个性化你的阅读

编程狂人

Programming Madman

NO.22

推酷

关于推酷

推酷是专注于IT圈的个性化阅读社区。我们利用智能算法，从海量文章资讯中挖掘出高质量的内容，并通过分析用户的阅读偏好，准实时推荐给你最感兴趣的内容。我们推荐的内容包含科技、创业、设计、技术、营销等内容，满足你日常的专业阅读需要。我们针对IT人还做了个活动频道，它聚合了IT圈最新最全的线上线下活动，使IT人能更方便地找到感兴趣的活动信息。

关于周刊

《编程狂人》是献给广大程序员们的技术周刊。我们利用技术挖掘出那些高质量的文章，并通过人工加以筛选出来。每期的周刊一般会在周二的某个时间点发布，敬请关注阅读。

本期为精简版 周刊完整版链接：

http://www.tuicool.com/mags/535db616d91b140c0000047d

欢迎下载推酷客户端
体验更多阅读乐趣

干嘛不在企业中使用Node.js呢?

译者：@andrewleeson

(注* 其实有很多新生的技术或工具是很优秀的，但很多人都不敢尝试，特别是企业不敢在自己的项目中使用新技术。新技术有很多优势的地方，当然也会因为新出的原因 而有一些漏洞，但我们应该正确的面对，根据自己项目的需要选择更好的技术，而不是一味的用那些陈旧繁琐的老技术，好技术都是在使用中不断完善的！本文作者 通过一些例子说明了新老技术取舍方面的问题)

我的女儿14岁。像她父亲一样，她热爱音乐且爱玩音乐。与所有其它叛逆的青少年一样， 为看格莱美奖而坐几个小时，但其实并不是热衷于整个表演，仅仅只是其中几个而已。因此她将我作为她的信息推送服务。我的任务就是坐在沙发 上（手上拿着ipad，我不会傻到几个小时一样盯着电视看），当她感兴趣的表演到来时提醒她。需要我提醒她的表演明星有：洛德, 林赛白金汉, 特伦特· 雷泽诺。

在她的房间，她在五斗柜上创建了一个“神殿”，它由碎南瓜乐队的梅隆牧羊犬和无限悲伤专辑， 佛利伍麦克乐队的精 选集，所有吸血鬼周末乐队的CD, 洛德的纯粹的女主角，西尔维娅·普拉特的钟形罩（书本，不是CD）的CD盒子组成。几年前她就已经退出了披头士队伍 了，如果不是关于约翰和保罗，那就没有什么重要的。

同时，我17岁的儿子喜欢听着莫扎特的《安魂曲》来玩多人在线射击游戏。而有时候他又会顺序地听赶时髦乐队的所有音乐。除了权力的游戏（电视名），他的床头时间阅读一些波斯、埃及最昏暗时期编年史的组成和一些很少听说的中国历史。

对于我的孩子来说，时间并不值钱，只有高品质的时光才值钱。对孩子们来说，.莫扎特，史蒂薇·尼克斯，大卫加恩，斯拉·柯尼希以及洛德才是他们值得的时光。

这将我带向了本文的主题。因此一个来自我老家隔壁的同事这样评论我的文章：

@dglozic 博客上有趣的#node.js文章：dejanglozic.com/tag/node-js/ 。并不是全新的东西， 但有些Node在企业方面的观点。

Zlatko Đurić (@zladuric)2014年3月18日

有 多种方式看待node.js吗？有用“startup-y“方式看待node.js的吗？有用”enterprise-y”的角度来看待node.js的 吗？是不是觉得前者更酷，后者是更让人讨厌，成年化，”享受它所有的乐趣“的方式？更重要的是，当创业公司和民营小公司用新鲜光亮的技术时，企业是否应该 只用使用时间较长的，被证实的技术。

我在IBM工作，所有IT行业里最大最古老的企业（已经103年的历史了）。我甚至应该被允许写、把玩且推进（上帝拒绝）Node.js在IBM中的发展，更不必说让其成为 JazzHub的新微服务架构的中心环?

