \(video\)](#)
 - ☐ Coursera: 字符串的算法
-

终面

这一部分有一些短视频，你可以快速的观看和复习大多数重要概念。
这对经常性的巩固很有帮助。

综述:

- ☐ 2-3 分钟的短视频系列 (23 个)
 - [Videos](#)
- ☐ 2-5 分钟的短视频系列 - Michael Sambol (18 个):
 - [Videos](#)

排序:

- ☐ 归并排序: <https://www.youtube.com/watch?v=GCae1WNvnZM>

书籍

Google Coaching 里提到的

阅读并做练习:


- ☐ 算法设计手册 (Skiena)
 - 书 (Kindle 上可以租到):
 - [Algorithm Design Manual](#)
 - Half.com 是一个资源丰富且性价比很高的在线书店.
 - 答案:
 - [解答](#)
 - [解答](#)
 - [勘误表](#)

read and do exercises from the books below. Then move to coding challenges (further down below) 一旦你理解了每日计划里的所有内容，就去读上面所列的书并完成练习，然后开始读下面所列的书并做练习，之后就可以开始实战写代码了（本文再往后的部分）

首先阅读:

-  [Programming Interviews Exposed: Secrets to Landing Your Next Job, 2nd Edition](#)

然后阅读 (这本获得了很多推荐，但是不在 **Google coaching** 的文档里):



-  [Cracking the Coding Interview, 6th Edition](#)
 - 如果你看到有人在看 "The Google Resume", 实际上它和 "Cracking the Coding Interview" 是同一个作者写的，而且后者是升级版。

附加书单

这些没有被 Google 推荐阅读，不过我因为需要这些背景知识所以也把它们列在了这里。

-  C Programming Language, Vol 2
 - [练习的答案](#)
-  C++ Primer Plus, 6th Edition
-  《Unix 环境高级编程》 [The Unix Programming Environment](#)
-  《编程珠玑》 [Programming Pearls](#)
-  [Algorithms and Programming: Problems and Solutions](#)

如果你有时间

-  [Introduction to Algorithms](#)
-  [Elements of Programming Interviews](#)
 - 如果你希望在面试里用 C++ 写代码，这本书的代码全都是 C++ 写的
 - 通常情况下能找到解决方案的好书。

编码练习和挑战

一旦你学会了理论基础，就应该把它们拿出来练练。尽量坚持每天做编码练习，越多越好。

编程问题预备:

-  不错的介绍 (摘自 [System Design](#) 章节): 算法设计:
-  [如何找到解决方案](#)
-  [如何剖析 Topcoder 题目描述](#)
-  [Topcoders 里用到的数学](#)
-  [动态规划 – 从入门到精通](#)

- [MIT 面试材料](#)
- [针对编程语言本身的练习](#)

编码练习平台:

- [LeetCode](#)
- [TopCoder](#)
- [Project Euler](#) (数学方向为主)
- [Codewars](#)
- [HackerRank](#)
- [Codility](#)
- [InterviewCake](#)
- [InterviewBit](#)
- [模拟大公司的面试](#)

当你临近面试时

- ☐ 搞定代码面试 (videos):
 - [Cracking The Code Interview](#)
 - [Cracking the Coding Interview](#) - 全栈系列
 - [Ask Me Anything: Gayle Laakmann McDowell](#) ([Cracking the Coding Interview](#) 的作者)

你的简历

- [10 条小贴士让你写出一份还算不错的简历](#)
- [这是搞定面试的第一个关键步骤](#)

当面试来临的时候

随着下面列举的问题思考下你可能会遇到的 20 个面试问题

每个问题准备 2-3 种回答

准备点故事，不要只是摆一些你完成的事情的数据，相信我，人人都喜欢听故事

- 你为什么想得到这份工作？
- 你解决过的最有难度的问题是什么？
- 面对过的最大挑战是什么？
- 见过的最好或者最坏的设计是怎么样的？
- 对某项 Google 产品提出改进建议。
- 你作为一个个体同时也是团队的一员，如何达到最好的工作状态？
- 你的什么技能或者经验是你的角色中不可或缺的？为什么？
- 你在某份工作或某个项目中最享受的是什么？

- 你在某份工作或某个项目中面临过的最大挑战是什么？
- 你在某份工作或某个项目中遇到过的最蛋疼的 Bug 是什么样的？
- 你在某份工作或某个项目中学到了什么？
- 你在某份工作或某个项目中哪些地方还可以做的更好？

