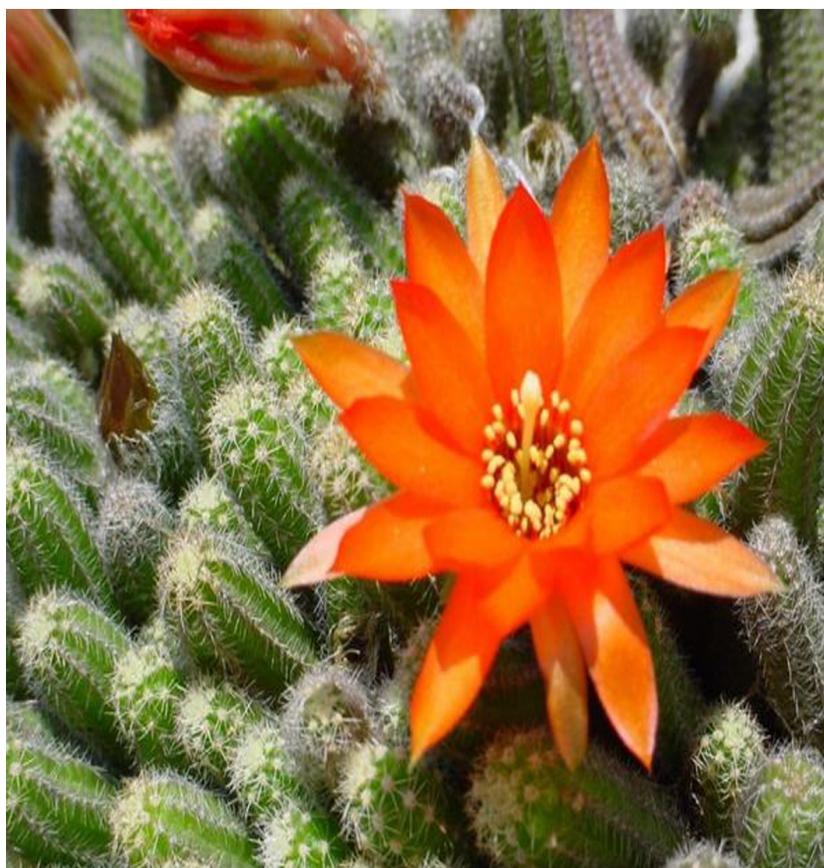


架构师

ARCHITECT



本期专题 | Topic

构建高可伸缩性的WEB交互式系统

特别专栏 | Column

腾讯大数据之TDW计算引擎解析
——Shuffle

梁定安：解密腾讯SNG云运维平台“织云”

推荐文章 | Article

看板如何奏效
理解Spark的核心RDD
缓存一致性入门



卷首语

金秋 10 月，又到了 2014 年 QCon 上海的季节。

记得去年 11 月 1 日是 QCon 第一次到上海，那日清晨我跟老公大人带着大包小包的喜糖，从浙江老家一路奔赴光大会展中心。社区编辑记者小分队在采访间忙里忙外，雪丰不知什么时候跟 Twitter 的 Raffi 聊了那么多东西，博学多才的 Mars 跟 Intel 的 Andrew Xia 聊了半天终于让我明白了啥是 Spark，青涩的 Steve 同学面对 Facebook 前端工程师 Hedger Wang 献上了自己的处女访，伯薇拉着平安科技的美女讲师问个不停，Kevin 则拉着不知从哪里蹦出来的 Roy 厉突如其来地占领了采访间……

每一次参加 QCon 大会都留下很多美好的回忆。回想最初参加 QCon 的时候谁也不认识，采访完了之后就在会场到处晃悠，找各种机会向身旁的陌生人搭讪，却总是说不了两句话就不知道该说什么了。后来，随着认识的朋友越来越多，每年 QCon 上都能看到越来越多熟悉的身影，跟他们打招呼，互相介绍新的朋友认识，找个桌子或者外面饭馆听他们聊天——那感觉可真是不错。

其实 InfoQ 组织的活动——无论是 QCon、ArchSummit 还是 QClub 等各路沙龙，都是有现场摄像的，会后都会发布到 InfoQ 网站上。采访内容也同样如此。所以如果你在现场只是听课骗礼品的话，那就太可惜了！QCon 在很多方面的设计都是为了营造合适的环境，让大家打开自己去跟别人交流——尤其是因为技术人这个物种比较 shy，会务组的同学们特别下了很多功夫。就好比有些人抱怨为什么早上的主题演讲不提前放人入场，而是让所有人都挤在会场的门口那里——这是因为根据经验，大家只要进入会场坐下后，往往就会玩手机玩到演讲开始了。所以下次当你在门口跟人挤作一团的时候，你就明白这是什么意思了：放下手机，跟身旁的同学们搭讪吧！

在下虽然不是男神也不是女神，但作为编辑，也算是试过不少种类的搭讪方式，在这里跟大家推荐一下：想要快速跟陌生人建立“严肃的”关系，最有效的方式就是采访！采访本身在操作上没什么难的，核心无非是“问正确的问题”而已。InfoQ 社区编辑团队现在在组建“采访小分队”，如果你想磨练你的搭讪技巧，现在就发邮件至 editors@cn.infoq.com 吧！

本期主编：杨赛

目录

卷首语

华三通信研发副总裁王飓谈传统通信技术与云计算

Databricks 连城谈 Spark 的现状

闪存将改变数据库存储引擎的设计

关于代码审查的几点建议

构建高可伸缩性的 WEB 交互式系统（上）

构建高可伸缩性的 WEB 交互式系统（中）

构建高可伸缩性的 WEB 交互式系统（下）

看板如何奏效

理解 Spark 的核心 RDD

缓存一致性入门

腾讯大数据之 TDW 计算引擎解析——Shuffle

梁定安：解密腾讯 SNG 云运维平台“织云”

李大维：互联网人做硬件创业容易产生的七大误解

可伸缩 NoSQL 数据库的五条建议



促进软件开发领域知识与创新的传播



中文 | 英文 | 日文 | 葡文 |

华三通信研发副总裁王飓谈传统通信技术与云计算

作者 杨赛

云计算无疑是 IT 界近几年最火的话题之一。很多互联网公司，传统 IT 公司，甚至是初创公司，都将视角投到了云计算上。

在即将于 10 月 16 日-18 日举行的 QCon 上海技术大会上，华三通信研发副总裁王飓将做题为《SDN 控制器集群中的分布式技术实践》的主题演讲。在大会召开前，InfoQ 中文站就传统通信技术与云计算的关系对他进行了采访。

InfoQ：在云计算时代，通信技术历史的积累您感觉有多少是有用的？新的技术又是主要来自哪里？

王飓：通信技术历史的积累其实大部分都是有用的。云计算核心关注的是虚拟化，并没有破坏 TCP/IP 的通讯层次模型，也没有彻底改变网络上需要的各种网络设备的种类（比如依然需要交换机/路由器/防火墙）。只是这些设备的控制管理方式（SDN 是改为了集中式管控）和形态（产生了虚拟形态如 VNF）发生了变化。当然个别协议在新的 SDN 架构上可能没用了，或者需要进化，这就是很自然的事情了，任何协议都有生命周期。

InfoQ：基于已有的体系改造，要比新建设一套体系更加困难。无论如何，过去有很多积累要放弃，给新的技术腾地儿。您目前跟通信领域的同行们沟通，感觉行内整体目前的心态是偏激进的（主动引入新技术革自己的命），还是偏保守的（用各种手段拦着新技术发展，给原来的积累多争取一些时间）？偏激进的那批人主要在哪儿？

王飓：我感觉业内大部分人心态是比较积极的。现在谈革命可能还不够准确，就像我上面说的，云计算时代，并不是完全颠覆已有的东西，那些过去的积累并不会一下子失去价值。

其实，大概在 6、7 年以前，我也迷茫过：数据通讯网络发展到当前的状况，是否就差不多了，剩下的只是提高一下带宽，提高一下芯片的容量就可以了？这就好像 19 世纪末一些物理学家认为经典物理学大厦已经建成，后人只有拾遗补阙的份了。所幸，这个世界的精彩总是超过我们的想象。云计算 /SDN 等技术的发展，无疑为我们的未来又重新增添了无穷的魅力。

当然，新技术是否就一定是好的？这个有待时间来验证。即使方向是对的，但具体的技术路线可能还是需要不断探索。所以，如果单纯是技术手段，则不存在什么保守或者激进的说法，任何技术行为都可以看成是一种尝试。在通信产业界，各种通信协议的竞争一直都有，而胜利从来不是单纯的看技术先进与否，适合的才是最好的。（当然，如果利用自己的垄断地位采用非正常的竞争手段是例外。）

我和我们公司都是这种变革的积极拥护者，这种变化，无论如何，最终都是让我们的技术更贴近我们的用户，让网络世界变的更美好。

InfoQ：有观点认为互联网是 IT 的消费者，并不是主要的 IT 贡献者，而主要的 IT 贡献者其实还是 IBM、思科、华为这些传统 IT 公司。现在 IT 越来越便宜，互联网公司获益最大，IT 公司、通信公司则很惨。但其实我们也看到像是 Google、Amazon 这样的互联网公司也在越来越多的往基础 IT 做投入。您如何评估现在互联网公司对于整体 IT 的贡献度？

王飓：如果 IT 这个词代表网络基础设施的话，可以认为互联网公司是 IT 的消费者，但别忘了最终的消费者是广大的用户，所有的评价应该以最终用户得到了什么作为评价。

技术本身是没有边界的，无论是互联网公司，还是所谓的传统 IT 公司，最终的目标都是满足和提升广大用户的使用体验。如果互联网公司认为现在的网络基础设施阻碍了他们提升用户体验，现有的传统 IT 公司无法给出他们满意的解决方案，则他们在这个方向投入就是很自然的事情。

至于说 IT 公司和通信公司，很惨应该还不准确，应该说是其利用技术和市场垄断来谋取高利润的时代结束了。

我们或许可以这样理解，互联网公司就是第三产业，是服务业，IT 公司就是第二产业，是制造业，完成了工业革命以后，第三产业的比重越来越大，附加值高，这是历史大趋势。这也是为什么十多年前思科、微软这样的公司市值最高，而现在的新贵是 Google、Facebook 和阿里巴巴。而当第三产业高度发展以后，肯定会反哺第二产业，这也就是为什么互联网公司开始向基础 IT 投入的原因。

想精确评价互联网公司对于整体 IT 的贡献度恐怕是很难的，但毫无疑问，他们的影响是正面的，推动了 IT 的发展，最终用户得到了个更优质、更方便、更便宜的服务。

InfoQ: 互联网公司作为用户，在网络方面倾向于 OpenFlow 这种完全由 Controller 掌控的网络结构，比如 Google 和 Facebook 都是 OpenFlow 的主要推手。但是这似乎是早期的一个观点，发展到现在，他们理想中的 Controller 还是没有实现，而且现在越来越多的声音也认为将所有控制逻辑放在 Controller 里面也未必合理。您对此是什么观点？您认为哪些节点应该使用 Controller，而转发层面应该保留哪些逻辑？

王飓：说这个问题，先看看我们的世界：今天的人类社会，是由一个个国家组成的，所有国家构成了一个松散的组织——联合国。每个国家都是一个自治的区域，有各种各样的政体，有的是松散的联盟，有的中央集权。看看互联网发展史，是否也是这样呢？我一直认为，网络虚拟空间，就是现实的折射。

所以，在一些封闭的空间里，比如一个数据中心，一个小的运营商网络，是比较适合使用 Controller 集中控制的。而在一些开放空间，边界不是很清晰的地方，还是需要传统的网络，这也是互联网诞生之初的设计思想：在一个无比广阔的空间里，网络自由互联、互通、自治，没有一个集中的控制点，任何对网络的攻击都只能影响局部，不会导致整个网络的崩溃。

至于转发层面需要保留哪些逻辑，现在大家争论还是很热烈的，但显然完全没有逻辑已经被证明不是最理想的，因为今天的设备，即使是冰箱和洗衣机都可以变得很聪明。这样看，完全把转发层面当成无逻辑的节点显然也是一种浪费。目前看，一些需要高速检测、本地链路快速切换等可能放到转发节点上更合适。

InfoQ: 现在互联网公司做云计算，传统 IT 公司也做云计算，运营商和通信公司也做云计算，初创企业也做云计算——这里说的云计算仅限于 IT 基础设施层面。就您的观察，你觉得他们要做的东西有什么不同？想要达到的目的有什么不同？对技术的需求又有什么不同？

王飓：云计算这个概念很大，大到不同的人可以有不同的理解。云计算的世界很广阔，广阔到可以容纳这些不同的力量一起来建设。

从方案上看，大体可以分为公有云，私有云和混合云三种；从内容上看有的偏重计算，有的偏重网络。

从技术本质上看，他们做的东西是类似的，但从需求角度看，又有明显的区别，各自的侧重点也不同。

互联网公司和运营商本身都是提供服务的，都是要解决自己的网络所面临的问题，相当于私有云，然后是建立公有云，提供公有云服务。

而通信公司则是希望借此机会进行转型，由卖设备变成卖服务，因为大家看到了，随着这个趋势（硬件标准化/软件开源化），设备的毛利率越来越低，价值链条中服务的部分会逐渐成为主体。这些通信公司即做公有云，也做私有云，甚至是混合云。这个服务和前面讲的互联网公司的服务不同，是指提供完整解决方案服务，自己并不运营。当然也不排除某些企业借此转型，变成了互联网公司。

初创企业就不说了，这个领域肯定是拿 VC 的热门，为什么不搏一下？而且初创企业没有历史包袱，显然在技术上是最有冲劲的，因为传统领域的蛋糕已经分完了，后来者要么重新做一个大蛋糕，要么就是打破原来壁垒，而推动一次技术革命显然是一个不错的选择。

InfoQ：感谢您接受我们的采访。

感谢臧秀涛对本文的审校。

查看原文：[华三通信研发副总裁王飓谈传统通信技术与云计算](#)



Databricks 连城谈 Spark 的现状

作者 张天雷

连城目前就职于 DataBricks，曾工作于网易杭州研究院和百度，也是《Erlang/OTP 并发编程实战》及《Erlang 并发编程（第一部分）》的译者。近日，InfoQ 中文站编辑跟连城进行了邮件沟通，连城在邮件中分享了自己对 Spark 现状的解读。

InfoQ：有专家侧重 Storm，您则是侧重 Spark，请简单谈谈这两者的区别和联系？

连城：Storm 是一个流处理系统，而 Spark 的适用范围则宽泛得多，直接涵盖批处理、流处理、SQL 关系查询、图计算、即席查询，以及以机器学习为代表的迭代型计算等多种范式。所以我想这个问题的初衷可能是想问 Storm 和 Spark 的流处理组件 Spark Streaming 之间的区别和联系？

Spark Streaming 相对于传统流处理系统的主要优势在于吞吐和容错。在吞吐方面，包括 Storm 在内的大部分分布式流处理框架都以单条记录为粒度来进行处理和容错，单条记录的处理代价较高，而 Spark Streaming 的基本思想是将数据流切成等时间间隔的小批量任务，吞吐量显著高于 Storm。在容错方面，Storm 等系统由于以单条记录为粒度进行容错，机制本身更加复杂，错误恢复时间较长，且难以并行恢复；Spark Streaming 借助 RDD 形成的 lineage DAG 可以在无须 replication 的情况下通过并行恢复有效提升故障恢复速度，且可以较好地处理 straggler。

除此之外，由于 Spark 整体建立在 RDD 这一统一的数据共享抽象结构之上，开发者不仅可以在单套框架上实现多种范式，而且可以在单个应用中混用多种范式。在 Spark 中，可以轻松融合批量计算和流计算，还可以在交互式环境下实现流数据的即席查询。Storm 相对于 Spark Streaming 最主要的优势在于处理延迟，但百毫秒至秒级延迟已经可以覆盖相当多的用例。更详细的分析比较可以参考 Matei Zaharia 博士的论文 An Architecture for Fast and General Data Processing on Large Clusters 的第四章。

InfoQ：2014年初您加入 Databricks 这个数据初创公司，当时是怎样一个契机触动了您？

连城：我于 2013 年六月第一次接触 Spark。此前函数式语言和分布式系统一直是我最为感兴趣的两个技术方向，而 Spark 刚好是这二者的一个很好的融合，这可以算是最初的契机。而深入接触之后，我发现 Spark 可以在大幅加速现有大数据分析任务的同时大幅降低开发成本，从而使得很多原先不可能的工作成为可能，很多困难的问题也得到了简化。Spark 的社区活跃度也进一步增强了我对 Spark 的信心。有鉴于此，去年十月份刚得知 Databricks 成立时便有心一试，并最终得偿所愿。😊

InfoQ：您之前翻译的图书都是跟并发和分布式相关，请您介绍一下 Spark 在并发和分布式上的设计？

连城：分布式系统设计的一大难点就是分布式一致性问题。一旦涉及可变状态的分布式同步，系统的复杂性往往会陡然上升。而 Spark 则较好地规避了这个问题。个人认为原因主要有二：

1. Spark 是一个大数据分析框架，其本身并不包含任何（持久）存储引擎实现，而是兼容并包现有的各种存储引擎和存储格式，所以规避了分布式存储系统中的分布式数据一致性问题。
2. 虽然函数式语言还远未成为主流，但在大数据领域，以不可变性（immutability）为主要特征之一的函数式编程却已经深入骨髓。扎根于函数式编程的 MapReduce 固然是一大原因，但我猜想另一方面可能是因为在大数据场景下，单一节点出错概率较高，容错代价偏大，因此早期工程实践中一般不会在计算任务中就地修改输入数据，而是以新增和/或追加文件内容的方式记录中间结果和最终结果，以此简化容错和计算任务的重试。这种对不可变性的强调，大大削减了大数据分析场景下的数据一致性问题的难度。上层框架也因此得以将注意力集中在容错、调度等更为高层的抽象上。

在并发方面，和 Hadoop 的进程级并行不同，Spark 采用的是线程级并行，从而大大降低了任务的调度延迟。借助于 Akka 的 actor model，Spark 的控制位面和数据位面并发通讯逻辑也相对精简（Akka 本身也的确是跟 Erlang 一脉相承）。

InfoQ：Spark 社区现在是空前火爆，您觉得其流行的主要原因是什么？

连城：可能是大家受 Hadoop 压迫太久了吧，哈哈，开玩笑的。我觉得原因有几点：

1. Spark 在大大提升大数据分析效率的同时也大大降低了开发成本，切实解决了大数据分析中的痛点。

2. 通过 RDD 这一抽象解决了大数据分析中的数据共享这一重要问题，从而使得开发者得以在单一应用栈上混合使用多种计算范式打造一体化大数据分析流水线，这大大简化了应用的开发成本和部署成本。
3. 简洁明了的接口。我曾经碰到这么一个真实案例，一位 Spark contributor 用 Spark 来做数据分析，但他的数据量其实很小，单机完全可以处理，他用 Spark 的主要理由就是接口简洁明了，写起来代码来“幸福指数”高 :-) 当然另一个重要原因则是今后数据量大起来之后可以很方便地 scale out。
4. 对兼容性的极致追求。面对资产丰富的 Hadoop 生态，Spark 的选择是全面兼容，互惠共赢。用户无须经历痛苦的 ETL 过程即可直接部署 Spark。这也是 Cloudera、MapR、Pivotal、Hortonworks 等 Hadoop 大厂商全面拥抱 Spark 的重要原因之一。

InfoQ：Spark 目前好像还没有完全大规模应用，您觉得开发者主要的顾虑在什么地方？

连城：由于大数据本身的重量，大数据分析是一个惯性很大的技术方向，相关新技术的推广所需要的时间也更加长久。我个人接触到的案例来看，Spark 用户的主要顾虑包括两点：

1. Scala 相对小众，认为相关人才培养和招聘上会比较吃力。这个问题我认为正在缓解，而且有加速的趋势。
2. 对数百、上千节点的大规模集群的稳定性的顾虑。实际上一千节点以上 Spark 集群的用例已经出现多个，其中 eBay 的 Spark 集群节点数已超过两千。

InfoQ：您最希望 Spark 下一版本能解决的技术难题是什么？

连城：我近期的工作主要集中于 Spark SQL，在 1.2 的 roadmap 中最为期待的还是正在设计当中的外部数据源 API。有了这套 API，用户将可以在 Spark SQL 中采用统一的方式注册和查询来自多种外部数据源的数据。Cassandra、HBase 等系统的深度集成将更加统一和高效。

InfoQ：在部署 Spark 集群、设计 Spark 应用时有哪些方面的问题需要考量？

连城：

- 集群部署方式，standalone、Mesos 和 YARN 各有千秋，需要按需选用。
- 在单集群规模上，也可以按需调整。Yahoo 和腾讯采用的是多个小规模卫星集群的部署模式，每个集群都有专用的目的，这种模式故障隔离更好，可以保证更好的 QoS。同时业内也不乏 eBay 这样的单体大集群案例，其主要点在于更高的集群资源利用率以及对大规模计算的应对能力。
- 与现存数据分析系统的对接。对于常见系统如 Kafka、Flume、Hive、支持 JDBC 的传统数据库，可以利用 Spark 提供的现成接口；对于 Spark 项目本身尚未涵盖的，或是私有系统的对接，可以考虑开发自定

义数据源 RDD。在 1.2 版本以后，也可以考虑通过 Spark SQL 的外部数
据源 API 来对接现有结构化、半结构化数据。

- 合理挑选、组织需要 cache 的数据，最大限度地发挥 Spark 内存计算的
优势。
- 熟悉并合理选用恰当的组件。Spark 提供了多个可以互操作的组件，可
以极方便的搭建一体化的多范式数据流水线。
- 和所有其他基于 JVM 的大数据分析系统一样，规避 full GC 带来的停顿
问题。

采访者简介

张天雷（@小猴机器人），清华大学计算机系博士，熟悉知识挖掘，机器学习，社交网络
舆情监控，时间序列预测等应用。目前主要从事国产无人车相关的研发工作。

查看原文：[Databricks 连城谈 Spark 的现状](#)

闪存将改变数据库存储引擎的设计

作者 马德奎

过去十年，固态硬盘（俗称闪存）已经从根本上改变了计算机信息处理技术。在客户端，U 盘取代了 CD；在服务器端，它有高于 RAM 和磁盘驱动器的性价比。但在过去的几年里，数据库才刚刚开始赶上这一趋势，而且大部分仍然依赖于针对旋转磁盘内部数据结构和存储管理的优化来提升性能。

近日，[O'Reilly Media](#) 资深编辑 [Andy Oram](#) 发表了一篇文章，他基于对数位数据库专家的采访，详细介绍了闪存如何改变了数据库存储引擎的设计，其中包括 Aerospike、Cassandra、FoundationDB、RethinkDB 和 Tokutek 的代表人物。对于正在设计应用程序和寻找最佳存储方案的读者而言，他们给出的各种方法会有一定的指导意义。

根据介绍，闪存影响数据库存储引擎设计的关键特性如下：

- 随机读：闪存不同于传统磁盘，它像内存一样，不管两次读的物理距离相差多远，它都可以以同样的速度提供数据。不过，它每次会读取整个块，所以，应用程序可能仍然会受益于访问局部性。比如，如果本次读与上次读的位置相近，那么本次操作可能可以直接从内存或者缓存读取数据。
- 吞吐量：有记录的原始吞吐量已达到每秒几十万次的读/写，这比磁盘高两个数量级，甚至更高。而且，随着磁盘密度的提高，吞吐量还在增长。
- 延时：据 FoundationDB CEO David Rosenthal 说，通常，闪存的读延时大约为 50 到 100 微秒。而 RethinkDB CEO Slava Akhmechetat 指出，闪存至少比磁盘快 100 倍。不过，闪存的延时已经达到了极限。
- 并行：闪存驱动器提供多个控制器或者单个性能更高的控制器。这对于能够使用多个线程和内核的数据库设计大有裨益，它可以将工作负载划分成许多独立的读写操作。

那么，这些特性对数据库存储引擎的设计有什么影响呢？为了说明这个问题，Oram 介绍了一些企业的现行做法。

Aerospike 是第一款从设计之初就选择了闪存的数据库产品。它将索引存储在 RAM 中，其它数据存储在闪存中。这样，他们可以在 RAM 中快速查找索引，然后从多个闪存驱动器中并行检索数据。由于索引在 RAM 中更新，向闪存写数据的次数就大大减少了。

Cassandra 通过排序数据实现了访问局部性。它的基本数据结构是日志结构的合并树（LSM-树）。和闪存一起使用时，该结构可以显著减少写操作。据项目负责人 Jonathan

Ellis 说，为了保证 LSM-树的效率，Cassandra 承担了许多碎片整理工作，而大部分应用程序都把这项工作留给文件系统来做。而据 Rosenthal 说，FoundationDB 团队的做法则与此相反，他们依赖闪存控制器解决写碎片问题。闪存控制器可以完成 LSM 在数据库引擎层面所做的工作。现在，大部分闪存控制器都提供了这些算法。这里有一点需要注意，实现访问局部性会增加写操作的开销。在闪存吞吐量如此大的情况下，这部分开销可能会超过多次读操作的开销。

Tokutek 提供了一个聚簇数据库 TokuDB，他们发现聚簇是检索范围数据的理想选择。TokuDB 的压缩比很高（在 MySQL 或 MariaDB 上为 5 比 1 或 7 比 1，在 MongoDB 上为 10 比 1），这有效地减少了读写开销，并降低了存储成本。而且据[官方介绍](#)，它所使用的分形树索引结构减少了写操作次数，延长了闪存的使用寿命。

Aerospike、FoundationDB、RethinkDB 和 Tokutek 都是用 MVCC 或类似的概念连续写入新版本数据，并在稍后清理老版本数据，而不是直接用新值替换已存数据。因此，数据库的一个写请求会变成多个操作，这称为[写入放大](#)，是闪存的一个缺点。但据 Bulkowski 说，通过将索引存储在内存中，Aerospike 的写入放大仅为 2，而在其它应用程序中，这个值通常为 10。

此外，按照 Rosenthal 的说法，闪存的速度和并发为数据库设计带来了最大的变化。他说，“在传统关系型数据的设计中，每个连接一个线程，这在磁盘是瓶颈的时代可以工作的很好，但现在，线程成了瓶颈。”因此，FoundationDB 内部使用它自己的轻量级进程。在闪存延迟无法再改善的情况下，并发显得更重要了。而 Bulkowski 则表示，由于大量的并发，深队列在闪存上比在旋转型磁盘上工作的更好。

总之，这些新的数据库存储引擎设计已经抛弃了许多传统的设计方案。为了利用这些新的发展成果，应用程序开发人员应该重新审视他们的数据库模式和访问模式了。

感谢[郭董](#)对本文的审校。

查看原文：[闪存将改变数据库存储引擎的设计](#)

关于代码审查的几点建议

作者 李士窑

Code Review 即代码审查是软件开发中常用的手段，它和 QA 测试相比，更容易发现架构以及时序相关等较难发现的问题，还可以帮助团队成员统一编程风格，提高编程技能等。代码审查被公认为是一个提高代码质量的有效手段。目前很多开发团队虽然进行了代码审查，但是他们可能没有有效、合理的进行代码审查，以致没有很好达到代码审查的目的。近日，BIDS 贸易科技有限公司的 CTO [Jim Bird](#) 总结了关于代码审查的一些建议。现对这些建议进行了一个全面的梳理，具体内容如下。

1、代码审查不要太正式

目前，有很多研究表明正式代码的评审会议会延误开发进度和增加开发成本。尽管可能只需要几周的时间进行代码评审，但是只有 4% 的缺陷是在会议期间发现的，其余所有的权限是靠代码审查者自己发现和处理的。只有采用简短、轻量的代码审查才是有效的发现问题在代码检查，这样的代码审查更适合迭代、增量开发，为开发者提供更快的反馈。

2、代码审查人员要尽可能少

并不是代码审查人员越多就能发现越多 Bug，只有合理数量的审查人员才能够更加有效地审查代码。研究表明，平均来说，一个代码审查人员能够发现 Bug 的一半，第二审稿人会发现剩余新问题的一半。多个人同 2 个人发现问题的数量没有太大差异，故两个人进行代码审查是比较合适的。另外，还由于社会惰性的存在，更多代码审查人员意味着多人在寻找同样的问题，使得审查人员积极性、主动性不高，更加不利于代码审查工作的有效进行。

3、需要有经验的开发者进行代码审查

研究充分表明，代码审查的有效性依赖于审查人的技能和对问题领域以及代码的熟悉程度。如果让新加入团队的成员进行代码审查的话，并不利于他们的成长，且对于代码审查来说也是一种非常糟糕的方式。只有擅长阅读代码、程序调试、非常熟悉语言、框架、对应的问题的人才最适合代码审查，才能够高效发现问题、提供更多有价值的反馈。新的、没有经验的开发者只适合检查代码的变化和使用静态分析工具并和另一位评论人员共同代码审查。

4、实质重于方式

完全按照编码规格标准进行的代码审查是一个浪费开发人员宝贵时间的方式。代码审查的实质是确认代码能够正确的运行，发现安全漏洞、功能错误、代码错误、设计失误、安全验证和防范、恶意代码等。而不是单单按照编码规范完全保证代码格式一致，而丢失了代码审查的实质。