让 我们来看看我的孩子们是如何接近它的：因为他们还小，没有思维定势，他们可能会在前端完全使用Node.js，因为对乏味的事他们没有多少耐心 ，且 Node.js能允许他们在重复构建接口的时候更快。如果他们有一些高性能的任务需要实现，他们会毫不犹豫的用Java来写，或要求更高的话用C或 C++，或他们可能会考虑GO语言。他们将会增加i18n的支持，以便他们能给英语不是很好的亲戚展示他们在做的东西。他们会增加安全机制，因为当站点上线时，一些无聊的黑客会攻击它（我女儿9岁时，她的的Club Penguin 账号被攻击过很多次）。他们可能会做所有的这些事情，因为这些都是常识，他们很重要，他们很要紧且如果没有他们你不能说你已经完成了。

当 你有客户且为他们负责任的时候，你在做”enterprise-y”事情。NPM刚刚成为一个公司，你知道他们的第一笔订单是什么吗？订购外部安全审计。

对我来说这非常“企业化”。Uber商业模式你听起来可能很怪，但它是一个很严肃的商业（严肃到他们能在其扩展的市场上通过说客来创建一个pushback）,且他们从2011年开始就在他们的高度系统上使用Node.js。AirBNB同样也是如此，虽然他们是2013年才开始加入到 Node.js队伍中的。

我用这两个公司作为例子是因为他们比较新，且没有经典“企业化”的固定模式。然而，请仔细看看他 们的站点——包括用下列菜单选择语言。我可以解释一个经典的笑话，将“你知道你是一个乡巴佬.....”解释为“当你需要处理i18n时你知道你在编写 一个企业类别的软件”。当然，如果你是一个想给美国政府销售产品的企业，这并不是一个张力目标——你必须遵守规定508，甚至是在运行时。欧洲委员会也有 同样的需求。但是通过一个政府坚持要添加i18n且能访问你的商业web应用，你没有必要被打败——只因它有道理，因为通过此你可以接触到更多的人。而且 这也不只限于node.js——仔细检查广受欢迎的BootStrap工具箱例子和搜索“aria”单词——你会发现43条记录，因为所有的组件都支持 wai-aria访问。

现在我们进入到Node.js年轻生命最激动人心的阶段——企业应用中的node.js。到处都有迹象表明，从2013的节点峰会和2014的节点日期间推荐的公司用node.js重新改造他们的系统到企业微服务架构中文章和讨论的巨浪。

最 近几周我已经看到node.js和企业语气方面的显著变化。在过去，讨论可能会因为还未找到平衡点的结论而更激烈，好像前面的node.js营有一点点不 安全，因此一般的论述会是这样的“node.js已经为企业准备好了吗？”。现在，来自由快速成熟的重要的成功故事所构成的Node.js社区支持的语气已经转移了。讨论的问题再也不是Node.js是否已经为企业准备好了——Node.js已经在企业中，且这个变化给Node.js提供了最重要的成长。 在某种意义上，现在论述应该是“我们怎么会讨论这个问题的？”，之前问题的答案已经在很多人的脑海中了，因此他们都响亮的回答“是啊，我们怎么会讨论这个 问题的！”。现在人们已经忙着享受node.js的乐趣了。

有时候，有历史意义的意见会很有帮助，这就有个例子。在 1995年，Sun公司以和C++在桌面应用程序上二分天下的战略将Java和JVM公布于世。它是解释型的，它速度慢，它有很多BUG，但它是全新的， 令人兴奋的，承诺“一次编写，到处运行”，没有重新编辑，它社区的人数增长得很快。1996年，我被要求尝试着写一个允许IBM中间件开发工具靠一个有效 的GUI在AIX，OS/2和windows NT下运行而不必再维护三份不同的代码的框架的原型。我用Java写了且叫做JFace。这个框架最终以更 长的代码（虽然完全重写）而结束，后来以Eclipse平台而出名。