问面试官的问题

我会问的一些：（可能我已经知道了答案但我想听听面试官的看法或者了解团队的前景）：

- 团队多大规模？
- 开发周期是怎样的？会使用瀑布流/极限编程/敏捷开发么？
- 经常会为 **deadline** 加班么？或者是有弹性的？
- 团队里怎么做技术选型？
- 每周平均开多少次会？
- 你觉得工作环境有助于员工集中精力吗？
- 目前正在做什么工作？
- 喜欢这些事情吗？
- 工作期限是怎麼样的？

当你获得了梦想的职位

我还能说些什么呢，恭喜你！

- 我希望在 [Google](#) 的第一天就知道的 10 件事

坚持继续学习。

得到这份工作只是一个开始。

```
*****
*****
*****
*****
```

下面的内容都是可选的。这些是我的推荐，不是 **Google** 的。
通过学习这些内容，你将会得到更多的有关 **CS** 的概念，并将为所有的软件 engineering 工作做更好的准备。

```
*****
*****
*****
*****
```

附加的学习

- Unicode
 - ☐ 每一个软件开发者的绝对最低限度, 必须要知道的关于 Unicode 和字符集知识
 - ☐ 关于处理文本需要的编码和字符集, 每个程序员绝对需要知道的知识
- 字节顺序
 - ☐ 大、小端字节序
 - ☐ 大端字节 Vs 小端字节(视频)
 - ☐ 大、小端字节序的里里外外(Big And Little Endian Inside/Out) (视频)
 - 内核开发者的讨论非常技术性, 如果大多数都超出了你的理解范围, 不要太担心。
 - 前半段已经足够了。
- Emacs and vi(m)
 - Yegge 的建议, 从一个很早以前的亚马逊招聘信息中而来: 熟悉基于 unix 的代码编辑器
 - vi(m):
 - 使用 vim 进行编辑 01 - 安装, 设置和模式 (视频)
 - VIM 的冒险之旅
 - 4 个视频集:
 - vi/vim 编辑器 - 课程 1
 - vi/vim 编辑器 - 课程 2
 - vi/vim 编辑器 - 课程 4
 - vi/vim 编辑器 - 课程 3
 - 使用 Vi 而不是 Emacs
 - emacs:
 - 基础 Emacs 教程 (视频)
 - 3 个视频集:
 - Emacs 教程 (初学者) -第 1 部分- 文件命令, 剪切/复制/粘贴, 自定义命令
 - Emacs 教程 (初学者) -第 2 部分- Buffer 管理, 搜索, M-x grep 和 rgrep 模式
 - Emacs 教程 (初学者) -第 3 部分- 表达式, 声明, ~/.emacs 文件和包机制
 - Evil 模式: 或许, 我是怎样对 Emacs 路人转粉的 (视频)
 - 使用 Emacs 开发 C 程序
 - (或许) 深度组织模式:管理结构 (视频)
- Unix 命令行工具
 - 下列内容中的优秀工具由的 Yegge 推荐, Yegge 目前致力于 Amazon 人事招聘处。
 - ☐ bash
 - ☐ cat
 - ☐ grep
 - ☐ sed
 - ☐ awk
 - ☐ curl or wget
 - ☐ sort
 - ☐ tr
 - ☐ uniq
 - ☐ strace