5、合理安排 Bug 和可维护性问题代码的审查时间分配

发现代码中的 Bug 是很难的，在别人的代码找到的 Bug 更难。研究表明，代码审查人员找到 Bug 和可维护性、可读性问题的比例是 25:75，故消耗在了代码可读性、可维护性等问题上和 Bug 上的代码审查时间应该合理分配。

6、尽量使用静态代码分析工具以提高审查效率

工欲善其事，必先利其器。静态代码分析工具可以帮助程序开发人员自动执行静态代码分析，快速定位代码隐藏错误和缺陷；帮助代码设计人员更专注于分析和解决代码设计缺陷；显著减少在代码逐行检查上花费的时间，提高了软件可靠性并节省软件开发和测试成本。

7、二八定律处理高风险代码

审查所有的代码并没有太大的意义，应该把审查的重点放在高风险的代码和容易引起高风险的修改或者重构的代码上。旧而复杂、处理敏感数据、处理重要业务逻辑和流程、大规模重构以及刚加入团队的开发者实现的代码都是审查的重点。

8、从代码审查中尽量获得最大的收益

虽然代码审查是发现 Bug、提高开发人员代码编写质量的重要方式，但是它也增加了代码开发成本。如果没有合理、有效的进行代码审查，将有可能影响项目进度和破坏团队文化。故我们要紧抓代码审查的实质性问题，尽早和经常性的进行非正式的代码审查；选择精而少的人员并运用二八定律审查高风险的代码，同时，还需要合理分配 Bug 以及可维护性问题的代码审查时间，才可以从代码审查中获得最大的收益。

感谢郭蕾对本文的审校。

查看原文：[关于代码审查的几点建议](#)

ArchSummit

全球架构师峰会 2014

2014.12.19-20 北京国际会议中心



互联网金融
研发体系构建
云计算解决方案专场
电商，不是搭个平台就能赢

- 转型中的SNS
- 智能硬件，更懂你
- 移动互联网，随时随地
- 云计算与大数据，从技术选型说起

构建高可伸缩性的 WEB 交互式系统（上）

作者 蔡剑飞

可伸缩性是一种对软件系统处理能力的设计指标，高可伸缩性代表一种弹性，在系统扩展过程中，能够保证旺盛的生命力，通过很少的改动，就能实现整个系统处理能力的增长。

在系统设计的时候，充分地考虑系统的可伸缩性，一方面能够极大地减少日后的维护开销，并帮助决策者对于投资所能获得的回报进行更加精准的估计；另一方面，高可伸缩性的系统往往会有更好的容灾能力，从而提供更好的用户体验。

WEB 交互式系统的可伸缩性主要体现在两个方面：

- 平台的可伸缩性：随着 WEB 技术的发展，越来越多的平台开始使用 WEB 技术来构建系统，一方面不同的平台提供的环境支持存在着各种差异；另一方面随着平台的发展，不断的会有一些旧平台退出历史舞台，新平台转而成为主流平台；因此构建的 WEB 系统需要能够快速的响应此类变化就需要其具备良好的平台伸缩性。
- 模块的可伸缩性：随着系统功能不断增删更新需求的变化，系统可能会变得越来越复杂，冗余信息也可能会越来越多，改动所带来的影响范围也可能会越来越大，因此良好的模块伸缩性可保证系统具有良好的可维护性，让系统始终处于最佳状态。

WEB 交互式系统的主要应用包括：

- 桌面端/移动端网站类系统（如 [网易云课堂](#)、[易信 WebIM](#)、[Lofter 移动 WEB 版](#) 等）
- 移动混合应用（如 [网易云相册 IPad 版](#)、[Lofter](#) 等）
- 桌面混合应用（如 [网易云音乐 PC 版](#)、[网易邮箱助手](#) 等）

本系列文章主要分为两个主要部分对可伸缩性进行阐述，分别是平台的可伸缩性和模块的可伸缩性。本文是系列文章的第一篇，讨论平台的可伸缩性。

平台的可伸缩性

WEB 交互式系统对平台的可伸缩性主要表现为：

- 可扩展性：对于新兴平台能够快速进行支持。
- 可缩减性：对于过时的平台冗余信息能够以最小的修改方式剔除。

我们先介绍一下 WEB 交互式系统的目标平台的情况。

平台分类

根据系统所在容器的差异，我们将平台分为浏览器平台和混合应用平台两大类。各分类的详细说明见下文所述。

浏览器平台

按引擎划分

浏览器平台，按照主流引擎可以划分为以下几类：



引擎	说明
Trident	由微软研发的排版引擎，代表浏览器有 Internet Explorer
Webkit	由 Apple、Google、Adobe 等公司推动的开源的排版引擎，代表浏览器有 Apple Safari、Google Chrome
Gecko	由 Mozilla 基金会支持的开源排版引擎，代表浏览器有 Mozilla Firefox
Presto	由 Opera Software 研发的商用排版引擎，代表浏览器有 Opera，由于 Opera 从 15 以后就开始采用新的 Blink 引擎，因此 Presto 也将逐步淡出我们的目标平台
Blink	由 Google 和 Opera Software 基于 Webkit 引擎研发的排版引擎，代表浏览器有 Chrome 28+、Opera 15+

按功能划分

各引擎的浏览器版本根据对标准、规范的支持程度进行划分，可分为以下几类：



由于目前国内基于 Trident 的 Internet Explorer 浏览器还占有大量的市场份额，包括低版本的 Internet Explorer 浏览器，因此我们将浏览器分成三个等级：

标准性	说明
差	主要针对低版本的 Trident 引擎（如 IE6 浏览器）平台，这部分平台对规范和标准的支持程度比较差，在适配时需要做大量额外的适配工作来实现相应功能，因此如果产品的目标平台定位需要支持此平台则会有一定的性能损耗
中	主要针对中间版本的 Trident 引擎（如 IE7-9 浏览器）平台，这部分平台对规范和标准有一定的支持，但是也存在若干功能需要做额外的适配工作来实现
好	主要针对对规范、标准支持比较好的平台，按照标准实现的功能无需做额外的适配工作，因此如果产品的目标平台定位为此平台将取得比较好的用户体验和性能，如移动产品、混合应用等

混合应用平台

根据混合应用的宿主平台的差异，我们将混合应用的目标平台分为以下几类：



宿主	说明
Android	Android 系统的混合应用，浏览器引擎会自动适配至 Webkit
iOS	iOS 系统的混合应用，浏览器引擎会自动适配至 Webkit
WinPhone	Windows Phone 系统的混合应用，浏览器引擎会自动适配至 Trident
PC	桌面应用，采用 CEF 做为容器，浏览器引擎会自动适配至 Webkit

平台适配

AOP (Aspect-Oriented Programming)：面向切面的编程范式，其核心思想是将横切关注点从主关注点中分离出来，因此特定领域的问题代码可以从标准业务逻辑中分离出来，从而使得主营业务逻辑和领域性业务逻辑之间不会存在任何耦合性。

这里我们可以借鉴 AOP 思想来实现平台的适配策略，结合不同的平台实现逻辑，我们可以认为对于使用规范、标准来实现业务逻辑的部分为我们的主关注点，而不同平台可以做为

若干的切面关注点进行封装，各平台只需关注自己平台下对标准的修正逻辑即可，因此可以通过增加、删除平台修正的切面逻辑来实现对不同平台的适配。

实现时我们首先提取标准业务逻辑，然后各平台根据实际情况实现对业务逻辑的修正：



- 标准业务逻辑：主关注点，这里主要是使用根据 W3C、ES 标准来实现的业务逻辑。
- 前置平台修正逻辑：领域特定关注点，主要是根据平台特性对标准在该平台下的修正，修正逻辑会先于标准逻辑执行。
- 后置平台修正逻辑：同前置平台修正逻辑，也是领域特定关注点，修正逻辑会在标准逻辑执行后再执行。

根据此思路我们对比以下两段代码：

代码一：目前常用的平台适配方式

```

function doSomething(){
    if(isTrident){
        // TODO trident implement
    }else if(isWebkit){
        // TODO webkit implement
    }else if(isGecko){
        // TODO gecko implement
    }else if(isPresto){
        // TODO presto implement
    }else{
        // TODO w3c implement
    }
}

// 上层应用使用
doSomething(1,2,3);
  
```

此方式对所有平台的修正逻辑均在主逻辑中实现，存在以下弊端：

- 对平台特有的修正逻辑耦合在主逻辑中，平台特有的更新必然引起主逻辑的更新。
- 对于新增或删除平台的支持必须修改到主营业务逻辑。
- 无法分离不必要的平台修正，比如基于 webkit 引擎的移动平台应用不需要其他平台的修正逻辑。

代码二：借鉴 AOP 思想的平台适配方式

```

function doSomething(){
    // TODO w3c/es implement
}
// 上层应用使用
  
```

```
doSomething(1,2,3);
```

针对 Trident 平台适配的逻辑，比如 trident.js 中

```
// trident implement
doSomething = doSomething._$aop(
    function(_event){
        // TODO trident implement
    },
    function(_event){
        // TODO trident implement
    }
);
```

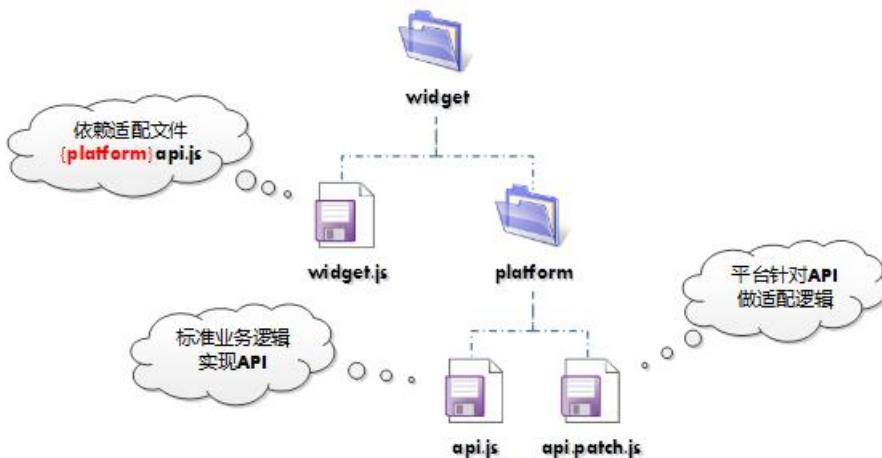
对比代码一，我们可以发现借鉴 AOP 思想的接口适配方式分离了标准业务逻辑和平台特有业务逻辑，是否增加平台特有业务逻辑并不会影响主营业务逻辑的执行，而对于平台修正逻辑的切入则可以直接通过配置的方式灵活的进行增删，因此我们可以从中得到以下好处：

- 主逻辑和平台特有逻辑无耦合性，可随意分离、整合。
- 对于新增平台适配只需新加平台特有逻辑即可，而无需影响到主营业务逻辑。
- 可通过配置控制支持的目标平台，有选择性的导出平台特有业务逻辑。

实现举例

[NEJ 框架](#)借鉴 AOP 思想提供了配置式的平台适配系统，对于这部分的详细信息可参阅 NEJ 的[《依赖管理系统》](#)和[《平台适配系统》](#)了解更为详细的信息，以下仅举例说明 NEJ 中适配的使用方式。

一个典型的适配控件结构如下图所示：

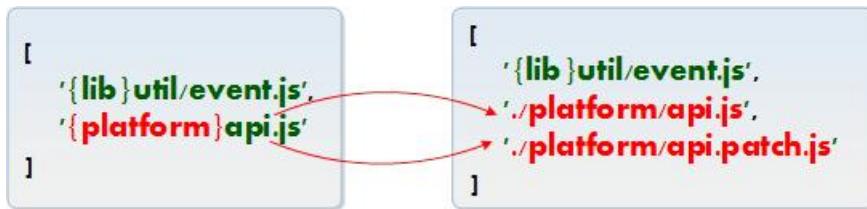


这里的 widget.js 是控件业务逻辑实现文件，在此控件的实现中会依赖到存在平台差异的

API，其依赖代码如下所示：

```
NEJ.define([
  'util/event',
  '{platform}api.js'
],function(t,h,p){
  // TODO
});
```

这里对 {platform}api.js 的处理方式如下图所示，这里的./相对于当前的代码文件即 widget.js 文件所在的目录



这里的 api.js 文件为需平他适配 API 的标准实现逻辑，而 api.patch.js 文件则利用 NEJ.patch 接口对各平台做按需适配逻辑，同时打包时也根据 NEJ.patch 接口中对平台的条件识别做按需输出，由于 api.patch.js 文件最终会按需输出，因此在此文件中除了使用 NEJ.patch 做平台适配逻辑外，不允许包含其它业务逻辑。

```
// 此文件只能定义 NEJ.patch 不可执行其他业务逻辑
// 打包输出时仅根据平台配置输出所需处理逻辑
// 实际情况看需求，可将平台相关部分逻辑独立到单独的模块中
```

```
NEJ.define([
  './hack.js'
],function(h){
  // 针对 trident 平台的处理逻辑
  NEJ.patch('TR',function(){
    // TODO
  });

  // 针对 gecko 平台的处理逻辑
  NEJ.patch('GR',[

  ],function(fh){
    // TODO
  });

  // 针对 IE6 平台的处理逻辑
  NEJ.patch('TR==2.0',[ './hack.ie6.js' ]);

  // 针对 IE7-IE9 的处理逻辑
  NEJ.patch('3.0<=TR<=5.0',function(){
    // TODO
  });
  // 这里必须同 hack.js 文件的返回值一致
  return h;
});
```

最后我们只需要配置产品的目标平台即可输出平台对应的适配，而不会存在其他平台的额外影响：

```
<script src="/path/to/nej/define.js?p=wk|gk|td"></script>
<script src="/path/to/nej/define.js?p=cef"></script>
```

平台变更

通过以上实现举例我们可以看到当平台发生变更时我们可以快速进行扩展或缩减

平台扩展

当有新平台需要作为系统目标平台时，我们只需要做以下工作：

- 增加平台配置识别符，如 nxw。
- 识别该平台与标准存在的差异，增加平台特有业务逻辑至 patch。
- 系统对平台配置部分增加新添的识别符，如

原平台适配

```
<script src="/path/to/nej/define.js?p=wk|gk|td"></script>
```

新增平台适配

```
<script src="/path/to/nej/define.js?p=wk|gk|td|nxw"></script>
```

即可完成对平台的扩展，而不会影响到原有的业务逻辑。

平台缩减

当系统适配的目标平台由于某种原因退出历史舞台时，系统也需要将该平台的冗余代码从系统中剔除，我们只需要做以下工作：

系统对平台配置部分删除要剔除的平台标识，如：

原平台适配

```
<script src="/path/to/nej/define.js?p=wk|gk|td"></script>
```

缩减后平台适配

```
<script src="/path/to/nej/define.js?p=wk"></script>
```

即可完成对平台的缩减，而无需修改任何业务逻辑。

以上即是有关平台可扩展性的介绍。下一篇将阐述模块的可扩展性，敬请期待！

本作品采用[知识共享署名 4.0 国际许可协议](#)进行许可。

作者介绍

蔡剑飞，网易杭州研究院前端高级技术专家，2005 年加入网易，先后参与过网易邮箱、网易博客、网易相册、网易云音乐等产品的开发，从 2010 年开始开发 NEJ 框架，现在负责 NEJ 框架的维护及培训。他的邮箱是 genify@163.com，微博是 genify，欢迎大家来信交流！

感谢[张云龙](#)对本文的审校。

查看原文：[构建高可伸缩性的 WEB 交互式系统（上）](#)

构建高可伸缩性的 WEB 交互式系统（中）

作者 蔡剑飞

在[《构建高可伸缩性的 WEB 交互式系统》](#)的第一篇，我们介绍了 Web 交互式系统中平台的可伸缩性。本文将描述模块的可伸缩性。

模块的可伸缩性

WEB 交互式系统对模块的可伸缩性同样表现为：

- 可扩展性：对于系统新增的功能需求能够快速响应支持。
- 可缩减性：对于系统退化的模块能够以最小的修改方式剔除。

这里我们提供一套模块调度的系统架构模式，用于支持单页富应用系统的设计架构、模块拆分、模块重组、调度管理等功能。

模块

我们定义的模块是指：从系统中拆分出来的、可与用户进行交互完成一部分完整功能的独立单元。

模块组成

因为这里描述的模块可独立与用户完成交互功能，因此模块会包含以下元素：

- 样式：定义模块的效果。
- 结构：定义模块的结构。
- 逻辑：实现模块的功能。

以上元素对于一个 WEB 系统开发者来说并不陌生，而我们只需要寻求一种形式将这些内容封装起来即可。

模块封装

从模块的组成我们可以看到系统中分离出来的模块可能会长成这个样子，比如 module.html 就是我们分离出来的一个模块。

当然这里也可以用脚本文件封装，样式和结构采用注入形式。下面以 html 文件封装举例：

```
<!-- 模块样式 -->
<style>
    .m-mdl-1 .a{color:#aaa;}
    .m-mdl-1 .b{color:#bbb;}

    /* 此处省略若干内容 */
</style>

<!-- 模块结构 -->
<div class="m-mdl-1">
    <p class="a">aaaaaaaaaaaaaaaaaa</p>
    <p class="b">bbbbbbbbbbbbbbbbbbbb</p>

    <!-- 此处省略若干内容 -->
</div>

<!-- 模块逻辑 -->
<script>
    (function(){
        var a = 'aaa';
        var b = 'bbb';

        // 此处省略若干内容
    })();
</script>
```

这个模块在用户需要时加载到客户端，并展现出来跟用户进行交互，完成功能。但是我们会发现，如果系统预加载了此模块或者模块在 parse 时，这些内容会被直接执行，而这个结果并不是我们需要的，因此我们需要将模块的各元素文本化处理。文本化处理有多种方式，如作为文本 script、textarea 等标签内容，因此 module.html 里的模块我们可以封装成如下样子，以 textarea 举例：

```
<!-- 模块样式 -->
<textarea name="css">
    .m-mdl-1 .a{color:#aaa;}
    .m-mdl-1 .b{color:#bbb;}

    /* 此处省略若干内容 */
</textarea>

<!-- 模块结构 -->
<textarea name="html">
    <div class="m-mdl-1">
        <p class="a">aaaaaaaaaaaaaaaaaa</p>
        <p class="b">bbbbbbbbbbbbbbbbbb</p>

        <!-- 此处省略若干内容 -->
    </div>
</textarea>

<!-- 模块逻辑 -->
<textarea name="js">
```

```
(function(){
    var a = 'aaa';
    var b = 'bbb';

    // 此处省略若干内容
})();
</textarea>
```

管理依赖

从系统中拆分出来的模块之间是存在有一定关系的，如一个模块的呈现必须依赖另外一个模块的呈现。下面我们会以一个简单的例子来讲解模块之间的依赖管理，如下图是我们的一个单页应用系统：

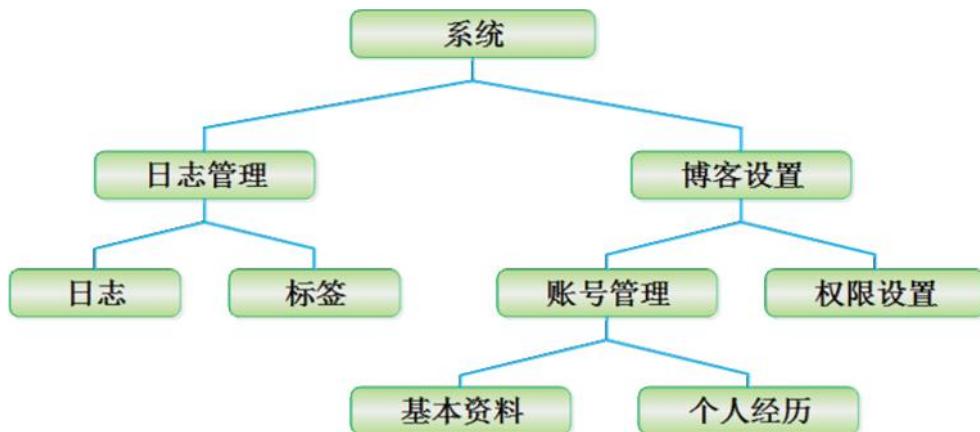
标签	Log ID
tag-1409106504605(0)	tag-1409106504605(0)
tag-1409106504605(1)	tag-1409106504605(1)
tag-1409106504605(2)	tag-1409106504605(2)
tag-1409106504605(3)	tag-1409106504605(3)
tag-1409106504605(4)	tag-1409106504605(4)
tag-1409106504605(5)	tag-1409106504605(5)
tag-1409106504605(6)	tag-1409106504605(6)
tag-1409106504605(7)	tag-1409106504605(7)
tag-1409106504605(8)	tag-1409106504605(8)
tag-1409106504605(9)	tag-1409106504605(9)
tag-1409106504605(10)	tag-1409106504605(10)
tag-1409106504605(11)	tag-1409106504605(11)
tag-1409106504605(12)	tag-1409106504605(12)
tag-1409106504605(13)	tag-1409106504605(13)
tag-1409106504605(14)	tag-1409106504605(14)
tag-1409106504605(15)	tag-1409106504605(15)
tag-1409106504605(16)	tag-1409106504605(16)
tag-1409106504605(17)	tag-1409106504605(17)
tag-1409106504605(18)	tag-1409106504605(18)

从上图不难看出整个系统包含以下几部分内容：

- 日志管理
 - 日志：日志列表，可切换收件箱/草稿箱/回收站/标签。
 - 标签：标签列表，可转至日志按标签查看列表。

- 博客设置
 - 账号管理
 - 基本资料: 用户基本资料设置表单。
 - 个人经历: 个人经历填写表单。
 - 权限设置: 权限设置表单

而这些模块之间的层级关系则如下所示:



针对交互式系统的这种层级架构典型的模式可以参阅:

- [PAC \(Presentation—Abstraction—Control\) 模式](#)
- [HMVC \(Hierarchical model-view-controller\) 模式](#)

然而在 WEB 交互式系统的实践过程中我们发现这种模式会存在一些缺陷:

- 由于每个父模块自己维护了所有的子模块，因此父子模块之间耦合性过强，父模块必须耦合所有子模块。
- 由于模块之间不能直接越级调用，因此子模块需要其他模块协助时必须层层向上传递事件，如果层级过深则会影响到系统效率。
- 模块的增删等变化导致的变更涉及的影响较大，删除中间节点上的模块可能导致相邻的若干模块的变更。
- 多人协作开发系统时存在依赖关系的模块会导致开发人员之间的紧密耦合。

在这里，我们给出了一种基于模块标识的依赖管理配置方案，可以彻底的将模块进行解耦，每个模块可以独立完整的完成自己的交互功能，而系统的整合则可以通过配置的方式灵活的重组各模块，模块的增删操作只需修改配置即可完成，而无需影响到具体业务逻辑。

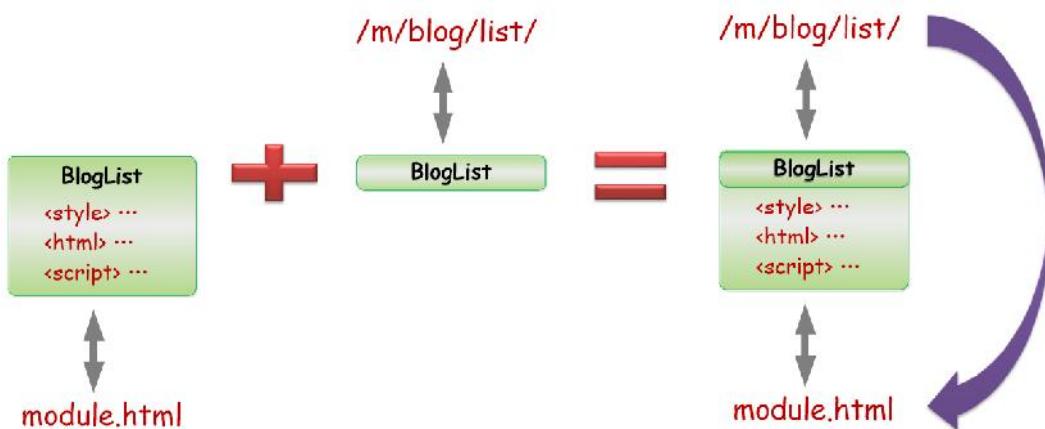
下文我们会通过以上例子来讲解此方案的原理和实际操作方式。

模块标识

因为本方案会基于模块标识做配置，因此在介绍方案之前我们先介绍一下模块标识，这里我们给模块标识取名为 **UMI**（Uniform Module Identifier）统一模块标识，下文简称 UMI，遵循以下规则约定：

- 格式同 URI 的 Path 部分，如 /m/m0/
- 必须以“/”符开始
- 私有模块必须以“?”开始
- 承载模块的依赖关系，如 /m/m0/ 和 /m/m1/ 表明这两个标识对应模块的父模块标识均为 /m

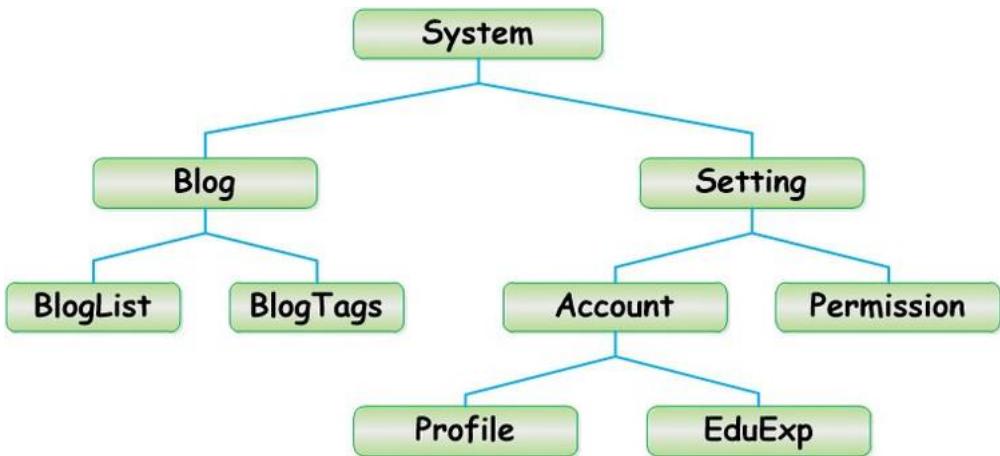
每个 UMI 均可唯一标识一个模块及模块在系统中的依赖关系，在模块章节我们介绍了一个模块可以用一个 html 进行封装，因此我们可以得到以下结果：



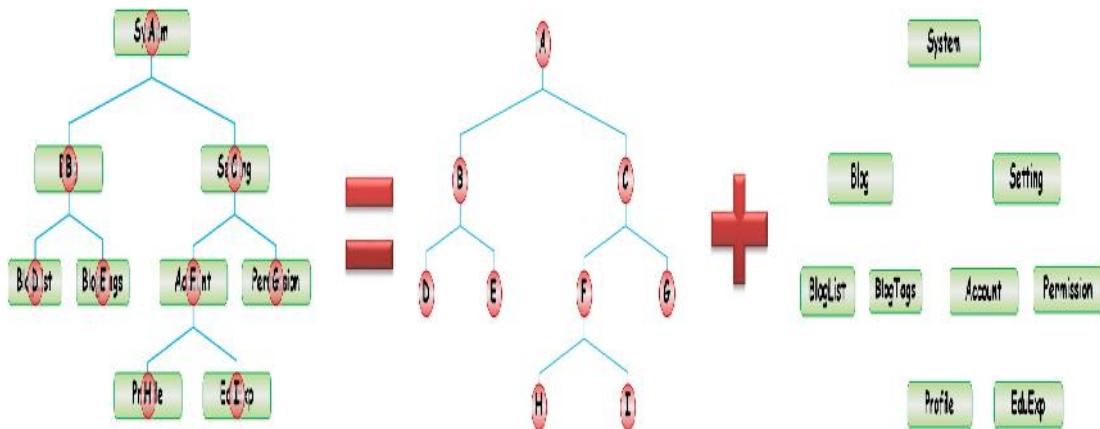
每个 UMI 均可映射到一个模块实现文件，这样我们就可以将模块从具体实现中解耦出来，对模块的增删修改操作只需调整 UMI 和模块文件的映射关系即可，而无需涉及具体业务逻辑的修改。