这个故事的重点是在我被要求用Java解决问题 时，Java才面世2年，且Eclipse平台的巨大成就取决于Java,此时Java面世4年。那时，Java有很多的BUG,而且比现在的 node.js要慢。实际上，Java一直比其实语言慢，然而当node.js被用于设计时（有重要的I/O活动的系统），能提高性能。我敢肯定，相同时 期人们对Java也有相应的疑问，但这并不能阻止他们不断前行，解决掉这些疑问。

某种意义上，企业中的一些人就像是有着疯狂的青春年华，然后忘却它，将自己变得很守旧的父母。正如上面例子中说的，我们曾经很大胆，我们可以再次放开胆。5年后，一些新的东西会出现，然后我们还要再次讨论这样的问题：

你的代码不会被后代称赞，会被斥责。

——Robert King (@hrobertking) 2014年2月28日

现 在任何Node.js的讨论不会成为“enterprise-y”——我们有伟大的HWPS（每秒Hello World次数）的数字，现在需要解决像 i18n，安全，规模，大型开发团队的微服务独立版本控制，服务间的消息传递，集群，持续集成，零停机时间的应用部署等这些真实的问题。让我们像我孩子们 对待音乐一样来对待它吧——开放思想，没有偏见，没有先入为主的观念，只有优点。

Node.js不过只是工具箱中的一个工具而已，有目的的使用它会给你带来很大的好处。感受新生，再次热爱你的工作，很乐意地动手，代码会因为你开放的思想而免费地飞到你的口袋中。

译文链接：http://dejanglozic.com/2014/03/24/node-js-and-enterprise-why-not/?utm_source=ourjs.com

原文链接：http://ourjs.com/detail/5356a495ed9add0e2600000a

CSS 最核心的几个概念

作者：@GeekPlux

My brain has two parts: left & right. My left brain has nothing right, my right brain has nothing left.

本文将讲述 CSS 中最核心的几个概念，包括：盒模型、position、float等。这些是 CSS 的基础，也是最常用的几个属性，它们之间看似独立却又相辅相成。为了掌握它们，有必要写出来探讨一下，如有错误欢迎指正。

元素类型

HTML 的元素可以分为两种：

- 块级元素（block level element）
- 内联元素（inline element 有的人也叫它行内元素

两者的区别在于以下三点：

1. 块级元素会独占一行（即无法与其他元素显示在同一行内，除非你显示修改元素的 display 属性），而内联元素则都会在一行内显示。
2. 块级元素可以设置 width、height 属性，而内联元素设置无效。
3. 块级元素的 width 默认为 100%，而内联元素则是根据其自身的内容或子元素来决定其宽度。

最常见块级元素应该是 吧，内联元素有 <a> <img> 等等，完整的元素列表可以谷歌一下。

具体来说一下吧,

```
.example {
    width: 100px;
    height: 100px;
}
```

我们为 设置上面的样式，是有效果的，因为其是块级元素，而对 设置上面的样式是没用的。要想让 也可以改变宽高，可以

通过设置 display: block; 来达到效果。当 display 的值设为 block 时，元素将以块级形式呈现；当 display 值设为 inline 时，元素将以内联形式呈现。

