

# 个人学习笔记整理

---

代码阅读笔记、论文阅读报告及一些零散知识点整理

整理： 刘冰

qiao.liubing@gmail.com

开始时间： 7月4日，2018年

最后一次更新时间： 二零一八年九月

版本： Version 0.01

Copyright © Bing, Liu.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled “GNU Free Documentation License”.

# 目录

<b>目录</b>	<b>i</b>	<b>第三章 深度学习框架</b>	<b>27</b>
<b>第一章 大规模机器学习框架</b>	<b>1</b>	3.1 张量 . . . . .	27
1.1 背景 . . . . .	1	3.2 基于张量的各种操作 . . . . .	28
1.1.1 简单模型 or 复杂模型 . . . . .	1	3.3 计算图 . . . . .	28
1.1.2 数据并行 vs 模型并行 . . . . .	2	3.4 自动微分工具 . . . . .	30
1.2 并行算法演进 . . . . .	8	3.5 BLAS/cuBLAS/cuDNN 等扩展包 . . . . .	30
1.2.1 MapReduce 路线 . . . . .	8	3.6 总结 . . . . .	30
1.2.2 MPI 技术 . . . . .	10		
1.3 参数服务器演进 . . . . .	11	<b>第四章 分布式系统的基本要素</b>	<b>33</b>
1.3.1 历史演进 . . . . .	11	4.1 Vector Clock . . . . .	33
1.3.2 基础架构 . . . . .	12	4.1.1 . . . . .	33
1.3.3 同步协议 . . . . .	14	4.1.2 . . . . .	33
1.3.4 核心技术 . . . . .	17	4.2 一致性哈希 . . . . .	33
1.4 大规模机器学习的四重境界 . . . . .	17	4.2.1 分布式缓存问题 . . . . .	33
1.4.1 境界 1 . . . . .	17	4.2.2 一致性哈希算法 . . . . .	34
1.4.2 境界 2 . . . . .	18		
1.4.3 境界 3 . . . . .	18	<b>第五章 机器学习 &amp; 深度学习基础</b>	<b>39</b>
1.4.4 境界 4 . . . . .	19	5.1 梯度下降 . . . . .	39
1.4.5 TensorFlow . . . . .	19	5.1.1 损失函数 . . . . .	39
1.5 其他 . . . . .	19	5.1.2 过拟合 . . . . .	39
1.5.1 资源管理 . . . . .	19	5.1.3 正则化 . . . . .	39
1.5.2 设备 . . . . .	19	5.2 Inference & Training . . . . .	39
1.6 结语 . . . . .	20	5.2.1 . . . . .	39
<b>第二章 分布式系统论文阅读</b>	<b>21</b>	<b>第六章 RDMA 相关知识点整理</b>	<b>41</b>
2.1 论文阅读要求 . . . . .	21	6.1 . . . . .	41
2.2 分布式存储 . . . . .	22	6.1.1 . . . . .	41
2.2.1 GFS . . . . .	22	6.1.2 . . . . .	41
2.3 分布式计算 . . . . .	22	<b>第七章 算法与数据结构</b>	<b>43</b>
2.3.1 MapReduce . . . . .	22	7.1 基本数据结构 . . . . .	43
2.3.2 Spark . . . . .	22	7.2 分而治之 . . . . .	43
2.4 参数服务器 . . . . .	23	7.3 贪心 . . . . .	43
2.4.1 ASP . . . . .	23	7.3.1 . . . . .	43
2.4.2 SSP . . . . .	23	7.3.2 . . . . .	43
2.4.3 Parameter Server . . . . .	24	7.4 动态规划 . . . . .	43
2.4.4 GeePS . . . . .	26	7.4.1 动态规划的本质 . . . . .	43

7.4.2 状态的定义 . . . . .	43	8.3.1 进程、线程与协程 . . . . .	61
7.4.3 状态转移方程 . . . . .	44	8.3.2 select / poll / epoll . . . . .	61
7.4.4 动态规划迷思 . . . . .	44	8.3.3 RPC . . . . .	61
7.5 回溯法 . . . . .	45	8.4 面试回顾及整理 . . . . .	62
7.6 分支限界法 . . . . .	45	8.4.1 微信支付 . . . . .	62
<b>第八章 秋招准备</b>	<b>47</b>	<b>第九章 信息论与计算机</b>	<b>65</b>
8.1 算法题 . . . . .	47	9.1 信息论基础 . . . . .	65
8.1.1 判断是否有环? . . . . .	47		
8.1.2 排序算法 . . . . .	47	<b>第十章 杂的言</b>	<b>67</b>
8.1.3 二分查找 . . . . .	48	10.1 每月知识点归档 . . . . .	67
8.2 C / C++ . . . . .	48	10.2 每周整理为一些集中的点 . . . . .	67
8.2.1 基础知识 . . . . .	48	10.2.1 缓存 . . . . .	67
8.2.2 C++ 语言特性 . . . . .	57	10.3 每日杂项整理 . . . . .	67
8.2.3 一些实现 . . . . .	61	10.3.1 20180710 . . . . .	67
8.3 操作系统 . . . . .	61	10.3.2 20180711 . . . . .	67

# 第一章 大规模机器学习框架

本章内容主要是对知乎上 *carbon zhang* 的专栏文章**大规模机器学习框架的四种境界<sup>1</sup>**进行的总结与整理。

## 1.1 背景

自从 Google 发表著名的 GFS<sup>2</sup>、MapReduce<sup>3</sup>、Big Table<sup>4</sup>三篇 paper 以后，互联网正式迎来了大数据时代。

有了 GFS，即分布式文件系统，我们有能力积累海量的数据样本，比如在线广告的曝光和点击数据，天然具有正负样本的特性，积累一两个月往往就能轻松获得百亿、千亿级的训练样本。这样海量的样本如何存储？用什么样的模型可以学习海量样本中有用的模式（patterns）？这些问题不止是工程问题，也值得每个做算法的同学去深入思考。

### Note 1.1 GFS

Google 在 2003 年发表 GFS 论文，对其中的 *Falut Tolerance* 有很深刻的印象（第一遍看 GFS 论文的时候）。

### Note 1.2 BigTable

*BigTable* 是什么？那篇论文我还没有看过。

### 1.1.1 简单模型 or 复杂模型

在深度学习概念提出之前，我们大都使用的是传统的机器学习算法，例如 LR（逻辑回归）、SVM（支持向量机）、感知机等，这些算法种类很少且相对固定；那时候要解决一个实际的问题，更多的工作主要集中在**特征工程**方面。而特征工程本身并没有很系统化的指导理论（至少目前没有看到系统介绍特征工程的书籍），所以很多时候特征的构造技法显得光怪陆离，是否有用也取决于问题本身、数据样本、模型以及运气。

如果给这种方式起一个名字的话，大概是简单模型 + 复杂特征；简单模型说的是算法比如 LR、SVM 本身并不复杂，参数和表达能力基本呈现一种线性关系，易于理解；复杂特征则是指特征工程方面不断尝试使用各种奇技淫巧构造的可能有用、可能没用的特征。这部分特征的构造方式可能会有各种 trick，比如窗口滑动、离散化、归一化、开方、平方、笛卡尔积、多重笛卡尔积等等；因为特征工程本身并没有

<sup>1</sup><https://zhuanlan.zhihu.com/p/29968773>

<sup>2</sup>The Google File System

<sup>3</sup>MapReduce: Simplified Data Processing on Large Clusters

<sup>4</sup>Bigtable: A Distributed Storage System for Structured Data

特别系统的理论和总结，因此需要大量阅读和自己业务场景一样或类似的文章，从里面学习作者分析、理解数据的方法以及对应的构造特征的技法，久而久之，有望形成自己的知识体系。

深度学习概念提出以后，人们发现通过深度神经网络可以进行一定程度的表示学习（representation learning），例如在图像领域，通过 CNN 提取图像特征（feature）并在此基础上进行分类，此方法（AlexNet<sup>5</sup>）一举打破了之前算法的天花板，而且是以极大的差距打破。这给所有算法工程师带来了新的思路，既然深度学习本身有提取特征的能力，干嘛还要苦苦的自己去做人工特征设计呢？

### Note 1.3 AlexNet

*AlexNet*

深度学习虽然一定程度上缓解了特征工程的压力，但这里要强调两点：

1. 缓解并不等于彻底解决。除了图像这种特定领域，在个性化推荐等领域，深度学习目前还没有完全取得绝对的优势；究其原因，可能还是数据自身内在结构的问题，使得在其他领域目前还没有发现类似图像 + CNN 这样的完美 CP。
2. 深度学习在缓解特征工程的同时，也带来了模型复杂、不可解释的问题。算法工程师在网络结构设计方面一样要花很多心思来提升效果。

概括起来，深度学习代表的简单特征 + 复杂模型是解决实际问题的另一种方式。

两种模式孰优孰劣还难有定论，以点击率预测为例，在计算广告领域往往以海量特征 + LR 为主流，根据 VC 维理论，LR 的表达能力和特征个数成正比，因此海量的 feature 也完全可以使 LR 拥有足够的描述能力。而在个性化推荐领域，深度学习刚刚萌芽，目前 google play 采用了 WDL 的结构<sup>6</sup>，YouTube 采用了双重 DNN 的结构<sup>7</sup>。

### Note 1.4 VC 维理论

待补充

不管是哪种模式，当模型足够庞大的时候，都会出现模型参数一台机器无法存放的情况。比如百亿级 feature 的 LR 对应的权重  $w$  有好几十个 G，这在很多单机上存储都是困难的，大规模神经网络则更复杂，不仅难以单机存储，而且参数和参数之间还有逻辑上的强依赖，要对超大规模的模型进行训练势必要借用分布式系统的技法，本文主要是系统总结这方面的一些思路。

### Note 1.5 VGG16 每一层参数所占的空间大小

待补充 <https://zhuanlan.zhihu.com/p/31558973>

## 1.1.2 数据并行 vs 模型并行

数据并行和模型并行是理解大规模机器学习框架的基础概念，这两个概念在知乎问题『谈谈你对“GPU/CPU 集群下做 Data/Model Parallelism 的区别”』<sup>8</sup> 中沐帅给出了一个非常直观且经典的解释：

<sup>5</sup>ImageNet Classification with Deep Convolutional Neural Networks

<sup>6</sup>Wide & Deep Learning for Recommender Systems

<sup>7</sup>Deep Neural Networks for YouTube Recommendations

<sup>8</sup><https://www.zhihu.com/question/31999064>

举个栗子。假设我准备盖一座双子楼，我有两个选择：一个是把人分成两组，每组建一栋楼然后装修好，最后把两个连起来。二是仍然分两组，第一组先把楼盖好，第二组负责装修。

第一个方案的好处是并行度高。但要求每个组又要懂建房又要懂装修。第二方案要求低点，一个组会建，一个组会装修就行了。坏处是，一个组在干活的时候另外一个组可能在赋闲。

换成“data/model parallelism”，这里一个组是一个cpu或者一个gpu。第一个方案是 data parallelism，第二方案是 model parallelism。技能要求可以简单看成是对内存的需求。当模型很大不能 fit 进单机内存或者单 GPU 内存（这个可能性高多了）的时候，我们一般用 model parallelism，不然用 data parallelism，因为通常快一些。但某些情况下，model parallelism 的并行程度也没那么低，例如 deep neural network。换回到那个栗子，每一层楼就是 neural network 的一层。这时候我可以流水盖楼：

- 第一天：一组盖第一层
- 第二天：一组盖第二层，二组装修第一层
- 第三天：一组盖第三层，二组装修第二层 …
- 或者只要楼高，可以基本接近 2 倍加速。

第一个方案和数据并行类似，第二个方案则道出了模型并行的精髓。

数据并行理解起来比较简单，当样本比较多的时候，为了使用所有样本来训练模型，不妨把数据分布到不同的机器上，然后每台机器都来对模型参数进行迭代，如图 1.1 所示：

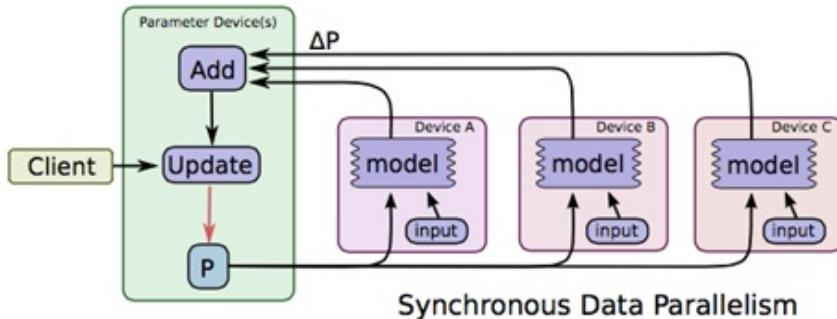


图 1.1: 数据并行

图片取材于 TensorFlow 的 paper<sup>9</sup>，图中 ABC 代表三台不同的机器，上面存储着不同的样本，模型 P 在各台机器上计算对应的增量  $\Delta P$ ，然后在参数存储的机器上进行汇总和更新，这就是数据并行。先忽略 synchronous，这是同步机制相关的概念，在第 1.3.3 节会有专门介绍。

数据并行概念简单，而且不依赖于具体的模型，因此数据并行机制可以作为框架的一种基础功能，对所有算法都生效。与之不同的是，模型并行因为参数间存在依赖关系（其实数据并行中参数更新也可能会依赖所有的参数，但区别在于往往是依赖于上一个迭代的全量参数。而模型并行往往是同一个迭代内的参数之间有强依赖关系，比如 DNN 网络的不同层之间的参数依照 BP 算法形成的先后依赖），无法类比数据并行这样直接将模型参数分片而破坏其依赖关系，所以模型并行不仅要对模型分片，同时需要调度器来控制参数间的依赖关系。而每个模型的依赖关系往往不同，所以模型并行的调度器因模型而异，较难做到完全通用。关于这个问题，CMU 的 Erix Xing 在这里<sup>10</sup> 有所介绍，感兴趣的可以参考。

<sup>9</sup>TensorFlow:Large-Scale Machine Learning on Heterogeneous Distributed Systems

<sup>10</sup><https://www.jianshu.com/p/00736aa21dc8>

**Note 1.6 CMU Erix Xing 的分享**

待补充，有文字版整理以及演讲的 PPT

**Note 1.7 现在模型并行的必要性**

*ResNet50* 模型的整个参数是  $100M$  左右？现在没有模型并行的需求？全连接层的去除？大家都是在使用着数据并行？  
必要性？

**Note 1.8 对于模型并行的调研或相关论文可以推后**

先考察一下现在模型并行的必要性，再开始接触这方面的论文。

模型并行的问题定义可以参考 Jeff Dean 这篇 paper<sup>11</sup>，其作为 TensorFlow 的前身，其中图：

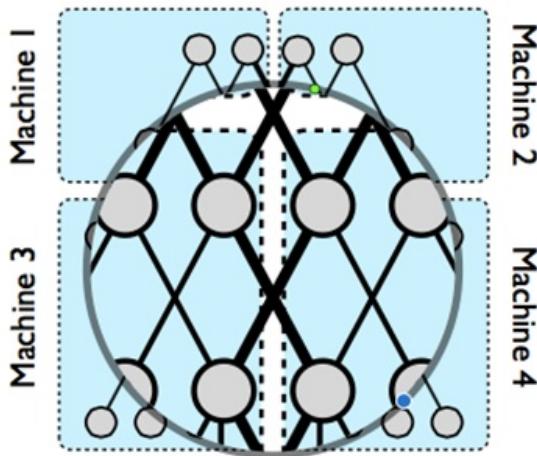


图 1.2: 模型并行

解释了模型并行的物理图景，当一个超大神经网络无法存储在一台机器上时，我们可以切割网络存到不同的机器上，但是为了保持不同参数分片之间的依赖，如图中粗黑线的部分，则需要在不同的机器之间进行 concurrent 控制；同一个机器内部的参数依赖，即图中细黑线部分在机器内即可完成控制。

黑线部分如何有效控制呢？如下图所示：

在将模型切分到不同机器以后，我们将参数和样本一起在不同机器间流转，图中 ABC 代表模型的不同部分的参数；假设 C 依赖 B，B 依赖 A，机器 1 上得到 A 的一个迭代后，将 A 和必要的样本信息一起传到机器 2，机器 2 根据 A 和样本对 P2 更新得到，以此类推；当机器 2 计算 B 的时候，机器 1 可以展开 A 的第二个迭代的计算。了解 CPU 流水线操作的同学一定感到熟悉，是的，模型并行是通过数据流水线来实现并行的。想想那个盖楼的第二种方案，就能理解模型并行的精髓了。

上图则是对控制模型参数依赖的调度器的一个示意图，实际框架中一般都会用 DAG（有向无环图）

<sup>11</sup>Large Scale Distributed Deep Networks

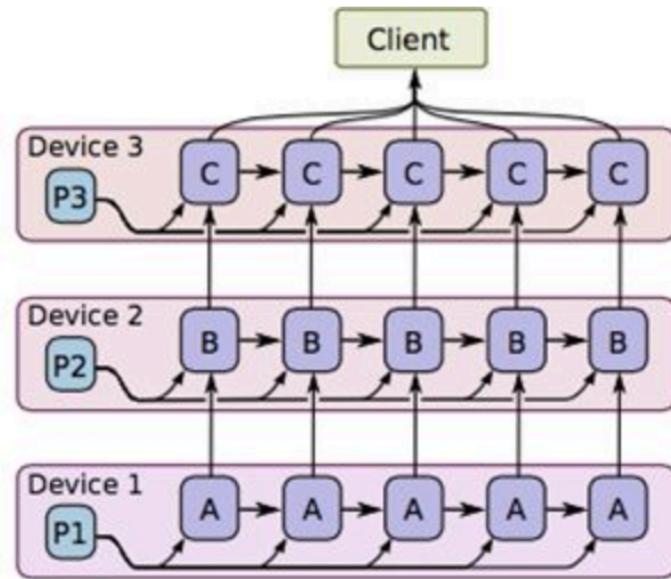


图 1.3: 模型并行中的训练过程

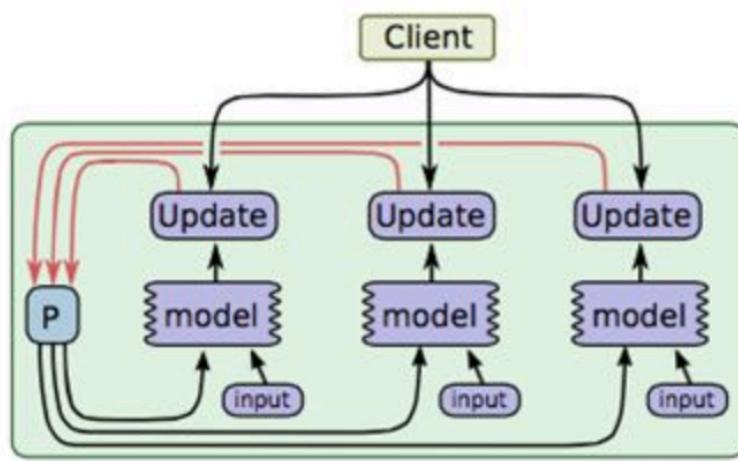


图 1.4: Concurrent 控制

调度技术来实现类似功能，未深入研究，以后有机会再补充说明。

理解了数据并行和模型并行对后面参数服务器的理解至关重要，但现在让我先荡开一笔，简单介绍一下并行计算框架的一些背景信息。

### 待消化之后补充：

GPU 上的数据/模型并行与 CPU 集群上的并没有本质区别。从 Fundamental 的角度来说，都是要解决几个问题：

1. 通过并行化来做性能加速
2. 把不可行变为可行（单机内存无法 hold 的 huge model size）

从手段上来说，抽象来看都是这几种：

1. 将计算与通信开销 overlap 以后，提高计算资源的 utilization rate[1]
2. 结合具体的算法业务场景，对网络的拓扑结构进行优化，在不明显损失精度的情况下，减少并行计算的同步过程所需要传输的数据量（典型的例子就是 CNN 里卷积层的 local connectivity, weight sharing, 以及 pooling 的设计理念 [7]）
3. 将算法与系统相结合，在不明显损失精度的情况下，改善计算资源的 utilization rate（典型的例子就是 Hogwild 为代表的 ASGD 算法 [2] 以及 Eric Xing 的团队提出的 SSP[3]）