- ☐ [tcpdump](#)
- 信息资源 (视频)
 - ☐ [Khan Academy 可汗学院](#)
 - ☐ 更多有关马尔可夫的内容:
 - ☐ [Core Markov Text Generation](#) 马尔可夫内容生成
 - ☐ [Core Implementing Markov Text Generation](#) 马尔可夫内容生成补充
 - ☐ [Project = Markov Text Generation Walk Through](#) 一个马尔可夫内容生成器的项目
 - 关于更多信息, 请参照下方 MIT 6.050J 信息和系统复杂度的内容.
- 奇偶校验位 & 汉明码 (视频)
 - ☐ [入门](#)
 - ☐ [奇偶校验位](#)
 - ☐ 汉明码(Hamming Code):
 - [发现错误](#)
 - [修正错误](#)
 - ☐ [检查错误](#)
- 系统熵值 (系统复杂度)
 - 请参考下方视频
 - 观看之前, 请先确定观看了信息论的视频
 - ☐ [信息理论, 克劳德·香农, 熵值, 系统冗余, 数据比特压缩 \(视频\)](#)
- 密码学
 - 请参考下方视频
 - 观看之前, 请先确定观看了信息论的视频
 - ☐ [可汗学院](#)
 - ☐ [密码学: 哈希函数](#)
 - ☐ [密码学: 加密](#)
- 压缩
 - 观看之前, 请先确定观看了信息论的视频
 - ☐ 压缩 (视频):
 - ☐ [压缩](#)
 - ☐ [压缩熵值](#)
 - ☐ [由上而下的树 \(霍夫曼编码树\)](#)
 - ☐ [额外比特 - 霍夫曼编码树](#)
 - ☐ [优雅的压缩数据 \(无损数据压缩方法\)](#)
 - ☐ [Text Compression Meets Probabilities](#)
 - ☐ [数据压缩的艺术](#)
 - ☐ (可选) 谷歌开发者: [GZIP 还差远了呢!](#)
- 网络 (视频)

- ☐ 可汗学院
- ☐ 网络传输协议中的数据压缩
- ☐ TCP/IP 和 OSI 模型解析!
- ☐ TCP/IP 教程: 传输数据包.
- ☐ HTTP
- ☐ SSL 和 HTTPS
- ☐ SSL/TLS
- ☐ HTTP 2.0
- ☐ 视频
- ☐ 子网络解密 - 第五部分 经典内部域名指向 CIDR 标记
- 计算机安全
 - MIT
 - ☐ 威胁模型: 入门
 - ☐ 控制攻击
 - ☐ 缓冲数据注入和防御
 - ☐ 优先级区分
 - ☐ 能力
 - ☐ 在沙盒中运行原生代码
 - ☐ 网络安全模型
 - ☐ 网络安全应用
 - ☐ 标志化执行
 - ☐ 网络安全
 - ☐ 网络协议
 - ☐ 旁路攻击
- 释放缓存
 - ☐ Java 释放缓存; 片段化数据 (视频)
 - ☐ 编译器 (视频)
 - ☐ Python 释放缓存 (视频)
 - ☐ 深度解析: 论释放缓存在 JAVA 中的重要性
 - ☐ 深度解析: 论释放缓存在 Python 中的重要性(视频)
- 并行/并发编程
 - ☐ Coursera (Scala)
 - ☐ 论并行/并发编程如何提高 Python 执行效率 (视频)
- 设计模式
 - ☐ UML统一建模语言概览 (视频)
 - ☐ 主要有如下的设计模式:
 - ☐ s(strategy)
 - ☐ singleton
 - ☐ adapter
 - ☐ prototype

- ☐ decorator
- ☐ visitor
- ☐ factory, abstract factory
- ☐ facade
- ☐ observer
- ☐ proxy
- ☐ delegate
- ☐ command
- ☐ state
- ☐ memento
- ☐ iterator
- ☐ composite
- ☐ flyweight
- ☐ 第六章 (第 1 部分) - 设计模式 (视频)
- ☐ 第六章 (第 2 部分) - Abstraction-Occurrence, General Hierarchy, Player-Role, Singleton, Observer, Delegation (视频)
- ☐ 第六章 (第 3 部分) - Adapter, Facade, Immutable, Read-Only Interface, Proxy (video)
- ☐ 视频
- ☐ Head First 设计模型
 - 尽管这本书叫做设计模式：重复使用模块，但是我还是认为Head First是对于新手来说很不错的书。
- ☐ 基于实际操作对于入门开发者的建议
- 信息传输, 序列化, 和队列化的系统
 - ☐ Thrift
 - 教程
 - ☐ 协议缓冲
 - 教程
 - ☐ gRPC
 - gRPC 对于JAVA开发者的入门教程（视频）
 - ☐ Redis
 - 教程
 - ☐ Amazon的 SQS 系统 (队列)
 - ☐ Amazon的 SNS 系统 (pub-sub)
 - ☐ RabbitMQ
 - 入门教程
 - ☐ Celery
 - Celery入门
 - ☐ ZeroMQ
 - 入门教程
 - ☐ ActiveMQ
 - ☐ Kafka
 - ☐ MessagePack
 - ☐ Avro
- 快速傅里叶变换