模块依赖

在解决了模块与实现分离的问题后，我们接下来需要将层级式的模块扁平化来解耦模块之间的依赖关系。回到前面的例子，模块之间的层级关系如下图所示：

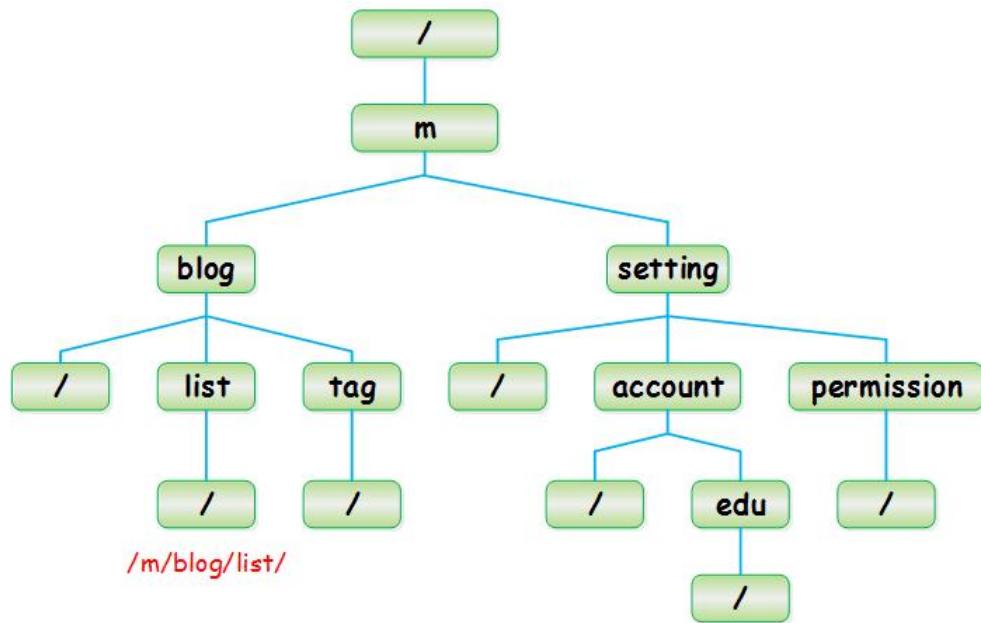


如果我们将图中的依赖关系进行抽象分离后，可以发现所有的模块即可呈现扁平的状态：

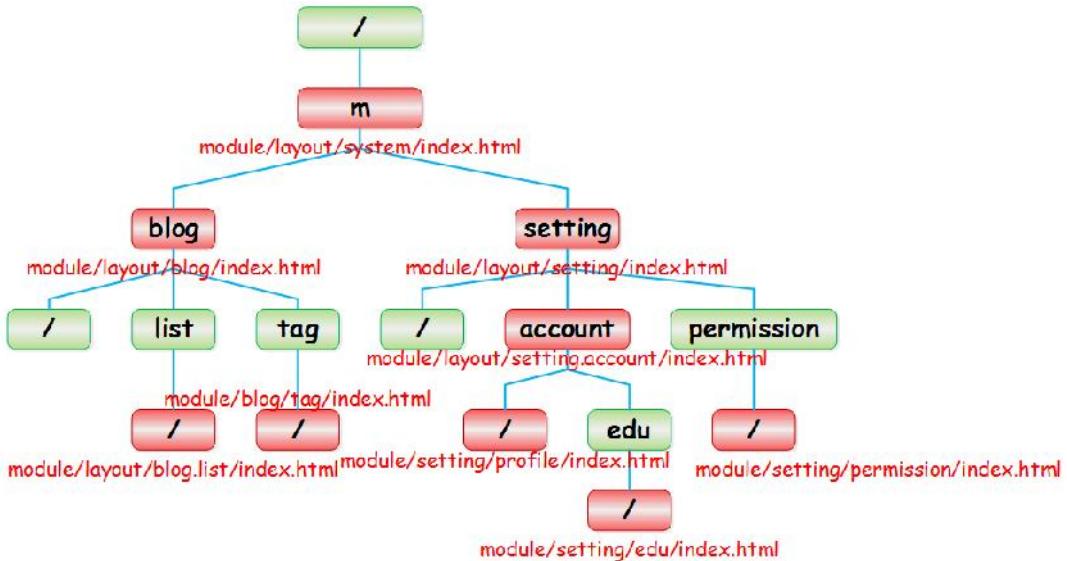


而对于模块之前的依赖关系的管理，在所有系统中都是一致的，但是每个模块的具体功能实现是由系统来决定的，不同的系统是截然不同的，因此本方案提供的解决方案主要是用来维护模块之间的依赖关系的。

从上图我们可以比较清楚的看到模块之间的依赖关系呈现树状结构，因此我们会以树的结构来组织维护模块之间的依赖关系，我们称之为依赖关系树。而当我们把这棵树上的任意节点与根节点之间的路径用“/”分隔序列化后，发现刚好与我们提供的 UMI 是匹配的，因此组成系统的模块的 UMI 可以跟依赖关系树的节点一一对应起来，如下图所示：



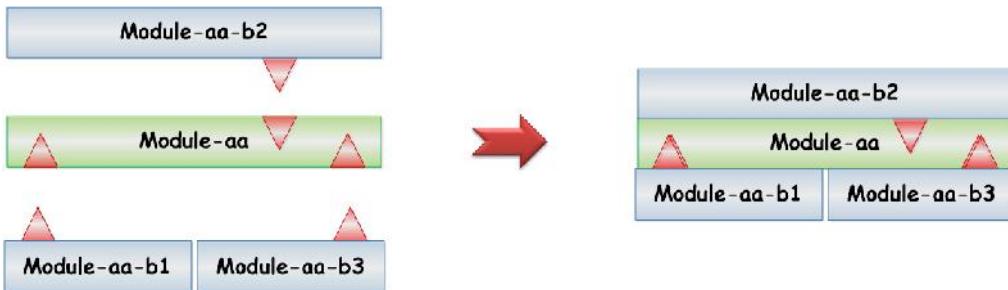
在模块标识章节我们介绍了 UMI 与模块封装文件可以相互映射，因此依赖关系树上的节点可以直接与模块的实现文件做一一对应，如下图所示：



至此，我们将垂直层级依赖的模块通过依赖关系树分解成了无任何关系的扁平模块结构。

模块组合

模块只需要有个呈现容器即可渲染出来，因此模块如果需要能够做任意组合，只需将模块分成两种类型：提供容器的模块，和使用容器的模块即可。当然，一个模块可同时兼具提供容器和使用容器的功能，提供容器的模块和使用容器的模块可任意组合。



对于模块组合的配置代码范例：

```

'/m/blog/list':{
  module:'module/layout/blog.list/index.html',
  composite:{
    box:'/?/blog/box/',
    tag:'/?/blog/tag/',
    list:'/?/blog/list/',
    clazz:'/?/blog/class/'
  }
}

```

调度策略

在将模块扁平化后，各模块就可以安排给不同的开发人员进行功能实现和测试了，各模块完成后根据依赖关系树进行系统整合，系统整合后各模块会遵循一定的调度策略进行调度。

模块状态

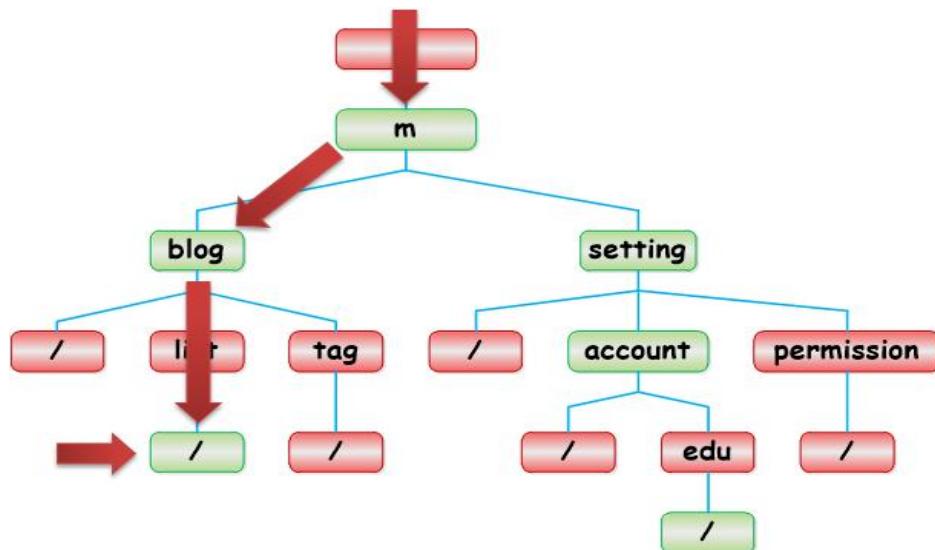
根据模块调度的阶段划分，模块的状态可以分为以下四种：

- 模块构建：构建模块结构。
- 模块显示：将模块渲染到指定的容器中。
- 模块刷新：根据外界输入的参数信息获取数据并展示（这里主要做数据处理）。
- 模块隐藏：模块放至内存中，回收由显示和刷新阶段产生的额外数据及结构。

调度策略主要控制模块在这几个阶段之间的转换规则。

模块显示

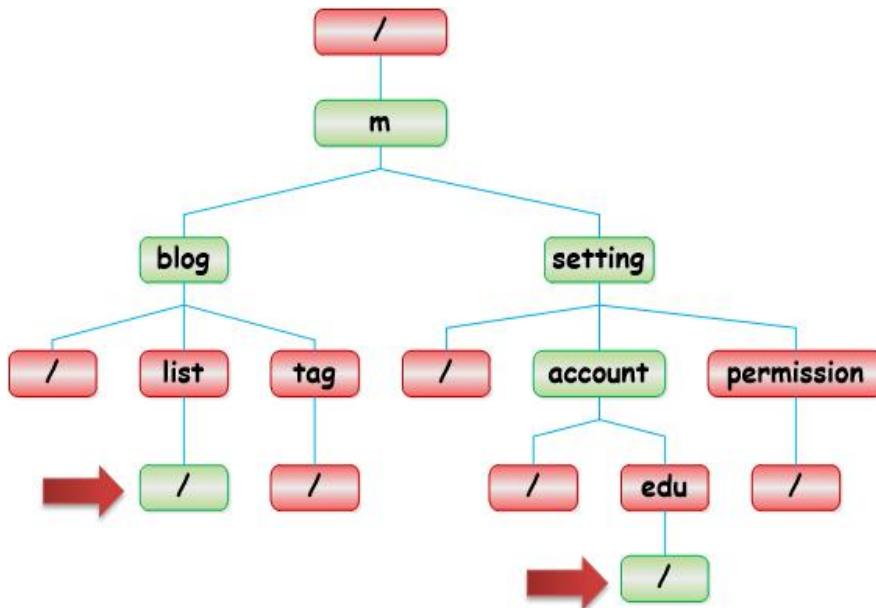
当用户请求显示一个模块时各模块会遵循以下步骤进行调度，假设请求显示 /m/blog/list/ 模块：



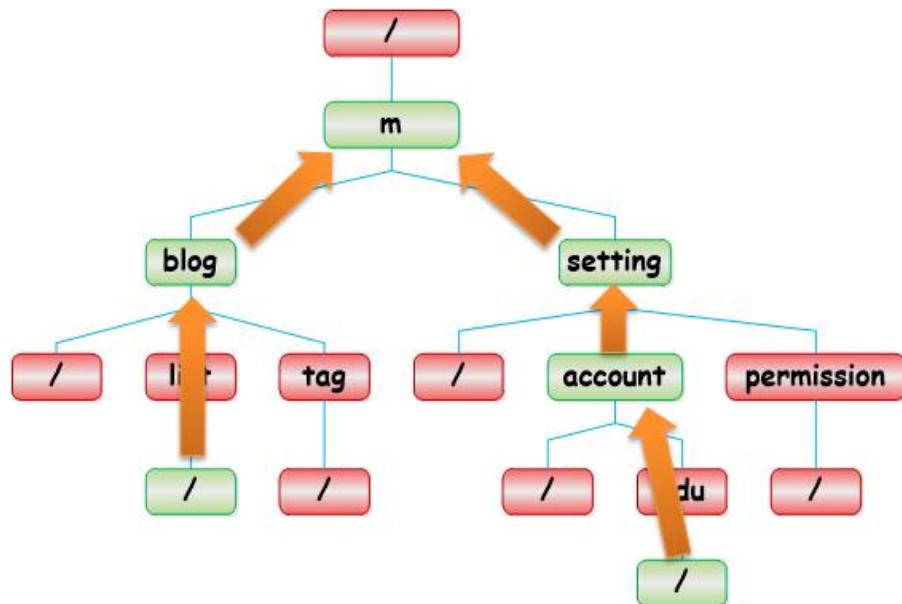
1. 检查目标节点到根节点路径上注册的模块，如果注册的是模块的实现文件地址，则请求载入模块实现文件。
2. 如果节点所在的模块的所有祖先节点已显示，则当前模块可被显示出来，否则等待祖先模块的显示调度。
3. 模块载入后根据第二步骤原则尝试调度目标模块的显示。

模块切换

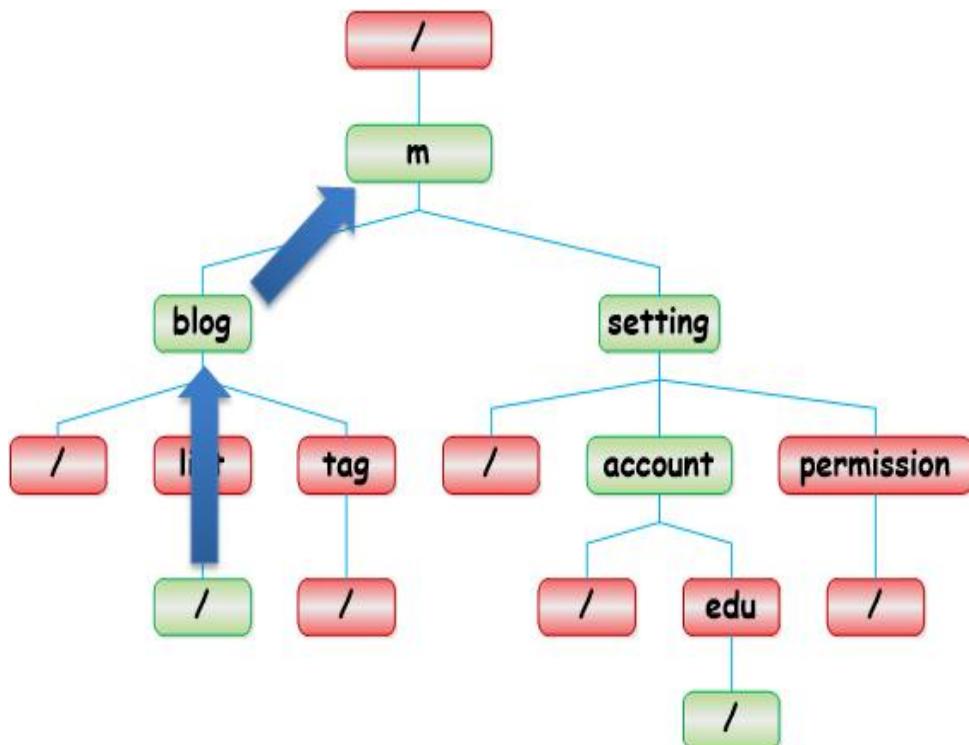
当用户从一个模块切换到另外一个模块时各模块遵循以下步骤调度，假设从 /m/blog/list/ 切换到 /m/setting/account/edu/ 模块：



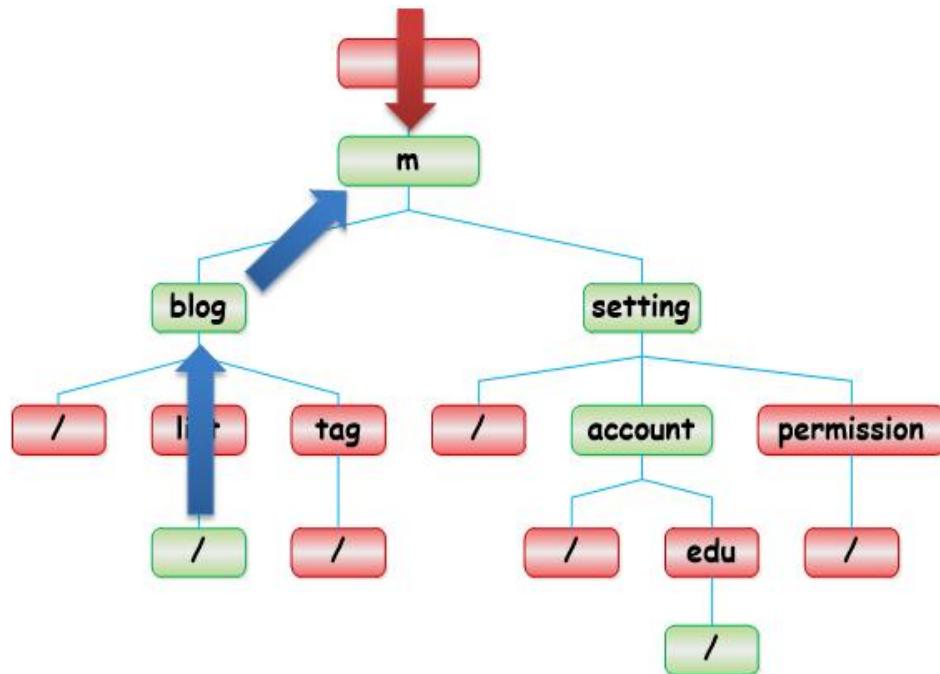
1. 查找源模块与目标模块的公共父节点。



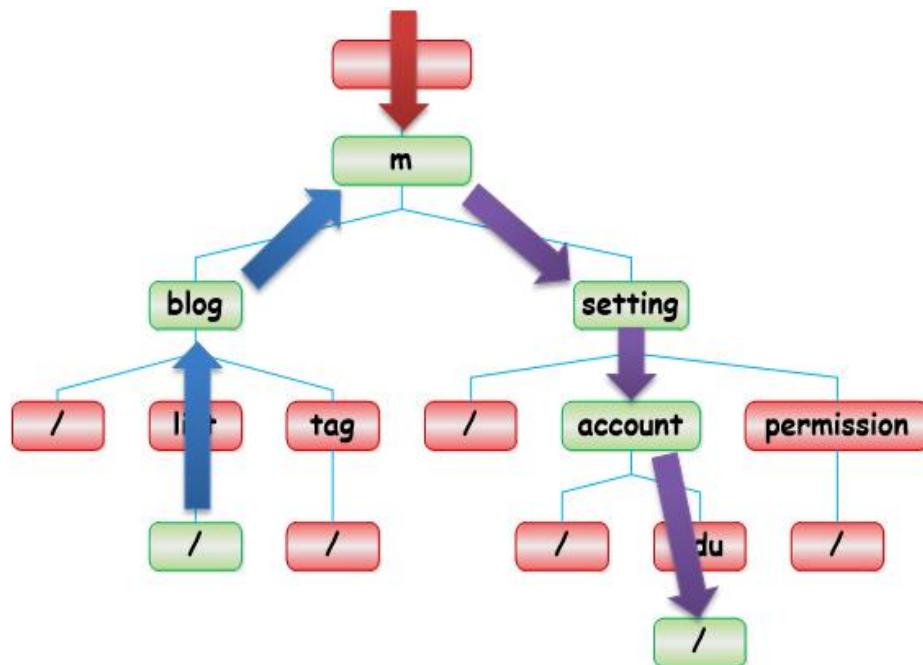
2. 从源节点到公共节点之间的模块调度隐藏操作。



3. 从根节点到公共节点之间的模块调度刷新操作。



4. 从公共节点到目标节点之间的模块调度显示操作。



消息通道

大部分时候我们不建议使用模块之前的消息通信，实践中也存在一些特殊情况会需要模块之前的消息通信，这里提供两种方式的消息通讯：

- 点对点的消息：一个模块发送消息时明确指定目标模块的 UMI。
- 观察订阅消息：一个模块可以对外申明发布了什么样的消息，有需要的模块可以订阅该模块 UMI 上的消息。

上面介绍了模块可伸缩性的一些原理。在本系列的最后一篇文章中，我们将以网易的 NEJ 框架为例，对上述原则进行说明。敬请期待！

本作品采用[知识共享署名 4.0 国际许可协议](#)进行许可。

感谢[张云龙](#)对本文的审校。

查看原文：[构建高可伸缩性的 WEB 交互式系统（中）](#)

构建高可伸缩性的 WEB 交互式系统（下）

作者 蔡剑飞

本文是《构建高可伸缩性的 WEB 交互式系统》系列文章的第三篇，以网易的 NEJ 框架为例，对模块的可伸缩性进行分析介绍。

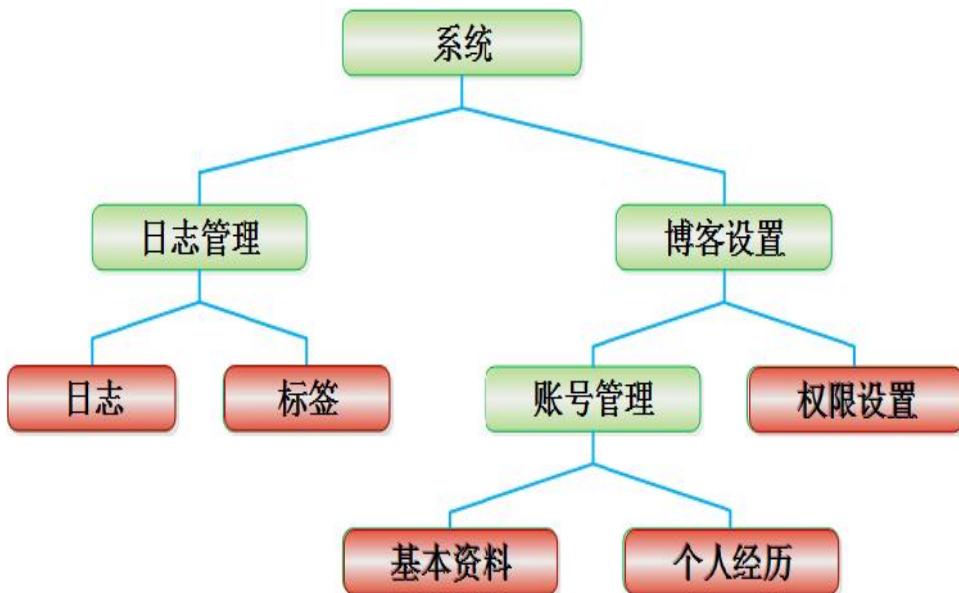
实例分析

[NEJ 框架](#)根据前两篇的描述对此套架构模式做了实现，下面我们用具体实例讲解如何使用 NEJ 中的模块调度系统来拆分一个复杂系统、开发测试模块、整合系统等。

系统分解

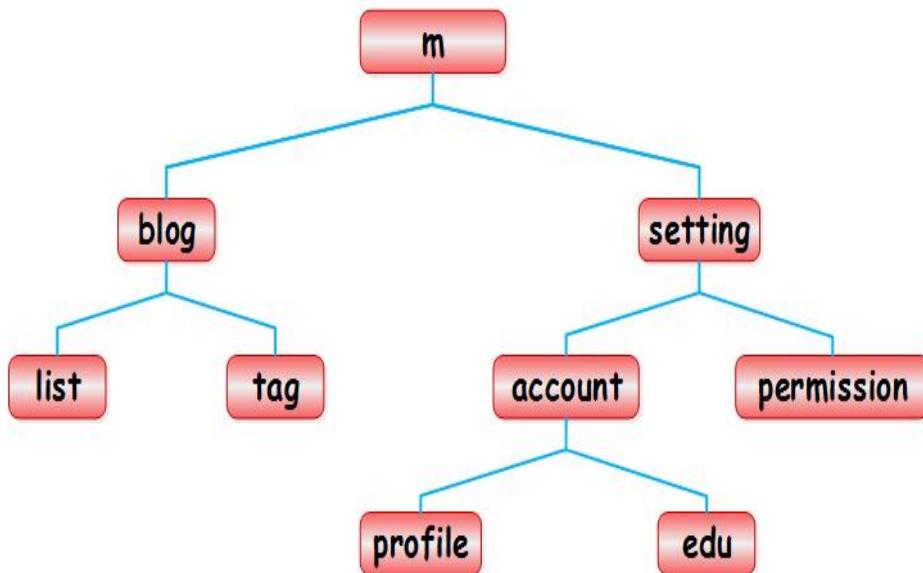
绘制层级关系图

当我们拿到一个复杂系统时，根据交互稿可以绘制出组成系统的模块的层级关系图，并确定系统对外可访问的模块。



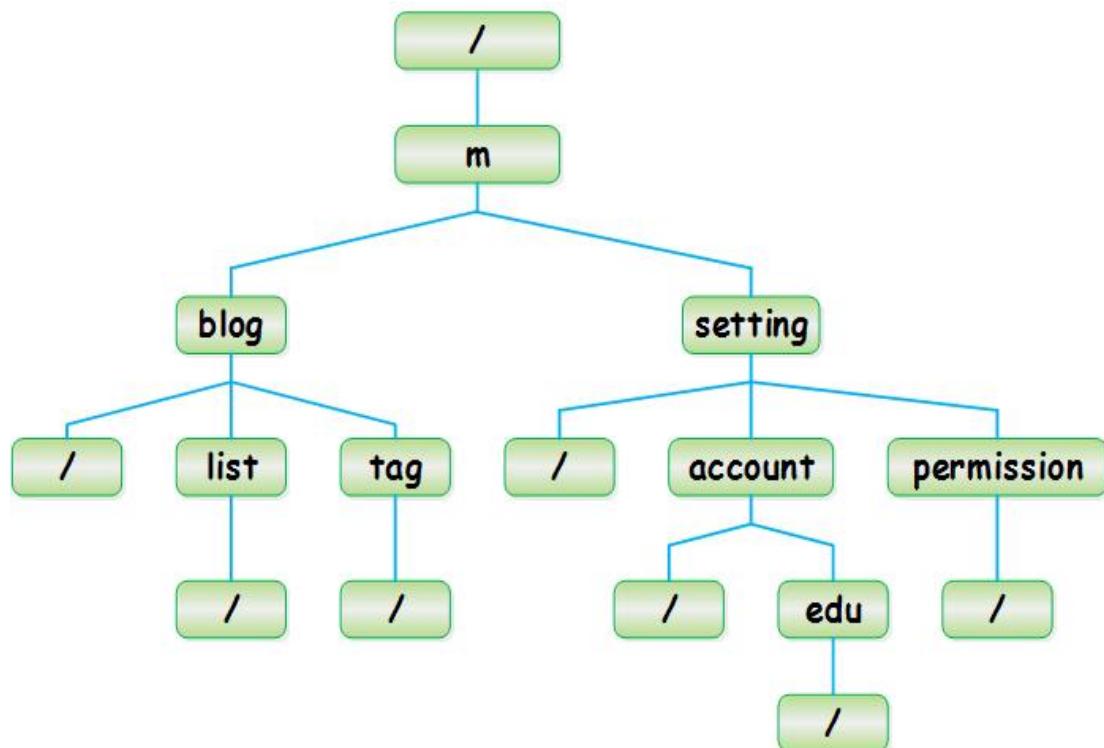
抽象依赖关系树

从模块的层级关系图中，我们可以非常方便的抽象出模块的依赖关系树：



然后，我们将抽象出来的依赖关系树根据 UMI 规则进行格式化。格式化的主要操作包括：

- 增加一个名称为“/”的根结点（也可将“m”结点改为“/”）
- 每个结点增加“/”的子节点作为默认节点

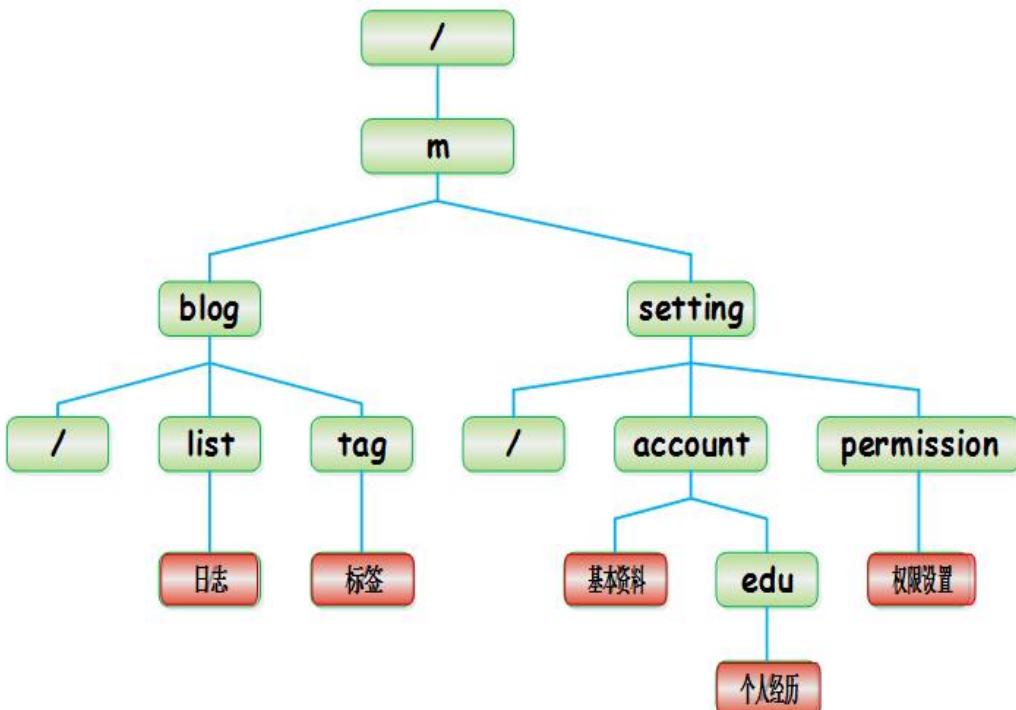


至此输出的依赖关系树，具有以下特性：

- 任何一个结点（除根结点外）到根结点路径上的结点名称用“/”分隔组合起来即为结点的 UMI 值，如 list 结点的 UMI 值为 /m/blog/list。
- 任何结点上的模块都依赖于他祖先结点（注册有模块）上的模块存在，如 blog 结点和 list 结点均注册有模块，则 list 结点上的模块显示必须以 blog 结点上的模块的显示为先决条件。