具体到操作层面，因为 CPU 与 GPU 在计算特点上的差异，还是会引入一些区别：

1. 相较于 CPU 而言，GPU 更强大的『naive』浮点算术能力（这里提到『naive』是为了突出 GPU 硬件设计上忽略复杂的控制结构，而把更多的芯片资源用于 computation intensive 的计算逻辑并辅之以 high latency/high bandwidth[8] 的显存资源，来获取到浮点计算吞吐能力的显著提升 — no free lunch 的又一个典型 use case）使得它跟通信的性能 gap 要更为突显，也就使得 GPU 集群上，因为计算与通信的 gap 导致的性能 degradation 会更显著。在 GPU 集群里，引入高速通信链路 (InfiniBand) 以及特定的硬件技术支持 (GPU Direct RDMA[11]) 也都是为了缓解计算与通信的 gap(对于 GPU 与 CPU 计算能力的差异，既不能低估，也不应该过分高估，参见 Intel 发在 ISCA10 上的这篇文章 [5])。
2. GPU 的访存特点也使得 GPU 计算平台上能 hold 的有效模型尺寸通常来说是远小于 CPU 平台上的（以比较主流的 Nvidia Tesla K40M[6] 为例，显存 12GiB），这也使得 GPU 平台上在处理比较大的模型的时候，会比 CPU 平台更早地遇到模型尺寸的瓶颈，需要考虑 model parallelism。而 GPU 异构计算的复杂性和非黑盒特性也使得 GPU 平台上做 model parallelism 需要考虑更多的 constraint( global memory/shared memory/register file 这些不同存储单元的延迟及带宽差异)，相应会要求更为精巧的设计。

总的来说，我个人以为 GPU 和 CPU 集群上的分布式机器学习系统的搭建并不存在本质上的差异，但是存在操作层面的区别。这些区别的有效 handle 非常依赖于强工程能力，特别是良好的体系结构背景与机器学习 common sense 的有效结合。具体来说，结合硬件的具体指标，有效平衡好计算与通信的耗时比例，以及 CGMA(Computation to Global Memory Access) ratio[14]，对于保证 GPU 集群上的程序性能是有着重要帮助的。即便是有了 CUDA-Aware MPI[10](OpenMPI 从 1.7.4[12] 开始就已经既支持 InfiniBand 也支持 GPU Direct 了)，让编程界面变得跟开发常规 MPI 程序更像了，但是对底层硬件性能指标的精确把握才可能让我们真正用好这些编程 API 完成分布式机器学习系统的开发工作。

再从一个更大的 picture 来看，分布式机器学习系统，本质上跟常规的分布式系统并没有差异。虽然在 Eric Xing 的文章 [9] 里提到了 pure-system 或 pure-algorithm 的视角看待分布式学习系统的 limitation。但实际上从工业界应用的角度来看 mitigate 这个 limitation 所需要的代价其实并不会非常之高（我个人以为 Xing 教授在 Large Scale Learning 方面关于异步更新场景下收敛性证明的一些理论性工作对于分布式机器学习系统的未来演进有着方向性的重要意义和价值，但是仅从工业应用的角度来看还并没有构成严格的必要前提条件，有则更佳，欠乏其实并不会影响实际的业务推进和应用）。所以我还是会更愿意把

分布式机器学习系统看成分布式系统在特定业务场景下的一个 use case。有扎实的分布式系统经验的人，加以专门的 Machine Learning 相关的培训，其实并不难成长为分布式机器学习系统的专家（五年前，能够用 MPI 实现并行 LR 算法，可能还算是相当不错的技术成绩。而现在，一个基础扎实的应届生，给他合适的 mentorship，其实花两三个月的时间也就能够实现一个可用于支持线上生产训练任务的版本了）。

现在，因为技能本身的 diversity，拥有分布式机器学习系统开发经验的工程师还是稀缺资源，但是我会有一种感觉，随着时间的推移，人才的流动，会有更多的人能够掌握相关的经验技能，这个技能的稀缺性也会渐渐地下降到一个稳定值。大规模机器学习系统的 Cross-domain 性质所带来的固有难度还是存在一个固有的门坎，但是在我看来，这些门坎更多是一些 know-how 性质的门坎，而不是 fundamental 的门坎，是可以通过建立有效的培养机制来解决掉的。举例来说，不理解 L1 和 L2 正则为什么一个对应于 Laplace 先验，另一个对应于高斯先验，并不会影响我们去实现相应的分布式 L1/L2 LR 算法；同样，对矩阵分解理论缺乏精细的理解和把控，也不会构成我们去实现一个 SVD Feature 或是 Tensor Decomposition 算法的本质障碍；没有办法把 EM 算法的原理推敲得很清楚，也不会影响我们去实现一个并行版的 PLSA 主题模型。从工程实现的角度去完成一个 academia community 设计出来的大规模机器学习算法和自己从头设计一个大规模机器学习算法所需要的背景知识还是不一样的。前者更适合工业界的同学来完成，后者则更适合 academia 的同学来担当。（注意，我上面说的都是大规模算法的实现，而不是大规模算法的应用，我个人的观点，真正要把这些算法有效地应用到业务场景里，难度反而更大，不是仅仅靠良好的 engineering practice 和 machine learning common sense 就可以做到的，而会需要更扎实的数据统计以及数据挖掘/机器学习的专业背景）

个人来说，我更期待学术界能够从更 fundamental 的层面为 large scale learning 提供支持。因为现在的 large scale learning 系统在工业界的应用，我觉得还更多是一种以 embarrassingly parallel 为代表的暴力式方法带来性能优化。data parallelism/model parallelism 不外如是，ASGD 也算是一个“暴力”性质的例子（异步 SGD 虽然看起来很美，但是从实际的 implementation 上来看，在不同数据集上保证良好的收敛性并不是件容易的事情，在带来计算资源的更有效 utilization rate 的同时，ASGD 固有的随机性其实让达到相同收敛点所需耗费的时间更长，变数也更多。G 家的 Paper[4] 虽然 claim 说在 non-convex, non-sparse 场景下，他们使用 ASGD+Adagrad 也能够获得不错的收敛效果，但是缺失了严格的理论证明总会让人猜测有一些论文中没有透露的 tricks 才使得他们保证了 ASGD 在不同数据集上的收敛能力）。这种“暴力美学”性质的优化工业界有足够的能力去 handle，而工业界因其自身定位特别是业务压力，未必有能力和精力 handle 的则是一些通过算法设计改良将算法时间复杂度或空间复杂度大幅降下来的 case，典型的例子就是 Newton 法之于 Gradient Descent，LBFGS 算法之于原生的 BFGS 算法。以及对算法改良的 theory justification，比如 Eric Xing 的团队在 bounded delay 场景下异步更新收敛性证明方面做过的一些工作。

最后简单 summarize 一下：

1. GPU 与 CPU 集群上 Model/Data Parallelism 不存在本质上的区别，更多是存在工程细节上的区别。对这些工程细节的有效 handle 依赖于强体系结构背景和机器学习 common sense 的有机结合。
2. 大规模机器学习系统本质上可以视为分布式系统的一个特例。虽然现在从事大规模机器学习系统开发还算能力，但是开发大规模机器学习系统所需的技能经验是可以通过有效的培养机制加以养成的。
3. 为了担心误导，特地加入对 2 的补充。只是开发出可以用于支持线上常规业务 (CTR Prediction/搜索推荐/文本潜层语义挖掘，以及比较成熟的 Deep Learning 应用场景—Speech and CV) 的大规模机器学习系统的技能经验是不难培养的，但是真正顶尖的相关能力则是不太容易培养的（似乎任何行当都是如此）。在资讯发达，行业交流频繁的年代，只是写一个可以 work 并支持线上业务基本需求的大规模并行算法并不存在本质困难（就算是一些现在不知道应该怎么做的算法，随着时间推移也不难从别人那里听说个大概，自然也就可以完成相应的实现了），但是结合业务场景，进行算法定制，来帮助业务建立起技术壁垒，就需要在 engineering practice, machine learning common sense

之外，对 domain knowledge(既指代技术知识，也包括业务知识) 建立起更深刻的理解和把握了，这种能力是需要通过长时间的思考，沉淀，持续地追踪最新的学术进展，同时 touch 多个 domain 才可能建立起来的，非量产品。

Alex2014 One Weird Trick for Parallelizing CNN

Niu2011 Hogwild!: A lock-free approach to parallelizing stochastic gradient descent

Qirong2013 More Effective Distributed ML via a Stale Synchronous Parallel Parameter Server

Jeff2012 Large scale distributed deep networks

Victor2010 Debunking the 100X GPU vs. CPU Myth: An Evaluation of Throughput Computing on CPU and GPU

Nvidia 官网 <http://www.nvidia.com/content/PDF/kepler/nvidia-tesla-k40.pdf>

Jarret2009 What is the Best Multi-Stage Architecture for Object Recognition?

Patterson Why Latency Lags Bandwidth, and What it Means to Computing

Eric2014 <http://www.cs.cmu.edu/~epxing/talks/IEEE2014-BigData-Tutorial.pdf>

Nvidia 官网 NVIDIA GPUDirect <https://developer.nvidia.com/gpudirect>

Nvidia 官网 <http://on-demand.gputechconf.com/gtc/2013/presentations/S3047-Intro-CUDA-Aware-MPI-NVIDIA-GPUDirect.pdf>

OpenMPI GitHub <https://github.com/open-mpi/ompi/>

David B.Kirk2013 Programming Massively Parallel Processors, 2nd Edition, Chapter 5, Section 5.1

整理自：<https://www.zhihu.com/question/31999064/answer/54185461>

### Note 1.9 模型并行带来的两方面益处

对神经网络进行划分，比如水平、垂直划分等等，模型并行带来了两方面的益处

- 能处理更大规模的模型，因为单机内存有限，特别是 GPU 内存
- 带来了流水线优化，提升了计算效率。

但也有缺点，比如在水平划分模型时，中间的某一层计算需要上一层所有的数据都计算完毕，如果有数据未完成，整个计算都会延迟，也就是木桶效应（计算效率由最慢的计算决定，而水平模型划分加剧了这类风险）。这里就可以用体系结构的方法来处理这些问题了。

但数据并行和模型并行，二者本质上没有区别，都是为了提高并行度，增加计算效率，最大化利用计算资源。但是模型并行还能够带来对更大规模的神经网络的处理能力。

整理自：<https://www.zhihu.com/question/31999064/answer/106715799>

### Note 1.10 GPU 内存



## 1.2 并行算法演进

### 1.2.1 MapReduce 路线

从函数式编程中的受到启发，Google 发布了 MapReduce 的分布式计算框架，通过将任务切分成多个叠加的 Map+Reduce 任务，来完成复杂的计算任务，示意图如下

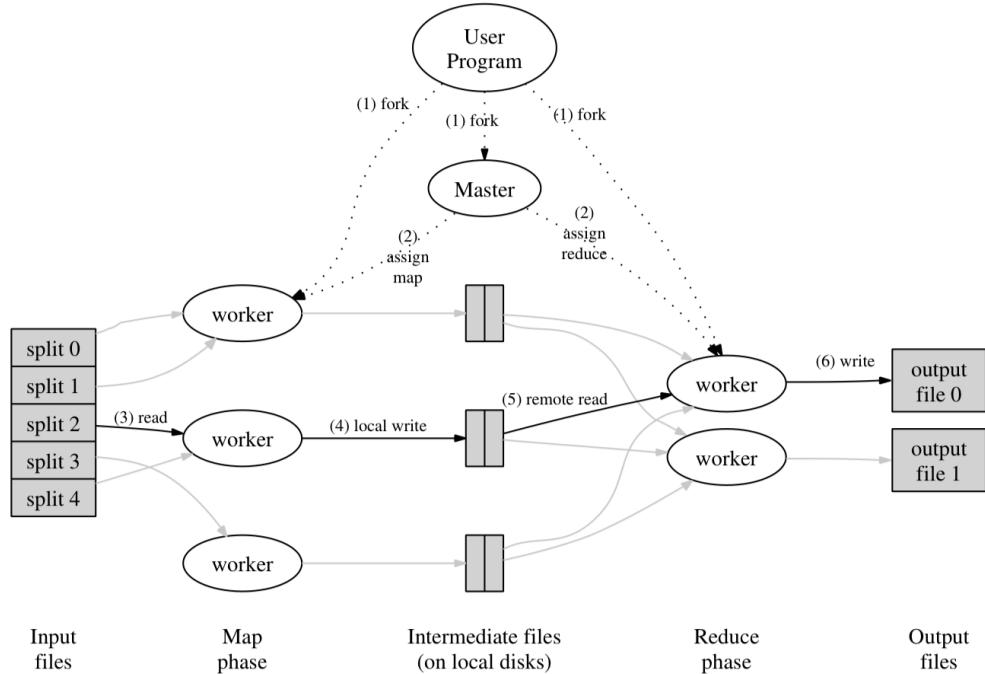


图 1.5: MapReduce 的整个流程

**Note 1.11 MapReduce**

MapReduce 是一种编程模型。阅读 *MapReduce* 论文，写阅读笔记。

MapReduce 的主要问题有两个：

1. 原语的语义过于低级，直接使用其来写复杂算法，开发量比较大；
2. 依赖于磁盘进行数据传递，性能跟不上业务需求。

为了解决 MapReduce 的两个问题，Matei 提出了一种新的数据结构 RDD<sup>12</sup>，并构建了 Spark 框架。Spark 框架在 MapReduce 语义之上封装了 DAG 调度器，极大降低了算法使用的门槛。

**Note 1.12 RDD**

RDD 弹性数据集。阅读 *Spark* 论文，写阅读笔记。

较长时间内 Spark 几乎可以说是大规模机器学习的代表，直至后来沐帅的参数服务器进一步开拓了大规模机器学习的领域以后，Spark 才暴露出一点点不足。如下图：

从图中可以看出，Spark 框架以 Driver 为核心，任务调度和参数汇总都在 Driver，而 Driver 是单机结构，所以 Spark 的瓶颈非常明显，就在 Driver 这里。当模型规模大到一台机器存不下的时候，Spark 就无法正常运行了。所以从今天的眼光来看，Spark 只能称为一个中等规模的机器学习框架。腾讯开源的 Angel 通过修改 Driver 的底层协议将 Spark 扩展到了一个高一层的境界。后面还会再详细介绍这部分。

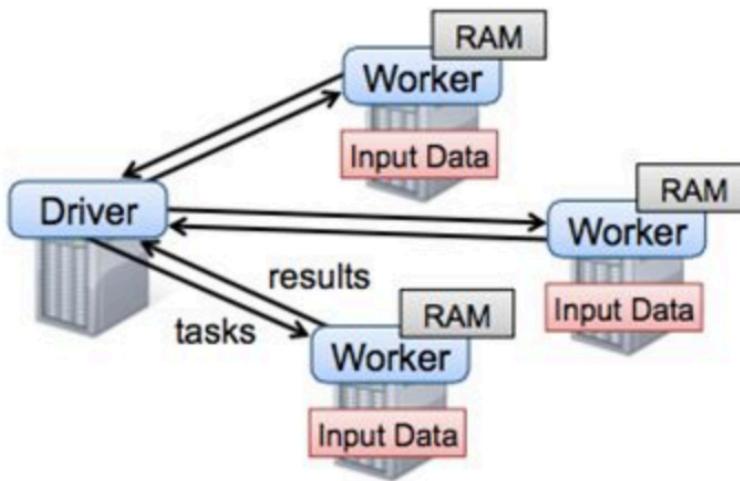


图 1.6: Spark 的架构

**Note 1.13 Spark**

*Spark* 是一个内存计算框架。

MapReduce 不仅是一个框架，还是一种思想，Google 开创性的工作为我们找到了大数据分析的一个可行方向，时至今日，仍不过时。只是逐渐从业务层下沉到底层语义应该处于的框架下层。

### 1.2.2 MPI 技术

沐帅在知乎问题『MPI 在大规模机器学习领域的前景如何？<sup>13</sup>』中对 MPI 的前景做了简要介绍：

这个取决于场景，如果要在超算（super computer）上跑机器学习，用 MPI 是不错的选择。但在云上，不管是公有云的例如 AWS/Azure/GCP 或者私有云，MPI 没有太多必要。

MPI 定义的是接口，具体用的时候我们是用某个特定的实现，例如 openmpi 或者 mpich2。为了简单，这里统一叫 MPI。

用 MPI 实现机器学习是没问题的，不管是对高维稀疏模型还是深度学习。MPI 的接口在一些算法的实现上很方便，另外一些地方（例如异步）绕一绕也是可以的。例如虽然 MXNet 没有用 MPI，不过使用 MPI 来实现个 kvstore 的 backend 也是可行的。

MPI 的一大优势是支持各种网络硬件，例如 infiniband 和 Intel Omni Path，或者网络拓扑结构，例如 cray 的 dragonfly。通常 MPI 会做各种针对性的优化从而得到不错的性能。

这个优势主要体现在超算上。而对于云，通常使用常见的网络硬件（例如 Ethernet）和连接结构（multi-rooted tree），MPI 做的优化通常不会有太大效果。在云上，MPI 的一大问题是容灾。任何一个节点出问题会导致整个任务失败，会导致运营成本增加。（回答里面有提到百度有 MPI 集群。我当年是最大的用户，一度使用超过 50% 的节点。对于这一点我是深有体会，例如凌晨 3 点起来重启任务。）

所以结论是，如果要在云上跑机器学习的话，MPI 前景不大。但如果是使用 Top500 类似的超算，MPI 是不错的选择。

<sup>13</sup><https://www.zhihu.com/question/55119470>

和 Spark 不同，MPI 是类似 socket 的一种系统通信 API，只是支持了消息广播等功能。因为对 MPI 研究不深入，这里简单介绍下优点和缺点吧。优点是系统级支持，性能杠杠的；缺点也比较多，一是和 MapReduce 一样因为原语过于低级，用 MPI 写算法，往往代码量比较大。另一方面是基于 MPI 的集群，如果某个任务失败，往往需要重启整个集群，而 MPI 集群的任务成功率并不高。阿里的鲲鹏系统<sup>14</sup>给出了下图：



图 1.7: MPI 集群的任务成功率

从图中可以看出，MPI 作业失败的几率接近五成。MPI 也并不是完全没有可取之处，正如沐帅所说，在超算集群上还是有场景的。对于工业届依赖于云计算、依赖于 commodity 计算机来说，则显得性价比不够高。当然如果在参数服务器的框架下，对单组 worker 再使用 MPI 未尝不是个好的尝试，鲲鹏系统正是这么设计的。

#### Note 1.14 MPI

因为对 MPI 在之前基本就没有怎么了解过，考察一下，如果 MPI 现在使用的还比较广泛，这个可能需要整理成一个新的章节来。

## 1.3 参数服务器演进

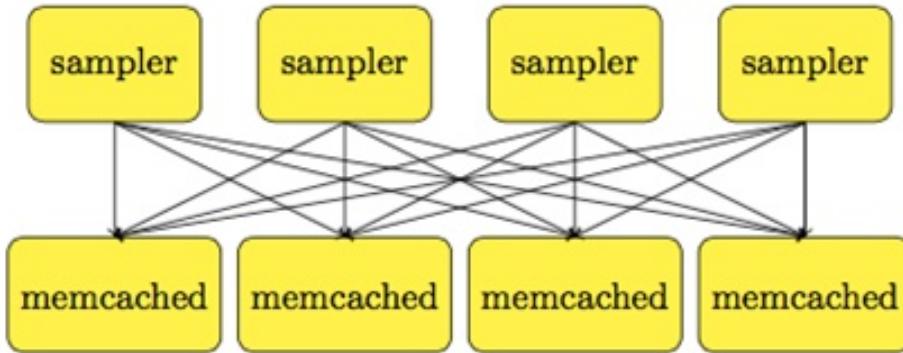
### 1.3.1 历史演进

沐帅在 OSDI2014 的一篇文章<sup>15</sup>中将参数服务器的历史划分为三个阶段，第一代参数服务器萌芽于沐帅的导师 Smola 的 paper<sup>16</sup>，如下图所示：

<sup>14</sup>KunPeng:Parameter Server based Distributed Learning Systems and Its Applications in Alibaba and Ant Financial

<sup>15</sup>Scaling Distributed Machine Learning with the Parameter Server

<sup>16</sup>An Architecture for Parallel Topic Models



**Figure 3: Each sampler keeps on processing the subset of data associated with it. Simultaneously a synchronization thread keeps on reconciling the local and global state tables.**

图 1.8: xxxxxx 待补充

这个工作中仅仅引入 memcached 来存放 key-value 数据，不同的处理进程并行对其进行处理。另一篇文章<sup>17</sup>中也有类似的想法。第二代参数服务器叫 application-specific 参数服务器，主要针对特定应用而开发，其中最典型的代表应该是 TensorFlow 的前身 DistBelief<sup>18</sup>。