- ☐ [什么是傅立叶变换？论傅立叶变换的用途](#)
- ☐ [什么是傅立叶变换？\(视频\)](#)
- ☐ [关于 FFT 的不同观点 \(视频\)](#)
- ☐ [FTT 是什么](#)
- 布隆过滤器
 - 给一个布隆过滤器m比特和k个哈希函数，所有的注入和相关测试都会是通过。
 - [布隆过滤器](#)
 - [布隆过滤器 | 数据挖掘 | Stanford University](#)
 - [教程](#)
 - [如何写一个布隆过滤器应用](#)
- van Emde Boas 树
 - ☐ [争论: van Emde Boas 树 \(视频\)](#)
 - ☐ [MIT课堂笔记](#)
- 更深入的数据结构
 - ☐ [CS 61B 第 39 课: 更深入的数据结构](#)
- 跳表
 - "有一种非常迷幻的数据类型" - Skiena
 - ☐ [随机化: 跳表 \(视频\)](#)
 - ☐ [更生动详细的解释](#)
- 网络流
 - ☐ [5分钟简析Ford-Fulkerson \(视频\)](#)
 - ☐ [Ford-Fulkerson 算法 \(视频\)](#)
 - ☐ [网络流 \(视频\)](#)
- 不相交集 & 联合查找
 - ☐ [不相交集](#)
 - ☐ [UCB 61B - 不相交集; 排序 & 选择\(视频\)](#)
 - ☐ Coursera (not needed since the above video explains it great):
 - ☐ [概览](#)
 - ☐ [初级实践](#)
 - ☐ [树状结构](#)
 - ☐ [合并树状结构](#)
 - ☐ [路径压缩](#)
 - ☐ [分析选项](#)
- 快速处理数学
 - ☐ [整数运算, Karatsuba 乘法 \(视频\)](#)

- ☐ [中国剩余定理 \(在密码学中的使用\) \(视频\)](#)
- 树堆 (Treap)
 - 一个二叉搜索树和一个堆的组合
 - ☐ [树堆](#)
 - ☐ [数据结构：树堆的讲解\(video\)](#)
 - ☐ [集合操作的应用\(Applications in set operations\)](#)
- 线性规划 (Linear Programming) (视频)
 - ☐ [线性规划](#)
 - ☐ [寻找最小成本](#)
 - ☐ [寻找最大值](#)
- 几何：凸包 (Geometry, Convex hull) (视频)
 - ☐ [Graph Alg. IV: 几何算法介绍 - 第 9 课](#)
 - ☐ [Graham & Jarvis: 几何算法 - 第 10 课](#)
 - ☐ [Divide & Conquer: 凸包, 中值查找](#)
- 离散数学
 - 查看下面的视频: (这里没看到视频= =)
- 机器学习 (Machine Learning)
 - ☐ 为什么学习机器学习?
 - ☐ [谷歌如何将自己改造成一家「机器学习优先」公司?](#)
 - ☐ [智能计算机系统的大规模深度学习 \(视频\)](#)
 - ☐ [Peter Norvig: 深度学习和理解与软件工程和验证的对比](#)
 - ☐ [谷歌云机器学习工具 \(视频\)](#)
 - ☐ [谷歌开发者机器学习清单 \(Scikit Learn 和 Tensorflow\) \(视频\)](#)
 - ☐ [Tensorflow \(视频\)](#)
 - ☐ [Tensorflow 教程](#)
 - ☐ [Python 实现神经网络实例教程 \(使用 Theano\)](#)
 - 课程:
 - ☐ [很棒的初级课程：机器学习 - 视频教程 - 看第 12-18 集复习线性代数 \(第 14 集和第 15 集是重复的\)](#)
 - ☐ [机器学习中的神经网络](#)
 - ☐ [Google 深度学习微学位](#)
 - ☐ [Google/Kaggle 机器学习工程师微学位](#)
 - ☐ [无人驾驶工程师微学位](#)
 - ☐ [Metis 在线课程 \(两个月 99 美元\)](#)
 - 资源:
 - 书籍: Data Science from Scratch: First Principles with Python:
<https://www.amazon.com/Data-Science-Scratch-Principles-Python/dp/149190142X>
 - 网站: Data School: <http://www.dataschool.io/>