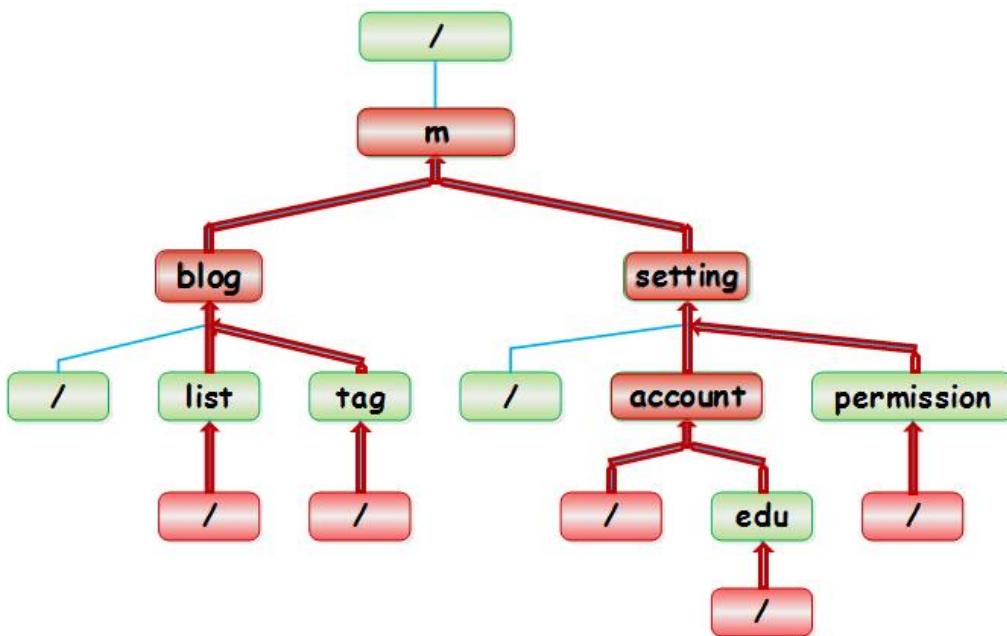
确定对外模块注册节点

五个对外可访问的模块：日志、标签、基本资料、个人经历、权限设置。在依赖关系树中找到合适的结点（叶子结点，层级关系树在依赖关系树中对应的结点或“/”结点）来注册对外可访问的模块：



确定布局模块注册节点

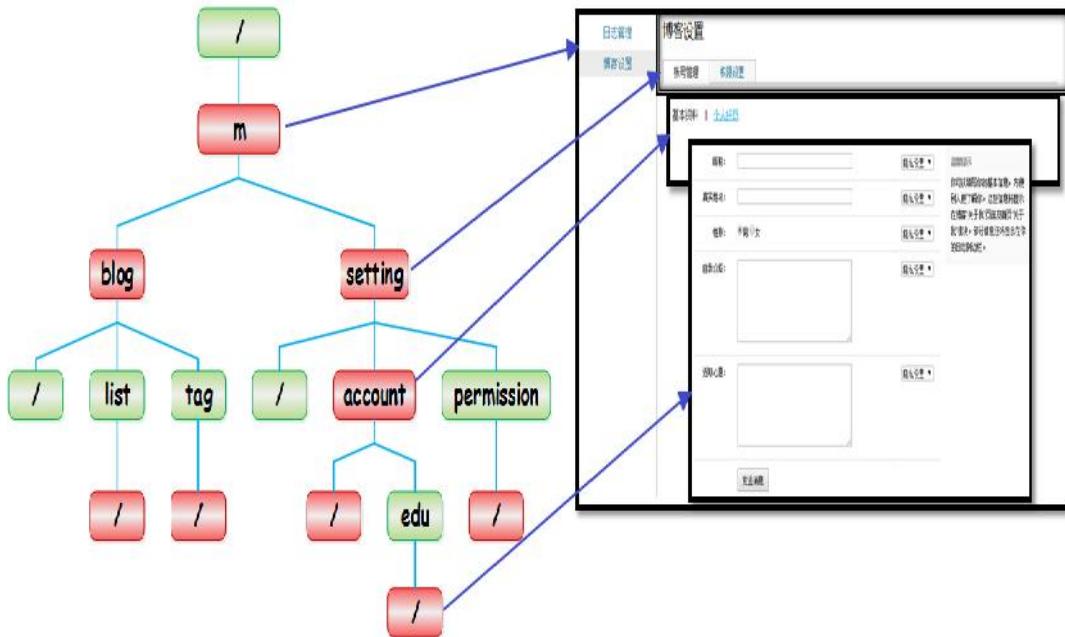
从可访问模块注册的结点往根结点遍历，凡碰到两模块交叉的结点即为布局模块注册结点，系统所需的组件相关的模块可注册到根结点，这样任何模块使用的时候都可以保证这些组件已经被载入。



映射模块功能

原则：结点的公共父结点实现结点上注册的模块的公共功能。

举例：blog 结点和 setting 结点的公共父结点为 m 结点，则我们可以通过切换 blog 模块和 setting 模块识别不变的功能即为 m 模块实现的功能，同理其他模块。

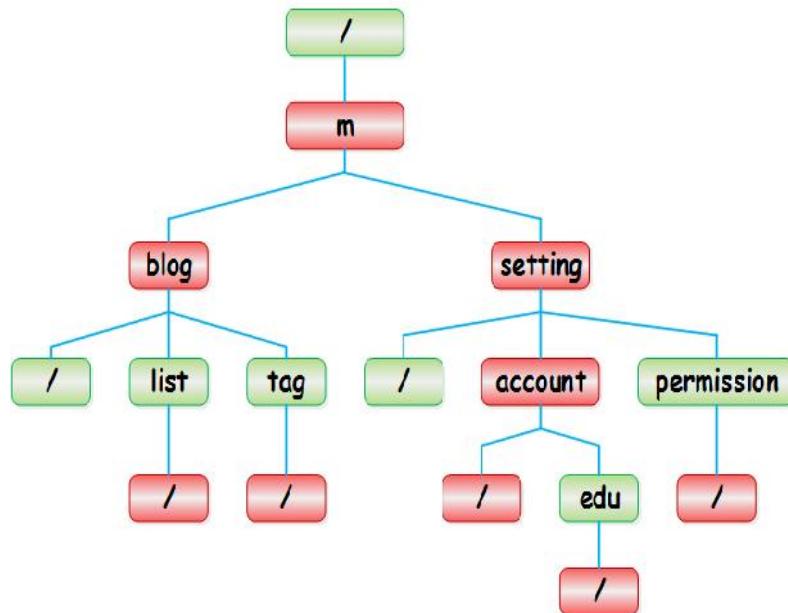


分解复杂模块

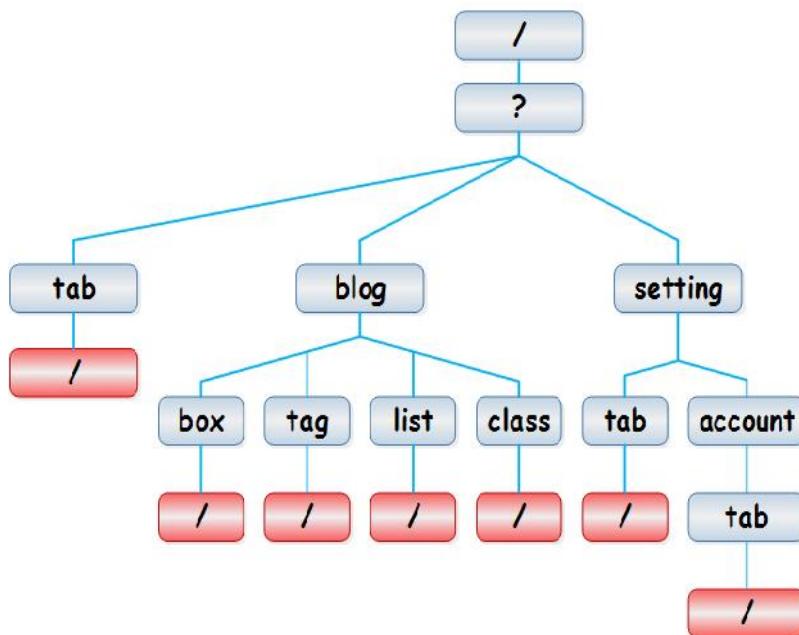
进一步分解复杂模块，一般需要分解的模块包括：

- 可共用模块，比如日志列表，可以在日志管理页面呈现，也可以在弹层中显示。
- 逻辑上无必然联系的模块，如日志模块中日志列表与右侧的按标签查看的标签列表之间没有必然的联系，任何一个模块的移除或添加都不会影响到另外一个模块的业务逻辑。

至此我们可以得到两棵系统分解后的依赖关系树——对外模块依赖关系树：



以及私有模块依赖关系树：



绘制模块功能规范表

本例中为了说明分解过程，将所有可分解的模块都做了分解。实际项目看具体情况，比如这里的/m 模块组合的/?/tab/模块的功能可以直接在/m 模块中实现，而不需要新建一个/?/tab/模块来实现这个功能。

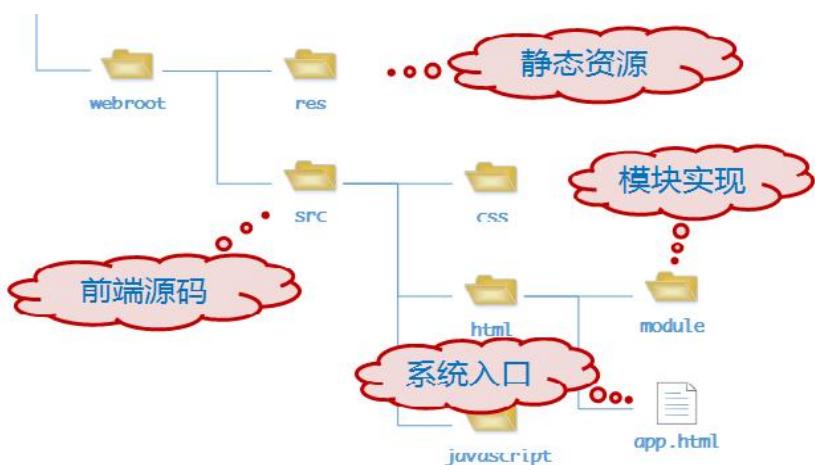
规范表范例如下所示：

名称	描述
模块UMI	/m
实现文件	module/layout/system/index.html
模块构造	wd.m.l_ssModuleLayoutSystem
模块功能	<ol style="list-style-type: none">提供左右布局容器日志管理/博客设置切换选中状态同步 (组合/?/tab/)

构建目录

项目目录

项目目录的构建如下图所示：



各目录说明

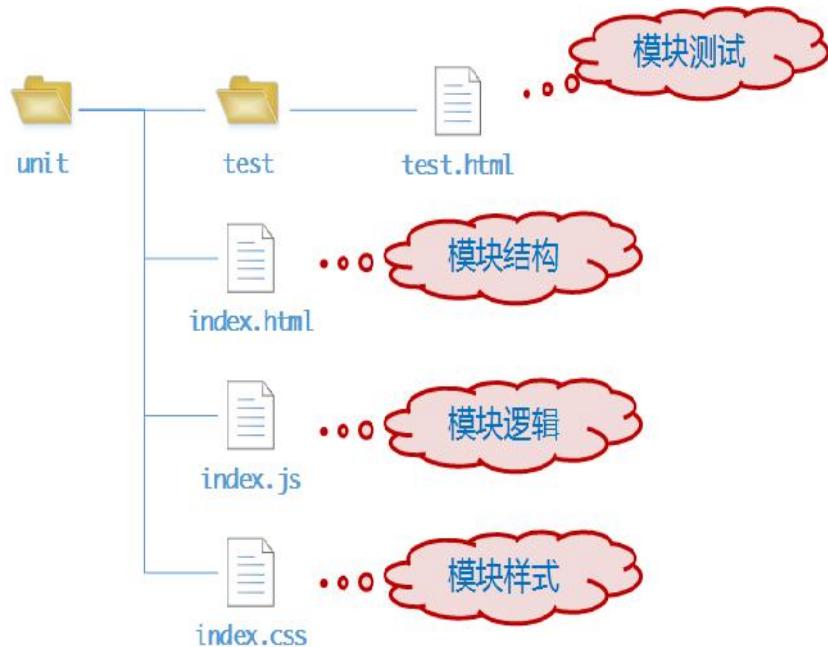
```
webroot          项目前端开发相关目录  
| - res          静态资源文件目录，打包时可配置使用到该目录下的静态资源带版本信息  
| - src          前端源码目录，最终发布时该目录不会部署到线上  
    | - html  
        | - module    单页面模块目录，系统所有模块的实现均在此目录下  
        | - app.html   单页面入口文件
```

模块单元目录

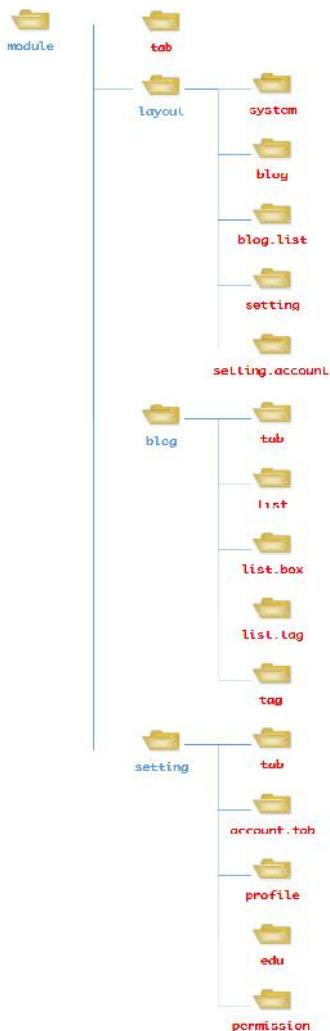
根据模块封装规则一个模块单元由以下几部分组成：

- 模块测试：模块实现的功能可以通过模块测试页面独立进行测试。
- 模块结构：模块所涉及的结构分解出来的若干模板集合。
- 模块逻辑：根据模块规范实现的模块业务逻辑，从模块基类继承。
- 模块样式：模块特有的样式，一般情况下这部分样式可以直接在 css 目录下实现。

结构范例如下所示：



至此我们可以得到所有模块的目录结构，如下所示：



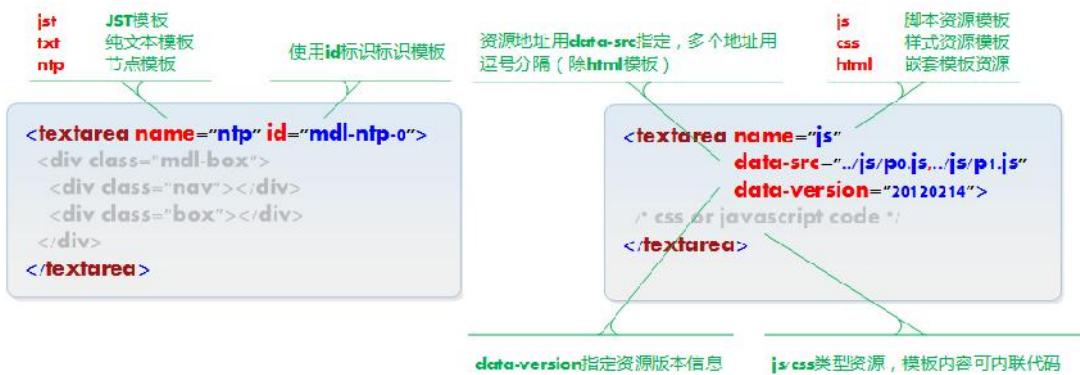
模块实现

结构

这里我们假设系统的静态页面已经做完，这里的模块实现只是在原有结构的基础上进行结构分解和业务逻辑的实现，结构部分内容主要将模块相关的静态结构拆分成若干 NEJ 的模板。注意：

- 模板中的外联资源如 css, js 文件地址如果使用的是相对路径则均相对于模块的 html 文件路径。
- 模板集合中的外联资源必须使用@TEMPLATE 标记标识，这个在后面打包发布章节会详细介绍。

NEJ 模板说明



模块结构举例

```

<meta charset="utf-8"/>

<textarea name="txt" id="m-ifrm-module">
  <div class="n-login">
    <div class="iner j-flag">
      <span class="cls j-flag"></span>
      <span class="min j-flag">--</span>
    </div>
    <div class="cnt j-cnt"></div>
  </div>
</textarea>

<!-- @TEMPLATE -->
<textarea name="js" data-src=".index.css"></textarea>
<textarea name="js" data-src=".index.js"></textarea>
<!-- /@TEMPLATE -->

```

逻辑

依赖 util/dispatcher/module 模块，我们从`$_$ModuleAbstract` 扩展一个项目的模块基类，完成项目中模块特有属性、行为的抽象。

```

/*
 * -----
 * 项目模块基类实现文件
 * @version 1.0
 * @author genify(caijf@corp.netease.com)
 * -----
 */
NEJ.define([
  'base/klass',
  'util/dispatcher/module'
],function(_k,_t,_p){
  // variable declaration
  var _pro;
  /**

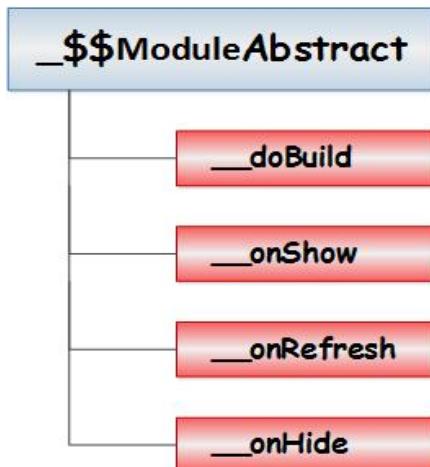
```

```
* 项目模块基类对象
* @class  {_$$Module}
* @extends {_$$ModuleAbstract}
* @param  {Object} 可选配置参数，已处理参数列表如下所示
*/
_p._$$Module = _k._$klass();
_pro = _p._$$Module._$extend(_t._$$ModuleAbstract);
/** 
 * 操作
* @param  {Object}
* @return {Void}
*/
_pro.__doSomething = function(_args){
    // TODO
};

// TODO

return _p;
});
```

根据模块状态的划分，我们在实现一个模块时需要实现以下几个接口：



各阶段对应的接口：

- 构建 `__doBuild`: 构建模块结构，缓存模块需要使用的节点，初始化组合控件的配置参数。
- 显示 `__onShow`: 将模块放置到指定的容器中，分配组合控件，添加相关事件，执行 `__onRefresh` 的业务逻辑。
- 刷新 `__onRefresh`: 根据外界输入的参数信息获取数据并展示（这里主要做数据处理）。
- 隐藏 `__onHide`: 模块放至内存中，回收在 `__onShow` 中分配的组合控件和添加的事件，回收 `__onRefresh` 中产生的视图（这里尽量保证执行完成后恢复到 `__doBuild` 后的状态）。

具体模块实现举例：

```
/*
 * -----
 * 项目模块实现文件
 * @version 1.0
 * @author genify(caijf@corp.netease.com)
 * -----
 */
NEJ.define([
    'base/klass',
    'util/dispatcher/module',
    '/path/to/project/module.js'
],function(_k,_e,_t,_p){
    // variable declaration
    var _pro;
    /**
     * 项目模块对象
     * @class {$_$ModuleDemo}
     * @extends {$_$Module}
     * @param {Object} 可选配置参数
     */
    _p._$ModuleDemo = _k._$klass();
    _pro = _p._$ModuleDemo._$extend(_t._$Module);
    /**
     * 构建模块，主要处理以下业务逻辑
     * - 构建模块结构
     * - 缓存后续需要使用的节点
     * - 初始化需要使用的组件的配置信息
     * @return {Void}
     */
    _pro.__doBuild = function(){
        this.__super();
        // TODO
    };
    /**
     * 显示模块，主要处理以下业务逻辑
     * - 添加事件
     * - 分配组件
     * - 处理输入信息
     * @param {Object} 输入参数
     * @return {Void}
     */
    _pro.__onShow = function(_options){
        this.__super(_options);
        // TODO
    };
    /**
     * 刷新模块，主要处理以下业务逻辑
     * - 分配组件，分配之前需验证
     * - 处理输入信息
     * - 同步状态
     * - 载入数据
     * @return {Void}
     */
    _pro.__onRefresh = function(_options){
        this.__super(_options);
    }
});
```

```
// TODO
};

/**
 * 隐藏模块，主要处理以下业务逻辑
 * - 回收事件
 * - 回收组件
 * - 尽量保证恢复到构建时的状态
 * @return {Void}
 */
_pro.__onHide = function(){
    this.__super();
    // TODO
};

// notify dispatcher
_e.$regist(
    'umi_or_alias',
    _p.$ModuleDemo
);

return _p;
});
```

消息

点对点消息

模块可以通过`__doSendMessage`接口向指定 UMI 的模块发送消息，也可以通过实现`__onMessage`接口来接收其他模块发给他的消息。

发送消息

```
_pro.__doSomething = function(){

    // TODO

    this.__doSendMessage(
        '/m/setting/account/',
        {
            a:'aaaaaa',
            b:'bbbbbbbbbb'
        }
    );
};
```

接收消息

```
_pro.__onMessage = function(_event){
    // _event.from 消息来源
    // _event.data 消息数据，这里可能是 {a:'aaaaaa',b:'bbbbbbbbbb'}
    // TODO
};
```

发布订阅消息

发布消息

```
_pro.__doSomething = function(){
    // TODO

    this.__doPublishMessage(
        'onok',{
            a:'aaaaaaaa',
            b:'bbbbbbbb'
        }
    );
};
```

订阅消息

```
_pro.__doBuild = function(){
    // TODO

    this.__doSubscribeMessage(
        '/m/message/account/','onok',
        this.__onMessageReceive._$bind(this)
    );
};
```

自测

创建 html 页面，使用模板引入模块实现文件：

```
<!-- template box -->
<div id="template-box" style="display:none;">
    <textarea name="html" data-src="../index.html"></textarea>
</div>
```

模块放至 document.mbody 指定的容器中：

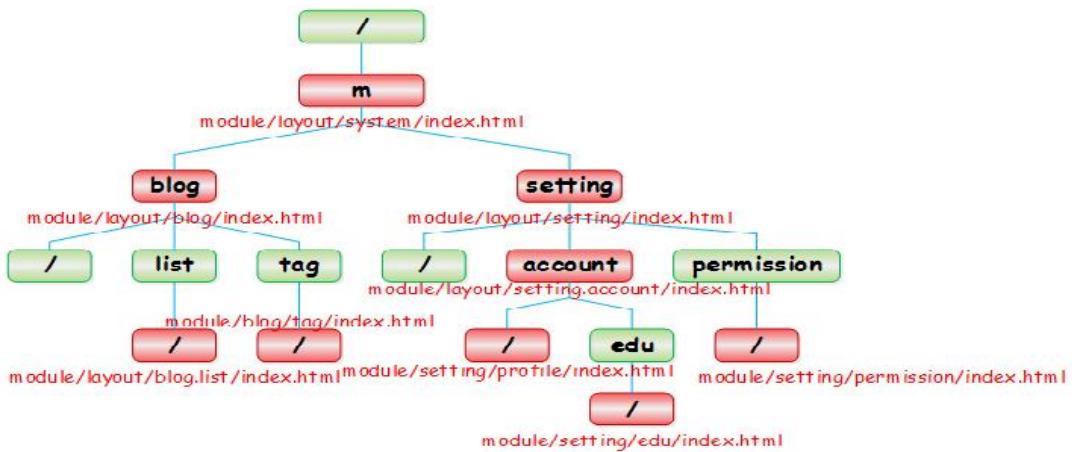
```
NEJ.define([
    'util/dispatcher/test'
],function(_e){
    document.mbody = 'module-id-0';
    // test module
    _e._$testByTemplate('template-box');
});
```

系统整合

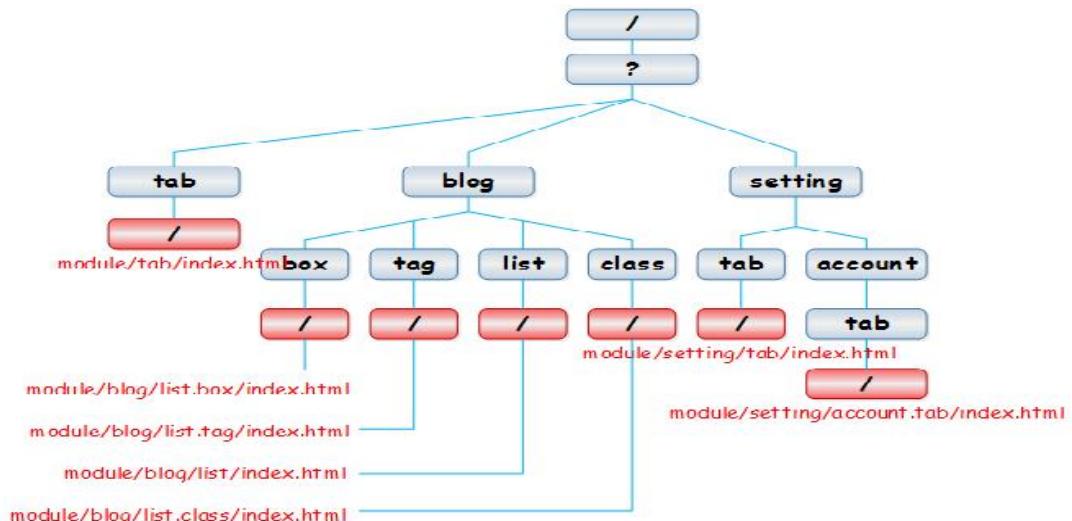
映射依赖关系树

系统整合时，我们只需要将依赖关系树中需要注册模块的节点同模块实现文件进行映射即可。

对外模块整合



私有模块整合



提取系统配置信息

规则配置举例

```

rules: {
    rewrite: {
        '404': '/m/blog/list/',
        '/m/blog/list/': '/m/blog/',
        '/m/setting/account/': '/m/setting/'
    },
    title: {
        '/m/blog/tag/': '日志标签',
    }
}
    
```

```

    '/m/blog/list/':'日志列表',
    '/m/setting/permission/':'权限管理',
    '/m/setting/account/':'基本资料',
    '/m/setting/account/edu/':'教育经历'
},
alias:{  

    'system-tab':'/?/tab/',
    'blog-tab':'/?/blog/tab/',
    'blog-list-box':'/?/blog/box/',
    'blog-list-tag':'/?/blog/tag/',
    'blog-list-class':'/?/blog/class/',
    'blog-list':'/?/blog/list/',
    'setting-tab':'/?/setting/tab/',
    'setting-account-tab':'/?/setting/account/tab/',

    'layout-system':'/m',
    'layout-blog':'/m/blog',
    'layout-blog-list':'/m/blog/list/',
    'layout-setting':'/m/setting',
    'layout-setting-account':'/m/setting/account',  

    'blog-tag':'/m/blog/tag/',
    'setting-edu':'/m/setting/account/edu/',
    'setting-profile':'/m/setting/account/',
    'setting-permission':'/m/setting/permission/'
}
}

```

模块配置举例

```

modules:{  

    '/?/tab/':'module/tab/index.html',
    '/?/blog/tab/':'module/blog/tab/index.html',
    '/?/blog/box/':'module/blog/list.box/index.html',
    '/?/blog/tag/':'module/blog/list.tag/index.html',
    '/?/blog/class/':'module/blog/list.class/index.html',
    '/?/blog/list/':'module/blog/list/index.html',
    '/?/setting/tab/':'module/setting/tab/index.html',
    '/?/setting/account/tab/':'module/setting/account.tab/index.html',  

    '/m':{  

        module:'module/layout/system/index.html',
        composite:{  

            tab:'/?/tab/'  

        }
    },
    '/m/blog':{  

        module:'module/layout/blog/index.html',
        composite:{  

            tab:'/?/blog/tab/'  

        }
    },
    '/m/blog/list':{  

        module:'module/layout/blog.list/index.html',
        composite:{  

            box:'/?/blog/box/',
            tag:'/?/blog/tag/',
            list:'/?/blog/list/',
        }
    }
}

```

```

        clazz:'/?/blog/class/'
    }
},
'/m/blog/tag/':'module/blog/tag/index.html',

'/m/setting':{
    module:'module/layout/setting/index.html',
    composite:{
        tab:'/?/setting/tab/'
    }
},
'/m/setting/account':{
    module:'module/layout/setting.account/index.html',
    composite:{
        tab:'/?/setting/account/tab/'
    }
},
'/m/setting/account/':'module/setting/profile/index.html',
'/m/setting/account/edu/':'module/setting/edu/index.html',
'/m/setting/permission/':'module/setting/permission/index.html'
}
}

```

模块组合

模块通过`_export`属性开放组合模块的容器，`_export`中的`parent`为子模块的容器节点，顶层模块（如“/m”）可以通过重写`_doParseParent`来明确指定应用所在容器。

```

_pro._doBuild = function(){
    this._body = _e._$html2node(
        _e._$getTemplate('module-id-12')
    );
    // 0 - box select
    // 1 - class list box
    // 2 - tag list box
    // 3 - sub module box
    var _list = _e._$getElementsByClassName(this._body,'j-flag');
    this._export = {
        box:_list[0],
        clazz:_list[1],
        tag:_list[2],
        list:_list[3],
        parent:_list[3]
    };
};
}

```

通过`composite`配置模块组合：

```

'/m/blog/list/':{
    module:'module/layout/blog.list/index.html',
    composite:{
        box:'/?/blog/box/',
        tag:'/?/blog/tag/',
        list:'/?/blog/list/',
        clazz:'/?/blog/class/'
    }
}

```

模块组合时可以指定组合模块的处理状态:

- **onshow** : 这里配置的组合模块仅在模块显示时组合, 后续的模块 refresh 操作不会导致组合模块的 refresh, 适合于模块在显示后不会随外界输入变化而变化的模块。
- **onrefresh** : 这里配置的模块在模块显示时组合, 后续如果模块 refresh 时也会跟随做 refresh 操作, 适用于组合的模块需要与外部输入同步的模块。
- 不指定 onshow 或者 onrefresh 的模块等价于 onrefresh 配置的模块。

```
composite:{  
    onshow:{  
        // 模块 onshow 时组合  
        // 组合的模块在模块 onrefresh 时不会刷新  
    },  
    onrefresh{  
        // 模块 onshow 时组合  
        // 组合的模块在模块 onrefresh 时也同时会刷新  
    }  
    // 这里配置的组合模块等价于 onrefresh 中配置的模块  
}
```

启动应用

根据配置启动应用

```
NEJ.define([
    'util dispatcher dispatcher'
], function(_e){
    _e._$startup({
        // 规则配置
        rules:{
            rewrite:{  
                // 重写规则配置
            },  
            title:{  
                // 标题配置
            },  
            alias:{  
                // 别名配置
                // 建议模块实现文件中的注册采用这里配置的别名
            }
        },
        // 模块配置
        modules:{
            // 模块 UMI 对应实现文件的映射表
            // 同时完成模块的组合
        }
    });
});
```

打包发布

打包发布内容详见 [NEJ 工具集](#) 相关文档。

系统变更

当系统需求变化而进行模块变更我们只需要开发新的模块或删除模块配置即可。

新增模块

如果增加一个全新的模块，则只需按照上面的逻辑实现步骤开发一个模块即可。如果新增的模块功能在系统中已经实现，则只需修改配置即可。如上例中我们需要在将日志管理下的标签模块在博客设置中也加一份，访问路径为/m/setting/tag/。

The screenshot shows a user interface for 'Blog Settings'. At the top, there are tabs: 'Blog Settings' (selected), 'Log Management', 'Access Control', and 'Log Tag'. Below the tabs is a table with two columns of tags and their counts. A tooltip on the right provides information about tags.