第三代参数服务器，也即是通用参数服务器框架是由百度少帅李沐正式提出的，和前两代不同，第三代参数服务器从设计上就是作为一个通用大规模机器学习框架来定位的。要摆脱具体应用、算法的束缚，做一个通用的大规模机器学习框架，首先就要定义好框架的功能；而所谓框架，往往就是把大量重复的、琐碎的、做了一次就不想再来第二次的脏活、累活进行良好而优雅的封装，让使用框架的人可以只关注于自己的核心逻辑。第三代参数服务器要对哪些功能进行封装呢？沐帅总结了这几点，我照搬如下：

- **高效的网络通信：**异步通信，计算与通信重叠（overlap）；
- **灵活的一致性模型：**不同的一致性模型其实是在模型收敛速度和集群计算量之间做 tradeoff，要理解这个概念需要对模型性能的评价做些分析，暂且留到下节再介绍；
- **弹性可扩展：**新节点的加入不需要重启整个运行的框架；
- **容灾容错：**大规模集群协作进行计算任务的时候，出现 Straggler 或者机器故障是非常常见的事，因此系统设计本身就要考虑到应对；没有故障的时候，也可能因为对任务时效性要求的变化而随时更改集群的机器配置。这也需要框架能在不影响任务的情况下能做到机器的热插拔；
- **易用性：**主要针对使用框架进行算法调优的工程师而言，显然，一个难用的框架是没有生命力的。

在正式介绍第三代参数服务器的主要技术之前，先从另一个角度来看下大规模机器学习框架的演进。

这张图可以看出，在参数服务器出来之前，人们已经做了多方面的并行尝试，不过往往只是针对某个特定算法或特定领域，比如 YahooLDA 是针对 LDA 算法的。当模型参数突破十亿以后，则可以看出参数服务器一统江湖，再无敌手。

### 1.3.2 基础架构

第三代参数服务器的基本架构

<sup>17</sup>Piccolo:Building fast, distributed programs with partitioned tables

<sup>18</sup>Large Scale Distributed Deep Networks

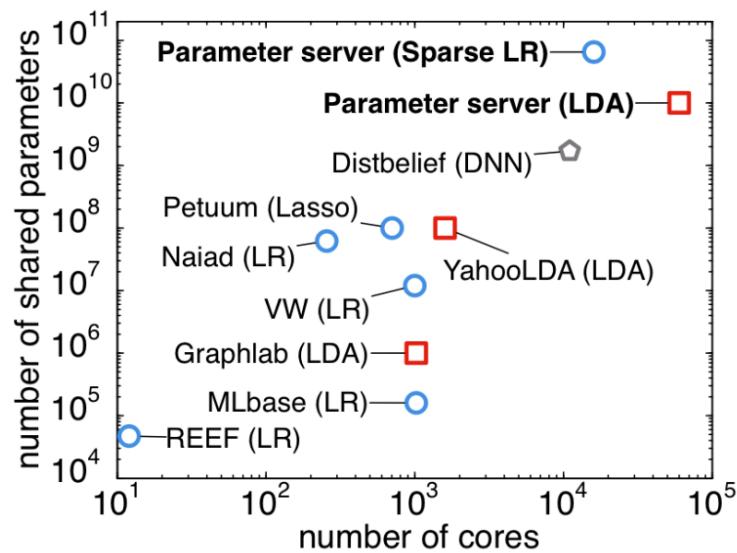


图 1.9: 参数服务器与其他系统执行机器学习任务的比较

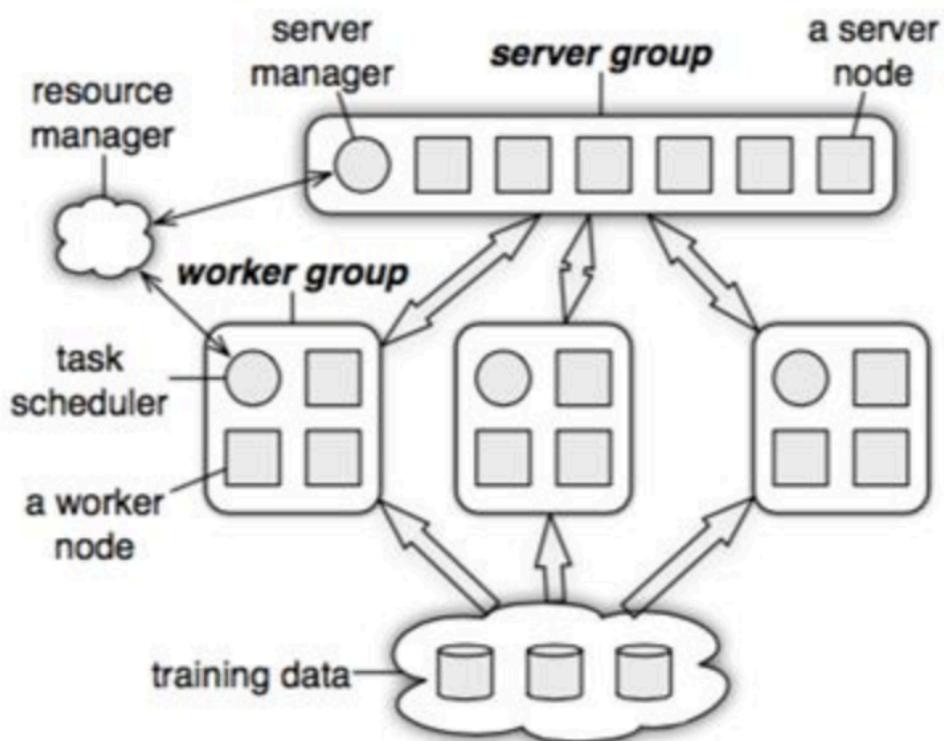


Figure 4: Architecture of a parameter server communicating with several groups of workers.

图 1.10: xxxxxxx 待补充

上图的 resource manager 可以先放一放，因为实际系统中这部分往往是复用现有的资源管理系统，比如 yarn 或者 mesos；底下的 training data 毫无疑问需要类似 GFS 的分布式文件系统的支持；剩下的部分就是参数服务器的核心组件了。

图中画了一个 server group 和三个 worker group；实际应用中往往也是类似，server group 用一个，而 worker group 按需配置；server manager 是 server group 中的管理节点，一般不会有逻辑，只有当有 server node 加入或退出的时候，为了维持一致性哈希而做一些调整。

worker group 中的 task schedule 则是一个简单的任务协调器，一个具体任务运行的时候，task schedule 负责通知每个 worker 加载自己对应的数据，然后去 server node 上拉取一个要更新的参数分片，用本地数据样本计算参数分片对应的变化量，然后同步给 server node；server node 在收到本机负责的参数分片对应的所有 worker 的更新后，对参数分片做一次 update。

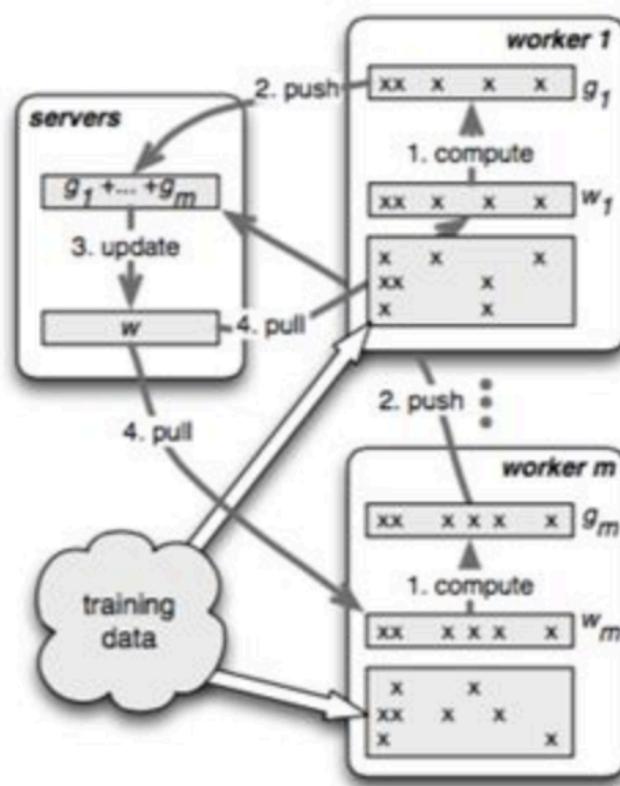


图 1.11: xxxxxxx 待补充

如图所示，不同的 worker 同时并行运算的时候，可能因为网络、机器配置等外界原因，导致不同的 worker 的进度是不一样的，如何控制 worker 的同步机制是一个比较重要的课题。详见下节分解。

#### ps-lite 代码详细解析

### 1.3.3 同步协议

本节假设读者已经对随机梯度优化算法比较熟悉，如果不熟悉的同学请参考吴恩达经典课程机器学习中对 SGD 的介绍，或者我之前多次推荐过的书籍《**最优化导论**》。

我们先看一个单机算法的运行过程，假设一个模型的参数切分成三个分片 k1, k2, k3；比如你可以假设是一个逻辑回归算法的权重向量被分成三段。我们将训练样本集合也切分成三个分片 s1, s2, s3；在单机运行的情况下，我们假设运行的序列是 (k1, s1)、(k2, s1)、(k3, s1)、(k1, s2)、(k2, s2)、(k3,

s2) ... 看明白了吗？就是假设先用 s1 中的样本一次对参数分片 k1、k2、k3 进行训练，然后换 s2；这就是典型的单机运行的情况，而我们知道这样的运行序列最后算法会收敛。

现在我们开始并行化，假设 k1、k2、k3 分布在三个 server node 上，s1、s2、s3 分布在三个 worker node 上，这时候如果我们还要保持之前的计算顺序，则会变成怎样？work1 计算的时候，work2 和 work3 只能等待，同样 work2 计算的时候，work1 和 work3 都得等待，以此类推；可以看出这样的并行化并没有提升性能；但是也算简单解决了超大规模模型的存储问题。

为了解决性能的问题，业界开始探索这里的一致性模型，最先出来的版本是前面提到的 ASP 模式<sup>19</sup>，就是完全不顾 worker 之间的顺序，每个 worker 按照自己的节奏走，跑完一个迭代就 update，然后继续，这应该是大规模机器学习中的 freestyle 了，如图所示

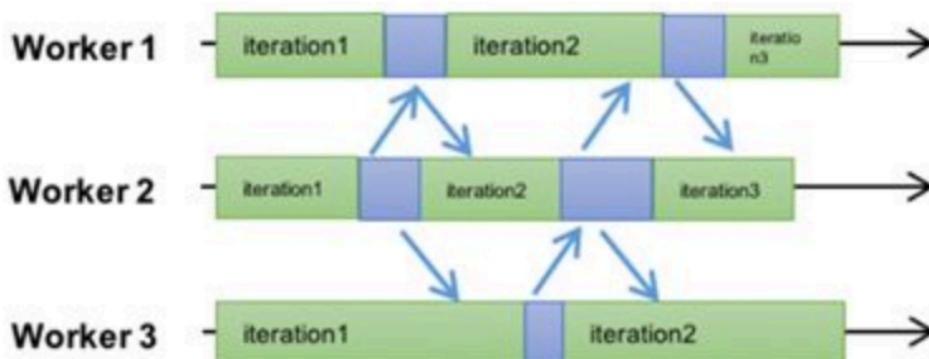


图 1.12: ASP 模式

ASP 的优势是最大限度利用了集群的计算能力，所有的 worker 所在的机器都不用等待，但缺点也显而易见，除了少数几个模型，比如 LDA，ASP 协议可能导致模型无法收敛。也就是 SGD 彻底跑飞了，梯度不知道飞到哪里去了。

在 ASP 之后提出了另一种相对极端的同步协议 BSP，Spark 用的就是这种方式，如图所示

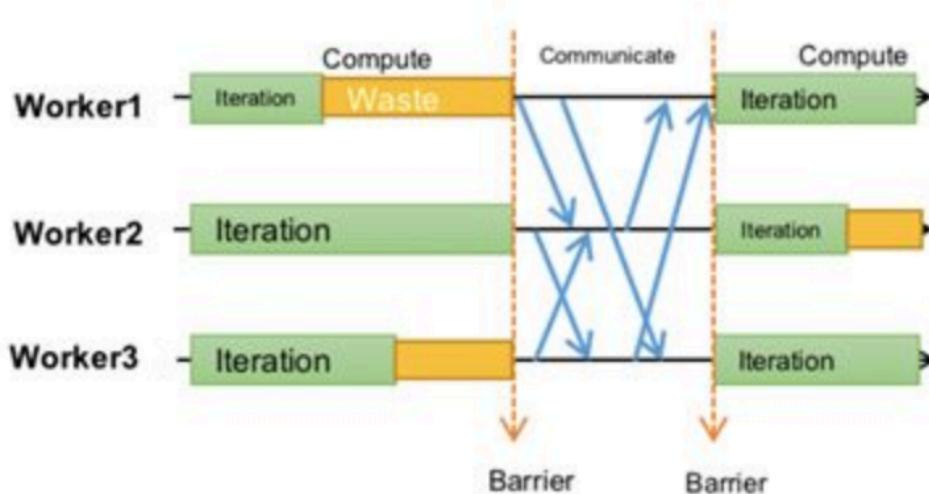


图 1.13: BSP 模式

每个 worker 都必须在同一个迭代运行，只有一个迭代任务所有的 worker 都完成了，才会进行一次

<sup>19</sup>An Architecture for Parallel Topic Models

worker 和 server 之间的同步和分片更新。这个算法和严格一致的算法非常类似，区别仅仅在于单机版本的 batch size 在 BSP 的时候变成了所有 worker 的单个 batch size 求和得到的总的 batch size 替换。毫无疑问，BSP 的模式和单机串行因为仅仅是 batch size 的区别，所以在模型收敛性上是完全一样的。同时，因为每个 worker 在一个周期内是可以并行计算的，所以有了一定的并行能力。

以此协议为基础的 Spark 在很长时间内成为机器学习领域实际的霸主，不是没有理由的。此种协议的缺陷之处在于，整个 worker group 的性能由其中最慢的 worker 决定；这个 worker 一般称为 straggler。读过 GFS<sup>20</sup>文章的同学应该都知道 straggler 的存在是非常普遍的现象。

#### Note 1.15 GFS 中的 Straggler

阅读 GFS 中的 Straggler 部分

能否将 ASP 和 BSP 做一下折中呢？答案当然是可以的，这就是目前我认为最好的同步协议 SSP；SSP 的思路其实很简单，既然 ASP 是允许不同 worker 之间的迭代次数间隔任意大，而 BSP 则只允许为 0，那我是否可以取一个常数  $s$ ？如图所示

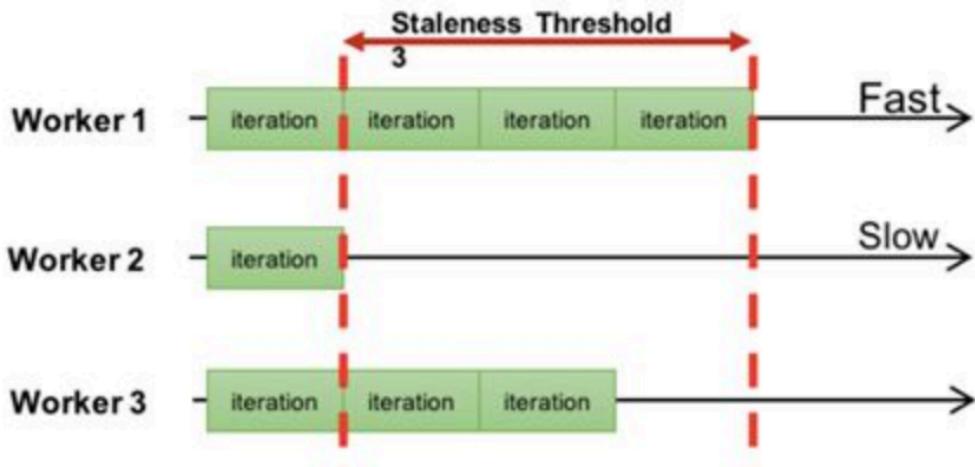


图 1.14: SSP 模式

不同的 worker 之间允许有迭代的间隔，但这个间隔数不允许超出一个指定的数值  $s$ ，图中  $s=3$ 。

SSP 协议的详细介绍参见<sup>21</sup>，CMU 的大拿 Eric Xing 在其中详细介绍了 SSP 的定义，以及其收敛性的保证。理论推导证明常数  $s$  不等于无穷大的情况下，算法一定可以在若干次迭代以后进入收敛状态。其实在 Eric 提出理论证明之前，工业界已经这么尝试过了：)

#### Note 1.16 从另一个角度看数据并行

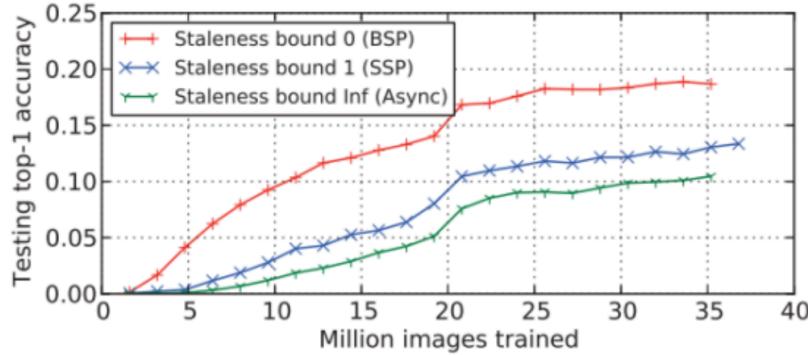
在随机梯度下降中，数据并行使用参数服务器（Parameter Server）来做同步，同步的策略就是我们上面提到的 ASP、BSP 以及 SSP。在 GeePS 中，提到同步策略对更新效率的影响，BSP 最好，SSP 次优，ASP 最差；但 BSP 每轮都需要同步，训练一批数据时间会最长，SSP 次优，ASP 最快。所以这就是一个 tradeoff 的问题了：如何选择同步策略？

GeePS: Scalable deep learning on distributed GPUs with a GPU-specialized parameter server

整理自：<https://www.zhihu.com/question/31999064/answer/106715799>

<sup>20</sup>The Google File System

<sup>21</sup>More Effective Distributed ML via a Stale Synchronous Parallel Parameter Server



**Figure 15.** AdamLike model top-1 accuracy as a function of the number of training images processed, for BSP, SSP with slack 1, and Async.