- Go 语言
 - ☐ 视频:
 - ☐ 为什么学习 Go 语言?
 - ☐ Go 语言编程
 - ☐ Go 语言之旅
 - ☐ 书籍:
 - ☐ Go 语言编程入门 (免费在线阅读)
 - ☐ Go 语言圣经 (Donovan & Kernighan)
 - ☐ Go 语言新手训练营

--

一些主题的额外内容


我为前面提到的某些主题增加了一些额外的内容，之所以没有直接添加到前面，是因为这样很容易导致某个主题内容过多。毕竟你想在本世纪找到一份工作，对吧？

- ☐ 动态规划的更多内容 (视频)
 - ☐ 6.006: 动态规划 I: 斐波那契数列, 最短路径
 - ☐ 6.006: 动态规划 II: 文本匹配, 二十一点/黑杰克
 - ☐ 6.006: 动态规划 III: 最优加括号方式, 最小编辑距离, 背包问题
 - ☐ 6.006: 动态规划 IV: 吉他指法, 拓扑, 超级马里奥.
 - ☐ 6.046: 动态规划: 动态规划进阶
 - ☐ 6.046: 动态规划: 所有点对最短路径
 - ☐ 6.046: 动态规划: 更多示例
- ☐ 图形处理进阶 (视频)
 - ☐ 异步分布式算法: 对称性破缺, 最小生成树
 - ☐ 异步分布式算法: 最小生成树
- ☐ MIT 概率论 (mathy, and go slowly, which is good for mathy things) (视频):
 - ☐ MIT 6.042J - 概率论概述
 - ☐ MIT 6.042J - 条件概率 Probability
 - ☐ MIT 6.042J - 独立
 - ☐ MIT 6.042J - 随机变量
 - ☐ MIT 6.042J - 期望 I
 - ☐ MIT 6.042J - 期望 II
 - ☐ MIT 6.042J - 大偏差
 - ☐ MIT 6.042J - 随机游走
- ☐ Simonson: 近似算法 (视频)

视频系列

坐下来享受一下吧。"netflix and skill" 😊

- ☐ 个人的动态规划问题列表 (都是短视频哟)
- ☐ x86 架构, 汇编, 应用程序 (11 个视频)
- ☐ MIT 18.06 线性代数, 2005 年春季 (35 个视频)
- ☐ 绝妙的 MIT 微积分: 单变量微积分
- ☐ 计算机科学 70, 001 - 2015 年春季 - 离散数学和概率理论
- ☐ 离散数学 (19 个视频)
- ☐ CSE373 - 算法分析 (25 个视频)
 - Skiena 的算法设计手册讲座
- ☐ UC Berkeley 61B (2014 年春季): 数据结构 (25 个视频)
- ☐ UC Berkeley 61B (2006 年秋季): 数据结构 (39 个视频)
- ☐ UC Berkeley 61C: 计算机结构 (26 个视频)
- ☐ OOSE: 使用 UML 和 Java 进行软件开发 (21 个视频)
- ☐ UC Berkeley CS 152: 计算机结构和工程 (20 个视频)
- ☐ MIT 6.004: 计算结构 (49 视频)
- ☐ 卡内基梅隆大学 - 计算机架构讲座 (39 个视频)
- ☐ MIT 6.006: 算法介绍 (47 个视频)
- ☐ MIT 6.033: 计算机系统工程 (22 个视频)
- ☐ MIT 6.034 人工智能, 2010 年秋季 (30 个视频)
- ☐ MIT 6.042J: 计算机科学数学, 2010 年秋季 (25 个视频)
- ☐ MIT 6.046: 算法设计与分析 (34 个视频)
- ☐ MIT 6.050J: 信息和熵, 2008 年春季 (19 个视频)
- ☐ MIT 6.851: 高等数据结构 (22 个视频)
- ☐ MIT 6.854: 高等算法, 2016 年春季 (24 个视频)
- ☐ MIT 6.858 计算机系统安全, 2014 年秋季
- ☐ 斯坦福: 编程范例 (17 个视频)
 - C 和 C++ 课程
- ☐ 密码学导论

- [本系列更多内容 \(不分先后顺序\)](#)
-  [大数据 - 斯坦福大学 \(94 个视频\)](#)

计算机科学课程

- [在线 CS 课程目录](#)
- [CS 课程目录 \(一些是在线讲座\)](#)