Tag	Count
www (0)	修改
问号 (1)	修改
firefox (1)	修改
sprite (1)	修改
window (1)	修改
textarea (1)	修改
插件 (1)	修改
页面 (1)	修改
cmt (1)	修改
html5 (1)	修改
历史 (2)	修改
ie (2)	修改
manager (2)	修改
css (3)	修改
iframe (4)	修改
浏览器 (6)	修改
opera (1)	修改
stylesheet (1)	修改
html (1)	修改
image (1)	修改
name (1)	修改
onpropertychange (1)	修改
document (1)	修改
嵌入 (1)	修改
数据 (1)	修改
dispatcher (1)	修改
a (2)	修改
管理器 (2)	修改
template (2)	修改
模板 (3)	修改
module (4)	修改
javascript (10)	修改
空格 (1)	修改
笔记 (1)	修改
优化 (1)	修改
chrome (1)	修改
abord (1)	修改
fireevent (1)	修改
样式 (1)	修改
渲染 (1)	修改
交互 (1)	修改
模次调度 (1)	修改
href (2)	修改
hash (2)	修改
模次化 (2)	修改
模次 (3)	修改
bug (5)	修改

温馨提示
标签是由用户定义的、概括文章内容的关键词，比日志分类更准确、更具体。经常使用标签，可以让读者更加方便地找到感兴趣的文章。

修改规则配置：

```
rules: {
  // ...
  alias: {
    // ...
    'blog-tag': ['/m/blog/tag/', '/m/setting/tag/']
  }
}
```

修改模块配置：

```
modules: {
```

```
// ...
'/m/setting/tag/':'module/blog/tag/index.html'
}
```

如果要在?/setting/tab 模块的结构模板中增加一个标签即可。

```
<textarea name="txt" id="module-id-8">
<div class="ma-t w-tab f-cb">
    <a class="itm fl" href="#/setting/account/" data-id="/setting/account/">账号管理</a>
    <a class="itm fl" href="#/setting/permission/" data-id="/setting/permission/">权限设置</a>
    <a class="itm fl" href="#/setting/tag/" data-id="/setting/tag/">日志标签</a>
</div>
</textarea>
```

删除模块

将退化的模块从系统中删除只需要将模块对应的 UMI 配置从模块配置中删除即可，而无需修改具体业务逻辑。

总结

随着 WEB 技术的快速发展，单页面系统（SPA）的应用变得越来越广泛，随着此类系统复杂度的增加，其对平台及模块的伸缩性方面需求变得越来越重要。对于这两方面，业界给出了不少解决方案，本文我们主要探讨了网易 NEJ 框架在这些方面给出的解决方案。网易在单页面系统方面也做了多年的实践和技术积累，如近几年的[网易云音乐 PC 版](#)、[易信 WebIM](#)、[网易邮箱助手](#)等，早些年的[网易相册](#)、[网易邮箱](#)等，移动端的[网易云相册 IPad 版](#)、[Lofter Android 版](#)等产品，均是此类单页面系统的应用实践。实践过程中对这方面有兴趣的同学可进一步做交流。

本作品采用[知识共享署名 4.0 国际许可协议](#)进行许可。

感谢[张云龙](#)对本文的审校。

查看原文：[构建高可伸缩性的 WEB 交互式系统（下）](#)

看板如何奏效

作者 Amr Noaman Abdel-Hamid，译者 李清玉

最近，越来越多的人开始对看板产生兴趣，因为它是管理软件开发和持续改进的简单有效的方法。但是，看板如何（或者说为什么）奏效？是因为它暴露了系统，并对需求的可视化的追踪吗？还是因为限制了在线制品（work in progress）的数量，并且减少了浪费在任务切换上所消耗的精力？或许是因为通过简单的测量，例如周期时间（cycle time）和产能（Throughput）给经理们提供了频繁和有力度的反馈？本文依据排队理论和利特尔法则¹（Little's Law¹）深入研究看板的细节。并且通过案例分析，我们会阐述看板开发系统中，经理们所面临的三个典型的问题，并提出如何解决这些问题的方法。对于看板如何奏效，本文会揭露一些基本的概念和深入的见解。

软件系统中的利特尔法则

利特尔法则（根据 John Little 命名）是看板方法所建立的基础思想之一。在软件开发中，利特尔法则是这样描述的：

$$\text{在线制品数量 (WIP)} = \text{产能 (Th)} * \text{周期时间 (CT)}$$

在线制品数量（Work in Process）=在开发系统中，未完成条目的平均数量（缺陷，用户故事，变更请求，等等）。

产能 (Th) = 团队在单位时间内的产出

周期时间 (CT) = 团队完成一个条目所花费的平均时间

利特尔法则的动态性是令人惊奇的。它解释了软件开发的许多复杂性并激发我们做出决定。为了分析下一个案例的动态性，我们使用效应图²（Diagrams of Effects），它是一款非常好的工具，可以分析非线性系统，超过两个效应的系统或者分析影响系统行为。

案例 1：提高团队产能

Adam 是一个团队的教练，这个团队有 2 个开发和 1 个测试，负责维护公司大量的产品。在 2013 年，公司产品的市场宣传上提高了投入，并成功地将客户数目提高了一倍。现在，Adam 的团队收到越来越多的支持请求。然而，CEO 却不愿扩大团队的规模。

在这个例子中，为了满足客户需求的增长，团队必须提高产能。根据利特尔法则 ($Th = WIP / CT$)，为了提高团队产能，需要减少周期时间或者增加在线制品数量。因为团队的能力是固定的，不可能减少周期时间。所以，简单的解决方案就是增加在线制品数量。

可问题是：增加在线制品数量就真的能提高产能吗？答案是否定的。通过增加更多的在线制品数量提高的产能有一个极限，产能在到达极限之后会开始下降，如下图所示：

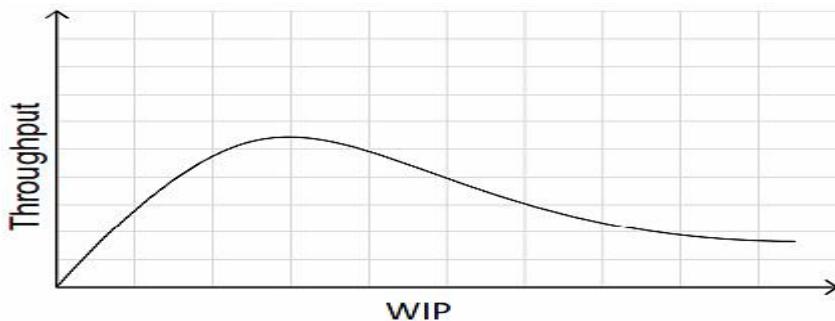


图 1 产能与在线制品数量的关系图

如下图所描述的，增加在线制品数量会刺激团队优化他们的工作，从他们的交付流程中减少某些浪费（黄色区域），直到团队可能达到的最大产能（绿色峰值）。在此之后，更多的在线制品数量不会带来任何改进；相反，由于工作压力和任务切换则会降低团队的产能（红色区域）。

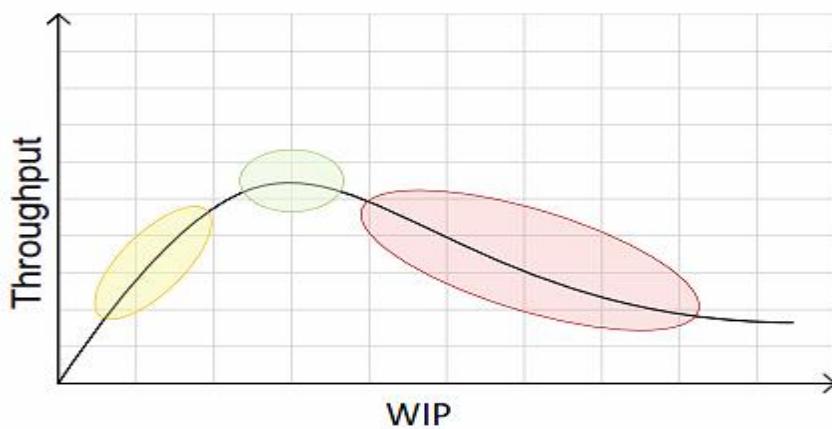


图 2 根据在线制品数量的总和，团队对在线制品数量增加的响应图

在红色区域，由于外部因素和团队内部出现的一些问题，团队招架不住，从而导致生产率降低。下面的效应图分析了团队在红色区域的动态变化。

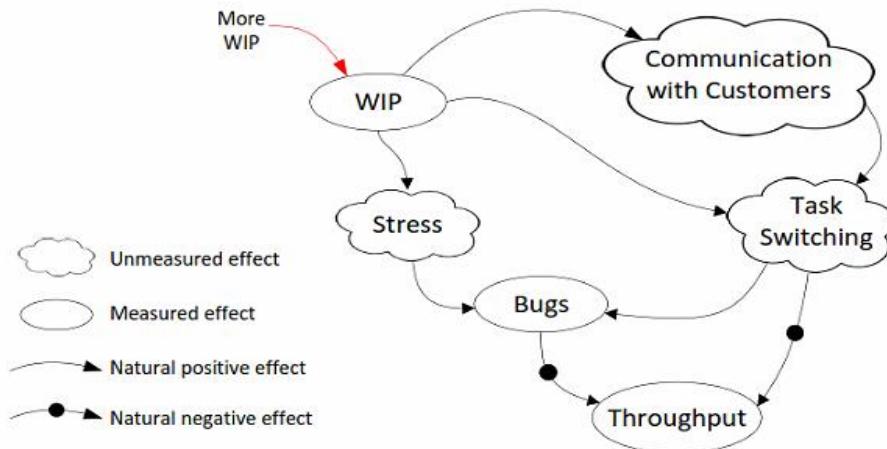


图 3 超越团队能力后增加在线制品数量会降低团队生产率和产能

该图显示了在超出团队能力之上，增加在线制品数量的所产生的效应。这会增加与客户的沟通增多，增加任务的切换并增大团队的压力。在压力和任务频繁切换下工作会导致更多的缺陷，最终会降低生产率，因此，相应的产能就会降低。

为了理解这一决定的影响，下面的模型图显示了加强效应：增加在线制品数量会引起效率降低，因而堆积的需求就会上升，从而导致在线制品数量的上升，以此类推。这个系统会持续循环，因此产能会持续下降，直至团队崩溃。

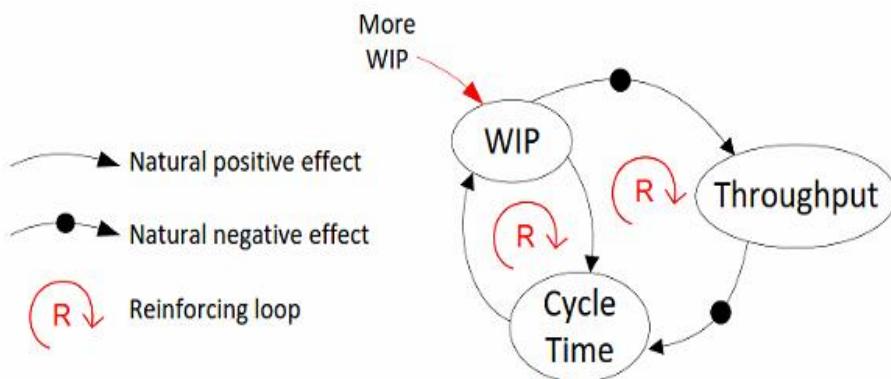


图 4 超越团队能力后增加在线制品数量会引起 2 个增强回路，在这种动态下，在线制品数量会持续增加，直至开发系统崩溃

说明：两个连续的负面效应=正面效应。

总结一下，如果你们团队的能力是固定的，想要增加产能，你可以选择同时增加团队规模以及在线制品数量。如果做不到这一点，那么你只有一个选择：降低周期时间，也就是发现和去除浪费。

案例 2：稳定周期时间

Ismail 是开发经理，负责在服务等级协议 SLA (Service Level Agreement) 里所规定的时间内为客户交付变更的内容。Ismail 和他的团队收到的客户请求数量是波动的。有时候比平时的多，导致周期时间超出了 SLA 规定的时间，另外一些时候需求比例比较低，不能占满团队的工作量。下图解释了为什么高输入率会增加周期时间。

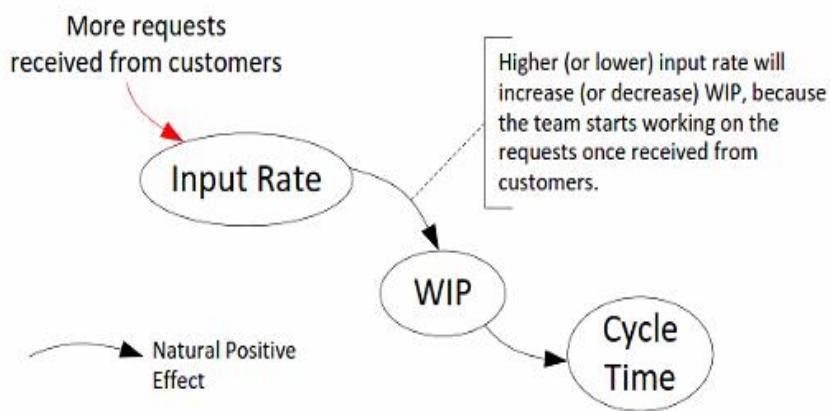


图 5 根据利特尔法则，高输入率导致周期时间延长。同样，低输入率导致周期时间缩短

为了稳定周期时间，利特尔法则表明周期时间与在线制品数量成正比，与产能成反比。所以，如果 Ismail 能够稳定这两个参数，周期时间才可以相应地稳定。

为了做到这一点，Ismail 既要控制在线制品数量，也要控制团队能力（团队成员的数量或者专注于处理客户请求的团队时间百分比），从而可以响应过高或过低的输入率。这两个参数会同时影响周期时间，增加在线制品数量会延长周期时间，然而，增加团队能力会减少周期时间。所以两者效应会相互抵消。

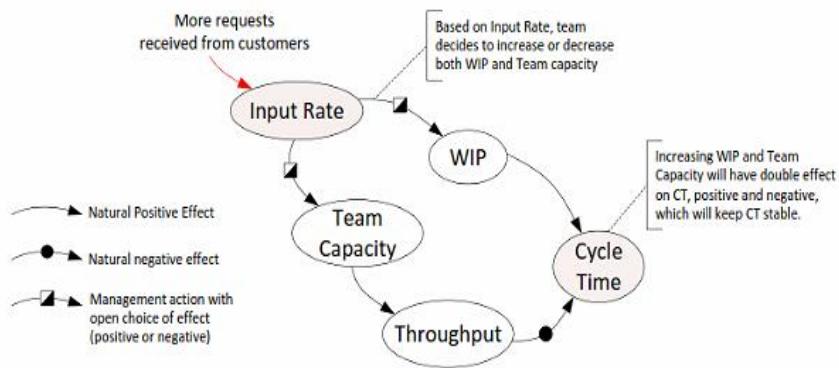


图 6 如果经理可以增加/减少在线制品数量和团队能力，他们就可以稳定周期时间和优化团队利用率与绩效

因此，总结如下，如果团队经历波动的输入率，他们需要控制两个参数，在线制品数量上限和团队能力。通过控制这两个参数，团队才可以稳定周期时间并优化团队利用率。

案例 3：不要太多的快速通道

快速通道的工作方式（在看板的白板中使用高优先级的通道）也许可以简单的解决重复的问题报告和不确定的服务要求，特别是针对不满意的客户。在很多情况下，为了缓解特殊的案例或者对抱怨强烈的客户做出回应，很可能会无原则地使用快速通道。

快速通道会消耗团队的部分时间和工作，会使开发速度减慢，从而增加平均周期时间。根据利特尔法则，它会显著地降低团队的产出。

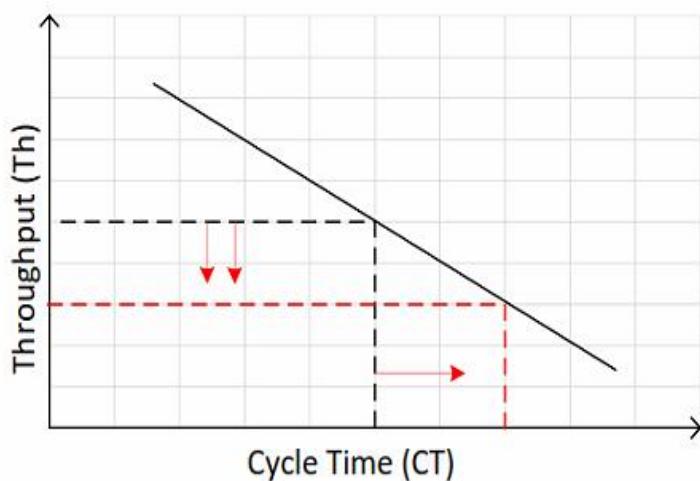


图 7 根据利特尔法则，如果在线制品数量固定，周期时间越长，产出就会越低

然而，实际发生的情况是，在周期时间和产出之间的线性关系也会发生变化，如下图所示：

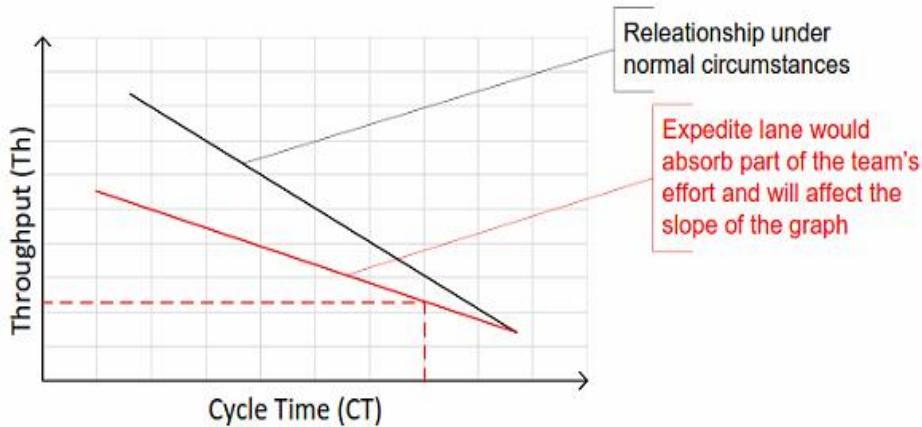


图 8 由于周期时间的延长和产出与周期时间的关系图的变换这两个因素，产出会极大地降低

在更多严重的例子中，团队可能会把任务切换到快速通道的任务上去，然后开始立即处理这个请求，并把正在进行中的其他任务搁置在一旁。应急式的上下文切换会对产出有更加严重的负面影响。下图解释了这个影响：

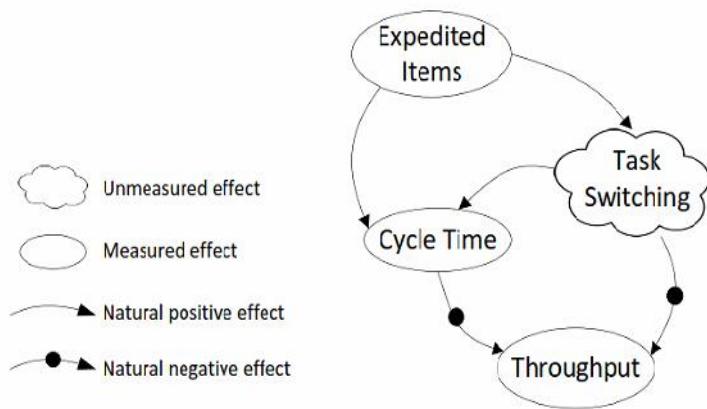


图 9 快速通道的任务延长了正在进行中的任务的周期时间，最终会负面影响产出

总结第三个例子，要意识到快速通道是一个陷阱，它可能会对整个团队的工作效率有负面影响，而且可能导致降低平均周期时间。虽然快速通道可能在一些规模较小的紧急案例中有所帮助，但也可能会给计划外的负面的动态变化打开一道门。

结论

因此，在这三个案例中，我们根据排队理论阐述了看板如何奏效。它是非常简单而且有效的管理工具。作为一个经理或者主管，你需要控制几个参数：在线制品数量和团队能力。并且，你有衡量指标，例如周期时间和产出，因此对于流程的有效性，你可以简单的衡量并快速的得到反馈。在后三个例子中，我们指出了使用看板时要注意的三个基本问题：

1. 试图研究团队能力，而不是压榨团队，让他们超出团队能力之上额外工作。绘制在线制品数量与产出的关系图可以让你知道团队可以承受的最大在线制品数量值。
2. 可以通过对多个参数的控制实现稳定团队开发进度。就像在第二个例子中，你可以控制在线制品数量和团队能力，从而达到稳定的周期时间。
3. 小心快速通道陷阱。实际上，他就是违反流程的一个后门。如果不小心使用，它会破坏团队的生产率。

• 利特尔法则：

在[一本书的章节中](#)解释了利特尔法则，该书由麻省理工学院发表，John Little 解释了这个法则并且把理论与实际结合起来。它是一本极好的书籍，非常简单并且深入到了利特尔法则的精髓。

本书的这一章节非常好的解释了一个问题，是利特尔法则在原始的公式中（把到达率（Arrival Rate）作为公式的一个参数）和把它应用到生产系统中（用产能代替了到达率（Arrival Rate））的区别。解释如下：

利特尔法则的陈述如下：

$$L = \lambda W$$

L = 在排队系统中的平均数量

λ = 系统中的有效到达率

W = 系统中一个条目的平均等待时间

John Little 解释说这个法则是强大的，通用的并且精确的保留了排队理论中所给的必要条件：“对开始【当系统为空】和当系统为空站点有一个有限的观察窗口。”（第 88 页）。

你可能会注意到，这个公式和文章中讨论的公式不同。实际上，对于原始的利特尔法则和在软件上下文中描述的公式($WIP = Th * CT$)有两个基本的不同。原始的公式提到的是输入与到达率，后面的公式提到的是产出率和产能。第二个问题是

在利特尔公式中陈述的条件：系统应该开始于 0 个条目，终止于 0 个条目。在软件中，我们很少看到一个系统没有任何维护的请求。

为了解决这些不同，Little 对于这个法则保留了更加微妙的条件，那就是：在系统中没有条目进入并且丢失，或未完成。他称为“保守流动”（第 93 页）。如果我们把这个条件应用到系统中，我们可以轻易地得到输入率（Input Rate）= 产出率（Output Rate），因此，我们就可以将有效到达率（ λ ）和产能（ th ）关联起来。

对于第二个条件（系统应该开始并终止于 0 个条目）。Little 解释说这个法则“仍然适用，至少是个近似值，只要我们选择的时间间隔足够长。”（第 93 页）

- [效应图（Diagrams of Effects）：](#)

效应图第一次引入是在著名的 four-volume series 中：Gerald Weinberg 写的质量软件管理。它是一个很好的开启器，试图理解系统展现的非线性行为的动态性，它与软件开发团队的系统很相像。

效应图与[因果回路图 CLD（Causal Loop Diagrams）](#)很像，但在记号上稍有不同，并且在系统中创建人为干预模型非常强大。一个效应图主要包括一些节点和箭头，每一个节点对应一个可测的量。简单的箭头对应一个效应（正面或负面的），从一个源头节点到一个目标节点。这里有一个全面的材料：[如何绘画和使用效应图手册](#)。

关于作者

Amr Noaman Abdel-Hamid 是一名敏捷教练、培训师和实践者，他的生活愿景就是将敏捷和精益思想传播到埃及和中东地区。Amr 是 [Agile Academy](#) 的共同创始人，该产品能够帮助团队和组织以最大的潜能交付软件。Amr 还是埃及 Lean&Agile Network 的创始成员，并有幸发起了埃及的 GoAgile 项目，该项目是在埃及由埃及政府为了推广敏捷所发起的采用敏捷活动。从那时起，Amr 已经培训了超过 400 名的实践者，并且指导了很多团队。Amr 经常发表演讲、是几个行业报告的作者，并且在他的博客中分享他的思想：[敏捷软件开发的故事](#)。你可以通过 [email](#)、[Linked-in](#) 或者 Twitter @amrnoaman 联系 Amr。

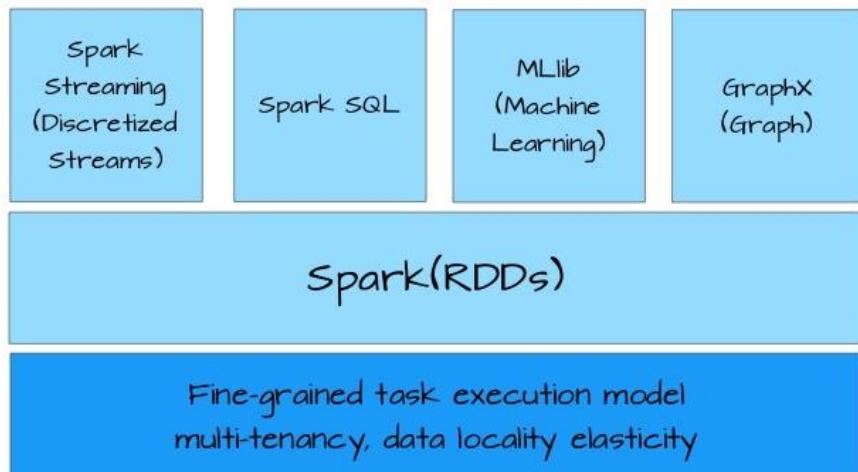
英文原文链接：<http://www.infoq.com/articles/how-kanban-works>

感谢[邵思华](#)对本文的审校。

查看原文：[看板如何奏效](#)

理解 Spark 的核心 RDD

作者 张逸



与许多专有的大数据处理平台不同，Spark 建立在统一抽象的 RDD 之上，使得它可以以基本一致的方式应对不同的大数据处理场景，包括 MapReduce，Streaming，SQL，Machine Learning 以及 Graph 等。这即 Matei Zaharia 所谓的“设计一个通用的编程抽象（Unified Programming Abstraction）。这正是 Spark 这朵小火花让人着迷的地方。