图 1.15: ASP、BSP 以及 SSP 各同步策略对更新效率的影响

顺便提一句，考察分布式算法的性能，一般会分为 statistical performance 和 hard performance 来看。前者指不同的同步协议导致算法收敛需要的迭代次数的多少，后者是单次迭代所对应的耗时。两者的关系和 precision/recall 关系类似，就不赘述了。有了 SSP, BSP 就可以通过指定  $s=0$  而得到。而 ASP 同样可以通过指定  $s=\infty$  来达到。

### 1.3.4 核心技术

除了参数服务器的架构、同步协议之外，本节再对其他技术做一个简要的介绍，详细的了解请直接阅读沐帅的博士论文和相关发表的论文<sup>22</sup>。

热备、冷备技术：为了防止 server node 挂掉，导致任务中断，可以采用两个技术，一个是对参数分片进行热备，每个分片存储在三个不同的 server node 中，以 master-slave 的形式存活。如果 master 挂掉，可以快速从 slave 获取并重启相关 task。除了热备，还可以定时写入 checkpoint 文件到分布式文件系统来对参数分片及其状态进行备份。进一步保证其安全性。

server node 管理：可以使用一致性哈希技术来解决 server node 的加入和退出问题，如图所示

当有 server node 加入或退出的时候，server manager 负责对参数进行重新分片或者合并。注意在对参数进行分片管理的情况下，一个分片只需要一把锁，这大大提升了系统的性能，也是参数服务器可以实用的一个关键点。

## 1.4 大规模机器学习的四重境界

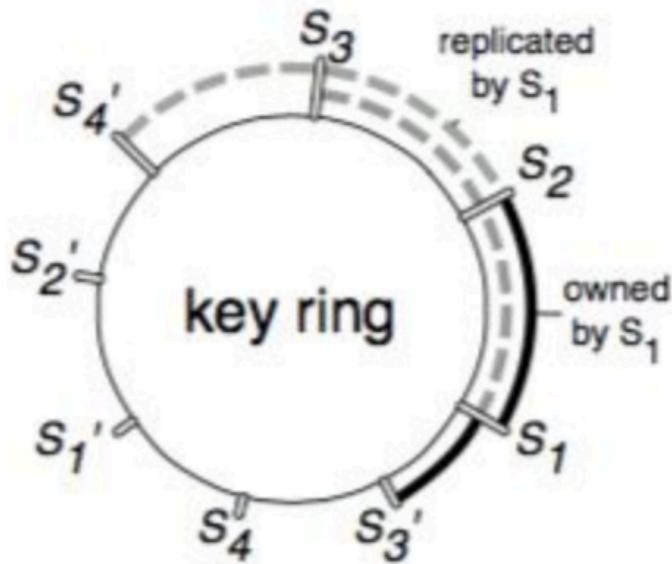
到这里可以回到我们的标题了，大规模机器学习的四重境界到底是什么呢？这四重境界的划分是作者个人阅读总结的一种想法，并不是业界标准，仅供大家参考。

### 1.4.1 境界 1

#### 参数可单机存储和更新

此种境界较为简单，但仍可以使用参数服务器，通过数据并行来加速模型的训练。

<sup>22</sup><http://www.cs.cmu.edu/~muli/>



**Figure 7: Server node layout.**

图 1.16: Server node layout

### 1.4.2 境界 2

#### 参数不可单机存储，可以单机更新

此种情况对应的是些简单模型，比如 sparse logistic regression；当 feature 的数量突破百亿的时候，LR 的权重参数不太可能在一台机器上完全存下，此时必须使用参数服务器架构对模型参数进行分片。但是注意一点，SGD 的更新公式

$$w' = w - \alpha$$

其中可以分开到单个维度进行计算，但是单个维度的  $w_i = f(x)x_i$ 。

这里的  $f(w)$  表示是全部参数  $w$  的一个函数，具体推导比较简单，这里篇幅所限就不赘述了。只是想说明 worker 在计算梯度的时候可能需要使用到上一轮迭代的所有参数。而我们之所以对参数进行分片就是因为我们无法将所有参数放到一台机器，现在单个 worker 有需要使用所有的参数才能计算某个参数分片的梯度，这不是矛盾吗？可能吗？

答案是可能的，因为单个样本的 feature 具有很高的稀疏性 (sparseness)。例如一个百亿 feature 的模型，单个训练样本往往只在其中很小一部分 feature 上有取值，其他都为 0 (假设 feature 取值都已经离散化了)。因此计算  $f(w)$  的时候可以只拉取不为 0 的 feature 对应的那部分  $w$  即可。有文章统计一般这个级别的系统，稀疏性往往在 0.1% (or 0.01%，记得不是很准，大致这样) 以下。这样的稀疏性，可以让单机没有任何阻碍的计算  $f(w)$ 。

目前腾讯开源的 Angel 和 AILab 正在做的系统都处于这个境界。而原生 Spark 还没有达到这个境界，只能在中小规模的圈子里厮混。

### 1.4.3 境界 3

#### 参数不可单机存储，不可单机更新，但无需模型并行

境界 3 顺延境界 2 而来，当百亿级 feature 且 feature 比较稠密的时候，就需要计算框架进入到这层境界了，此时单个 worker 的能力有限，无法完整加载一个样本，也无法完整计算  $f(w)$ 。怎么办呢？其实很简单，学过线性代数的都知道，矩阵可以分块。向量是最简单的矩阵，自然可以切成一段一段的来计算。只是调度器需要支持算符分段而已了。

#### 1.4.4 境界 4

##### 参数不可单机存储，不可单机更新，需要模型并行

进入到这个层次的计算框架，可以算是世界一流了。可以处理超大规模的神经网络。这也是最典型的应用场景。此时不仅模型的参数不能单机存储，而且同一个迭代内，模型参数之间还有强的依赖关系，可以参见姐夫对 Distbelief 的介绍里的模型切分。

此时首先需要增加一个 coordinator 组件来进行模型并行的 concurrent 控制。同时参数服务器框架需要支持 namespace 切分，coordinator 将依赖关系通过 namespace 来进行表示。

一般参数间的依赖关系因模型而已，所以较难抽象出通用的 coordinator 来，而必须以某种形式通过脚本 parser 来生产整个计算任务的 DAG 图，然后通过 DAG 调度器来完成。对这个问题的介绍可以参考 Erix Xing 的分享<sup>23</sup>。

#### 1.4.5 TensorFlow

目前业界比较知名的深度学习框架有 Caffe、MXNet、Torch、Keras、Theano 等，但目前最炙手可热的应该是 Google 发布的 TensorFlow。这里单独拿出来稍微分解下。

前面不少图片引自此文<sup>24</sup>，从 TF 的论文来看，TF 框架本身是支持模型并行和数据并行的，内置了一个参数服务器模块，但从开源版本所曝光的 API 来看，TF 无法用来 10B 级别 feature 的稀疏 LR 模型。原因是已经曝光的 API 只支持在神经网络的不同层和层间进行参数切分，而超大规模 LR 可以看做一个神经单元，TF 不支持单个神经单元参数切分到多个参数服务器 node 上。

当然，以 Google 的实力，绝对是可以做到第四重境界的，之所以没有曝光，可能是基于其他商业目的考量，比如使用他们的云计算服务。

综上，个人认为如果能做到第四重境界，目前可以说的上是世界一流的大规模机器学习框架。仅从沐帅的 ppt 里看他曾经达到过，Google 内部应该也是没有问题的。第三重境界应该是国内一流，第二充应该是国内前列吧。

### 1.5 其他

#### 1.5.1 资源管理

本文没有涉及到的部分是资源管理，大规模机器学习框架部署的集群往往资源消耗也比较大，需要专门的资源管理工具来维护。这方面 yarn 和 mesos 都是佼佼者，细节这里也就不介绍了

#### 1.5.2 设备

除了资源管理工具，本身部署大规模机器学习集群本身对硬件也还是有些要求的，虽然理论上来说，所有 commodity 机器都可以用来搭建这类集群，但是考虑到性能，我们建议尽量用高内存的机器 + 万兆及以上的网卡。没有超快速的网卡，玩参数传递和样本加载估计会比较苦逼。

<sup>23</sup><https://www.jianshu.com/p/00736aa21dc8>

<sup>24</sup>TensorFlow:Large-Scale Machine Learning on Heterogeneous Distributed Systems

## 1.6 结语

从后台转算法以来，长期沉浸于算法推理的论文无法自拔，对自己之前的后台工程能力渐渐轻视起来，觉得工程对算法的帮助不大。直到最近一个契机，需要做一个这方面的调研，才豁然发现，之前的工程经验对我理解大规模机器学习框架非常有用，果然如李宗盛所说，人生每一步路，都不是白走的。

在一个月左右的调研中，脑子每天都充斥这各种疑问和困惑，曾经半夜 4 点醒来，思考同步机制而再也睡不着，干脆起来躲卫生间看书，而那天我一点多才睡。当脑子里有放不下的问题的时候，整个人会处于一种非常亢奋的状态，除非彻底想清楚这个问题，否则失眠是必然的，上一次这种状态已经是很多年前了。好在最后我总算理清了这方面的所有关键细节。以此，记之。Carbonzhang 于 2017 年 8 月 26 日凌晨！

## 致谢

感谢 wills、janwang、joey、roberty、suzi 等同学一起讨论，特别感谢 burness 在 TF 方面的深厚造诣和调研。因为本人水平所限，错漏难免，另外还有相当多的细节因为篇幅限制并未一一展开，仅仅是较高抽象层面上简述了下大规模机器学习框架的关键思路，其他如分片向量锁、通信协议、时钟逻辑、DAG 调度器、资源调度模块等均为展开来讲，希望以后有机会能补上。

# 第二章 分布式系统论文阅读

整理阅读的分布式系统以及大数据处理方面的论文。

## 2.1 论文阅读要求

Cornell University CS6453<sup>1</sup>给的阅读 paper 和准备 presentation 的建议

**Paper Reviews:** Please write constructive reviews. Here is a rough outline.

- Summary of problem being solved (1-2 lines)
- Why is the problem interesting? Perhaps, its a new problem? Perhaps, its an old problem but with a new twist (e.g., new workloads, new environment, new hardware)? Perhaps, its just an old classical problem? (1-2 lines)
- What are the main insights in the proposed solution? What is the main technical contribution? How does the solution advance the state-of-the-art? (4-5 lines)
- Can you think of cases where the proposed solution may not work well? (4-5 lines)
- What are the next few problems that you would solve in this space? What do you think is the holy grail in this direction? (5-10 lines)

**Paper Presentations:** Please plan for 30 minutes. Here is a rough outline.

- What is the problem being solved? (2-3 slides)
- Why is the problem interesting? Perhaps, its a new problem? Perhaps, its an old problem but with a new twist (e.g., new workloads, new environment, new hardware)? Perhaps, its just an old classical problem? (1-2 slides)
- What is the most related work and how is this paper different? (3-4 slides)
- What is the main technical contribution? What are the main insights used to build the proposed solution? How does the solution advance the state-of-the-art? (2-3 slides)
- Techniques used in the paper to solve the problem (4-5 slides)
- Can you think of cases where the proposed solution may not work well? (2-3 slides)
- What are the next few problems that you would solve in this space? What do you think is the holy grail in this direction? (4-5 slides)

所选择阅读并记录的论文参考自 Cornell CS6453 Spring2017<sup>2</sup>和 MIT 6.824 Spring2018<sup>3</sup>

---

<sup>1</sup><http://www.cs.cornell.edu/courses/cs6453/2017sp/index.html>

<sup>2</sup><http://www.cs.cornell.edu/courses/cs6453/2017sp/schedule.html>

<sup>3</sup><https://pdos.csail.mit.edu/6.824/schedule.html>

## 2.2 分布式存储

### 2.2.1 GFS

#### - The Google File System

- 【20180707】阅读了 Abstract & Introduction 部分
- 

2003 年 Google 发表这篇论文，文中详细介绍了 Google 使用的分布式文件系统，其支持大规模分布式的数据密集型应用。那时候的计算机硬件特别昂贵且性能很差（相较于现在，性价比应该是很低的），因此需要特别关注错误容忍（容错性，Fault Tolerance），集群中的任何一个节点可能在任何一个时间点宕机。

GFS 吸取了在其之前的众多分布式文件系统的特性，包括性能、可扩展性、可靠性、可用性 (performance, scalability, reliability, and availability)，同时也根据 Google 具体的应用负载 (application workloads) 和应用环境 (technological environment) 进行了适配，重新审视了传统分布式文件系统设计上的选择 (choices)，并提出了不同的系统设计考量。

1. 集群中的组件故障 (failures) 是正常情况。在部署的分布式集群环境中，使用了成百上千台廉价的机器作为存储节点，这些组件宕掉是很正常的情况，因此所构建的系统需要满足（每一点都需要在之后补充上）：
  - constant monitoring
  - error detection
  - fault tolerance
  - automatic recovery
2. 存储的文件非常大，比如 TB、PB 级别的数据
3. 文件极少更新，更确切地说，文件作为某些计算的数据读入，并且不时有额外的数据追加到文件尾部
4. 将文件系统的 API 与应用程序协同设计，以增加整个系统的灵活性

#### Note 2.1 GFS 中为实现容错做了哪些手段？

待补充

## 2.3 分布式计算

### 2.3.1 MapReduce

- MapReduce: Simplified Data Processing on Large Clusters
- 
- 

### 2.3.2 Spark

- Spark: Cluster Computing with Working Sets
- Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing

- 
- 

## 2.4 参数服务器

### 2.4.1 ASP

- An Architecture for Parallel Topic Models
- 
- 

2010 年的一篇文章，提出了参数服务器的概念，比较有开创性。

### 2.4.2 SSP

- More Effective Distributed ML via a Stale Synchronous Parallel Parameter Server
- 【20180703】全文阅读，但跳过收敛性证明
- 【20180707】重读 & 整理，同样跳过收敛性证明

这是分布式机器学习一个里程碑式的文章，提出了一个很 generic 的算法，提供了在某些合理假设下的收敛性证明（优化），并且在这个基础上提出了一个合理的编程模型，最后给了一个系统实现。<sup>4</sup>

在 Spark/Hadoop 中每个 worker 的新一次迭代都需要等待所有 workers 上一次迭代的结束，要用到上一次迭代的结果 (BSP)；而对于 ASP，每一个 worker 执行调度器分配给自己的那部分任务，不用等待其他 workers 的执行。一个很直接的想法就是能否寻找一个折衷的同步策略，即不是像 BSP 那样每次迭代都强制所有 workers 对上一次迭代进行同步，也不是像 ASP 那样各 workers 随意执行，而是每个 worker 都按照自己的节奏执行程序，但要求执行最快的那个 worker 与执行最慢的那个 worker 之间不能超过所指定的迭代数，例如 3 个迭代，如图 2.1 所示。

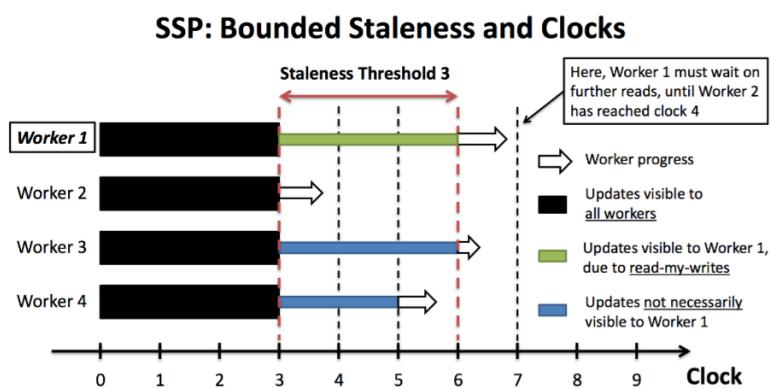


图 2.1: SSP 模型示例

<sup>4</sup><https://zhuanlan.zhihu.com/p/29032307>

### Note 2.2 摘录 SSP 论文中的描述

We begin with an informal explanation of SSP: assume a collection of  $P$  workers, each of which makes additive updates to a shared parameter  $x \leftarrow x + \mu$  at regular intervals called *clocks*. Clocks are similar to iterations, and represent some unit of progress by an ML algorithm. Every worker has its own integer-valued clock  $c$ , and workers only commit their updates at the end of each clock. Updates may not be immediately visible to other workers trying to read  $x$  — in other words, workers only see effects from a “stale” subset of updates. The idea is that, with staleness, workers can retrieve updates from caches on the same machine (*fast*) instead of querying the parameter server over the network (*slow*). Given a user-chosen staleness threshold  $s \geq 0$ , SSP enforces the following bounded staleness conditions:

- The slowest and fastest workers must be  $\leq s$  clocks apart — otherwise, the fastest worker is forced to wait for the slowest worker to catch up.
- When a worker with clock  $c$  commits an update  $\mu$ , that  $\mu$  is timestamped with time  $c$ .
- When a worker with clock  $c$  reads  $x$ , it will always see effects from all  $\mu$  with timestamp  $\leq c - s - 1$ . It may also see some  $\mu$  with timestamp  $> c - s - 1$  from other workers.
- *Read-my-writes*: A worker  $p$  will always see the effects of its own updates up.

### 2.4.3 Parameter Server

- Scaling Distributed Machine Learning with the Parameter Server
- Communication Efficient Distributed Machine Learning with the Parameter Server

- 【20180625】阅读过好几遍了，打印纸质版、MarginNote 上，在笔记上手写了笔记
- 【20180707】重读 & 整理
- 【20180708】整理，差论文 2.2 3.4 的理解与整理，可能需要看过 ps-lite 代码之后才能理解的更清晰

沐帅在 OSDI 2014 上发表这篇工作，引出第三代参数服务器框架，在 worker 节点上分布式存放数据和工作负载，server 节点上保存着全局共享的参数，各节点之间异步通信，支持灵活的一致性模型，可扩展并且容错性很好 (continuous fault tolerance 是什么意思)。

当处理大规模的机器学习（深度学习）模型时，会面临这样一个问题：多台机器节点之间需要共享一些状态信息，包括参数、集群信息，用户 profiles 以及其他一些需要在各节点之间进行交流的信息。另外，单机已经解决不了目前快速增长的数据和参数了，在实际的系统中训练的数据量已经达到 TB、PB 级别，训练过程中可能会产生  $10^9 \sim 10^{12}$  的参数量（这两个数据是在 2014 年年初时候的数据，现在需要更新了）。模型参数被所有的 worker 节点所共享并且会被频繁访问到，这就会带来很多问题和挑战：

1. 访问这些巨量的参数，需要大量的网络带宽支持
2. 很多机器学习算法是连续型的（顺序的），只有上一次迭代完成（各个 worker 上都完成）之后，才能进行下一次迭代，这导致如果各机器之间的性能差距太大（木桶效应），就会造成性能的极大损失（因为系统整体的性能由性能最差的那台机器所决定）
3. 在分布式系统中容错能力是非常重要的，在真实环境的部署下，算法最终是在云上（云计算）运行的（这种环境下，机器是不可靠的，并且所提交的 job 可能会被其他任务抢占）

沐帅将参数服务器的发展分为三个阶段：

第一代 缺少灵活性和性能 — 仅使用 memcached 分布式 (key, value) 存储，来作为同步机制。YahooLDA 通过改进这一机制，增加了一个专门的服务器，提供用户能够自定义的更新操作 (set/get/update)

第二代 Petuum 使用 bounded delay 模型来改进 YahooLDA，但是却限制了 worker 的线程模型

第三代 即在这篇 paper 中提出来，能够解决这些局限性

参数服务器与通用的分布式系统有什么区别呢？通用的分布式系统通常是每次迭代都强制同步，在十几个节点上它们的性能可以表现的很好，但是在大规模集群中，这样的每次迭代强制同步的机制（BSP）会因为木桶效应变得很慢。Mahout 基于 Hadoop，MLI 基于 Spark，它们（MLI 和 Spark）采用的都是 Iterative MapReduce 架构，能够保持迭代之间的状态，并且执行策略也更加优化了，但是由于这两种方法均采用了同步迭代的通信方式，使得它们很容易因为个别机器的低性能导致全局性能的降低。为了解决这个问题，GraphLab 采用图形抽象的方式进行异步通信，但是它缺少以 MapReduce 为基础架构的弹性扩展性，并且它使用粗粒度的 snapshots 来进行恢复，这两点都会阻碍到可扩展性。

参数服务器正是吸取了两者的优点，1) MLI 和 Spark 保持各个迭代之间的状态信息，2) GraphLab 异步通信，并且解决了 GraphLab 在可扩展性方面的劣势。

**Distributed Subgradient Descent** 论文的 2.2 节

### 架构

一个参数服务器的实例可以同时运行多个算法。参数服务器的节点以 group 进行分组，包含一个 server group 和多个 worker group，如图 2.2 所示。

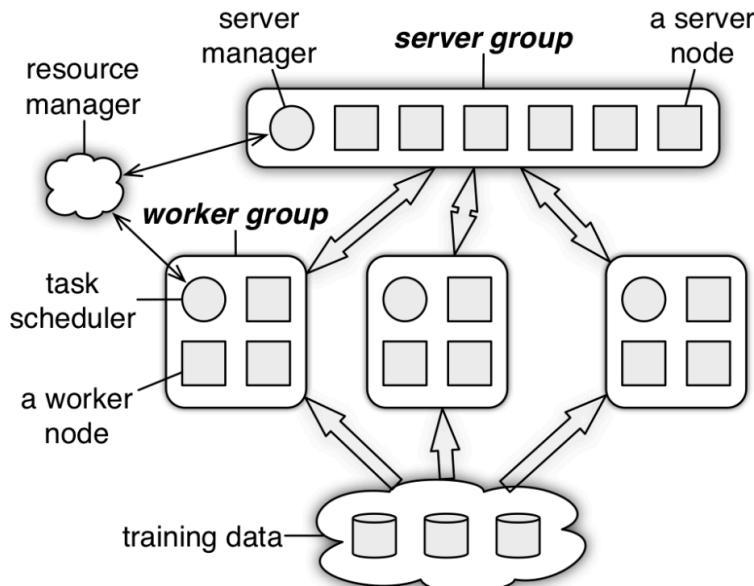


图 2.2: 参数服务器的架构

在 server group 中的每一个 server 节点保存着部分全局共享的参数，每个 server 节点之间会进行通信，以进行备份或者迁移节点上的参数，这是为了整个系统的容错性和可扩展性考虑。在 server group 中有一个节点的角色是 server manager，保存各 server 节点的元数据信息（包括节点是否存活，分配在其上的共享参数的范围），需要考虑如何保持一致性。

每一个 worker group 上都运行着一个应用。每一个 worker 节点上都本地保存着部分的训练数据，用来计算一个本地的梯度值。worker 节点将计算得到的梯度值 push 给 server 节点，或者从 server 节点上 pull 下已更新的参数；各 worker 节点之间不进行通信。每一个 worker group 中同样有一个 task scheduler 节点，负责分配任务给 group 中的所有 worker 节点，并监测各 worker 的执行进度，在新的 worker 节点添加进该 group 或有 worker 节点被移除时，scheduler 节点负责重新调度未完成的任务。

参数服务器支持独立的参数命名空间（namespace）。

沐帅提出的第三代参数服务器包含以下几个要点：

- 所有共享参数的形式都是 (key, value) vectors — 假设 keys 是有序的，不存在的 key 对应的 value 值为 0
- 基于 range 的 push/pull 操作，这样 worker 节点每次与 server 节点进行通信时可以选择性的仅传递少量需要更新的参数，更可控也更灵活
- server 节点用于聚集各 worker 节点发送过来的各子梯度值，同时，server 节点也支持执行用户指定的操作。支持这样的操作是很有好处的，因为 server 节点通常保存有更完整并且更 up-to-date 的共享参数信息
- 异步操作，回调的实现
- 各 worker 节点异步的执行任务，各节点之间可能会出现数据不一致，从而使得算法的收敛过程变慢。然而，有一些算法对数据的一致性要求没有那么严格，即有一些算法在数据出现不一致时，也能够在最终收敛并得到准确的结果。

对一致性协议的考量，需要在系统性能跟算法收敛速率之间做一个 tradeoff，同时考虑：

1. 算法对于参数非一致性的敏感程度
2. 训练数据的特征之间的关联度
3. 硬件设备的能力

考虑到用户使用的时候会有不同的情况，参数服务器为用户提供了多种任务依赖方式，1) Sequential, 2) Eventual, 3) Bounded delay

- 用户可自定义的过滤器。对于机器学习优化问题来说，比如梯度下降，并不是每次计算的梯度对于最终优化都是有价值的，用户可以通过自定义的规则过滤一些不必要的通信，再进一步压缩带宽耗费：

1. 发送很小的梯度值是低效的 — 因此可以自定义设置，只在梯度值很大的时候发送
2. 更新接近最优情况的值是低效的 — 因此只在非最优情况下发送，可通过 KKT 条件来判断

#### 2.4.4 GeePS

- GeePS: Scalable deep learning on distributed GPUs with a GPU-specialized parameter server
- 
-

# 第三章 深度学习框架

本章内容主要是对博客 **The Anatomy of Deep Learning Frameworks<sup>1</sup>** 的总结与整理。

可以考虑看看这个课程 CSE 599W: Systems for ML<sup>2</sup>

许多初学者觉得深度学习框架抽象，虽然调用了几个函数/方法，计算了几个数学难题，但始终不能理解这些框架的全貌。

为了更好地认识深度学习框架，也为了给一些想要自己亲手搭建深度学习框架的朋友提供一些基础性的指导，日前来自苏黎世联邦理工学院计算机科学系的硕士研究生 Gokula Krishnan Santhanam 在博客上撰文，概括了大部分深度学习框架都会包含的五大核心组件，为我们详细剖析了深度学习框架一般性的内部组织结构。本文为雷锋网编译<sup>3</sup>。

Gokula Krishnan Santhanam 认为，大部分深度学习框架都包含以下五个核心组件：

1. 张量 (Tensor)
2. 基于张量的各种操作
3. 计算图 (Computation Graph)
4. 自动微分工具 (Automatic Differentiation)
5. BLAS、cuBLAS、cuDNN 等扩展包

## 3.1 张量

张量是所有深度学习框架中最核心的组件，因为后续的所有运算和优化算法都是基于张量进行的。几何代数中定义的张量是基于向量和矩阵的推广，通俗一点理解的话，我们可以将标量视为零阶张量，矢量视为一阶张量，那么矩阵就是二阶张量。

举例来说，我们可以将任意一张 RGB 彩色图片表示成一个三阶张量（三个维度分别是图片的高度、宽度和色彩数据）。如图 5.2.1 所示，是一张普通的水果图片，按照 RGB 三原色表示，其可以拆分为三张红色、绿色和蓝色的灰度图片，如果将这种表示方法用张量的形式写出来，就是图中最下方的那张表格。

图中只显示了前 5 行、320 列的数据，每个方格代表一个像素点，其中的数据 [1.0, 1.0, 1.0] 即为颜色。假设用 [1.0, 0, 0] 表示红色，[0, 1.0, 0] 表示绿色，[0, 0, 1.0] 表示蓝色，那么前面 5 行的数据则全是白色。

将这一定义进行扩展，我们也可以用四阶张量表示一个包含多张图片的数据集，其中的四个维度分别是：图片在数据集中的编号，图片高度、宽度，以及色彩数据。

将各种各样的数据抽象成张量表示，然后再输入神经网络模型进行后续处理是一种非常必要且高效的策略。因为如果没有这一步骤，我们就需要根据各种不同类型的数据组织形式定义各种不同类型的数据操作，这会浪费大量的开发者精力。更关键的是，当数据处理完成后，我们还可以方便地将张量再转换回想要的格式。例如 Python Numpy 包中 numpy.imread 和 numpy.imsave 两个方法，分别用来将图片转换成张量对象（即代码中的 Tensor 对象），和将张量再转换成图片保存起来。

<sup>1</sup>[https://medium.com/@gokul\\_uf/the-anatomy-of-deep-learning-frameworks-46e2a7af5e47](https://medium.com/@gokul_uf/the-anatomy-of-deep-learning-frameworks-46e2a7af5e47)

<sup>2</sup><http://dlsys.cs.washington.edu/>

<sup>3</sup><https://www.leiphone.com/news/201701/DZeAwe2qgx8JhbU8.html>

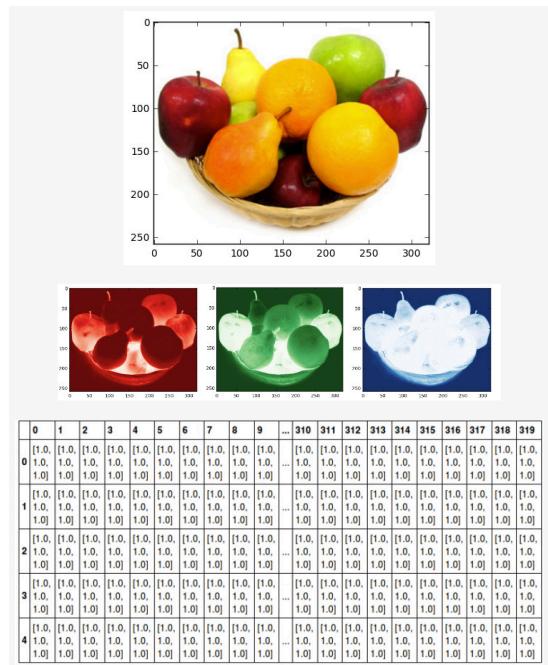


图 3.1: 水果的三种表示

### 3.2 基于张量的各种操作

有了张量对象之后，下面一步就是一系列针对这一对象的数学运算和处理过程。如图 5.2.1 所示。

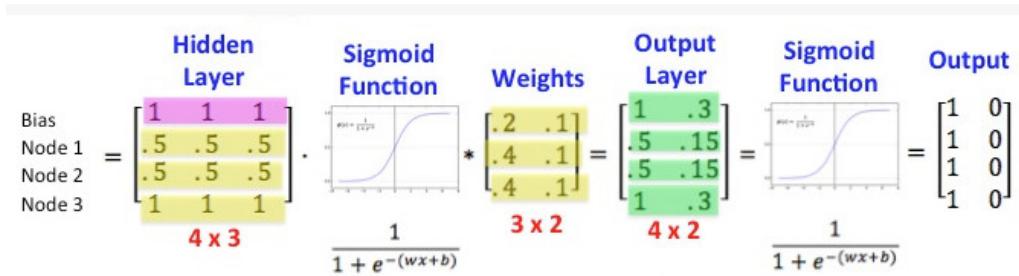


图 3.2: 基于张量的各种操作

其实，整个神经网络都可以简单视做为了达到某种目的，针对输入张量进行的一系列操作的过程。而所谓的“学习”就是不断纠正神经网络的实际输出结果和预期结果之间误差的过程。这里的一系列操作包含的范围很宽，可以是简单的矩阵乘法，也可以是卷积、池化和 LSTM 等稍复杂的运算。而且各框架支持的张量操作通常也不尽相同，详细情况可以查看其官方文档。

需要指出的是，大部分的张量操作都是基于类实现的（而且是抽象类），而并不是函数（这一点可能要归功于大部分的深度学习框架都是用面向对象的编程语言实现的）。这种实现思路一方面允许开发者将各种类似的操作汇总在一起，方便组织管理。另一方面也保证了整个代码的复用性、扩展性和对外接口的统一。总体上让整个框架更灵活和易于扩展，为将来的发展预留了空间。

### 3.3 计算图

有了张量和基于张量的各种操作之后，下一步就是将各种操作整合起来，输出我们需要的结果。

但不幸的是，随着操作种类和数量的增多，有可能引发各种意想不到的问题，包括多个操作之间应该并行还是顺次执行，如何协同各种不同的底层设备，以及如何避免各种类型的冗余操作等等。这些问题有可能拉低整个深度学习网络的运行效率或者引入不必要的 Bug，而计算图正是为解决这一问题产生的。

计算图首次被引入人工智能领域是在 2009 年的论文《Learning Deep Architectures for AI》。当时的示意图如下图 5.2.1 所示，作者用不同的占位符 (\*, +, sin) 构成操作结点，以字母 x、a、b 构成变量结点，再以有向线段将这些结点连接起来，组成一个表征运算逻辑关系的清晰明了的“图”型数据结构，这就是最初的计算图。

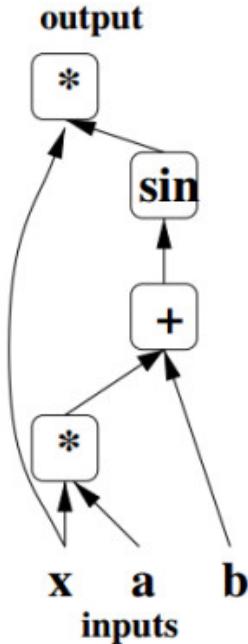


图 3.3: 最初的计算图

将计算图作为前后端之间的中间表示 (Intermediate Representations) 可以带来良好的交互性，开发者可以将 Tensor 对象作为数据结构，函数/方法作为操作类型，将特定的操作类型应用于特定的数据结构，从而定义出类似 MATLAB 的强大建模语言。

需要注意的是，通常情况下开发者不会将用于中间表示得到的计算图直接用于模型构造，因为这样的计算图通常包含了大量的冗余求解目标，也没有提取共享变量，因而通常都会经过依赖性剪枝、符号融合、内存共享等方法对计算图进行优化。

目前，各个框架对于计算图的实现机制和侧重点各不相同。例如 Theano 和 MXNet 都是以隐式处理的方式在编译中由表达式向计算图过渡。而 Caffe 则比较直接，可以创建一个 Graph 对象，然后以类似 Graph.Operator(xxx) 的方式显示调用。

因为计算图的引入，开发者得以从宏观上俯瞰整个神经网络的内部结构，就好像编译器可以从整个代码的角度决定如何分配寄存器那样，计算图也可以从宏观上决定代码运行时的 GPU 内存分配，以及分布式环境中不同底层设备间的相互协作方式。除此之外，现在也有许多深度学习框架将计算图应用于模型调试，可以实时输出当前某一操作类型的文本描述。

### 3.4 自动微分工具

计算图带来的另一个好处是让模型训练阶段的梯度计算变得模块化且更为便捷，也就是自动微分法。正如前面提到的，因为我们可以将神经网络视为由许多非线性过程组成的一个复杂的函数体，而计算图则以模块化的方式完整表征了这一函数体的内部逻辑关系，因此微分这一复杂函数体，即求取模型梯度的方法就变成了在计算图中简单地从输入到输出进行一次完整遍历的过程。与自动微分对应，业内更传统的做法是符号微分。

符号微分即常见的求导分析。针对一些非线性过程（如修正线性单元 ReLU）或者大规模的问题，使用符号微分法的成本往往非常高昂，有时甚至不可行（即不可微）。因此，以上述迭代式的自动微分法求解模型梯度已经被广泛采用。并且由于自动微分可以成功应对一些符号微分不适用的场景，目前许多计算图程序包（例如 Computation Graph Toolkit）都已经预先实现了自动微分。

另外，由于每个节点处的导数只能相对于其相邻节点计算，因此实现了自动微分的模块一般都可以直接加入任意的操作类中，当然也可以被上层的微分大模块直接调用。

### 3.5 BLAS/cuBLAS/cuDNN 等扩展包

现在，通过上述所有模块，我们已经可以搭建一个全功能的深度学习框架：将待处理数据转换为张量，针对张量施加各种需要的操作，通过自动微分对模型展开训练，然后得到输出结果开始测试。这时还缺什么呢？答案是运算效率。

由于此前的大部分实现都是基于高级语言的（如 Java、Python、Lua 等），而即使是执行最简单的操作，高级语言也会比低级语言消耗更多的 CPU 周期，更何况是结构复杂的深度神经网络，因此运算缓慢就成了高级语言的一个天然的缺陷。

目前针对这一问题有两种解决方案。

第一种方法是模拟传统的编译器。就好像传统编译器会把高级语言编译成特定平台的汇编语言实现高效运行一样，这种方法将高级语言转换为 C 语言，然后在 C 语言基础上编译、执行。为了实现这种转换，每一种张量操作的实现代码都会预先加入 C 语言的转换部分，然后由编译器在编译阶段将这些由 C 语言实现的张量操作综合在一起。目前 pyCUDA 和 Cython 等编译器都已经实现了这一功能。

第二种方法就是前文提到的，利用脚本语言实现前端建模，用低级语言如 C++ 实现后端运行，这意味着高级语言和低级语言之间的交互都发生在框架内部，因此每次的后端变动都不需要修改前端，也不需要完整编译（只需要通过修改编译参数进行部分编译），因此整体速度也就更快。

除此之外，由于低级语言的最优化编程难度很高，而且大部分的基础操作其实也都有公开的最优解决方案，因此另一个显著的加速手段就是利用现成的扩展包。例如最初用 Fortran 实现的 BLAS（基础线性代数子程序），就是一个非常优秀的基本矩阵（张量）运算库，此外还有英特尔的 MKL（Math Kernel Library）等，开发者可以根据个人喜好灵活选择。

值得一提的是，一般的 BLAS 库只是针对普通的 CPU 场景进行了优化，但目前大部分的深度学习模型都已经开始采用并行 GPU 的运算模式，因此利用诸如 NVIDIA 推出的针对 GPU 优化的 cuBLAS 和 cuDNN 等更据针对性的库可能是更好的选择。

运算速度对于深度学习框架来说至关重要，例如同样训练一个神经网络，不加速需要 4 天的时间，加速的话可能只要 4 小时。在快速发展的人工智能领域，特别是对那些成立不久的人工智能初创公司而言，这种差别可能就会决定谁是先驱者，而谁是追随者。

### 3.6 总结

原文作者在文末指出：为了向开发者提供尽量简单的接口，大部分深度学习框架通常都会将普通的概念抽象化，这可能是造成许多用户感知不到上述五点核心组件的重要原因。

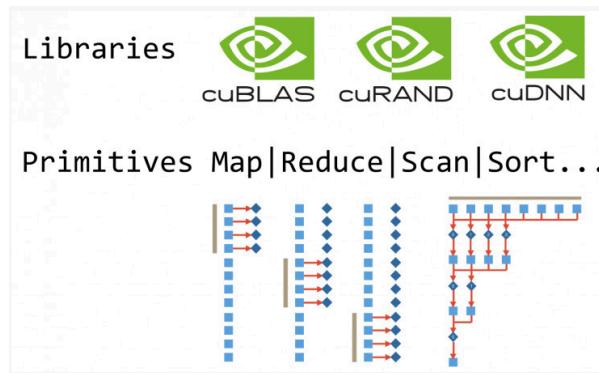


图 3.4: BLAS/cuBLAS/cuDNN 等扩展包

而这也正是作者写本文的初衷：他希望开发者能够通过了解不同框架之间的一些相似特性，更好地认识和使用一个深度学习框架。另一方面，对于那些不仅对学会使用深度学习框架感兴趣，还打算亲手搭建一个深度框架的朋友，作者认为了解各框架的内部组成和一些共性的特征也是迈向成功的重要一步。他真诚地相信，一个优秀的工程师不仅应该“知其然”，更应该“知其所以然”。



# 第四章 分布式系统的基本要素

## 4.1 Vector Clock

4.1.1

4.1.2

## 4.2 一致性哈希

本节整理自博客『一致性哈希算法及其在分布式系统中的应用<sup>1</sup>』

本节将会从实际应用场景出发，介绍一致性哈希算法（Consistent Hashing）及其在分布式系统中的应用。首先本节会描述一个在日常开发中经常会遇到的问题场景，借此介绍一致性哈希算法以及这个算法如何解决此问题；接下来会对这个算法进行相对详细的描述，并讨论一些如虚拟节点等与此算法应用相关的话题。

### 4.2.1 分布式缓存问题

假设我们有一个网站，最近发现随着流量增加，服务器压力越来越大，之前直接读写数据库的方式不太给力了，于是我们想引入 Memcached 作为缓存机制。现在我们一共有三台机器可以作为 Memcached 服务器，如图 4.1 所示。

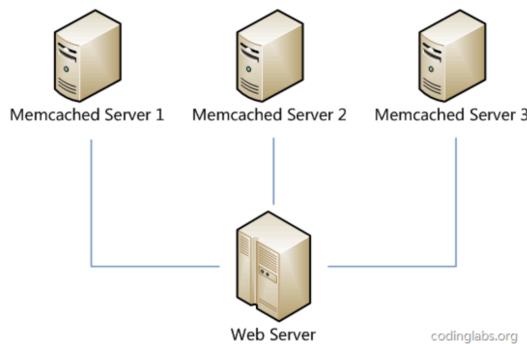


图 4.1: 三台 Memcached 服务器

很显然，最简单的策略是将每一次 Memcached 请求随机发送到一台 Memcached 服务器，但是这种策略可能会带来两个问题：

1. 同一份数据可能被存在不同的机器上而造成数据冗余

<sup>1</sup><http://blog.codinglabs.org/articles/consistent-hashing.html>

2. 有可能某数据已经被缓存但是访问却没有命中，因为无法保证对相同 key 的所有访问都被发送到相同的服务器

因此，随机策略无论是时间效率还是空间效率都非常不好。

要解决上述问题只需做到如下一点：保证对相同 key 的访问会被发送到相同的服务器。很多方法可以实现这一点，最常用的方法是计算哈希。例如对于每次访问，可以按如下算法计算其哈希值：

$$h = \text{Hash}(key) \% 3$$

其中，Hash 是一个从字符串到正整数的哈希映射函数。这样，如果我们将 Memcached Server 分别编号为 0、1、2，那么就可以根据上式和 key 计算出服务器编号 h，然后去访问。

这个方法虽然解决了上面提到的两个问题，但是存在一些其它的问题。如果将上述方法抽象，可以认为通过：

$$h = \text{Hash}(key) \% N$$

这个算式计算每个 key 的请求应该被发送到哪台服务器，其中，N 为服务器的台数，并且服务器按照 0 ~ (N-1) 编号。这个算法的问题在于容错性和扩展性不好。所谓容错性是指当系统中某一个或几个服务器变得不可用时，整个系统是否可以正确高效运行；而扩展性是指当加入新的服务器后，整个系统是否可以正确高效运行。

现假设有一台服务器宕机了，那么为了填补空缺，要将宕机的服务器从编号列表中移除，后面的服务器按顺序前移一位并将其编号值减一，此时每个 key 就要按  $h = \text{Hash}(key) \% (N - 1)$  重新计算；同样，如果新增了一台服务器，虽然原有服务器编号不用改变，但是要按  $h = \text{Hash}(key) \% (N + 1)$  重新计算哈希值。因此系统中一旦有服务器变更，大量的 key 会被重定位到不同的服务器从而造成大量的缓存不命中。而这种情况在分布式系统中是非常糟糕的。