要理解 Spark，就需得理解 RDD。

RDD 是什么？

RDD，全称为 Resilient Distributed Datasets，是一个容错的、并行的数据结构，可以让用户显式地将数据存储到磁盘和内存中，并能控制数据的分区。同时，RDD 还提供了一组丰富的操作来操作这些数据。在这些操作中，诸如 map、flatMap、filter 等转换操作实现了 monad 模式，很好地契合了 Scala 的集合操作。除此之外，RDD 还提供了诸如 join、groupBy、reduceByKey 等更为方便的操作（注意，reduceByKey 是 action，而非 transformation），以支持常见的数据运算。

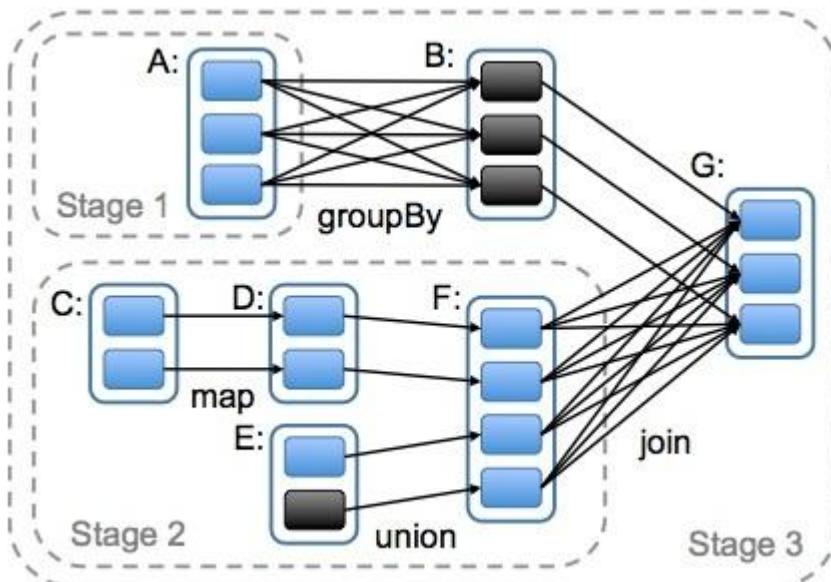
通常来讲，针对数据处理有几种常见模型，包括：Iterative Algorithms，Relational Queries，MapReduce，Stream Processing。例如 Hadoop MapReduce 采用了 MapReduces 模型，Storm 则采用了 Stream Processing 模型。RDD 混合了这四种模型，使得 Spark 可以应用于各种大数据处理场景。

RDD 作为数据结构，本质上是一个只读的分区记录集合。一个 RDD 可以包含多个分区，每个分区就是一个 dataset 片段。RDD 可以相互依赖。如果 RDD 的每个分区最多只能被一个 Child RDD 的一个分区使用，则称之为 narrow dependency；若多个 Child RDD 分区都可以依赖，则称之为 wide dependency。不同的操作依据其特性，可能会产生不同的依赖。例如 map 操作会产生 narrow dependency，而 join 操作则产生 wide dependency。

Spark 之所以将依赖分为 narrow 与 wide，基于两点原因。

首先，narrow dependencies 可以支持在同一个 cluster node 上以管道形式执行多条命令，例如在执行了 map 后，紧接着执行 filter。相反，wide dependencies 需要所有的父分区都是可用的，可能还需要调用类似 MapReduce 之类的操作进行跨节点传递。

其次，则是从失败恢复的角度考虑。narrow dependencies 的失败恢复更有效，因为它只需要重新计算丢失的 parent partition 即可，而且可以并行地在不同节点进行重计算。而 wide dependencies 牵涉到 RDD 各级的多个 Parent Partitions。下图说明了 narrow dependencies 与 wide dependencies 之间的区别：



本图来自 Matei Zaharia 撰写的论文 An Architecture for Fast and General Data Processing on Large Clusters。图中，一个 box 代表一个 RDD，一个带阴影的矩形框代表一个 partition。

RDD 如何保障数据处理效率？

RDD 提供了两方面的特性 persistence 和 partitioning，用户可以通过 persist 与 partitionBy 函数来控制 RDD 的这两个方面。RDD 的分区特性与并行计算能力(RDD 定义了 parallelize 函数)，使得 Spark 可以更好地利用可伸缩的硬件资源。若将分区与持久化二者结合起来，就能更加高效地处理海量数据。例如：

```
input.map(parseArticle _).partitionBy(partitioner).cache()
```

partitionBy 函数需要接受一个 Partitioner 对象，如：

```
val partitioner = new HashPartitioner(sc.defaultParallelism)
```

RDD 本质上是一个内存数据集，在访问 RDD 时，指针只会指向与操作相关的部分。例如存在一个面向列的数据结构，其中一个实现为 Int 的数组，另一个实现为 Float 的数组。如果只需要访问 Int 字段，RDD 的指针可以只访问 Int 数组，避免了对整个数据结构的扫描。

RDD 将操作分为两类：transformation 与 action。无论执行了多少次 transformation 操作，RDD 都不会真正执行运算，只有当 action 操作被执行时，运算才会触发。而在 RDD 的内部实现机制中，底层接口则是基于迭代器的，从而使得数据访问变得更高效，也避免了大量中间结果对内存的消耗。

在实现时，RDD 针对 transformation 操作，都提供了对应的继承自 RDD 的类型，例如 map 操作会返回 MappedRDD，而 flatMap 则返回 FlatMappedRDD。当我们执行 map 或 flatMap 操作时，不过是将当前 RDD 对象传递给对应的 RDD 对象而已。例如：

```
def map[U: ClassTag](f: T => U): RDD[U] = new MappedRDD(this, sc.clean(f))
```

这些继承自 RDD 的类都定义了 compute 函数。该函数会在 action 操作被调用时触发，在函数内部是通过迭代器进行对应的转换操作：

```
private[spark]
class MappedRDD[U: ClassTag, T: ClassTag](prev: RDD[T], f: T => U)
  extends RDD[U](prev) {

  override def getPartitions: Array[Partition] = firstParent[T].partitions

  override def compute(split: Partition, context: TaskContext) =
    firstParent[T].iterator(split, context).map(f)
}
```

RDD 对容错的支持

支持容错通常采用两种方式：数据复制或日志记录。对于以数据为中心的系统而言，这两种方式都非常昂贵，因为它需要跨集群网络拷贝大量数据，毕竟带宽的数据远远低于内存。

RDD 天生是支持容错的。首先，它自身是一个不变的(immutable)数据集，其次，它能够记住构建它的操作图（Graph of Operation），因此当执行任务的 Worker 失败时，完全可以通过操作图获得之前执行的操作，进行重新计算。由于无需采用 replication 方式支持容错，很好地降低了跨网络的数据传输成本。

不过，在某些场景下，Spark 也需要利用记录日志的方式来支持容错。例如，在 Spark Streaming 中，针对数据进行 update 操作，或者调用 Streaming 提供的 window 操作时，就需要恢复执行过程的中间状态。此时，需要通过 Spark 提供的 checkpoint 机制，以支持操作能够从 checkpoint 得到恢复。

针对 RDD 的 wide dependency，最有效的容错方式同样还是采用 checkpoint 机制。不过，似乎 Spark 的最新版本仍然没有引入 auto checkpointing 机制。

总结

RDD 是 Spark 的核心，也是整个 Spark 的架构基础。它的特性可以总结如下：

- 它是不变的数据结构存储。
- 它是支持跨集群的分布式数据结构。
- 可以根据数据记录的 key 对结构进行分区。
- 提供了粗粒度的操作，且这些操作都支持分区。
- 它将数据存储在内存中，从而提供了低延迟性。,

作者简介

张逸，现就职于 ThoughtWorks 中国。作为一名咨询师，主要为客户提供组织的敏捷转型、过程改进、企业系统架构、领域驱动设计、大数据、代码质量提升、测试驱动开发等咨询与培训工作。

查看原文：[理解 Spark 的核心 RDD](#)

缓存一致性入门

作者 曹知渊

本文是 RAD Game Tools 程序员 Fabian “ryg” Giesen 在其博客上发表的“Cache coherency primer”一文的翻译，经作者许可分享至 InfoQ 中文站。该系列共有两篇，本文系第一篇。

我计划写一些关于多核场景下数据组织的文章。写了第一篇，但我很快意识到有大量的基础知识我首先需要讲一下。在本文中，我就尝试阐述这些知识。

缓存（Cache）

本文是关于 CPU 缓存的快速入门。我假设你已经有了基本概念，但你可能不熟悉其中的一些细节。（如果你已经熟悉了，你可以忽略这部分。）

在现代的 CPU（大多数）上，所有的内存访问都需要通过层层的缓存来进行。也有些例外，比如，对映射成内存地址的 I/O 口、写合并（Write-combined）内存，这些访问至少会绕开这个流程的一部分。但这两者都是罕见的场景（意味着绝大多数的用户态代码都不会遇到这两种情况），所以在本文中，我将忽略这两者。

CPU 的读/写（以及取指令）单元正常情况下甚至都不能直接访问内存——这是物理结构决定的；CPU 都没有管脚直接连到内存。相反，CPU 和一级缓存（L1 Cache）通讯，而一级缓存才能和内存通讯。大约二十年前，一级缓存可以直接和内存传输数据。如今，更多级别的缓存加入到设计中，一级缓存已经不能直接和内存通讯了，它和二级缓存通讯——而二级缓存才能和内存通讯。或者还可能有三级缓存。你明白这个意思就行。

缓存是分“段”（line）的，一个段对应一块存储空间，大小是 32（较早的 ARM、90 年代 /2000 年代早期的 x86 和 PowerPC）、64（较新的 ARM 和 x86）或 128（较新的 Power ISA 机器）字节。每个缓存段知道自己对应什么范围的物理内存地址，并且在本文中，我不打算区分物理上的缓存段和它所代表的内存，这听起来有点草率，但是为了方便起见，还是请熟悉这种提法。具体地说，当我提到“缓存段”的时候，我就是指一段和缓存大小对齐的内存，不关心里面的内容是否真正被缓存进去（就是说保存在任何级别的缓存中）了。

当 CPU 看到一条读内存的指令时，它会把内存地址传递给一级数据缓存（或可戏称为 L1D\$，因为英语中“缓存”（cache）和“现金”（cash）的发音相同）。一级数据缓存会检查它是否有这个内存地址对应的缓存段。如果没有，它会把整个缓存段从内存（或者从更高级别的缓存，如果有的话）中加载进来。是的，一次加载整个缓存段，这是基于这样一个

假设：内存访问倾向于本地化（localized），如果我们当前需要某个地址的数据，那么很可能我们马上要访问它的邻近地址。一旦缓存段被加载到缓存中，读指令就可以正常进行读取。

如果我们只处理读操作，那么事情会很简单，因为所有级别的缓存都遵守以下规律，我称之为：

基本定律：在任意时刻，任意级别缓存中的缓存段的内容，等同于它对应的内存中的内容。

一旦我们允许写操作，事情就变得复杂一点了。这里有两种基本的写模式：直写（write-through）和回写（write-back）。直写更简单一点：我们透过本级缓存，直接把数据写到下一级缓存（或直接到内存）中，如果对应的段被缓存了，我们同时更新缓存中的内容（甚至直接丢弃），就这么简单。这也遵守前面的定律：缓存中的段永远和它对应的内存内容匹配。

回写模式就有点复杂了。缓存不会立即把写操作传递到下一级，而是仅修改本级缓存中的数据，并且把对应的缓存段标记为“脏”段。脏段会触发回写，也就是把里面的内容写到对应的内存或下一级缓存中。回写后，脏段又变“干净”了。当一个脏段被丢弃的时候，总是首先要进行一次回写。回写所遵循的规律有点不同。

回写定律：当所有的脏段被回写后，任意级别缓存中的缓存段的内容，等同于它对应的内存中的内容。

换句话说，回写模式的定律中，我们去掉了“在任意时刻”这个修饰语，代之以弱化一点的条件：要么缓存段的内容和内存一致（如果缓存段是干净的话），要么缓存段中的内容最终要回写到内存中（对于脏缓存段来说）。

直接模式更简单，但是回写模式有它的优势：它能过滤掉对同一地址的反复写操作，并且，如果大多数缓存段都在回写模式下工作，那么系统经常可以一下子写一大片内存，而不是分成小块来写，前者的效率更高。

有些（大多数是比较老的）CPU 只使用直写模式，有些只使用回写模式，还有一些，一级缓存使用直写而二级缓存使用回写。这样做虽然在一级和二级缓存之间产生了不必要的数据流量，但二级缓存和更低级缓存或内存之间依然保留了回写的优势。我想说的是，这里涉及到一系列的取舍问题，且不同的设计有不同的解决方案。没有人规定各级缓存的大小必须一致。举个例子，我们会看到有 CPU 的一级缓存是 32 字节，而二级缓存却有 128 字节。

为了简化问题，我省略了一些内容：缓存关联性（cache associativity），缓存组（cache sets），使用分配写（write-allocate）还是非分配写（上面我描述的直写是和分配写相结合的，而回写是和非分配写相结合的），非对齐的访问（unaligned access），基于虚拟地址的缓存。如果你感兴趣，所有这些内容都可以去查查资料，但我不准备在这里讲了。

一致性协议（Coherency protocols）

只要系统只有一个 CPU 核在工作，一切都没问题。如果有多个核，每个核又都有自己的缓存，那么我们就遇到问题了：如果某个 CPU 缓存段中对应的内存内容被另外一个 CPU 偷偷改了，会发生什么？

好吧，答案很简单：什么也不会发生。这很糟糕。因为如果一个 CPU 缓存了某块内存，那么在其他 CPU 修改这块内存的时候，我们希望得到通知。我们拥有多组缓存的时候，真的需要它们保持同步。或者说，系统的内存要在各个 CPU 之间无法做到与生俱来的同步，我们实际上是需要一个大家都遵守的方法来达到同步的目的。

注意，这个问题的根源是我们拥有多组缓存，而不是多个 CPU 核。我们也可以这样解决问题，让多个 CPU 核共用一组缓存：也就是说只有一块一级缓存，所有处理器都必须共用它。在每一个指令周期，只有一个幸运的 CPU 能通过一级缓存做内存操作，运行它的指令。

这本身没问题。唯一的问题就是太慢了，因为这下处理器的时间都花在排队等待使用一级缓存了（并且处理器会做大量的这种操作，至少每个读写指令都要做一次）。我指出这一点是因为它表明了问题不是由多核引起的，而是由多缓存引起的。我们知道了只有一组缓存也能工作，只是太慢了，接下来最好就是能做到：使用多组缓存，但使它们的行为看起来就像只有一组缓存那样。缓存一致性协议就是为了做到这一点而设计的。就像名称所暗示的那样，这类协议就是要使多组缓存的内容保持一致。

缓存一致性协议有多种，但是你日常处理的大多数计算机设备使用的都属于“窥探（snooping）”协议，这也是我这里要讲的。（还有一种叫“基于目录的（directory-based）”协议，这种协议的延迟性较大，但是在拥有多个处理器的系统中，它有更好的可扩展性。）

“窥探”背后的基本思想是，所有内存传输都发生在一条共享的总线上，而所有的处理器都能看到这条总线：缓存本身是独立的，但是内存是共享资源，所有的内存访问都要经过仲裁（arbitrate）：同一个指令周期中，只有一个缓存可以读写内存。窥探协议的思想是，缓存不仅仅在做内存传输的时候才和总线打交道，而是不停地在窥探总线上发生的数据交换，跟踪其他缓存在做什么。所以当一个缓存代表它所属的处理器去读写内存时，其他处

理器都会得到通知，它们以此来使自己的缓存保持同步。只要某个处理器一写内存，其他处理器马上就知道这块内存存在它们自己的缓存中对应的段已经失效。

在直写模式下，这是很直接的，因为写操作一旦发生，它的效果马上会被“公布”出去。但是如果混着回写模式，就有问题了。因为有可能在写指令执行过后很久，数据才会被真正写到物理内存中——在这段时间内，其他处理器的缓存也可能会傻乎乎地去写同一块内存地址，导致冲突。在回写模型中，简单把内存写操作的信息广播给其他处理器是不够的，我们需要做的是，在修改本地缓存之前，就要告知其他处理器。搞懂了细节，就找到了处理回写模式这个问题的最简单方案，我们通常叫做 MESI 协议（译者注：MESI 是 Modified、Exclusive、Shared、Invalid 的首字母缩写，代表四种缓存状态，下面的译文中可能会以单个字母指代相应状态）。

MESI 以及衍生协议

本节叫做“MESI 以及衍生协议”，是因为 MESI 衍生了一系列紧密相关的一致性协议。我们先从原生的 MESI 协议开始：MESI 是四种缓存段状态的首字母缩写，任何多核系统中的缓存段都处于这四种状态之一。我将以相反的顺序逐个讲解，因为这个顺序更合理：

- 失效 (Invalid) 缓存段，要么已经不在缓存中，要么它的内容已经过时。为了达到缓存的目的，这种状态的段将会被忽略。一旦缓存段被标记为失效，那效果就等同于它从来没被加载到缓存中。
- 共享 (Shared) 缓存段，它是和主内存内容保持一致的一份拷贝，在这种状态下的缓存段只能被读取，不能被写入。多组缓存可以同时拥有针对同一内存地址的共享缓存段，这就是名称的由来。
- 独占 (Exclusive) 缓存段，和 S 状态一样，也是和主内存内容保持一致的一份拷贝。区别在于，如果一个处理器持有了某个 E 状态的缓存段，那其他处理器就不能同时持有它，所以叫“独占”。这意味着，如果其他处理器原本也持有同一缓存段，那么它会马上变成“失效”状态。
- 已修改 (Modified) 缓存段，属于脏段，它们已经被所属的处理器修改了。如果一个段处于已修改状态，那么它在其他处理器缓存中的拷贝马上会变成失效状态，这个规律和 E 状态一样。此外，已修改缓存段如果被丢弃或标记为失效，那么先要把它的内容回写到内存中——这和回写模式下常规的脏段处理方式一样。

如果把以上这些状态和单核系统中回写模式的缓存做对比，你会发现 I、S 和 M 状态已经有对应的概念：失效/未载入、干净以及脏的缓存段。所以这里的新知识只有 E 状态，代表独占式访问。这个状态解决了“在我们开始修改某块内存之前，我们需要告诉其他处理器”这一问题：只有当缓存段处于 E 或 M 状态时，处理器才能去写它，也就是说只有这两种状态下，处理器是独占这个缓存段的。当处理器想写某个缓存段时，如果它没有独占权，它必须先发送一条“我要独占权”的请求给总线，这会通知其他处理器，把它们拥有的同一缓存段的拷贝失效（如果它们有的话）。只有在获得独占权后，处理器才能开始修改数据——并且此时，这个处理器知道，这个缓存段只有一份拷贝，在我自己的缓存里，所以不会有任何冲突。

反之，如果有其他处理器想读取这个缓存段（我们马上能知道，因为我们一直在窥探总线），独占或已修改的缓存段必须先回到“共享”状态。如果是已修改的缓存段，那么还要先把内容回写到内存中。

MESI 协议是一个合适的状态机，既能处理来自本地处理器的请求，也能把信息广播到总线上。我不打算讲更多关于状态图的细节以及不同的状态转换类型。如果你感兴趣的话，可以在关于硬件架构的书中找到更多的深度内容，但对于本文来说，讲这些东西有点过了。作为一个软件开发者，你只要理解以下两点，就大有可为：

第一，在多核系统中，读取某个缓存段，实际上会牵涉到和其他处理器的通讯，并且可能导致它们发生内存传输。写某个缓存段需要多个步骤：在你写任何东西之前，你首先要获得独占权，以及所请求的缓存段的当前内容的拷贝（所谓的“带权限获取的读（Read For Ownership）”请求）。

第二，尽管我们为了一致性问题做了额外的工作，但是最终结果还是非常有保证的。即它遵守以下定理，我称之为：

MESI 定律：在所有的脏缓存段（M 状态）被回写后，任意缓存级别的所有缓存段中的内容，和它们对应的内存中的内容一致。此外，在任意时刻，当某个位置的内存被一个处理器加载入独占缓存段时（E 状态），那它就不会再出现在其他任何处理器的缓存中。

注意，这其实是我们已经讲过的回写定律加上独占规则而已。我认为 MESI 协议或多核系统的存在根本没有弱化我们现有的内存模型。

好了，至此我们（粗略）讲了原生 MESI 协议（以及使用它的 CPU，比如 ARM）。其他处理器使用 MESI 扩展后的变种。常见的扩展包括“O”（Owned）状态，它和 E 状态类似，也是保证缓存间一致性的手段，但它直接共享脏段的内容，而不需要先把它们回写到内存中（“脏段共享”），由此产生了 MOSEI 协议。还有 MERSI 和 MESIF，这两个名字代表同一种思想，即指定某个处理器专门处理针对某个缓存段的读操作。当多个处理器同时拥有某个 S 状态的缓存段的时候，只有被指定的那个处理器（对应的缓存段为 R 或 F 状态）才能对读操作做出回应，而不是每个处理器都能这么做。这种设计可以降低总线的数据流量。当然你可以同时加入 R/F 状态和 O 状态，或者更多的状态。这些都属于优化，没有一种会改变基本定律，也没有一种会改变 MESI 协议所确保的结果。

我不是这方面的专家，很有可能有系统在使用其他协议，这些协议并不能完全保证一致性，不过如果有，我没有注意到它们，或者没有看到有什么流行的处理器在使用它们。所以为了达到我们的目的，我们真的就可以假设一致性协议能保证缓存的一致性。不是基本

一致，不是“写入一会儿后才能保持一致”——而是完全的一致。从这个层面上说，除非硬件有问题，内存的状态总是一致的。用技术术语来说，MESI 以及它的衍生协议，至少在原理上，提供了完整的顺序一致性（sequential consistency），在 C++ 11 的内存模型中，这是最强的一种确保内存顺序的模型。这也引出了问题，为什么我们需要弱一点的内存模型，以及“什么时候会用到它们”？

内存模型

不同的体系结构提供不同的内存模型。到本文写作的时候为止，ARM 和 POWER 体系结构的机器拥有相对较弱的内存模型：这类 CPU 在读写指令重排序（reordering）方面有相当大的自由度，这种重排序有可能会改变程序在多核环境下的语义。通过“内存屏障（memory barrier）”，程序可以对此加以限制：“重排序操作不允许越过这条边界”。相反，x86 则拥有较强的内存模型。

我不打算在这里深入到内存模型的细节中，这很容易陷入堆砌技术术语中，而且也超出了本文的范围。但是我想说一点关于“他们如何发生”的内容——也就是，弱内存模型如何保证正确性（相比较于 MESI 协议给缓存带来的顺序一致性），以及为什么。当然，一切都归结于性能。

规则是这样的：如果满足下面的条件，你就可以得到完全的顺序一致性：第一，缓存一收到总线事件，就可以在当前指令周期中迅速做出响应。第二，处理器如实地按程序的顺序，把内存操作指令送到缓存，并且等前一条执行完后才能发送下一条。当然，实际上现代处理器一般都无法满足以上条件：

- 缓存不会及时响应总线事件。如果总线上发来一条消息，要使某个缓存段失效，但是如果此时缓存正在处理其他事情（比如和 CPU 传输数据），那这个消息可能无法在当前的指令周期中得到处理，而会进入所谓的“失效队列”（invalidation queue），这个消息等在队列中直到缓存有空为止。
- 处理器一般不会严格按照程序的顺序向缓存发送内存操作指令。当然，有乱序执行（Out-of-Order execution）功能的处理器肯定是这样的。顺序执行（in-order execution）的处理器有时候也无法完全保证内存操作的顺序（比如想要的内存不在缓存中时，CPU 就不能为了载入缓存而停止工作）。
- 写操作尤其特殊，因为它分为两阶段操作：在写之前我们先要得到缓存段的独占权。如果我们当前没有独占权，我们先要和其他处理器协商，这也需要一些时间。同理，在这种场景下让处理器闲着无所事事是一种资源浪费。实际上，写操作首先发起获得独占权的请求，然后就进入所谓的由“写缓冲”（store buffer）组成的队列（有些地方使用“写缓冲”指代整个队列，我这里使用它指代队列的一条入口）。写操作在队列中等待，直到缓存准备好处理它，此时写缓冲就被“清空”（drained）了，缓冲区被回收用于处理新的写操作。

这些特性意味着，默认情况下，读操作有可能会读到过时的数据（如果对应失效请求还在队列中没执行），写操作真正完成的时间有可能比它们在代码中的位置晚，一旦牵涉到乱序执行，一切都变得模棱两可。回到内存模型，本质上只有两大阵营。

在弱内存模型的体系结构中，处理器为了开发者能写出正确的代码而做的工作是最小化的，指令重排序和各种缓冲的步骤都是被正式允许的，也就是说没有任何保证。如果你需要确保某种结果，你需要自己插入合适的内存屏障——它能防止重排序，并且等待队列中的操作全部完成。

使用强一点的内存模型的体系结构则会在内部做很多记录工作。比如，x86 会跟踪所有在等待中的内存操作，这些操作都还没有完全完成（称为“退休”（retired））。它会把它们的信息保存在芯片内部的 MOB（memory ordering buffer，内存排序缓冲）。x86 作为部分支持乱序执行的体系结构，在出问题的时候能把尚未“退休”的指令撤销掉——比如发生页错误（page fault），或者分支预测失败（branch mispredict）的时候。我已经在我以前的文章[“好奇地说”](#)中提到过一些细节，以及和内存子系统的一些交互。主旨是 x86 处理器会主动地监控外部事件（比如缓存失效），有些已经执行完的操作会因为这些事件而被撤销，但不算“退休”。这就是说，x86 知道自己的内存模型应该是什么样子的，当发生了一件和这个模型冲突的事，处理器会回退到上一个与内存模型兼容的状态。这就是我在[以前另一篇文章](#)中提到的“清除内存排序机（memory ordering machine clear）”。最后的结果是，x86 处理器为内存操作提供了很强的一致性保证——虽然没有达到完美的顺序一致性。

无论如何，一篇文章讲这么多已经够了。我把它放在我的博客上。我的想法是将来的文章只要引用它就行了。我们看效果吧。感谢阅读！

查看参考原文：<http://fgiesen.wordpress.com/2014/07/07/cache-coherency/>

查看原文：[缓存一致性（Cache Coherency）入门](#)

腾讯大数据之 TDW 计算引擎 解析——Shuffle

作者 腾讯

腾讯分布式数据仓库（Tencent distributed Data Warehouse, 简称 TDW）基于开源软件 Hadoop 和 Hive 进行构建，并且根据公司数据量大、计算复杂等特定情况进行了大量优化和改造，目前单集群最大规模达到 5600 台，每日作业数达到 100 多万，已经成为公司最大的离线数据处理平台。为了满足用户更加多样的计算需求，TDW 也在向实时化方向发展，为用户提供更加高效、稳定、丰富的服务。

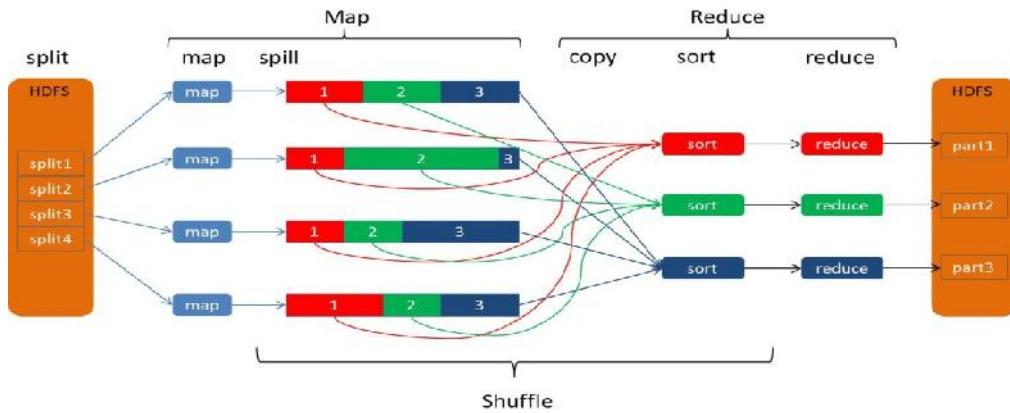
TDW 计算引擎包括两部分：一个是偏离线的 MapReduce，一个是偏实时的 Spark，两者内部都包含了一个重要的过程——Shuffle。本文对 Shuffle 过程进行解析，并对两个计算引擎的 Shuffle 过程进行比较，对后续的优化方向进行思考和探索，期待经过我们不断的努力，TDW 计算引擎运行地更好。

Shuffle——MapReduce 的 Shuffle 过程

Shuffle 的本义是洗牌、混洗，把一组有一定规则的数据尽量转换成一组无规则的数据，越随机越好。MapReduce 中的 shuffle 更像是洗牌的逆过程，把一组无规则的数据尽量转换成一组具有一定规则的数据。