一个设计良好的分布式哈希方案应该具有良好的单调性，即服务节点的增减不会造成大量哈希重定位。一致性哈希算法就是这样一种哈希方案。

### 4.2.2 一致性哈希算法

#### 算法简述

一致性哈希算法（Consistent Hashing）最早在这篇论文<sup>2</sup>中提出。简单来说，一致性哈希将整个哈希值空间组织成一个虚拟的圆环，如假设某哈希函数 H 的值空间为  $0 \sim 2^{32} - 1$ （即哈希值是一个 32 位无符号整形），整个哈希空间环如图 4.2 所示。整个空间按顺时针方向组织。0 和  $2^{32} - 1$  在零点中方向重

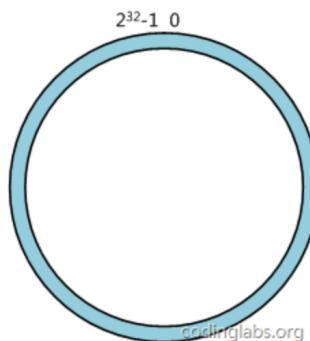


图 4.2: 哈希空间环

<sup>2</sup>Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web

合。

下一步将各个服务器使用 H 进行一个哈希，具体可以选择服务器的 ip 或主机名作为关键字进行哈希，这样每台机器就能确定其在哈希环上的位置，这里假设将上文中三台服务器使用 ip 地址哈希后在环空间的位置如图 4.3 所示

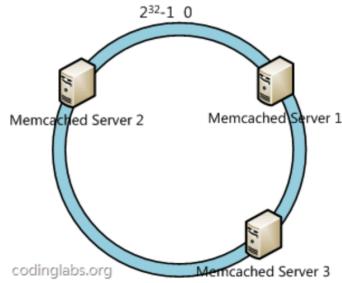


图 4.3: 三台服务器映射到哈希环上的位置

接下来使用如下算法定位数据访问到相应服务器：将数据 key 使用相同的函数 H 计算出哈希值 h，根据 h 确定此数据在环上的位置，从此位置沿环顺时针『行走』，第一台遇到的服务器就是其应该定位到的服务器。

例如我们有 A、B、C、D 四个数据对象，经过哈希计算后，在环空间上的位置如图 4.4

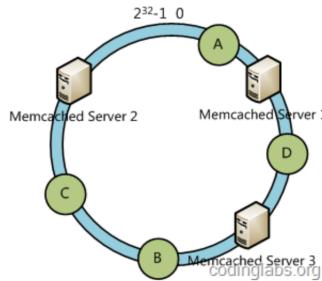


图 4.4: 四个数据对象映射到哈希环上的位置

根据一致性哈希算法，数据 A 会被定位到 Server 1 上，D 被定为到 Server 3 上，而 B、C 分别被定为到 Server 2 上。

#### 容错性与可扩展性分析

下面分析一致性哈希算法的容错性和可扩展性。现假设 Server 3 宕机了：

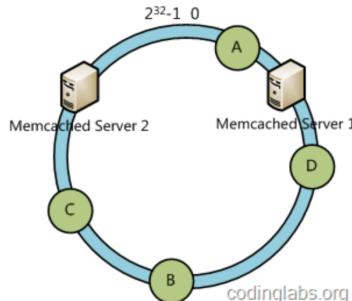


图 4.5: Server3 宕机

可以看到此时 A、C、B 不会受到影响，只有 D 节点被重定位到 Server 2。一般的，在一致性哈希算法中，如果一台服务器不可用，则受影响的数据仅仅是此服务器到其环空间中前一台服务器（即顺着逆时针方向行走遇到的第一台服务器）之间数据，其它不会受到影响。

下面考虑另外一种情况，如果我们在系统中增加一台服务器 Memcached Server 4：

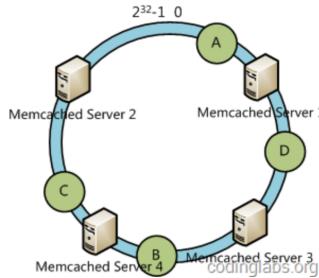


图 4.6: 增加 Server4

此时 A、D、C 不受影响，只有 B 需要重定位到新的 Server 4。一般的，在一致性哈希算法中，如果增加一台服务器，则受影响的数据仅仅是新服务器到其环空间中前一台服务器（即顺着逆时针方向行走遇到的第一台服务器）之间数据，其它不会受到影响。

综上所述，一致性哈希算法对于节点的增减都只需重定位环空间中的一小部分数据，具有较好的容错性和可扩展性。

### 虚拟节点

一致性哈希算法在服务节点太少时，容易因为节点分布不均匀而造成数据倾斜问题。例如我们的系统中有两台服务器，其环分布如图4.7：

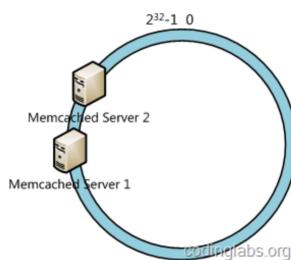


图 4.7: 节点太少造成的数据倾斜

此时必然造成大量数据集中到 Server 1 上，而只有极少量会定位到 Server 2 上。为了解决这种数据倾斜问题，一致性哈希算法引入了虚拟节点机制，即对每一个服务节点计算多个哈希，每个计算结果位置都放置一个此服务节点，称为虚拟节点。具体做法可以在服务器 ip 或主机名的后面增加编号来实现。例如上面的情况，我们决定为每台服务器计算三个虚拟节点，于是可以分别计算“Memcached Server 1#1”、“Memcached Server 1#2”、“Memcached Server 1#3”、“Memcached Server 2#1”、“Memcached Server 2#2”、“Memcached Server 2#3”的哈希值，于是形成六个虚拟节点：

同时数据定位算法不变，只是多了一步虚拟节点到实际节点的映射，例如定位到“Memcached Server 1#1”、“Memcached Server 1#2”、“Memcached Server 1#3”三个虚拟节点的数据均定位到 Server 1 上。这样就解决了服务节点少时数据倾斜的问题。在实际应用中，通常将虚拟节点数设置为 32 甚至更大，因此即使很少的服务节点也能做到相对均匀的数据分布。

### 总结

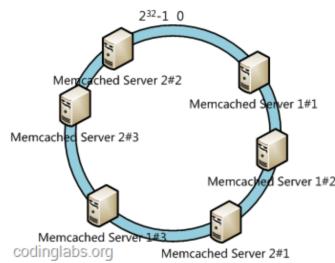


图 4.8: 使用虚拟节点来实现负载均衡

目前一致性哈希基本成为了分布式系统组件的标准配置，例如 Memcached 的各种客户端都提供内置的一致性哈希支持。本节只是简要介绍了这个算法，更深入的内容可以参看原论文。



# 第五章 机器学习 & 深度学习基础

## 5.1 梯度下降

5.1.1 损失函数

5.1.2 过拟合

5.1.3 正则化

## 5.2 Inference & Training

5.2.1



# 第六章 RDMA 相关知识点整理

## 6.1

6.1.1

6.1.2



# 第七章 算法与数据结构

## 7.1 基本数据结构

- 算法第四版 C++ 实现
- 浅谈算法和数据结构<sup>1</sup>

## 7.2 分而治之

### 7.3 贪心

#### 7.3.1

#### 7.3.2

## 7.4 动态规划

以下内容整理自：

- 知乎上问题『什么是动态规划？动态规划的意义是什么？』
- 算法导论 Chapter 15
- 算法 Chapter 6

#### 7.4.1 动态规划的本质

动态规划的本质，是对问题状态的定义和状态转移方程的定义。引用维基百科：

**Dynamic programming** is a method for solving a complex problem by **breaking it down into a collection of simpler subproblems**.

动态规划就是通过**拆分问题**，定义问题状态和状态之间的关系，使得问题能够以递推（或者说分治）的方式去解决。**如何拆分问题**，才是动态规划的核心。而**拆分问题**，靠的就是**状态的定义**和**状态转移方程的定义**。

#### 7.4.2 状态的定义

首先来看一个动态规划的教学必备题：

给定一个数列，长度为  $N$ ，求这个数列的最长上升（递增）子序列（LIS）的长度。以 1 7 2 8 3 4 为例，这个数列的最长递增子序列是 1 2 3 4，长度为 4；次长的为 3，包括 1 7 8、1 2 3、1 3 4。

---

<sup>1</sup><https://www.cnblogs.com/yangecnu/category/548373.html>

要解决这个问题，首先要**定义这个问题和这个问题的子问题**。有人可能会问了，题目都已经在这了，还需定义这个问题吗？需要，原因就是这个问题在字面上看，找不出子问题，而没有子问题，这个题目就没办法解决。

所以，重新定义这个问题：

给定一个数列，长度为  $N$ ，设  $F_k$  为：以数列中第  $k$  项结尾的最长递增子序列的长度。求  $F_1, \dots, F_N$  中的最大值。

显然，这个新问题与原问题等价。而对于  $F_k$  来讲， $F_1, \dots, F_{k-1}$  都是  $F_k$  的子问题：因为以第  $k$  项结尾的最长递增子序列（以下称 LIS），包含着以第  $1, \dots, k-1$  中某项结尾的 LIS。

上述的新问题  $F_k$  也可以叫做状态，定义中的『 $F_k$  为数列中第  $k$  项结尾的 LIS 的长度』，就叫做对状态的定义。之所以把  $F_k$  叫做『状态』而不是『问题』，一是因为避免跟原原题中的『问题』混淆，二是因为这个新问题是数学化定义的。

对状态的定义只有一种吗？当然不是。我们甚至可以二维的，以完全不同的视角定义这个问题：

给定一个数列，长度为  $N$ ，设  $F_{i,k}$  为：在前  $i$  项中的长度为  $k$  的最长递增子序列中，最后一位的最小值。其中  $1 \leq k \leq N$ 。若在前  $i$  项中，不存在长度为  $k$  的最长递增子序列，则  $F_{i,k}$  为正无穷。求最大的  $x$ ，使得  $F_{N,x}$  不为正无穷。

上述的  $F_{i,k}$  就是状态，定义中的『 $F_{i,k}$  为：在前  $i$  项中的长度为  $k$  的最长递增子序列中，最后一位的最小值』就是对状态的定义。

### 7.4.3 状态转移方程

上述状态定义好之后，状态和状态之间的关系式，就叫做**状态转移方程**。

比如，对于 LIS 问题，第一种定义：

设  $F_k$  为：以数列中第  $k$  项结尾的最长递增子序列的长度。

设  $A$  为题中数列，状态转移方程为：

$$F_1 = 1 \text{ (根据状态定义导出边界情况)}$$

$$F_k = \max(F_i + 1 | A_k > A_i, i \in (1 \dots k-1)) (k > 1)$$

用文字解释一下是：以第  $k$  项结尾的 LIS 的长度是，保证第  $i$  项比第  $k$  项小的情况下，以第  $i$  项结尾的 LIS 长度加一的最大值，取遍  $i$  的所有值 ( $i < k$ )。

第二种定义：

设  $F_{i,k}$  为：在前  $i$  项中的长度为  $k$  的最长递增子序列中，最后一位的最小值。

设  $A$  为题中数列，状态转移方程为：

$$\text{若 } A_i > F_{i-1,k-1}, F_{i,k} = \min(A_i, F_{i-1,k}); \text{ 否则 } F_{i,k} = F_{i-1,k}.$$

这里可以看出，状态转移方程，就是定义了问题和子问题之间的关系。可以看出，状态转移方程就是带有条件的递推式。

### 7.4.4 动态规划迷思

#### 缓存，重叠子问题，记忆化

这三个名词，都是在阐述递推式求解的技巧。以 Fibonacci 数列为例，计算第 100 项的时候，需要计算第 99 项和第 98 项；在计算第 101 项的时候，需要第 100 项和第 99 项。这时候你还需要重新计算

第 99 项吗？不需要，你只需要在第一次计算的时候把它记下来就可以了。上述的需要再次计算的『第 99 项』，就叫『重叠子问题』。如果没有计算过，就按照递推式计算，如果计算过，直接使用，就像『缓存』一样，这种方法，叫做『记忆化』，这是递推式求解的技巧。这种技巧，通俗的说叫『花费空间来节省时间』。**都不是动态规划的本质，不是动态规划的核心。**

### 递归

递归是递归是求解的方法，连技巧都算不上。

### 无后效性，最优子结构

上述的状态转移方程中，等式右边不会用到下标大于左边  $i$  或者  $k$  的值，这是『无后效性』的通俗上的数学定义，符合这种定义的状态定义，我们可以说她他具有『最有子结构』的性质，在动态规划中我们要做的，就是找到这种『最优子结构』。

在对状态和状态转移方程的定义过程中，满足『最优子结构』是一个隐含条件（否则根本定义不出来）。对状态和和『最优子结构』的进一步理解 **TODO**

#### Note 7.1 对状态和和『最优子结构』的进一步理解

动态规划是对于某一类问题的解决方法！重点在于如何鉴定『某一类问题』是动态规划可解的，而不是纠结解决方法是递归还是递推！

怎么鉴定动态规划可解的一类问题，需要从计算机是怎么工作的说起。计算机的本质是一个状态机，内存里存储的所有数据构成了当前的状态，CPU 只能利用当前的状态计算出下一个状态（不要纠结硬盘之类的外部存储，就算考虑它们也只是扩大了状态的存储容量而已，并不能改变下一个状态只能从当前状态计算出来这一条铁律）。当你企图使用计算机解决一个问题时，其实就是在思考如何将这个问题表达成状态（用哪些变量存储哪些数据）以及如何在状态中转移（怎样根据一些变量计算出另一些变量）。所以，所谓的时间复杂度就是为了支持你的计算所必需存储的状态最多有多少，所谓空间复杂度就是从初始状态到达最终状态中间需要多少步！

一个问题是该用递推、贪心、搜索还是动态规划，完全是由这个问题本身阶段间状态的转移方式决定的！

- 每个阶段只有一个状态 → 递推；
- 每个阶段的最优状态都是由上一个阶段的最优状态得到的 → 贪心；
- 每个阶段的最优状态是由之前所有阶段的状态的组合得到的 → 搜索；
- 每个阶段的最优状态可以从之前某个阶段的某个或某些状态直接得到而不管之前这个状态是如何得到的 → 动态规划。

『每个阶段的最优状态可以从之前某个阶段的某个或某些状态直接得到』，这个性质叫做**最优子结构**；而不管之前这个状态是如何得到的，这个性质叫做**无后效性**。

需要注意的是，一个问题可能有多种不同的状态定义和状态转移方程定义，存在一个有后效性的定义，不代表该问题不适合动态规划。动态规划方法要寻找符合『最优子结构』的状态和状态转移方程的定义，在找到之后，这个问题就亦可以以『记忆化地求解递推式』的方法来解决。而寻找到的定义，才是动态规划的本质。即，**动态规划是寻找一个对问题的观察角度，让问题能够以递推（或者说分治）的方式去解决，寻找看问题的角度，才是动态规划中最耀眼的宝石！**

## 7.5 回溯法

## 7.6 分支限界法



# 第八章 秋招准备

整理各面经上的经典面试题，将各个思路进行一个系统化的梳理。

## 8.1 算法题

### 8.1.1 判断是否有环？

如何判断一个链表是否有环？

1. 最简单也是最直接的想法，从头遍历链表，将已经遍历过的节点信息保存起来，可以在遍历一个新节点时，查看该节点之前是否已经遍历过了，增加一个 tag 标记；或者将已经遍历过的节点信息使用 HashMap 保存起来，当遍历到一个新节点时，直接查表看之前是否已经遍历过该节点。
2. 使用两个指针 slow 和 fast，在刚开始时这两个指针均指向链表首节点，slow 每次向前走一步 ( $slow = slow->next;$ )，fast 每次向前走两步 ( $fast = fast->next->next;$ )，需要我们自己去判断是否 next 合法，比较 slow 和 fast 指针指向的节点，如果相同，说明该链表有环；如果不同，继续进行下一次循环 (slow 向前走一步，fast 向前走两步)。当链表中确实存在环的时候，slow 和 fast 最终会指向同一个节点；若不存在环，可以通过 fast 达到链表结尾处来结束算法循环。

环形链表的问题可以引申出更多的算法问题，比如：

问题一 在一个有环链表中，如何知道环的长度？如何找出链表的入环点？

- slow 和 fast 第一次相遇可以用于判断链表是否有环，记录下该碰撞点 p，slow 和 fast 再次于节点 p 相遇即为环的长度。
- 碰撞点 p 到入环点的距离等于头节点到入环点的距离，因此，当 slow 和 fast 第一次相遇得到碰撞点 p 之后，从碰撞点 p 和头节点开始走（每次走一步），相遇的那个点就是入环点。

问题二 判断两个单向链表是否相交，如果相交，求出交点。

问题三 用于找到链表的中间元素。使用快慢指针（slow 和 fast）可以实现。

---

一个有向图用邻接矩阵表示，并且是有权图，现在怎么判断图中有没有环？

拓扑排序的思想可以借鉴。

### 8.1.2 排序算法

各种排序算法总结<sup>1</sup>

---

<sup>1</sup><http://www.cnblogs.com/wxisme/p/5243631.html>

实现快速排序？

**TODO** 待整理

---

被忽视的 partition 算法<sup>2</sup>

### 8.1.3 二分查找

实现最简单的二分查找，分析二分查找代码实现中需要注意的地方，另外整理可以使用二分查找思想来做的一系列题目。

## 8.2 C / C++

### 8.2.1 基础知识

**sizeof** 运算符

**sizeof** 运算符返回一条表达式或一个类型名字所占的字节数。**sizeof** 运算符满足右结合律，其所得的值是一个 **size\_t** 类型的常量表达式（在编译时即可确定值）。**sizeof** 运算符的运算对象有两种形式：

**sizeof (type)** or **sizeof expr**

在第二种形式中，**sizeof** 返回的是表达式结果类型的大小。与众不同的一点是，**sizeof** 并不实际计算其运算对象的值。

**sizeof** 运算符的结果部分地依赖于其作用的类型：

- 对 **char** 或者类型为 **char** 的表达式执行 **sizeof** 运算，结果得 1；
  - 对引用类型执行 **sizeof** 运算符得到被引用对象所占空间的大小；
  - 对指针执行 **sizeof** 运算得到指针本身所占空间的大小；
  - 对解引用指针执行 **sizeof** 运算得到指针指向的对象所占空间的大小，指针不需有效；
  - 对数组执行 **sizeof** 运算得到整个数组所占空间的大小。注意，**sizeof** 运算不会把数组转换成指针来处理；
  - 对 **string** 对象或 **vector** 对象执行 **sizeof** 运算只返回该类型固定部分的大小，不会计算对象中的元素占用了多少空间。
- 

### C 的结构体和 C++ 结构体的区别

- C 的结构体内不允许有函数存在，C++ 允许有内部成员函数，且允许该函数是虚函数。所以 C 的结构体是没有构造函数、析构函数、和 **this** 指针的。
- C 的结构体对内部成员变量的访问权限只能是 **public**，而 C++ 允许 **public**、**protected**、**private** 三种。
- C 语言的结构体是不可以继承的，C++ 的结构体是可以从其他的结构体或者类继承过来的。

以上都是表面的区别，实际区别就是面向过程和面向对象编程思路的区别：

C 的结构体只是把数据变量给包裹起来了，并不涉及算法。而 C++ 是把数据变量及对这些数据变量的相关算法给封装起来，并且给对这些数据和类不同的访问权限。C 语言中是没有类的概念的，但是 C 语言可以通过结构体内创建函数指针实现面向对象思想。

---

<sup>2</sup>[https://selfboot.cn/2016/09/01/lost\\_partition/](https://selfboot.cn/2016/09/01/lost_partition/)

### C++ struct 和 class 关键字

- `struct` 和 `class` 都可以用来定义一个类；
  - `struct` 和 `class` 的默认访问权限不太一样；
  - 类可以在它的第一个访问说明符之前定义成员，对这种成员的访问权限依赖于类定义的方式。如果使用 `struct` 关键字，则定义再第一个访问说明符之前的成员是 `public` 的；相反，如果使用 `class` 关键字，则这些成员是 `private` 的。
- 

### C++ 引用和指针

#### C 语言中的字符串

整理自 **C 程序设计语言（第 2 版）** 5.5 字符指针与函数，中文版页码 89

字符串常量是一个字符数组，例如：

```
'hello, world\n';
```

在字符串的内部表示中，字符数组以空字符 ‘\0’ 结尾，程序可以通过检查空字符找到字符数组的结尾。字符串常量占据的存储单元数也因此比双引号内的字符数大 1。

字符串常量最常用的用法也许是作为函数参数，例如：

```
printf("hello, world.\n");
```

当类似于这样的一个字符串出现在程序中时，实际上是通过字符指针访问该字符串的。在上述语句中，`printf` 接受的是一个指向字符数组第一个字符的指针。也就是说，字符串常量可通过一个指向其第一个元素的指针来访问。

除了作为函数参数外，字符串常量还有其他用法。假定指针 `pmassage` 的声明如下：

```
char *pmassage;
```

那么，语句

```
pmassage = "hello, world\n";
```

将把一个指向该字符数组的指针赋值给 `pmassage`。该过程并没有进行字符串的复制，而只是涉及到指针的操作。C 语言没有提供将整个字符串作为一个整体进行处理的运算符。下面两个定义之间有很大的差别：

```
char amessage[] = "hello, world\n"; // 定义了一个数组
char *pmassage = "hello, world\n"; // 定义了一个指针
```

上述声明中，`amessage` 是一个仅仅保存足以存放初始化字符串以及空字符 ‘\0’ 的一维数组。数组中的单个字符可以进行修改，但 `amessage` 始终指向同一存储位置。另一方面，`pmassage` 是一个指针，其初值指向一个字符串常量，之后它可以被修改以指向其他地址，但如果试图通过 `pmassage` 指针修改字符串的内容，结果是没有定义的（会出现段错误 `Segment Fault`）。

另外，在页码 85 中有描述：

在函数定义中，形式参数 `char s[]` 和 `char *s` 是等价的。我们通常更习惯于使用后一种形式，因为它比前者更直观地表明了这个参数是一个指针。如果将数组名传递给函数，函数可以根据情况判定是按照数组处理还是按照指针处理，随后根据相应的方式操作该参数。为了直观且恰当地描述函数，在函数中甚至可以同时使用数组和指针这两种表示方法。

另外我们可以从汇编的角度来理解：

第一个字符串是用数组开辟的，它是可以改变的变量。

而第二个字符串则是一个常量，也就是字面值。`pmassage` 只是指向它的指针而已，而不能改变指向的内容。

---

```

movq L_main.amessage(%rip), %rax
movq %rax, -22(%rbp)
movl L_main.amessage+8(%rip), %ecx
movl %ecx, -14(%rbp)
movw L_main.amessage+12(%rip), %dx
movw %dx, -10(%rbp)
movb \$72, -22(%rbp) %% TODO

```

---

源代码 8.1: amessage 数组表示的汇编代码

---

```
leaq L_.str(%rip), %rcx
```

---

源代码 8.2: pmessage 指针表示的汇编代码

可见用数组和用指针是完全不相同的。

要想通过指针来改变常量是错误，正确的写法应该是用 `const` 指针。

```
const char *pmassage = "hello, world\n";
```

---

### C++ 中 string 的使用小结

正是因为 C 风格字符串（以空字符结尾的字符数组）太过复杂难于掌握，不适合大程序的开发，所以 C++ 标准库定义了一种 `string` 类，定义在头文件 `<string>`。

可以使用 C 风格字符串来初始化 C++ 中的 `string` 对象实例，

```
string str("hello, world\n"); // correct
```

但是如果直接将 `string` 类型的对象赋值给 C 风格的字符串的话，编译器会报错，

```
char *pstr = str; // error
```

但是实际应用中这个问题也难以避免，很多时候还是需要将 `string` 类型的转化为 `char*` 来实现自定义的操作，C++ 标准库也为了和之前用 C 写的程序兼容，于是可以用 `string` 的 `c_str()` 函数。

```
char *pstr = str.c_str(); // error
```

`c_str()` 为了防止意外地修改 `string` 对象，返回的是 `const` 指针，所以上面这段代码是不能被编译的。正确的应该是用 `const` 指针。

```
const char *pstr = str.c_str(); // correct
```

这个 `c_str()` 方法在 C++ IO 流操作上也被广泛应用。在打开文件时，如果要指定文件名，可以用 C 风格的字符串。如果用到 `string` 类型的字符串作为文件名时，就必须调用 `c_str()` 方法将其转换为一个 C 风格字符串。

---

```

string filename; // 定义文件名称
cin >> filename;
ifstream.open(filename.c_str()); // 要使用 C 风格字符串

```

---

源代码 8.3: `c_str()` 在 C++ IO 流操作上的应用

对 `string` 有一定了解后，C++ 标准库还定义的一系列丰富的字符串操作，均基于 `string` 类型。从某一种程度上来说，`string` 就是一种字符容器。

标准库为 `string` 定义了很多方法，包括构造、插入（`insert`）、替换（`assign` 和 `replace`）、比较（`compare`）、查找（`find`）、删除（`erase`）、连接（`append`）以及对子串的操作（`substr`），并且每一类操作都有很多种重载。<sup>3</sup>

### `std::endl` 与 ‘\n’ 有什么区别？在执行的功能上？

`std::endl` 是一个被称为 **操纵符**（manipulator）的特殊值。写入 `std::endl` 的效果是结束当前行，并将与设备相关联的缓冲区（buffer）中的内容刷到设备中。缓冲刷新操作可以保证到目前为止程序所产生的所有输出都真正写入输出流，而不是仅停留在内存中等待写入流。

在调试程序时，通常会添加打印语句。这类语句应该保证『一直』刷新流，否则，如果程序崩溃，输出可能还留在缓冲区中，从而导致关于程序崩溃位置的错误推断。

‘\n’ 为 C++ 语言规定的转义序列之一，**换行符**

#### Note 8.1 StackExchange 上一个 CodeReview 上的解释

*The main difference in the two is that std::endl in addition to adding \n to the stream will flush the stream. The stream will already auto flush at the optimal times and forcing a manual flush will only result in a degradation in performance.*

何时适用 `std::endl`，何时适用 ‘\n’ 呢？

由于流操作符 `operator<<` 的重载，对于 ‘\n’ 和 ‘‘\n’’，输出效果相同。

对于有输出缓冲的流（例如 `std::cout`、`std::clog`），如果不手动进行缓冲区刷新操作，将在缓冲区满后自动刷新输出。不过对于 `std::cout` 来说（相对于文件输出流等），缓冲一般体现得并不明显。但是必要情况下使用 `std::endl` 代替 ‘\n’ 一般是个好习惯。

对于无缓冲的流（例如标准错误输出流 `std::cerr`），刷新是不必要的，可以直接使用 ‘\n’，过多的 `std::endl` 是影响程序执行效率低下的因素之一。

由于直接输入/输出和操作系统相关，可能需要切换内核态/用户态，需要一定的时间开销，频繁地进行操作会极大地降低输入/输出的效率，所以标准库对流的输入/输出操作使用缓冲。具体来讲，就是在内存中保存一个大小相对固定的区域（缓冲区）用来储存临时的输入或输出。当必要时，才向系统设备复制缓冲区的内容并清空缓冲区，这个过程称为刷新。

```
std::cout << std::endl;
is equivalent to
std::cout << '\n' << std::flush;
So,
• Use std::endl if you want to force an immediate flush to the output.
• Use '\n' if you are worried about performance (which is probably not the case if you are using
the << operator).
```

### C++ 中的声明与定义的关系

TODO

### C++ 中的四个关键字：const / extern / static / volatile

<sup>3</sup><http://www.cnblogs.com/gaojun/archive/2010/09/11/1824016.html>

**const**

- 修饰基本数据类型和类类型，说明其为常量，不能修改
- 修饰指针 **从右向左进行定义的理解**
  - 指向常量的指针 (pointer to const) **const** 在 \* 左边，指针指向的是常量，不能使用此指针来改变其所指对象的值
  - 常量指针 (const pointer) **const** 在 \* 右边，指针本身为常量，即不变的是指针本身的值而非指向的那个值。常量指针必须初始化，而且一旦初始化完成，则它的值（也就是存放在指针中的那个地址）就不能再改变了
- 修饰函数参数和返回值，与上一 item 中的『修饰指针』描述相同
- 修饰类中的数据成员，
- 修饰类中的成员函数

---

```
struct Sales_data {
    std::string isbn() const { return bookNo; }

    std::string bookNo;
};
```

---

源代码 8.4: C++ Primer 上 Sales\_data 类的例子

类中的成员函数在被调用时，成员函数通过一个名为 **this** 的额外隐式函数来访问调用它的那个对象。当我们调用一个成员函数时，用请求该函数的对象地址初始化 **this**。例如，如果调用 **total.isbn()**；，则编译器负责把 **total** 的地址传递给 **isbn** 的隐式形参 **this**，可以等价地认为编译器将该调用重写成了 **Sales\_data::isbn(&total)**；，其中，调用 **Sales\_data** 的 **isbn** 成员时传入了 **total** 的地址。

在成员函数内部，我们可以直接使用调用该函数的对象的成员，而无需通过成员访问运算符来做到这一点，因为 **this** 所指的正是这个对象。任何对类成员的直接访问都被看作 **this** 的隐式引用，也就是说，当 **isbn** 使用 **bookNo** 时，它隐式地使用 **this** 指向的成员，就像写为 **this->bookNo** 一样。**this** 形参是隐式定义的，我们可以在成员函数体内部使用 **this**

---

```
struct Sales_data {
    std::string isbn() const { return this->bookNo; }

    std::string bookNo;
};
```

---

源代码 8.5: C++ Primer 上 Sales\_data 类的例子，添加上 this

因为 **this** 总是指向『这个』对象，所以 **this** 是一个常量指针，我们不允许改变 **this** 中保存的地址。

默认情况下，**this** 的类型是指向类类型**非常量版本的常量指针**。例如在上面的 **Sales\_data** 的成员函数中，**this** 的类型是 **Sales\_data \*const**。尽管 **this** 是隐式的，它仍然需要遵循初始化规则，意味着（在默认情况下）我们不能把 **this** 绑定到一个常量对象上。这一情况也就使得我们不能在一个常量对象上调用普通的成员函数。

如果 **isbn** 是一个普通函数而且 **this** 是一个普通的指针参数，则我们应该把 **this** 声明成

```
const Sales_data *const
```

毕竟，在 `isbn` 的函数体内不会改变 `this` 所指的对象，所以把 `this` 设置为指向常量的指针有助于提高函数的灵活性。

然而，`this` 是隐式的并且不会出现在参数列表中，所以在哪里将 `this` 声明成指向常量的指针就是我们必须面临的问题。C++ 语言的做法是允许把 `const` 关键字放在成员函数的参数列表之后，此时，紧跟在参数列表后面的 `const` 表示 `this` 是一个指向常量的指针。称为**常量成员函数** (`const` member function)。

---

```
// 伪代码，说明隐式的 this 指针是如何使用的
// 下面的代码是非法的：因为我们不能显式地定义自己的 this 指针
// 谨记此处地 this 是一个指向常量的指针，因为 isbn 是一个常量成员
std::string Sales_data::isbn(const Sales_data *const this) {
    return this->bookNo;
}
```

---

源代码 8.6: C++ Primer 上 Sales\_data 类的 `isbn` 成员函数伪代码

因为 `this` 是一个指向常量的指针，所以常量成员函数不能改变调用它的对象的内容。在上面的例子中，`isbn` 可以读取调用它的对象的数据成员，但是不能写入新值。

#### extern

- 引用其他文件中的 `const` 变量

当以编译时初始化的方式定义一个 `const` 对象时，编译器将在编译过程中把用到该变量的地方都替换成为对应的值。为了执行此替换，编译器必须知道变量的初始值。如果程序包含多个文件，则每个用了 `const` 对象的文件都必须得能访问到它的初始值才行。要做到这一点，就必须在每一个用到变量的文件中都有对它的定义。为了支持这一用法，同时避免对同一变量的重复定义，默认情况下，`const` 对象被设定为仅在文件内有效。当多个文件中出现了同名的 `const` 变量时，其实等同于在不同文件中分别定义了独立的变量。

某些时候有这样一种 `const` 变量，它的初始值不是一个常量表达式，但又确实有必要在文件间共享。解决办法是，对于 `const` 变量不管是声明还是定义都添加 `extern` 关键字，这样只需要定义一次就可以了：

---

```
// example.cc 定义并初始化了一个常量，该常量能被其他文件访问
extern const int bufSize = fcn();
// example.h 头文件
extern const int bufSize; // 与 example.cc 中定义的 bufSize 是同一个
```

---

源代码 8.7: 在多个文件之间共享 `const` 对象，必须在变量的定义之前添加 `extern`

- C++ 程序有时候需要调用其他语言编写的函数，最常见的是调用 C 语言编写的函数。需使用 `extern` 关键字

#### static

C++ 的 `static` 有两种用法：面向过程程序设计中的 `static` 和面向对象程序设计中的 `static`。前者应用于普通变量和函数，不涉及类；后者主要说明 `static` 在类中的作用。

##### 面向过程设计中的 static

###### 1 静态全局变量

在全局变量前，加上关键字 `static`，该变量就被定义成为一个静态全局变量。静态全局变量有以下特点：

- 该变量在全局数据区分配内存；
- 未经初始化的静态全局变量会被程序自动初始化为 0（自动变量的值是随机的，除非它被显式初始化）；
- 静态全局变量在声明它的整个文件都是可见的，而在文件之外是不可见的

静态变量都在全局数据区分配内存，包括后面将要提到的静态局部变量。对于一个完整的程序，在内存中的分布情况如下图：

```
代码区
全局数据区
堆区
栈区
```

一般程序的由 `new` 产生的动态数据存放在堆区，函数内部的自动变量存放在栈区。自动变量一般会随着函数的退出而释放空间，静态数据（即使是函数内部的静态局部变量）也存放在全局数据区。全局数据区的数据并不会因为函数的退出而释放空间。定义全局变量就可以实现变量在文件中的共享，但定义静态全局变量还有几个好处：1) 静态全局变量不能被其它文件所用；2) 其它文件中可以定义相同名字的变量，不会发生冲突。

## 2 静态局部变量

在局部变量前，加上关键字 `static`，该变量就被定义成为一个静态局部变量。通常，在函数体内定义了一个变量，每当程序运行到该语句时都会给该局部变量分配栈内存。但随着程序退出函数体，系统就会收回栈内存，局部变量也相应失效。但有时候我们需要在两次调用之间对变量的值进行保存。通常的想法是定义一个全局变量来实现。但这样一来，变量已经不再属于函数本身了，不再仅受函数的控制，给程序的维护带来不便。静态局部变量正好可以解决这个问题。静态局部变量保存在全局数据区，而不是保存在栈中，每次的值保持到下一次调用，直到下次赋新值。

静态局部变量有以下特点：

- 该变量在全局数据区分配内存；
- 静态局部变量在程序执行到该对象的声明处时被首次初始化，即以后的函数调用不再进行初始化；
- 静态局部变量一般在声明处初始化，如果没有显式初始化，会被程序自动初始化为 0；
- 它始终驻留在全局数据区，直到程序运行结束。但其作用域为局部作用域，当定义它的函数或语句块结束时，其作用域随之结束

## 3 静态函数

在函数的返回类型前加上 `static` 关键字，函数即被定义为静态函数。静态函数与普通函数不同，它只能在声明它的文件当中可见，不能被其它文件使用。定义静态函数的好处：1) 静态函数不能被其它文件所用；2) 其它文件中可以定义相同名字的函数，不会发生冲突

### 面向对象的 `static` 关键字（类中的 `static` 关键字）

#### 4 静态数据成员

在类内数据成员的声明前加上关键字 `static`，该数据成员就是类内的静态数据成员。静态数据成员有以下特点：

- 对于非静态数据成员，每个类对象都有自己的拷贝。而静态数据成员被当作是类的成员。无论这个类的对象被定义了多少个，静态数据成员在程序中也只有一份拷贝，由该类型的所有的对象共享访问。也就是说，静态数据成员是该类的所有对象所共有的。对该类的多个对象来说，静态数据成员只分配一次内存，供所有对象共用。所以，静态数据成员的值对每个对象都是一样的，它的值可以更新；

- 静态数据成员存储在全局数据区。静态数据成员定义时要分配空间，所以不能在类声明中定义。
- 静态数据成员和普通数据成员一样遵从 `public`, `protected`, `private` 访问规则；
- 因为静态数据成员在全局数据区分配内存，属于本类的所有对象共享，所以，它不属于特定的类对象，在没有产生类对象时其作用域就可见，即在没有产生类的实例时，我们就可以操作它；
- 静态数据成员初始化与一般数据成员初始化不同。静态数据成员初始化的格式为：  
`< 数据类型 >< 类名 >::< 静态数据成员名 > = < 值 >`
- 类的静态数据成员有两种访问形式：  
`< 类对象名 >.< 静态数据成员名 >` or `< 类类型名 >::< 静态数据成员名 >`  
如果静态数据成员的访问权限允许的话（即 `public` 的成员），可在程序中，按上述格式来引用静态数据成员；
- 静态数据成员主要用在各个对象都有相同的某项属性的时候。比如对于一个存款类，每个实例的利息都是相同的。所以，应该把利息设为存款类的静态数据成员。这有两个好处，1) 不管定义多少个存款类对象，利息数据成员都共享分配在全局数据区的内存，所以节省存储空间；2) 一旦利息需要改变时，只要改变一次，则所有存款类对象的利息全改变过来了；
- 同全局变量相比，使用静态数据成员有两个优势：1) 静态数据成员没有进入程序的全局名字空间，因此不存在与程序中其它全局名字冲突的可能性；2) 可以实现信息隐藏。静态数据成员可以是 `private` 成员，而全局变量不能

## 5 静态成员函数

与静态数据成员一样，我们也可以创建一个静态成员函数，它是为类类型提供服务而不是为某一个类的具体对象服务。静态成员函数与静态数据成员一样，都是类的内部实现，属于类定义的一部分。普通的成员函数一般都隐含了一个 `this` 指针，`this` 指针指向类的对象本身，因为普通成员函数总是具体的属于某个类的具体对象的。通常情况下，`this` 是缺省的。但是与普通函数相比，静态成员函数由于不是与任何的对象相联系，因此它不具有 `this` 指针。从这个意义上讲，它无法访问属于类对象的非静态数据成员，也无法访问非静态成员函数，它只能访问其余的静态成员（包括静态成员变量和静态成员函数）。

关于静态成员函数，可以总结为以下几点：

- 出现在类体外的函数定义不能指定关键字 `static`；
- 静态成员之间可以相互访问，包括静态成员函数访问静态数据成员和访问静态成员函数；
- 非静态成员函数可以任意地访问静态成员函数和静态数据成员；
- 静态成员函数不能访问非静态成员函数和非静态数据成员；
- 由于没有 `this` 指针的额外开销，因此静态成员函数与类的全局函数相比速度上会有少许的增长；
- 调用静态成员函数，可以用成员访问操作符（`.`）和（`->`）为一个类的对象或指向类对象的指针调用静态成员函数，也可以直接使用如下格式：

`< 类名 >::< 静态成员函数名 >(< 参数表 >)`

调用类的静态成员函数

`static` 关键字总结：

- 修饰局部变量，把局部变量从栈区移动到静态数据区，改变声明周期，不改变作用域
- 修饰全局变量，改变作用域，由全工程可见变成本源文件可见
- 修饰函数，与修饰全局变量相同

- 修饰类变量，是类的一部分，只有唯一一份副本
- 修饰类成员函数，没有 this 指针，只能访问静态成员（包括静态成员变量和静态成员函数）

另外可以参看 C++ 中 static 作用和使用方法<sup>4</sup>

### volatile

- 谈谈 C/C++ 中的 volatile<sup>5</sup> 在多线程环境下，不应该假设 volatile 能够解决多线程中的某些问题，而应该选用 C++11 中的互斥量和条件变量，分别在头文件 mutex 和 condition\_varialbe。

---

```
// global shared data
std::mutex m; // #include <mutex>
std::condition_variable cv; // #include <condition_variable>
bool flag = false;

thread1() {
    flag = false;
    Type* value = new Type(/* parameters */);
    thread2(value);
    std::unique_lock<std::mutex> lk(m);
    cv.wait(lk, [](){ return flag; });
    apply(value);
    lk.unlock();
    thread2.join();
    if (nullptr != value) { delete value; }
    return;
}

thread2(Type* value) {
    std::lock_guard<std::mutex> lk(m);
    // do some evaluations
    value->update(/* parameters */);
    flag = true;
    cv.notify_one();
    return;
}
```

---

源代码 8.8: 使用互斥锁和条件变量实现多线程之间的数据同步，伪代码

- C/C++ 中 volatile 关键字详解<sup>6</sup>
- 易变性：两条语句，如果存在变量重用的话下一条语句会重新从内存中读取；
- 不可优化：保证含有 volatile 的指令一定会被执行，而不会被激进的消除；
- 顺序性：保证 volatile 变量间的顺序性，编译器不会乱序优化，但 CPU 还是可能乱序发射执行。