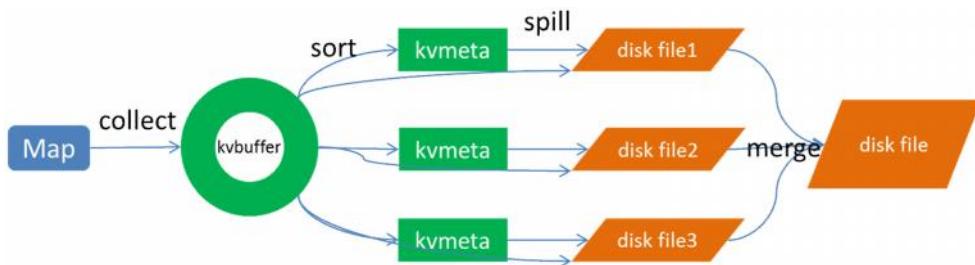
为什么 MapReduce 计算模型需要 shuffle 过程？我们都应该知道 MapReduce 计算模型一般包括两个重要的阶段：map 是映射，负责数据的过滤分发；reduce 是规约，负责数据的计算归并。Reduce 的数据来源于 map，map 的输出即是 reduce 的输入，reduce 需要通过 shuffle 来获取数据。

从 map 输出到 reduce 输入的整个过程可以广义地称为 shuffle。Shuffle 横跨 map 端和 reduce 端，在 map 端包括 spill 过程，在 reduce 端包括 copy 和 sort 过程，如图所示：



Spill 过程

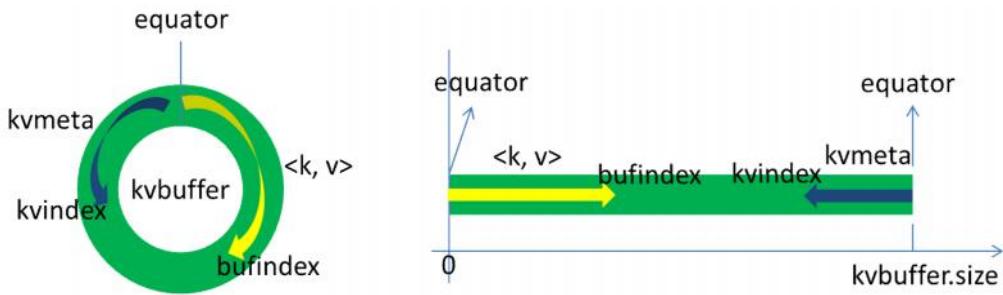
Spill 过程包括输出、排序、溢写、合并等步骤，如图所示：



Collect

每个 Map 任务不断地以`<key, value>`对的形式把数据输出到在内存中构造的一个环形数据结构中。使用环形数据结构是为了更有效地使用内存空间，在内存中放置尽可能多的数据。

这个数据结构其实就是一个字节数组，叫 **kvbuffer**，名如其义，但是这里面不光放置了`<key, value>`数据，还放置了一些索引数据，给放置索引数据的区域起了一个 **kvmeta** 的别名，在 **kvbuffer** 的一块区域上穿了一个 **IntBuffer**（字节序采用的是平台自身的字节序）的马甲。`<key, value>` 数据区域和索引数据区域在 **kvbuffer** 中是相邻不重叠的两个区域，用一个分界点来划分两者，分界点不是亘古不变的，而是每次 **spill** 之后都会更新一次。初始的分界点是 0，`<key, value>` 数据的存储方向是向上增长，索引数据的存储方向是向下增长，如图所示：



Kvbuffer 的存放指针 bufindex 是一直闷着头地向上增长，比如 bufindex 初始值为 0，一个 Int 型的 key 写完之后，bufindex 增长为 4，一个 Int 型的 value 写完之后，bufindex 增长为 8。

索引是对`<key, value>`在 Kvbuffer 中的索引，是个四元组，包括：value 的起始位置、key 的起始位置、partition 值、value 的长度，占用四个 Int 长度，kvmeta 的存放指针 kvindex 每次都是向下跳四个“格子”，然后再向上一个格子一个格子地填充四元组的数据。比如 kvindex 初始位置是 -4，当第一个`<key, value>`写完之后，`(kvindex+0)`的位置存放 value 的起始位置、`(kvindex+1)`的位置存放 key 的起始位置、`(kvindex+2)`的位置存放 partition 的值、`(kvindex+3)`的位置存放 value 的长度，然后 kvindex 跳到 -8 位置，等第二个`<key, value>`和索引写完之后，kvindex 跳到 -32 位置。

Kvbuffer 的大小虽然可以通过参数设置，但是总共就那么大，`<key, value>`和索引不断地增加，加着加着，Kvbuffer 总有不够用的那天，那怎么办？把数据从内存刷到磁盘上再接着往内存写数据，把 Kvbuffer 中的数据刷到磁盘上的过程就叫 spill，多么明了的叫法，内存中的数据满了就自动地 spill 到具有更大空间的磁盘。

关于 spill 触发的条件，也就是 Kvbuffer 用到什么程度开始 spill，还是要讲究一下的。如果把 Kvbuffer 用得死死得，一点缝都不剩的时候再开始 spill，那 map 任务就需要等 spill 完成腾出空间之后才能继续写数据；如果 Kvbuffer 只是满到一定程度，比如 80% 的时候就开始 spill，那在 spill 的同时，map 任务还能继续写数据，如果 spill 够快，map 可能都不需要为空闲空间而发愁。两利相衡取其大，一般选择后者。

Spill 这个重要的过程是由 spill 线程承担，spill 线程从 map 任务接到“命令”之后就开始正式干活，干的活叫 sortAndSpill，原来不仅仅是 spill，在 spill 之前还有个颇具争议性的 sort。

Sort

先把 Kvbuffer 中的数据按照 partition 值和 key 两个关键字升序排序，移动的只是索引数据，排序结果是 Kvmeta 中数据按照 partition 为单位聚集在一起，同一 partition 内的按照 key 有序。

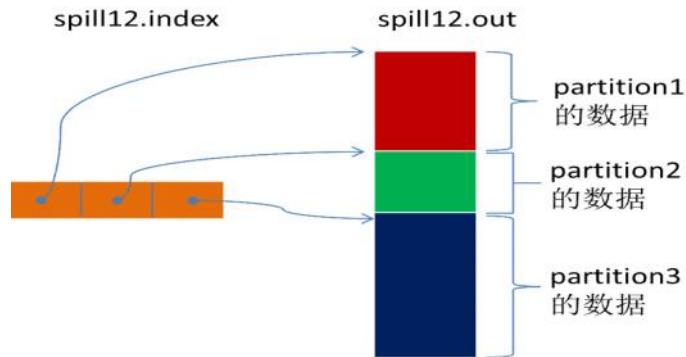
Spill

Spill 线程为这次 spill 过程创建一个磁盘文件：从所有的本地目录中轮训查找能存储这么大的空间的目录，找到之后在其中创建一个类似于“spill12.out”的文件。Spill 线程根据排过序的 kvmeta 挨个 partition 的把`<key, value>`数据吐到这个文件中，一个 partition 对应的数据吐完之后顺序地吐下个 partition，直到把所有的 partition 遍历完。一个 partition 在文件中对应的数据也叫段(segment)。

所有的 partition 对应的数据都放在这个文件里，虽然是顺序存放的，但是怎么直接知道某个 partition 在这个文件中存放的起始位置呢？强大的索引又出场了。有一个三元组记录某个 partition 对应的数据在这个文件中的索引：起始位置、原始数据长度、压缩之后的数据长度，一个 partition 对应一个三元组。

然后把这些索引信息存放在内存中，如果内存中放不下了，后续的索引信息就需要写到磁盘文件中了：从所有的本地目录中轮训查找能存储这么大的空间的目录，找到之后在其中创建一个类似于“spill12.out.index”的文件，文件中不光存储了索引数据，还存储了 crc32 的校验数据。(spill12.out.index 不一定在磁盘上创建，如果内存（默认 1M 空间）中能放得下就放在内存中，即使在磁盘上创建了，和 spill12.out 文件也不一定在同一个目录下。)

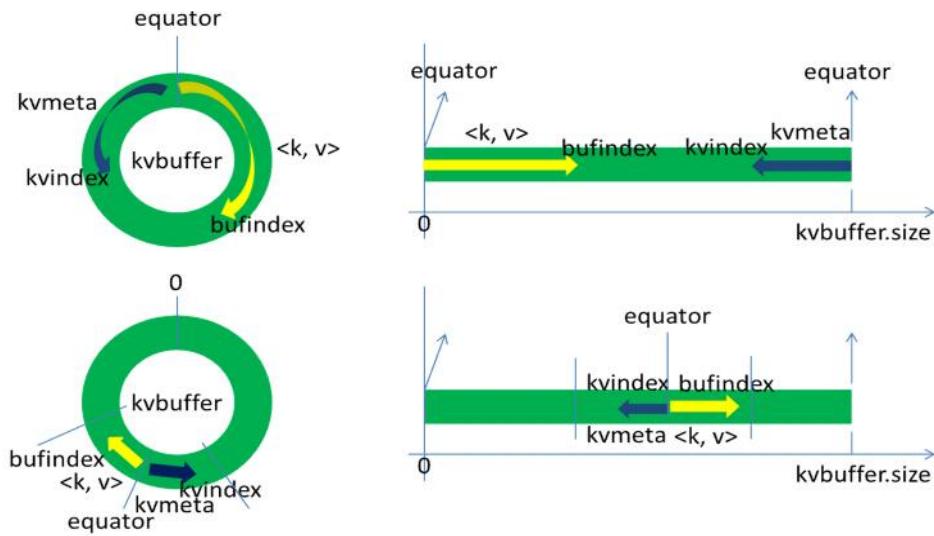
每一次 spill 过程就会最少生成一个 out 文件，有时还会生成 index 文件，spill 的次数也烙印在文件名中。索引文件和数据文件的对应关系如下图所示：



话分两端，在 spill 线程如火如荼的进行 sortAndSpill 工作的同时，map 任务不会因此而停歇，而是一无既往地进行着数据输出。Map 还是把数据写到 kvbuffer 中，那问题就来了：`<key, value>`只顾着闷头按照 bufindex 指针向上增长，kvmeta 只顾着按照 kvindex 向下增长，是保持指针起始位置不变继续跑呢，还是另谋它路？如果保持指针起始位置不变，很快 bufindex 和 kvindex 就碰头了，碰头之后再重新开始或者移动内存都比较麻烦，不可取。Map 取 kvbuffer 中剩余空间的中间位置，用这个位置设置为新的分界点，bufindex 指针移动到这个分界点，kvindex 移动到这个分界点的 -16 位置，然后两者就可以和谐地按照

自己既定的轨迹放置数据了，当 spill 完成，空间腾出之后，不需要做任何改动继续前进。

分界点的转换如下图所示：



Map 任务总要把输出的数据写到磁盘上，即使输出数据量很小在内存中全部能装得下，在最后也会把数据刷到磁盘上。

Merge

Map 任务如果输出数据量很大，可能会进行好几次 spill，out 文件和 index 文件会产生很多，分布在不同的磁盘上。最后把这些文件进行合并的 merge 过程闪亮登场。

Merge 过程怎么知道产生的 spill 文件都在哪了呢？从所有的本地目录上扫描得到产生的 spill 文件，然后把路径存储在一个数组里。Merge 过程又怎么知道 spill 的索引信息呢？没错，也是从所有的本地目录上扫描得到 index 文件，然后把索引信息存储在一个列表里。到这里，又遇到了一个值得纳闷的地方。在之前 spill 过程中的时候为什么不直接把这些信息存储在内存中呢，何必又多了这步扫描的操作？特别是 spill 的索引数据，之前当内存超限之后就把数据写到磁盘，现在又要从磁盘把这些数据读出来，还是需要装到更多的内存中。之所以多此一举，是因为这时 kvbuffer 这个内存大户已经不再使用可以回收，有内存空间来装这些数据了。（对于内存空间较大的土豪来说，用内存来省却这两个 I/O 步骤还是值得考虑的。）

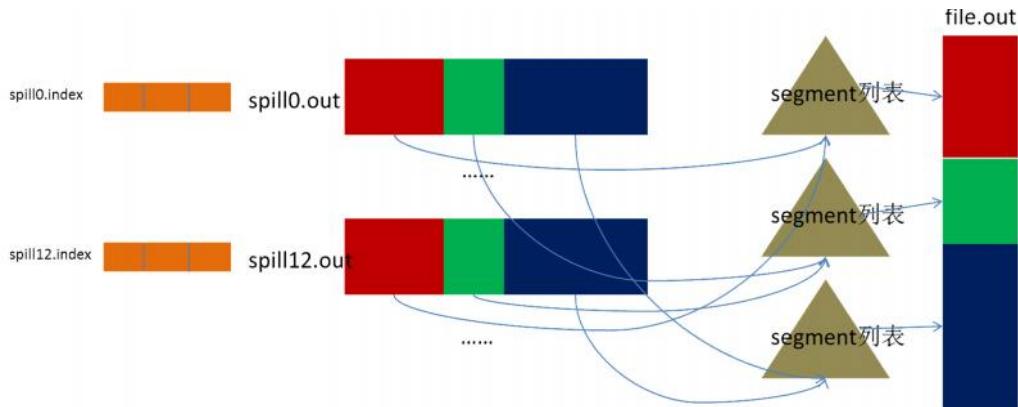
然后为 merge 过程创建一个叫 file.out 的文件和一个叫 file.out.index 的文件用来存储最终的输出和索引。

一个 partition 一个 partition 的进行合并输出。对于某个 partition 来说，从索引列表中查询这个 partition 对应的所有索引信息，每个对应一个段插入到段列表中。也就是这个 partition

对应一个段列表，记录所有的 spill 文件中对应的这个 partition 那段数据的文件名、起始位置、长度等等。

然后对这个 partition 对应的所有 segment 进行合并，目标是合并成一个 segment。当这个 partition 对应很多个 segment 时，会分批地进行合并：先从 segment 列表中把第一批取出来，以 key 为关键字放置成最小堆，然后从最小堆中每次取出最小的<key, value>输出到一个临时文件中，这样就把这一批段合并成一个临时的段，把它加回到 segment 列表中；再从 segment 列表中把第二批取出来合并输出到一个临时 segment，将其加入到列表中；这样往复执行，直到剩下的段是一批，输出到最终的文件中。

最终的索引数据仍然输出到 index 文件中。



Map 端的 shuffle 过程到此结束。

Copy

Reduce 任务通过 http 向各个 map 任务拖取它所需要的数据。每个节点都会启动一个常驻的 http server，其中一项服务就是响应 reduce 拖取 map 数据。当有 mapOutput 的 http 请求过来的时候，http server 就读取相应的 map 输出文件中对应这个 reduce 部分的数据通过网络流输出给 reduce。

Reduce 任务拖取某个 map 对应的数据，如果在内存中能放得下这次数据的话就直接把数据写到内存中。Reduce 要向每个 map 去拖取数据，在内存中每个 map 对应一块数据，当内存中存储的 map 数据占用空间达到一定程度的时候，开始启动内存中 merge，把内存中的数据 merge 输出到磁盘上一个文件中。

如果在内存中不能放得下这个 map 的数据的话，直接把 map 数据写到磁盘上，在本地目录创建一个文件，从 http 流中读取数据然后写到磁盘，使用的缓存区大小是 64K。拖一个

map 数据过来就会创建一个文件，当文件数量达到一定阈值时，开始启动磁盘文件 merge，把这些文件合并输出到一个文件。

有些 map 的数据较小是可以放在内存中的，有些 map 的数据较大需要放在磁盘上，这样最后 reduce 任务拖过来的数据有些放在内存中了有些放在磁盘上，最后会对这些来一个全局合并。

Merge Sort

这里使用的 merge 和 map 端使用的 merge 过程一样。Map 的输出数据已经是有序的，merge 进行一次合并排序，所谓 reduce 端的 sort 过程就是这个合并的过程。一般 reduce 是一边 copy 一边 sort，即 copy 和 sort 两个阶段是重叠而不是完全分开的。

Reduce 端的 shuffle 过程至此结束。

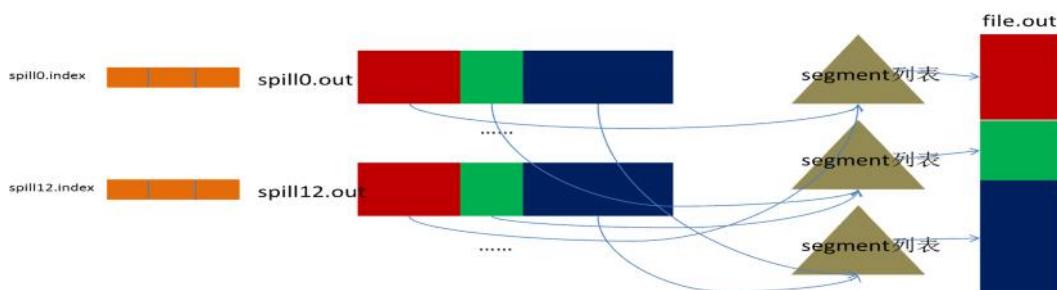
Spark 的 Shuffle 过程介绍

Shuffle Writer

Spark 丰富了任务类型，有些任务之间数据流转不需要通过 shuffle，但是有些任务之间还是需要通过 shuffle 来传递数据，比如 wide dependency 的 group by key。

Spark 中需要 shuffle 输出的 map 任务会为每个 reduce 创建对应的 bucket，map 产生的结果会根据设置的 partitioner 得到对应的 bucketId，然后填充到相应的 bucket 中去。每个 map 的输出结果可能包含所有的 reduce 所需要的数据，所以每个 map 会创建 R 个 bucket（R 是 reduce 的个数），M 个 map 总共会创建 M*R 个 bucket。

Map 创建的 bucket 其实对应磁盘上的一个文件，map 的结果写到每个 bucket 中其实就是写到那个磁盘文件中，这个文件也被称为 blockFile，是 DiskBlockManager 管理器通过文件名的 hash 值对应到本地目录的子目录中创建的。每个 map 要在节点上创建 R 个磁盘文件用于结果输出，map 的结果是直接输出到磁盘文件上的，100KB 的内存缓冲是用来创建 FastBufferedOutputStream 输出流。这种方式一个问题就是 shuffle 文件过多。



针对上述 shuffle 过程产生的文件过多问题，Spark 有另外一种改进的 shuffle 过程：consolidation shuffle，以期显著减少 shuffle 文件的数量。在 consolidation shuffle 中每个 bucket 并非对应一个文件，而是对应文件中的一个 segment 部分。Job 的 map 在某个节点上第一次执行，为每个 reduce 创建 bucket 对应的输出文件，把这些文件组织成 ShuffleFileGroup，当这次 map 执行完之后，这个 ShuffleFileGroup 可以释放为下次循环利用；当又有 map 在这个节点上执行时，不需要创建新的 bucket 文件，而是在上次的 ShuffleFileGroup 中取得已经创建的文件继续追加写一个 segment；当前次 map 还没执行完，ShuffleFileGroup 还没有释放，这时如果有新的 map 在这个节点上执行，无法循环利用这个 ShuffleFileGroup，而是只能创建新的 bucket 文件组成新的 ShuffleFileGroup 来写输出。

比如一个 job 有 3 个 map 和 2 个 reduce：(1) 如果此时集群有 3 个节点有空槽，每个节点空闲了一个 core，则 3 个 map 会调度到这 3 个节点上执行，每个 map 都会创建 2 个 shuffle 文件，总共创建 6 个 shuffle 文件；(2) 如果此时集群有 2 个节点有空槽，每个节点空闲了一个 core，则 2 个 map 先调度到这 2 个节点上执行，每个 map 都会创建 2 个 shuffle 文件，然后其中一个节点执行完 map 之后又调度执行另一个 map，则这个 map 不会创建新的 shuffle 文件，而是把结果输出追加到之前 map 创建的 shuffle 文件中；总共创建 4 个 shuffle 文件；(3) 如果此时集群有 2 个节点有空槽，一个节点有 2 个空 core 一个节点有 1 个空 core，则一个节点调度 2 个 map 一个节点调度 1 个 map，调度 2 个 map 的节点上，一个 map 创建了 shuffle 文件，后面的 map 还是会创建新的 shuffle 文件，因为上一个 map 还正在写，它创建的 ShuffleFileGroup 还没有释放；总共创建 6 个 shuffle 文件。

Shuffle Fetcher

Reduce 去拖 map 的输出数据，Spark 提供了两套不同的拉取数据框架：通过 socket 连接去取数据；使用 netty 框架去取数据。

每个节点的 Executor 会创建一个 BlockManager，其中会创建一个 BlockManagerWorker 用于响应请求。当 reduce 的 GET_BLOCK 的请求过来时，读取本地文件将这个 blockId 的数据返回给 reduce。如果使用的是 Netty 框架，BlockManager 会创建 ShuffleSender 用于发送 shuffle 数据。

并不是所有的数据都是通过网络读取，对于在本节点的 map 数据，reduce 直接去磁盘上读取而不再通过网络框架。

Reduce 拖过来数据之后以什么方式存储呢？Spark map 输出的数据没有经过排序，spark shuffle 过来的数据也不会进行排序，spark 认为 shuffle 过程中的排序不是必须的，并不是所有类型的 reduce 需要的数据都需要排序，强制地进行排序只会增加 shuffle 的负担。

Reduce 拖过来的数据会放在一个 HashMap 中，HashMap 中存储的也是<key, value>对，key

是 map 输出的 key，map 输出对应这个 key 的所有 value 组成 HashMap 的 value。Spark 将 shuffle 取过来的每一个<key, value>对插入或者更新到 HashMap 中，来一个处理一个。HashMap 全部放在内存中。

Shuffle 取过来的数据全部存放在内存中，对于数据量比较小或者已经在 map 端做过合并处理的 shuffle 数据，占用内存空间不会太大，但是对于比如 group by key 这样的操作，reduce 需要得到 key 对应的所有 value，并将这些 value 组一个数组放在内存中，这样当数据量较大时，就需要较多内存。

当内存不够时，要不就失败，要不就用老办法把内存中的数据移到磁盘上放着。Spark 意识到在处理数据规模远远大于内存空间时所带来的不足，引入了一个具有外部排序的方案。Shuffle 取过来的数据先放在内存中，当内存中存储的<key, value>对超过 1000 并且内存使用超过 70% 时，判断节点上可用内存如果还足够，则把内存缓冲区大小翻倍，如果可用内存不再够了，则把内存中的<key, value>对排序然后写到磁盘文件中。最后把内存缓冲区中的数据排序之后和那些磁盘文件组成一个最小堆，每次从最小堆中读取最小的数据，这个和 MapReduce 中的 merge 过程类似。

MapReduce 和 Spark 的 Shuffle 过程对比

	MapReduce	Spark
collect	在内存中构造了一块数据结构用于 map 输出的缓冲	没有在内存中构造一块数据结构用于 map 输出的缓冲，而是直接把输出写到磁盘文件
sort	map 输出的数据有排序	map 输出的数据没有排序
merge	对磁盘上的多个 spill 文件最后进行合并成一个输出文件	在 map 端没有 merge 过程，在输出时直接是对应一个 reduce 的数据写到一个文件中，这些文件同时存在并发写，最后不需要合并成一个
copy 框架	jetty	netty 或者直接 socket 流
对于本节点上的文件	仍然是通过网络框架拖取数据	不通过网络框架，对于在本节点上的 map 输出文件，采用本地读取的方式

copy过来的数据存放位置	先放在内存，内存放不下时写到磁盘	一种方式全部放在内存；另一种方式先放在内存
merge sort	最后会对磁盘文件和内存中的数据进行合并排序	对于采用另一种方式时也会有合并排序的过程

Shuffle 后续优化方向

通过上面的介绍，我们了解到，shuffle 过程的主要存储介质是磁盘，尽量的减少 io 是 shuffle 的主要优化方向。我们脑海中都有那个经典的存储金字塔体系，shuffle 过程为什么把结果都放在磁盘上，那是因为现在内存再大也大不过磁盘，内存就那么大，还这么多张嘴吃，当然是分配给最需要的了。如果具有“土豪”内存节点，减少 shuffle io 的最有效方式无疑是尽量把数据放在内存中。下面列举一些现在看可以优化的方面，期待经过我们不断的努力，TDW 计算引擎运行地更好。

MapReduce Shuffle 后续优化方向

- 压缩：对数据进行压缩，减少写读数据量。
- 减少不必要的排序：并不是所有类型的 reduce 需要的数据都是需要排序的，排序这个 nb 的过程如果不重要还是不要的好。
- 内存化：shuffle 的数据不放在磁盘而是尽量放在内存中，除非逼不得已往磁盘上放；当然了如果有性能和内存相当的第三方存储系统，那放在第三方存储系统上也是很好的；这个是个大招。
- 网络框架：netty 的性能据说要占优了。
- 本节点上的数据不走网络框架：对于本节点上的 map 输出，reduce 直接去读吧，不需要绕道网络框架。

Spark Shuffle 后续优化方向

Spark 作为 MapReduce 的进阶架构，对于 shuffle 过程已经是优化了的，特别是对于那些具有争议的步骤已经做了优化，但是 Spark 的 shuffle 对于我们来说在一些方面还是需要优化的。

- 压缩：对数据进行压缩，减少写读数据量。
- 内存化：Spark 历史版本中是有这样设计的：map 写数据先把数据全部写到内存中，写完之后再把数据刷到磁盘上；考虑内存是紧缺资源，后来修改成把数据直接写到磁盘了；对于具有较大内存的集群来讲，还是尽量地往内存上写吧，内存放不下了再放磁盘。

查看原文：[腾讯大数据之 TDW 计算引擎解析——Shuffle](#)

梁定安：解密腾讯 SNG 云运维平台“织云”

作者 刘宇

SNG 是腾讯体量最大、产品线最丰富的一个事业群，其覆盖了 QQ、手机 QQ、腾讯开放平台、腾讯云平台、广点通、移动分发平台应用宝在内的多条业务线。可见 SNG 的运维体系的庞大，早在 2013 年 QCon 北京大会上，腾讯业务运维 T4 专家、总监赵建春就在 QCon 分享过 [《海量 SNS 社区网站高效运维探索》](#)，当时引起了运维界的广泛关注；而整个 SNG 的运维又是如何运作的呢？

梁定安，2009 年加入腾讯运营部，先后从事系统运维、业务运维、运维规划和运营开发的工作，目前是社交平台业务运维组 Leader，可以说是整个 SNG 云平台的缔造者，也是今年 [QCon 上海 2014](#) 大会自动化运维的讲师，届时将分享《腾讯 SNG 织云自动化运维体系》的话题。

为什么会有织云？织云重点解决什么样的问题？面对错综复杂的业务，织云又是如何自寻突变的呢？梁定安会全面介绍这个平台的特性、底层技术组成、以及给 SNG 所带来的价值。

InfoQ： 梁定安你好，织云是什么时候开始做的？

梁定安：2012年底 CTO Tony 将云化战略推广到公司内部各个 BG，织云正是在公司云战略的背景下，为 SNG 量身定做的内部云管理平台，定位是为 SNG 自研业务提供虚拟化管理和自动化运维的平台，几乎所有 SNG 的业务（Qzone、QQ 秀、QQ 相册、QQ 音乐、QQ 等等）的运维操作都基于织云平台完成。

InfoQ： 织云定位为内部的自动化运维平台，那么它具备那些特征呢？

梁定安：织云平台定义了 SNG 业务的标准化运营规范，在平台中运维人员抽象出上层的管理节点，减少与统一运维对象，降低海量运维的复杂度，得益于运营环境的标准化建设，有更多通用的自动化工具被设计开发，配合流程引擎的驱动，使我们逐步迈入自动化的运维阶段。平台最大的特色是“一键上云”帮助 SNG 自研业务快速实现织云最初的上云目标；而“自动调度”则实现标准化服务的容量自动维护；还有我们基于内核 inotify 的一致性监控，保证了

配置资源与现网资源的一致性。此外，织云的核心模块还有：资源管理，包管理，配置管理，自动流程，中心文件源，权限中心，都是自动化运维必不可缺的重要功能模块之一。

InfoQ：“一键上云”实现没有那么容易吧？非标准化业务应该是很难接入，在实现这一目标时，你们又遇到那些难题？

梁定安：困难是必然存在的，我们第一个遇到的难题是虚拟化选型和适配改造，在 XEN 和 LXC 之间，我们选择更轻量成本更低的 LXC 作为织云平台的虚拟机，在运营过程中，由于虚拟机与实体机管理模式的差异，我们没少踩坑。如同母机下子机对 CPU、网卡流量抢占造成相互影响，子机多 crontab 集中调度，常用系统命令只显示母机状态，LXC 内网 NAT 管理改变原运维习惯等等问题，都被一一啃下，LXC 顺利本土化成功。

第二个难题是运维标准化的普及，像 QQ 秀、会员这类腾讯比较早期的业务，在当时没有标准化规范的背景下，现网的运维管理有诸多阻碍顺利上云的问题点，如程序混布、hardcode IP 地址、依赖非标准库等等，在接入织云前都要经过运维和开发的适配改造。为了顺利推动让业务开发配合运维做上云的改造，我们设计了织云的“自动调度”、“一致性监控”、“权限中心”等核心功能，让开发改造的工作量有了充足的利好回报，让旧业务标准化的改造如期完成。我们乐观的看到织云平台使 SNG 运维从 D/O 分离转型为 DevOps 模式。在织云平台中，没有运维和开发的严格区分，平台的用户会在既定的标准管理框架中，对统一管理对象——模块，进行录入或变更资源配置（包、配置文件、权限、目录、脚本），流程引擎则会按照既定的次序和调用不同的自动工具，完成用户的一键上云或其他变更需求，而这一切都会在织云的标准化体系保障中自动化的实现。