### C++ 中的四种强制类型转换:static\_cast / dynamic\_cast / const\_cast / reinterpret\_cast

<sup>4</sup><https://blog.csdn.net/artechnor/article/details/2312766>

<sup>5</sup><https://liam0205.me/2018/01/18/volatile-in-C-and-Cpp/>

<sup>6</sup>[https://www.cnblogs.com/yc\\_sunniwell/archive/2010/07/14/1777432.html](https://www.cnblogs.com/yc_sunniwell/archive/2010/07/14/1777432.html)

一个命名的强制类型转换具有如下形式：

```
cast-name<type>(expression);
```

其中，`type` 是转换的目标类型，`expression` 是要转换的值。如果 `type` 是引用类型，则结果是左值。`cast-name` 是 `static_cast`、`dynamic_cast`、`const_cast` 和 `reinterpret_cast` 中的一种。`cast-name` 指定了执行的是那种转换。

#### `static_cast`

- 允许执行任意的隐式转换和相反的转换动作（无论是否被允许）。也可以用于基础类型之间的标准转换；
- 如类的指针，允许子类类型的指针转换成父类类型的指针（这是一个有效的隐式转换），同时，也能够执行相反动作，转换父类为它的子类。

#### `dynamic_cast`

只用于对象的指针和引用。当用与多态类型时，它允许任意的隐式类型转换以及相反过程。不过，与 `static_cast` 不同，在后一种情况里（注：即隐式转换的相反过程），`dynamic_cast` 会检查操作是否有效。也就是说，它会检查转换是否会返回一个被请求的有效完整对象。`dynamic_cast` 支持运行时类型识别。

#### `const_cast`

操纵传递对象的 `const` 属性，或者是设置或者是移除；一般用于强制消除对象的常量性。一旦我们去掉了某个对象的 `const` 性质，编译器就不再阻止我们对该对象进行写操作了。如果对象本身不是一个常量，使用强制类型转换获得写权限是合法的行为。然而如果对象是一个常量，再使用 `const_cast` 执行写操作就会产生未定义的后果。

只有 `const_cast` 能改变表达式的常量属性，使用其他形式的命名强制类型转换改变表达式的常量属性都将引发编译器错误。

#### `reinterpret_cast`

转换一个指针为其他类型的指针，或者转换一个指针为整数类型，操作结果是简单的从一个指针到别的指针的值得二进制拷贝。

**注意：** 强制类型转换干扰了正常的类型检查，因此我们强烈建议程序员避免使用强制类型转换。

## 8.2.2 C++ 语言特性

STL<sup>7</sup>

### 各类容器的底层实现

<sup>7</sup>STL 源码剖析，侯捷 著

容器	实现
<b>vector</b>	数组，支持快速随机访问
<b>deque</b>	deque 是一个双端队列 (double-ended queue)，也是在堆中保存内容的。它的保存形式如下：[heap1] -> [heap2] -> [heap3] 每个堆保存好几个元素，然后堆与堆之间有指针指向，看起来像是 list 和 vector 的结合
<b>list</b>	双向链表，支持快速增删
<b>stack</b>	用 list 或 deque 实现，封闭头部
<b>queue</b>	用 list 或 deque 实现，封闭头部
<b>priority_queue</b>	底层数据结构一般为 vector 为底层容器，堆 heap 为处理规则来管理底层容器实现
<b>set</b>	红黑树，有序，不重复
<b>multiset</b>	红黑树，有序，可重复
<b>map</b>	红黑树，有序，不重复
<b>multimap</b>	红黑树，有序，可重复
<b>unordered_set</b>	拉链 hash 表，无序，不重复
<b>unordered_multiset</b>	拉链 hash 表，无序，可重复
<b>unordered_map</b>	拉链 hash 表，无序，不重复
<b>unordered_multimap</b>	拉链 hash 表，无序，可重复

注：拉链 hash 表，当桶内的链表过长影响性能时，将存储的结构由链表换为红黑树，来提高效率。

## 1 顺序容器

### - **vector**

1. 在内存中有连续的存储空间，支持快速随机访问，插入删除效率慢，空间分配以 2 的倍数动态增长，每次内存空间不足需要先分配新的 2 倍大小的空间，然后把原先空间中的元素拷贝到新空间中；
2. `clear` 不会消除 `vector` 在内存中的元素，可以改用 `swap` (临时变量)。

- **deque** 支持两端插入数据，内存空间分布是小片的连续，小片之间用链表相连，重新分配空间不需要拷贝原有元素。

- **list** 双向链表，内存空间不连续，通过指针进行数据访问，随机存储低效，插入删除低效。

## 2 关联容器

- **set / multiset**(允许重复) 内部有序，元素唯一，无法直接存储元素，只能通过迭代器间接存取。

- **map / multimap**(允许重复) 内部有序，红黑树中的每个节点在不保存数据时，占用 16 个字（父指针、左右孩子指针、标识红黑色的枚举值），占内存较大。

## 3 容器适配器

- **queue** `deque` 基础上封装，先进先出。

- **stack** `deque` 基础上封装，先进后出。

## 算法

如 `sort`、`search` 等，模板函数，`[first, last)` 的迭代器区间。

## 迭代器

一种『范型指针』，所有容器都有自己的迭代器，只有容器本身知道如何遍历自己的元素，对指针的`+`、`*`、`-`、`->`等操作进行重载。

迭代器失效的总结：

容器	失效情况
<code>vector</code>	<code>vector</code> 的迭代器在内存重新分配时失效（它所指向的元素在该操作的前后不再相同）。当把超过 <code>capacity() - size()</code> 个元素插入 <code>vector</code> 中时，内存会重新分配，所有的迭代器都将失效；否则，指向当前元素以后的任何迭代器都将失效。当删除元素时，指向被删除元素之后的任何元素的迭代器都将失效。
<code>deque</code>	增加任何元素都将使 <code>deque</code> 的迭代器失效。在 <code>deque</code> 的中间删除元素将使迭代器失效。在 <code>deque</code> 的头或尾删除元素时，只有指向该元素的迭代器失效。
<code>list</code>	增加任何元素都不会使迭代器失效。删除元素时，除了指向当前被删除元素的迭代器外，其他迭代器都不会失效。
<code>set</code>	如果迭代器所指向的元素被删除，则该迭代器失效。其他任何增加、删除元素的操作都不会使迭代器失效。
<code>map</code>	如果迭代器所指向的元素被删除，则该迭代器失效。其他任何增加、删除元素的操作都不会使迭代器失效。

## 仿函数

仿函数是一种重载了 `operator()` 的类，使类的行为类似函数。

## 适配器

用来修饰容器接口、迭代器接口或仿函数接口；如 `stack`、`queue` 用来修饰 `deque`。

## 空间适配器

为容器进行空间配置和管理。

## 虚函数

虚函数是一种在基类定义为 `virtual` 的函数，并在一个或多个派生类中再定义的函数。

## 实现

- 通过 `vtbl` 和 `vptr` 来实现。每个声明或继承了虚函数的类，都有自己的 `vtbl`，该类的每个对象都有指向 `vtbl` 的 `vptr`
- 继承：如果子类覆盖了父类的虚函数，将被放到虚函数表中原来父类虚函数的位置；在多继承的情况下，对于派生类，多重继承，就会有几个虚函数表，这些表按照派生的顺序依次排列，如果子类改写了父类的虚函数，那么就会用子类自己的虚函数覆盖虚函数表的相应的条目，如果子类有新的虚函数，那么就添加到第一个虚函数表的末尾。

需要添加一个图片。

## 性能

通过 `vptr` 类型找到 `vtbl`，在 `vtbl` 中找到指针指向的函数。单继承性能差不多，多继承会慢（多个 `vtbl` 中查找）。

## 占用空间

虚函数表会增加类的体积，在类的继承过程中，每个子类都有父类的虚函数表。对于子类对象，但继承情况下会多一个 vptr 指针的体积（4 字节）；多继承情况下会多 N 个 vptr 指针的体积（ $4 \times N$  字节）。

---

```

class A {
public:
    virtual void funa();
    virtual void funb();
    void func();
    static void fund();
    static int si;
private:
    int i;
    char c;
};

Q: sizeof(A);
A: 12 (32 位系统) 或 16 (64 位系统)
// 一个指针 (4 或者 8) + int (4) + char (1) 计算对齐

```

---

#### 源代码 8.9: 虚函数表占用空间示例

关于类占用的内存空间，有以下几点需要注意：

1. 如果类中含有虚函数，则编译器需要为类构建虚函数表，类中需要存储一个指针指向这个虚函数表的首地址，注意不管有几个虚函数，都只建立一张表，所有的虚函数地址都存在这张表里，类中只需要一个指针指向虚函数表首地址即可；
2. 类中的静态成员是被类所有实例所共享的，它不计入 `sizeof` 计算的空间；
3. 类中的普通函数或静态普通函数都存储在栈中，不计入 `sizeof` 计算的空间；
4. 类成员采用字节对齐的方式分配空间

### 纯虚函数

在基类中没有定义，但要求任何派生类都要定义自己的实现方法。

### 虚析构函数

用一个基类的指针删除一个派生类的对象时，派生类的析构函数会被调用。只有当一个类被用来作为基类的时候，才把析构函数写成虚函数。

### 虚函数类型

- 不能成为虚函数的函数类型
    - 内联函数，编译时展看，必须有实体；
    - 静态成员函数，属于类，必须有实体；
    - 构造函数，`vtbl` 在类被创建后产生，如果析构函数是虚函数，那么构造函数自己无法创建类的实例，也就没有 `vtbl` 去访问。
  - 可以成为虚函数的函数类型
    - 析构函数
-

### 多态是怎么实现的?

C++ 的多态性分为静态多态和动态多态:

- 静态多态性: 编译期间确定具体执行哪一项操作, 主要是通过函数重载和运算符重载来实现的;
  - 动态多态性: 运行时确定具体执行哪一项操作, 主要是通过虚函数来实现的。
- 

### 右值<sup>8</sup>

---

### 智能指针

- 谈谈代理类<sup>9</sup>
- 谈谈智能指针: 原理及其实现<sup>10</sup>
- 谈谈 C++ 中的右值引用<sup>11</sup>

## 8.2.3 一些实现

- 无锁数据结构<sup>12</sup>
  - C++ 并发编程实战
  - ZeroMQ 实现中用到的无锁队列
  - ps-lite 中实现的 SArray , 类似于标准库中的 vector
- 

### C++ 实现单例模式

---

### C++ 使用两个栈实现一个队列<sup>13</sup>

---

## 8.3 操作系统

### 8.3.1 进程、线程与协程

### 8.3.2 select / poll / epoll

### 8.3.3 RPC

- 深入理解 RPC<sup>14</sup>
- 谁能用通俗的语言解释一下什么是 RPC 框架? <sup>15</sup>

<sup>8</sup><https://liam0205.me/2016/12/11/rvalue-reference-in-Cpp/>

<sup>9</sup><https://liam0205.me/2017/11/26/surrogate-in-Cpp/>

<sup>10</sup><https://liam0205.me/2018/01/13/smart-pointer/>

<sup>11</sup><https://liam0205.me/2016/12/11/rvalue-reference-in-Cpp/>

<sup>12</sup><http://blog.jobbole.com/90811/>

<sup>13</sup><http://www.cnblogs.com/wanghui9072229/archive/2011/11/22/2259391.html>

<sup>14</sup><https://juejin.im/entry/57c866230a2b58006b204712>

<sup>15</sup><https://www.zhihu.com/question/25536695>

## 8.4 面试回顾及整理

### 8.4.1 微信支付

一面 2018 年 8 月 16 日 17:30

1. 个人自我介绍?
2. 如何查看一个进程的父进程号?

`ps -ef` 就可以看到, 平常多使用的是 `ps aux`。这两者有何区别呢?

3. 在一个目录下有很多个其他文件 (包含文件和目录), 其中有一个文件的文件名是乱码, 应该执行什么样的 Linux 命令将这个文件名乱码的文件给删除掉?

Unix/Linux 系统内部不使用文件名, 而使用 inode 号来识别文件。对于系统来说, 文件名只是 inode 号便于识别的别称或者绰号。表面上, 用户通过文件名, 打开文件。实际上, 系统内部这个过程分成三步: 首先, 系统找到这个文件名对应的 inode 号; 其次, 通过 inode 号, 获取 inode 信息; 最后, 根据 inode 信息, 找到文件数据所在的 block, 读出数据。

使用 `ls -i` 命令, 可以看到文件名对应的 inode 号。有时, 文件名包含特殊字符, 无法正常删除。这时, 直接删除 inode 节点, 就能起到删除文件的作用。

这里两步来走:

- 1) 查看包含乱码的文件名的 inode 号

```
ls -il
```

- 2) 利用 inode 号, 执行 find 和 xargs 命令删除该文件

```
- find ./ -inum 5725772 -exec rm -i  
- rm -i 'find ./ -inum 5725772'
```

这两个方法都比较好理解, 通过 `ls -i` 这个命令获得要删除文件的 inode 号, 然后使用 `find` 命令的 `-inum` 选项查找对应 inode 号的文件名, 然后将文件名通过 `find` 的 `-exec` 参数或者通 “反引号” 传递给 `rm` 命令

```
- find ./ -inum 5725772 | xargs rm -i
```

在使用 `find` 命令的 `-exec` 选项处理匹配到的文件时, `find` 命令将所有匹配到的文件一起传递给 `exec` 执行。但有些系统对能够传递给 `exec` 的命令长度有限制, 这样在 `find` 命令运行几分钟之后, 就会出现溢出错误。错误信息通常是“参数列太长”或“参数列溢出”。这就是 `xargs` 命令的用处所在, 特别是与 `find` 命令一起使用。

4. 当在一个浏览器的搜索框中输入网址时会发生怎样的一系列动作? 涉及到哪些技术?

- IP → TCP → HTTP
- DNS 域名解析服务 (域名 → IP)、ARP 地址解析协议 (IP → MAC Address)、ANT (子网 ↔ 内网)
- 负载均衡、一致性哈希, CDN
- etc...

5. 在实现 C++ 类时, 如果我们不自己实现拷贝构造函数和复制构造函数, 编译会为我们自动生成, 那么, 在什么情况下需要自己手动实现这两个函数呢?

6. STL 使用时, 在什么情况下选择使用 `std::vector`, 在什么情况下选用 `std::list`?

7. `std::vector` 对象是如何增长的?

8. 如何查找 5 亿个元素的 Top-100 (最大的 100 个元素)?

有两种思路:

1) 求最大的 Top-K 使用最小堆，求最小的 Top-K 使用最大堆

**思路** 使用堆排序来实现 Top-K 的思路很直接。最大（小）堆就是其堆顶元素最大（小），比所有的子节点都要大（小）。求最大的 Top-K 个元素，构建最小堆（堆中共有 K 个元素），堆顶的元素就是堆中最小的元素。堆就保存的是当前扫描到的元素中的 Top-K 个了，怎么理解呢？现在我们从文件头开始读取文件内容，读入文件中的前 K 个元素构建好最小堆（当前最小堆中包含元素为  $A_1 - A_K$ ，为当前读取得所有 K 个元素的 Top-K 组成的堆，假设文件中包含的元素以 A 作为标记，其索引是元素被读取到的顺序），接着读取文件中剩余的元素，例如接下来读取到的是文件中的 (K+1) 个元素  $A_{K+1}$ ，将  $A_{K+1}$  与已构建最小堆的堆顶元素进行比较，如果  $A_{K+1}$  较小，那就不用入堆，因为当前读到的所有元素  $A_1 - A_{K+1}$  的 Top-K（最大）就是最小堆中的 K 个元素了；如果  $A_{K+1}$  较大，将堆顶元素移出，将  $A_{K+1}$  入堆（这里需要删除堆顶元素并插入一个新的元素，需要调整堆，仍然为最小堆）。以上，即，最小堆中的 K 个元素始终保存着当前已经扫描过的元素的 Top-K（最大）元素，新扫描到的元素比堆顶小（比当前最大的 K 个元素中的最小的还要小）就不需要入堆了，新扫描到的元素比堆顶大，将堆顶元素移出，新扫描到的元素入堆，调整仍未最小堆。

**速记口诀** 最小的 K 个用最大堆，最大的 K 个用最小堆。

**时间复杂度**  $\mathcal{O}(n \log K)$

**适用场景** 实现的过程中，我们先用前 K 个数建立了一个堆，然后扫描所有元素来维护这个堆。这种做法带来了三个好处：

- 1) 不会改变数据的输入顺序（按顺序读的）；
- 2) 不会占用太多的内存空间（事实上，一次只读入一个数，内存只要求能容纳前 K 个数即可）；
- 3) 由于 2)，决定了它特别适合处理海量数据。

这三点，也决定了它最优的适用场景。

2) 借鉴快速排序的 partition 函数的思路，进行折半

**思路**

**时间复杂度**  $\mathcal{O}(n)$

**适用场景** 对照着堆排的解法来看，partition 函数会不断地交换元素的位置，所以它肯定会改变数据输入的顺序；既然要交换元素的位置，那么所有元素必须要读到内存空间中，所以它会占用比较大的空间，至少能容纳整个数组；数据越多，占用的空间必然越大，海量数据处理起来相对吃力。但是，它的时间复杂度很低，意味着数据量不大时，效率极高。

9. 你有什么问题想问我的么？

微信支付是如何日活跃量如此巨大的访问的，有用到哪些技术？NJS、RPC 模块、多线程库、手动实现了协程序（routine）等等

**准备一下**

针对你简历上写的、你曾经做过的、你最值得说道的一个项目或者是一个需求或者是一次优化不断追问。作为应聘者你一定要充分做好这方面的准备。建议首先要在你的简历中有意突出这个点，让面试官注意到然后想要来问你。接下来你要针对这个点去准备材料，可以从以下几个方面去准备：

- 为什么要做这个？
- 你是如何做这个的？
- 期间你遇到了什么问题？你又是如何解决的？
- 你做完这个带来了什么实际的效果？

- 你做完之后呢？有继续优化和改进吗？
- 等等

WXG 面委-技术 GM 面试 2018 年 9 月 5 日 17:30

# 第九章 信息论与计算机

之前看过一篇介绍信息论的文章，今天看到从信息论的角度来讨论排序算法平均时间复杂度，遂打算整理一下信息论的知识，有本书叫《信息简史》，结合信息论与计算机的知识

- 对论文《A Mathematical Theory of Communication》的简要解读<sup>1</sup>
- 谈谈基于比较的排序算法的复杂度下界<sup>2</sup>
- Mind Hacks 刘未鹏<sup>3</sup>

## 9.1 信息论基础

---

<sup>1</sup><http://blog.codinglabs.org/articles/simple-explain-of-amtoc.html>

<sup>2</sup><https://liam0205.me/2018/08/28/lower-bound-of-comparation-based-sort-algorithm/>

<sup>3</sup><http://mindhacks.cn/topics/algorithms/>



# 第十章 杂的言

这里主要进行一些杂项的整理，对于自己接触到的一些感兴趣的公众号上的文章，或者平常看书的一些感悟，还有就是实际工作学习中遇到的一些问题的整理，即杂的言。

可以每天早上半小时、晚上半小时整理公众号上看到的知识点，生成 notes.pdf，然后发送到手机上进行 review 修改，自己的一些知识点也可以整理在那里。一天整理一点，以每天的日期

每周对这七天的整理进行总结

## 10.1 每月知识点归档

每个月月末最后一周将知识点做一个归档，例如加到之前已有的章节中，或者是新建一个章节。

## 10.2 每周整理为一些集中的点

每一个 review 之后将 10.3 节的内容进行一个整理，梳理到不同的 subsection 中，subsection 以各项技术为标题，例如缓存 10.2.1。

### 10.2.1 缓存

## 10.3 每日杂项整理

将每天自己学习到的不能归类的杂项整理在这里，以时间为 subsection 的标题。每周来一个统一的 review，在固定的时间，**每周六**，另外每天晚上坐车的时候也可以 review 一下。

### 10.3.1 20180710

- Git 抓取某一非 master 分支代码?  
git branch -r 查看远程分支  
或者 git branch -a 查看所有分支  
其后直接 git checkout [分支名] 就可以了
- CLion 中比较两个 git 分支的差异：选中工程 -> 右键 -> git -> compare with branch

### 10.3.2 20180711

- 自定义 Git - 配置 Git <https://git-scm.com/book/zh/v1>
- git fetch 的含义
- Change the email address for a git commit. <https://gist.github.com/trey/9588090>
-