一键上云的成功，得益于 SNG 自研业务的标准化运维管理的良好基础，我们所有程序的包管理（pkg 包系统）、配置文件 svn 管理（CC 系统）、目录管理（中心文件源）、权限管理（标准 api）、名字服务（L5）的高覆盖率，都可以经过轻量的改造即可成为织云平台的功能之一。再辅以流程引擎，将上云的步骤串成自动化的流程。用户只需在织云平台上管理好模块与依赖资源的关系，织云便可以一键式的完成整个迁云的过程。

InfoQ: 织云的自动调度是实现业务动态扩缩容？你们又是如何控制“雪崩”的？面对业务的大量突发，是全自动，还是人工干预？

梁定安：是的，我们认为当一个模块可以灵活的实现扩容自动化时，它便具备了跨 IDC/城市迁移的调度能力。同理，我们对运维的业务按核心功能的不同分别抽象成不同类型的 SET/服务视图（包含多个模块），当整个 SET 的标准化程度且 SET 内的模块都具备自动调度的基础能力时，我们便可以对整个 SET 进行调度操，目前 Qzone、说说、广点通等重点业务的 SET 已经具备快速调度能力。

雪崩的预防是负载均衡管理中必须解决的一个问题，在 SNG 我们拥有一款十分出色的负载均衡+容错组件 L5，L5 利用名字服务将一个集群标识成一个 L5ID，主调方可以通过嵌入 L5api 并调用 get_route 来获取被调 L5ID 中的每个 IP: port，然后当请求结束后 update_route 来更新该 L5ID 的每个 IP:port 的成功与延时信息，L5 组件便可以通过全局数据的整合，在下个调用时间片动态的为 L5ID 下的每个 IP 分配合适的请求量，利用这个原理，L5 根据实际每个被调 IP:port 的请求量、成功率和延时的波动数据，可以计算出每个 IP: port 最大可支持的请求量，当遇到业务请求量陡增的场景，L5 会启动过载保护，在保证被调方饱和的请求量前提下，对新到的请求全部拒绝服务，以防止雪崩的发生。这些都是由 L5 组件全自动实现的。

织云的自动调度原理是对服务/模块的容量指标（CPU/流量）进行监控，当触发扩容阀值时，织云后台便自动触发部署、测试、灰度、上线这一系列的全自动流程，保证线上服务容量的可用，除非耗尽可用设备 buff，否则织云的自动调度功能辅以 L5 的优秀容错能力，可保持业务容量处于高可用的水平。

InfoQ: 一致性应该是多个方面的，包括上面有提到一致性监控，还有织云的另一大特色服务一致性管理，这里的关系与具体包含的内容都有那些？

梁定安：先介绍下一致性本身，这是一套 C/S 的监控程序，C 利用内核 inotify 订阅监控的对象，并通过动态上报的架构，在一个集群内选取一个 leader 汇总数据后，传输到 S 进行数据落地，实测性能可达监控 1000 个文件秒级感知，是我们实现织云前台实时监控现网环境的核心手段。

回到一致性管理，视乎场景的不同，具体可以分为两大类：资源的一致性和服务的一致性。资源的一致性，比较容易理解，织云自动调度中依赖的资源，包括：包（进程、版本、运行状态）、配置文件（md5）、目录（md5）、权限、后置脚本（md5）。通过一致性的管理，使织云能够在无人

职守的前提下，自动的变更运营环境。服务的一致性则与 SNG 业务的形态耦合比较重，如 Qzone 的多地容灾分布，每地的服务彼此一致，主要也是利用一致性的监控管理，服务一致性管理的会比资源一致性管理纬度要粗，往往只包含多个模块的包、通用目录、权限这 3 类。

InfoQ：织云的核心模块有：资源管理，包管理，配置管理，自动流程，一致性监控，中心文件源。对于开发团队而言，在织云做这些操作，和不使用织云有什么区别？大体的操作流程是什么样子的？

梁定安：织云提供给开发和运维团队的是工作效率的提升，让我逐一为大家介绍这些核心模块的功能。

资源管理，我们对现网操作的最小管理对象——模块，部署上线所依赖的所有资源，和操作这些资源的先后次序，都将录入到织云的资源配置，并存入模块的 CMDB 配置管理数据库，成为该模块必不可少的属性之一，结合自动流程可实现多种自动化的操作。原则上模块的资源配置只能由少数模块负责人修改，并且当资源发生变更时，织云提供锁表的能力，防止交叉篡改。没有织云的资源管理，运维/开发只能依赖 wiki、文档等古老的方式记录，并且无法自动化。包管理，是 SNG 一个最基础的系统，运维要求每个上线运营的程序，都必须按照 SNG 包管理规范打包，包管理本身提供了一套完整的框架，包括：进程名字与端口管理，启停方式统一管理，标准化的目录结构，完善的进程监控体系，灵活升级回滚的 svn 管理等等。包管理是一切标准化的基础，离开它，运营环境将一片狼藉。

配置管理，指的是配置文件管理，包括线上编辑、版本迭代、diff、前后置脚本、批量下发/回滚等等功能，是单文件管理的利器。没有它的话，就只能回到脚本发布时代了。

自动流程，指的是织云的流程引擎，主要功能是将不同的标准化工具串联起来，前者的输出可作为后续工具输入用途，支持流程分支汇总的能力，可通过串联不同的工具，拼装出不同的自动化流程，实现无人职守的各种操作。这就是人肉和自动化的差别。

一致性监控，主要就是秒级感知现网变化的能力。有了它，开发再也不会提让运维批量查一批 IP 的配置是否一致的需求了。

中心文件源，主要是为目录管理诞生的一套 svn 管理的系统，可根据策略不同，自动与现网同步或被同步，且支持大批量的分发能力。没有它，大伙还是只能靠脚本来进行目录类的发布管理，效率低下。

InfoQ：在织云上的业务部署监控、日志收集是如何实现的？而运维“织云”这个平台运维工程师又是如何保证平台的稳定性？

梁定安：托管于织云上的业务使用的是 SNG 内部统一的上报通道，将监控数据和日志上报到监控平台的 storm 集群中，监控数据处理集群利用 mongoDB、rabbitMQ 和 Infobright 对大量的流数据进行处理，最终产生监控告警或报表。织云负责提升运维效率，统一监控平台则负责提供监控告警管理。

织云平台本身的可用性，也是严格按照 SNG 可运营规范要求设计的。值得一提的是，织云为了保证具备自动调度能力的模块在关键时刻的可用性，我们特别设计了演习功能，每天都随机抽取演习，随时检验平台核心能力的可靠。

InfoQ：大部人都认为运维不如研发，作为一名运维工程师，你是如何看待这两者之间的关系的？

梁定安：干一行爱一行，既然选择了运维岗位，我认为就应该热爱运维工作并努力在这个岗位上做到卓越，否则就是对个人对工作的不负责。

谈谈运维和研发的区别，任何一款互联网产品，当具备一定的运营规模时，必然会有开发和运维的明确分工。开发专注于产品功能的实现，运维则会从质量、监控、容灾、成本、安全等纬度去支持产品的发展。

认为运维不如研发的看法，往往都是看到了处于初级阶段的运维。我认为优秀的运维团队，和开发团队之间是互惠互利、缺一不可的。以 SNG 社交平台业务运维团队举例，我们会根据业务的规模，提出很多优化建议，如优化 Qzone 的分布架构，制定跨地域、机房的容灾策略，对核心服务抽象成 SET 化管理，建设 SET 调度的智能决策和执行系统；降低业务的运营成本，引入更廉价的运营方案，根据不同的场景，提出用 SSD 硬盘存储替换内存存储，用虚拟化解决长尾服务的成本压力，或者利用 buff 设备提供离线计算能力；我们还建立了大数据分析系统，为移动业务提供运维 SDK，利用机器学习能力建立外网用户反馈智能分析告警的平台；为了让运维和研发能够更及时的查阅业务运营状态，我们还开发了手机侧的运维工具 MSNG，提供了关键服

务的核心指标展示和通用的告警处理工具；为了解决告警量大的问题，我们研发了告警预处理系统，告警根源分析系统等等，这些都是运维团队为产品和研发团队输送的超越运维范畴的价值。

也许运维这个岗位不常会在镁光灯下，成为耀眼的明星团队，但是目前运维团队的价值远未被发掘完，我们仍在积极探索未来可以触及的新方向。

查看原文：[梁定安：解密腾讯 SNG 云运维平台“织云”](#)

李大维：互联网人做硬件创业容易产生的七大误解

作者 杨赛

本文根据 InfoQ 中文站编辑跟李大维的一次沟通内容整理而成。

我在[第一篇](#)分享了为什么我认为现在是互联网人去做硬件创业的好时机，但是我们也看到不少互联网背景的同学真的来做硬件的时候，遇到了很多自己没有想到的问题。在我看来，很多问题其实是源于互联网人没有把“自己应该做什么”这件事情想清楚，结果做失败了之后，又对这件事产生了错误的理解，回到互联网领域去传播了这些误解。

今天我希望能够从我的角度跟大家分享为什么我们会产生这些误解，以及我们应该怎样做。

本文也是 InfoQ 中文站《[给软件人讲硬件](#)》系列访谈的第二篇。

误解之一：做硬件是个很难的事情

前些天我在福布斯的一个讨论智能硬件的论坛上，论坛里有为数不少的人跟我们说做硬件很难，难在哪里呢？

- 招聘不到电子领域的牛。
- 电子设计耗费很多时间和资源。
- 工厂生产的良率很低。

有时候我感觉我们互联网这一代总觉得自己什么都知道，不知道的 Google 也会知道，所以跑来做硬件也不去先跟传统行业学习，结果闹出很多笑话。上面这几个难点为什么难呢？

招聘不到电子大牛，问题是电子大牛为什么要跟你干？去年我接到很多互联网朋友的电话，说要去创业做智能硬件，可不可以给我介绍一个硬件的大牛？我说你这个硬件准备投资多少钱？几百万吧。几百万怎么聘一个电子大牛？人家要么在大公司，Intel、Marvell，要么在第一线做事情，每个月出货量 kk 级。你薪水没有诱惑力，做的东西也没有挑战，你怎么找电子大牛？

为什么电子设计会耗费很多时间？因为你招聘不到电子大牛，只好往二军找。实际上二军也不肯来，最后来的都是三军、四军的，比如硕士毕业两三年的，电子设计不 debug 一阵子是出不来的，怎么可能不慢。

至于工厂生产的良率低，这个就涉及到更多，简单说来就是因为你找的画 PCB 板的、找的 ID 设计也都是刚毕业的学徒。可能你找的画板的把上面下面都放了芯片，结果过 SMT 的锅炉要过两遍，过第二遍的时候芯片就烧了。ID 设计，做了个曲面，原型打样的时候看起来还像回事，市场宣传做完了之后才去找工厂，一去，人家工厂说，你这个画的是曲面，你知道现在曲面的制成良率只有 10% 多吗？结果你在外面跟人说这个曲面的表一只卖 500 块，成本要 2000 块，又出不了货。

所以，互联网企业一开始搞的轰轰烈烈，结果精力都用来付学费了。产品出不来总要给人家一个交代，就说做硬件好难。国内每一个失败的智能硬件，基本上都有这样的一个过程。

但是做硬件真的是很难的事情吗？有意思的是，这个问题跟下面这个问题正好是相互矛盾的：

误解之二：做工厂是个没什么技术含量的活儿？

现在很多互联网人做硬件创业，总觉得自己是乔布斯一样，去人家工厂觉得做工厂的什么都不懂，他们是小生意，我们是互联网的，我拿到 500 万 VC 的钱了，看不起做工厂的。

但这里面有一个背景。过去 20 年，全世界的焦点都在做互联网的人身上，光环都在我们身上，但我们则因此忘记了我们第三产业是一个周边产业，忘记了第二产业——生产业——到底有多大。做工厂的是小生意吗？我有个做手机板子的朋友，他说自己在深圳的生意算是中上。什么是中上呢？每月出 200 万片，一片 40 美金。算一算，这是 8000 万美金的月流水，一年的流水有上亿美金。像这种规模的工厂，在深圳有很多，而且大部分都不是上市公司，不像富士康或者台湾的厂子。

回头想想，互联网公司别说一亿的美金流水，哪怕一亿的人民币流水，早就被捧上天啦。你以这个心态进去，其实没有想清楚自己跟别人老板规模的差距，这也是很多互联网人走进去之后没办法成功的原因。

深圳工厂的实力到什么程度呢？我在几个星期前见过一只 5 寸屏的工程机，非常震撼。它跟现在小米、锤子、iPhone4 的设计一样的，两片玻璃，无边框，内核是 Qualcomm，中间是结构件。可以这么说，把 iPhone5s 摆在它旁边，你会以为它是 iPhone 的下一代，iPhone5s 看起来像是上一代的。这只工程机是一个深圳工厂的老板拿给我看的，他的工厂

不做组装，但是手机里面所有的零件都做——包括贴屏。当时我问他是给谁代工的，谁知他说不是代工。

这只手机是怎么做出来的？他说三个月前有一天起床，忽然想到一个问题：“我能不能做一个自己的手机呢？”他的工厂虽然规模不小，有上万人，但他自己就是绝大多数的股东，相当于只要不影响客户，他的生产线可以想干嘛就干嘛。于是他就在公司里找了两个人，跟他们说，你们到公司上下去调动资源，做一只手机出来吧。就这样，三个月做出来了这样一部，你能说他没有技术含量？在深圳，掌握这种资源的公司其实很多。

可能互联网人还是会说工厂的人不懂互联网思维，不懂互联网。的确他们是看不懂。他们第一看不懂的是，一家一点收入都没有的公司，怎么会有 VC 疯狂的追？公司身价都 10 亿了，公司财务还没有收过钱？第二看不懂的是，做硬件这么容易的事情，为什么做不出来，为什么会没有办法出货？第三看不懂的是，像是手环这种东西到底是有谁会买，这种一年最多卖几十万件的市场到底有谁会关注？但他们一直在观察互联网，想搞明白这些究竟是怎么一回事。

误解之三：厂家不愿意做小规模的生产？

有些创业者说他们想要下几千件的单，厂家不给他们做。的确，大部分厂家不愿意做小规模，因为生产本来就是大规模的。

但是事儿也得两说。

最近法国有一个手机品牌叫做 Wiko，在法国很火，做了三年，2013 年做到 18% 的智能手机市场份额，年出货量 250 万台，预计未来两年会超过三星。你问法国人知不知道 Wiko，他会很骄傲的说这是法国的品牌。而你把它打开来，会发现里面 90% 的东西都是来自一家叫天珑的公司。

这手机是怎么来的？这几个法国人做市场调研，发现在法国，100-300 美金这一档的智能手机有空缺，没有特别的品牌，他们就决定做个公司做这一段位。他们直接跑到了深圳华强北，市面上先看看样式，然后直接跑去跟公司谈，找到了天珑。天珑一听觉得还不错，就让这个团队回法国做市场，下面的所有东西天珑包办。你想，天珑一年的出货量是一千万，200 万台对他来说也是小量，但他为什么又做了？因为他是 Wiko 90% 的股东。Wiko 是他自己的东西。

今天你一个小公司跑去深圳，摆出一副“我是乔布斯”的样子，把别人都当作外包厂，做区区五千单的量还要 cost down，当然没有人甩你。所谓不做小量，不是真的不做，而是生态圈要重组。1 亿单跟 5k 单，不能享受同样的待遇。今年一些大厂也在想怎么介入，像富士

康就宣布在北京做一个打样中心，专门服务小团队。这是未来几年需要磨合的，两边各让一步。

误解之四：智能硬件的未来在大数据？

现在很多台面上做智能硬件、智能手环的，你看他们的 PPT，都提到大数据。

在我看来，只要提到大数据的，多半没救了。为什么？你放大数据在 PPT 上面，就等于说：对不起，我不知道我的手环还能干嘛。第一，全中国一亿的人每天走几步路的数据，你没法儿卖钱的，没法儿让用户再买更多东西，这数据基本上就是垃圾。第二，最近苹果宣布 Healthkit，基本上就是说，以后所有的大数据都跟你们没关系了。你不把数据给我，就不要用我的 Healthkit。所以苹果那个数据真的很大，可以说是大数据，其他的就算了。

这是另外一个互联网过来很大的问题，在于心太大。当然变成这样也是有原因的，因为公司拿不到下一轮 VC 就死了，所以要忽悠。

未来的十年会很有趣。我们要用更正确的东西引导互联网进入智能家居，而不是被“大方向”迷惑。

误解之五：物联网将会像互联网那样统一标准，因为这是开发者想要的？

最近听到有智能硬件创业的朋友在做物联网标准统一的工作，对此我可以分享一下我看到的东西。

2003-2005 年我在日本做顾问，当时开源硬件打入日本市场，后面那家公司是做高端芯片的，他们去全世界做芯片设计的公司拿方案，然后去松下、去索尼、日立，给他们看他们需要哪一款芯片，公司如果说要，他就跑到台湾自己开店，做芯片级的供应链整合。03 年的时候他们就有一个计划要推动 ZigBee——ZigBee 当时刚好成为 IEEE 的标准，所以大家都觉得很兴奋：智能家居要一统江山了。结果销售团队去松下讲 ZigBee 的好处，介绍了几个方案，松下很有兴趣，觉得 ZigBee 很好，我们想要，但是有一个条件：我要一个跟索尼不相容的 ZigBee。

这是兵家必争之地，没有人会愿意跟竞争对手的品牌相容。索尼也是一样，要跟松下不相容的 ZigBee——这可是 IEEE 的世界标准哦。其实互联网也是一样，你想整个 W3C 花了多少时间写 SOAP？谁管他啊，最后是一群野人跑来用 JSON，大家最后用 JSON 了。这就是现实，那就是我们如果需要连接，无论用什么方法总是能实现；如果不希望连接，标准委

员会的促进工作也起不到什么作用。兼容可以很贵，也可以很便宜；但是在没有利益的时候，没有人会去玩、去认真的做这件事。

中国有一个智能家居标准委员会，美的、海尔，反正苏宁逛一圈你能够看到的所有品牌都坐在里面。而这个委员会的目标，就是确保中国智能家居的标准永远出不来。你这些“委员”过去，是代表大电器公司的，而大电器公司文化非常重，所有东西的存在都是为了集团的利益：冰箱要照顾空调，空调要照顾洗衣机，大家要互相保护。公司内部是自己人，外面是竞争对手，你要是出去说自己跟别家的可以相容，回到公司人家要说你是叛徒。所以要去开会的人，目的就是要确保这个会议流产。

那么智能家居的互联互通最后可能会怎么达成呢？一线工厂是不可能的，我觉得会是源自于互联网公司跟二线厂商合作、生产以互联网为标准的家电。中国现在的二线厂很多是ODM，对他来讲，是否挂牌他不 care，只要有钱收就行。这样合作如果能推出体验更好的电冰箱，体验更好的洗衣机，还是很有前景的。其实现在的小米电视、乐视 TV，不就是互联网公司跟电视厂合作吗？这个模式已经有了，互联网公司做上层软件、品牌、市场、销售，二线厂做生产和售后。

你其实也不可能指望去抱一线工厂的大腿，因为你那几千台的量他们才不会做。而因为市场小，他们也不会跑来跟你竞争，而小厂则会跟你合作。你的上层软件可以让洗衣机的体验更好，洗衣机核心由二线厂提供，外壳可以到昆山打，团队可能就需要 10 个人，这样造一台洗衣机可能成本也就两千。这样一个制造背景的智能电视、智能洗衣机、智能冰箱，要实现互联互通的可能性更大。

但是有两点你要想好了：

第一，手机能够跟路由交流，能够跟洗衣机交流，能够跟冰箱交流，又怎么样？

第二，开发者不会为你的智能硬件开发应用。你卖硬件，你变现了，他帮你开发应用，他又卖什么？所以开发者会去做自己的智能硬件，卖自己的智能硬件。

关于这第二点我还想延伸说一下。

为什么 Google 的 Android Wear 在深圳根本没人 care？因为人家三年前就把 Android 做到手表上了。对于“智能硬件开发平台”这个事情，我可以看到的是在未来三个月，更有可能发力的一家公司是 MTK。他们现在有一个叫做 Linkit 的可穿戴式平台，这个平台的资料现在已经在深圳上百家 IDH 的手里面，包括 20 个不同的型号、几百片不同的公板，小到可以做手环，大到可以做高端 4G 通讯。你只要写 Linkit 的程序，未来你要做自己的硬件，只

需要选一块公板就可以，非常方便。这个东西现在还没有公开，我相信他幕布掀开的那天，就是软件人的好时代来临了。

如果你想做智能眼镜，还可以关注一下另外一家做高级公板的公司 Jorjin，他们也会来这次 QCon 上海。现在市面上你能看到的 Glass，除了 Google Glass 之外都是他们 ODM 的。他们会在 9 月宣布他们的 SDK，而这跟 Google 不同哦。Google Glass 是，你把 1500 美刀的眼镜买回去，Google 说我让你做的你才能做，不给你的你就别想做。而 Jorjin 他们的 SDK 是这样的：你看这是 MCU，这是 Wifi 模块，这是光学的部分，上面是 Linux，所有的代码都给你拿回家，想做什么就做什么吧！你需要的只是找人帮你做一个眼镜框。

误解之六：我妈妈会不会用你这个智能硬件？

现在这个想法好像还挺流行，那就是“我妈妈不会用的智能硬件是没什么前景的”。

对此我只能说，我做互联网 20 年，我妈妈一直到 5 年前才弄明白我到底是做什么的。在前面的 15 年，如果人人都问这个问题，那互联网早死了。

误解之七：做硬件要像乔布斯学习？

从现在台面上做硬件创业的，不难看出乔布斯的影响：无论是发布会的风格，还是自己从电子设计开始找大牛做硬件的想法，都在像乔布斯看齐。

我想这可能是因为互联网人要做硬件创业，但是 Google 上能找到的教材只有一本《乔布斯传》的原因吧。但这样一来，可是大大走了弯路。

就好比你要去做一个网站，第一件要做的事情是什么？无论如何不会是重写一个自己的 Web 服务器吧。那为什么你做网站的时候不会这样，跑来做硬件的时候却先来做这个事情呢？又有哪个网站会出来说，我们的竞争力在于我们有全世界最好的 Web 服务器？没有吧。那为什么你做网站的时候不这样，跑来卖硬件的时候又做这个事情呢？

再说了，就算是学习乔布斯，也没有学到位。苹果的硬件并不是设计师说怎么设计富士康就怎么做，实际上他们的设计师基本上都在富士康驻场，出来的设计可以直接试产测试良率，良率不好要拿回去改。这是一个互动的过程，不是说把前面的事情都做好了才拿给工厂。

未来 10 年的智能硬件是软件的天下，但可千万要想得开，发挥自己在软件上、在互联网思维上、在互动设计上的优势，在电子、硬件设计上找到好的伙伴，以良好、平等的合作心

态谈合作。中国目前在手机应用上领先世界，加上世界上最大的生产体系，未来智能硬件的前景是光明的！

分享者简介

李大维（David LI），从 1990 年起开始参与开源运动，曾经是免费软件基金会（Free Software Foundation）成员、Objectweb 董事会成员并参与 Apache 项目运作。在过去 20 年，David 发起并过众多开源软件项目并服务其中。2010 年他创立了新车间，中国第一个创客工坊，并在全国开启追随创客文化的风潮。近年来李大维又开始关注城市农业，并探究如何将开源精神融入到农业和园艺之中。在 2014 年 10 月 16-18 日的 [QCon 上海大会](#) 中，李大维将作为智能硬件专场的出品人，为大家展示领域内最核心的那一批人都在做些什么。

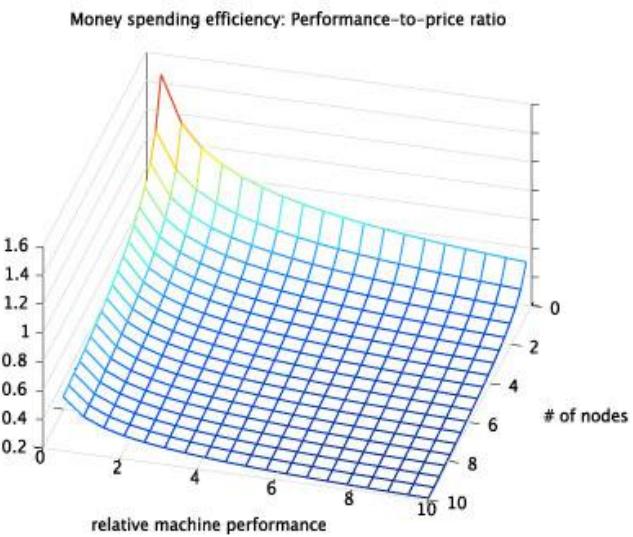
查看原文：[李大维：互联网人做硬件创业容易产生的七大误解](#)

可伸缩 NoSQL 数据库的五条建议

作者 张逸

Bigstep's Full Metal Cloud 公司的产品经理 Alex Bordei 分享了他们在 NoSQL 使用上关于可伸缩性的经验。Alex 给出了如下五条建议：

1. 永远不要假设规模是线性增长的。虽然硬件是可伸缩的，但软件并不能百分百地能利用硬件资源，却需要为这些用不到的硬件资源买单。因而，Alex 建议要寻找到硬件成本与硬件能力的切合点。下图展示了水平伸缩与垂直伸缩方面的性价比趋势：



2. 相信测试而非文档。Alex 告诫说，不要信任供应商的文档，虽然研读这些文档非常有用，但最好还是基于自己系统的情况对 NoSQL 进行测试。由于虚拟化和云技术使得搭建平台变得非常容易，因此不要找借口说没条件进行测试。
3. 体察细节：内存与 CPU 的度量数据。尤其对于内存数据库而言，内存的性能直接影响着数据库的性能。CPU 的指标同样需要引起重视。只有准确地获得这些硬件资源的使用效率，才能将钱花在刀刃上。

4. 不要忽略网络延迟。只要数据库是分布式的，必然需要占据网络带宽。无论是节点之间的通信，还是对数据建立副本，网络延迟以及吞吐量都直接制约着整个系统的性能。如果网速太慢，即使内存与 CPU 再好，对系统性能的改善也是杯水车薪。
5. 不要对 NoSQL 数据库做虚拟化。虚拟化是个好东西，但也得量力而为。由于它会影响到内存访问速度，而这一点对于 NoSQL 数据库而言却又至为关键。根据 Alex 的观察，对比虚拟环境，运行在纯硬件环境上的管理程序性能要提示 400% 左右。

感谢郭蕾对本文的审校。

查看原文：[可伸缩 NoSQL 数据库的五条建议](#)

架构师

www.infoq.com/cn/architect

每月8号出版



封面植物——白檀仙人掌



白檀仙人掌具有长长的鲜绿色的茎部，在茎上布满了短刺，是一种非常容易长出子球的仙人掌。子球生在较长的茎上，短短的就像花生一样，因此也称为花生仙人掌。这些子球在一般栽培下是很容易繁生的，因此健康的白檀仙人掌都是一丛丛地生长的。虽然繁茂的植株在一年四季都可以看到，没什么稀奇，但是每年在春夏季节才开的花才正是栽培者所期待的，白檀仙人掌的花朵呈漏斗状，鲜红色的花朵都是侧生的，花径约 4 厘米左右，只要数朵花就能将植株完全遮盖住。该属的仙人掌，无论是原生种还是园艺杂交种都具有相当不错的耐寒性，在 0℃ 下亦不会有损害，而且若经过冬天低温处理，会使花开得更加灿烂。

栽培：白檀仙人掌生性强健，栽培容易，喜阳光充足、通风良好的环境。生长季节可充分浇水。冬季低温休眠期，宜保持盆土干燥，可耐 1~2° C 低温。盛夏高温时节，需适当遮阳并注意通风，以预防红蜘蛛危害。

繁殖：白檀仙人掌极易孽生子球，可摘取子球扦插，成活率高。也可将子球嫁接在量天尺上，生长良好。

促进软件开发领域知识与创新的传播

架构师

ARCHITECT



本期专题 | Topic

特别专栏 | Column

腾讯大数据之TDW计算引擎解析
Shuffle

梁定安：解密腾讯SNG云运维平台“织云”

推荐文章 | Articles

看板如何奏效
理解Spark的核心RDD
缓存一致性入门**InfoQ**

2014年10月

架构师 10月刊

每月 8 号出版

本期主编：杨赛

流程编辑：丁晓昀

发行人：霍泰稳

读者反馈/投稿：editors@cn.infoq.comInfoQ 中文站新浪微博：<http://weibo.com/infoqchina>商务合作：sales@cn.infoq.com**本期主编：**杨赛，InfoQ 高级策划编辑

对软件开发、系统运维、互联网产品、移动技术、项目管理等方向感兴趣。写过一点 Flash 和前端，现在只是个伪码农。曾在 51CTO 创办了《Linux 运维趋势》电子杂志，偶尔也自己折腾系统。曾混迹于英联邦国家，学过物理，做过一些游戏汉化，练过点长拳，玩过足球、篮球、羽毛球等各类运动和若干乐器。喜欢读《失控》。