若既想让元素在行内显示，又能设置宽高，可以设置：

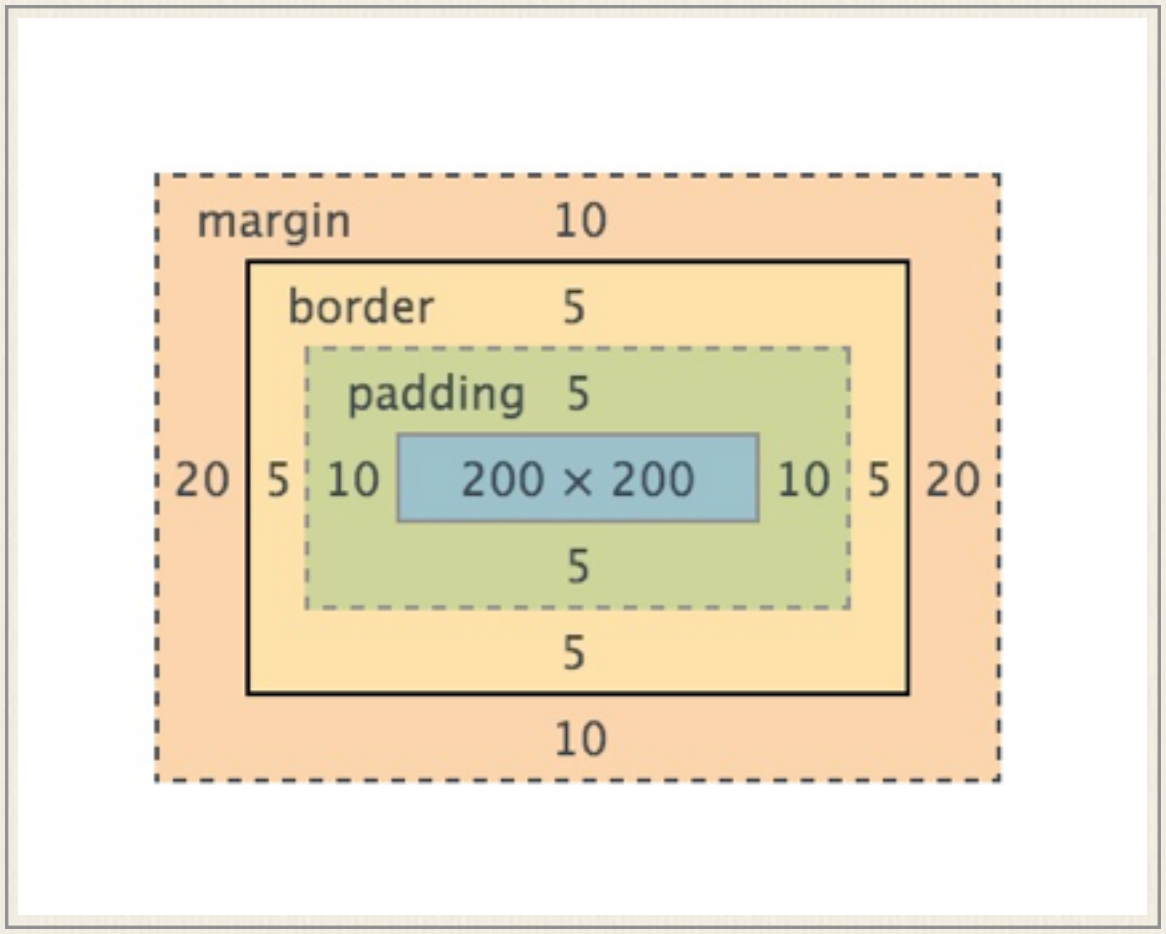
```
display: inline-block;
```

inline-block 在我看来就是让元素对外呈内联元素，可以和其他元素共处与一行内；对内则让元素呈块级元素，可改变其宽高。

HTML 代码是顺序执行的，一份无任何 CSS 样式的 HTML 代码最终呈现出的页面是根据元素出现的顺序和类型排列的。块级元素就从上到下排列，遇到内联元素则从左到右排列。这种无样式的情况下，元素的分布叫普通流，元素出现的位置应该叫正常位置（这是我瞎起的），同时所有元素会在页面上占据一个空间，空间大小由其盒模型决定。

盒模型

页面上显示的每个元素（包括内联元素）都可以看作一个盒子，即盒模型(box model)。请看Chrome DevTools 里的截图：



可以显而易见的看出盒模型由 4 部分组成。从内到外分别是：

```
content -> padding -> border -> margin
```

按理来说一个元素的宽度（高度以此类推）应该这样计算：

总宽度 = margin-left + border-left + padding-left + width + padding-right + border-right + margin-right
但是不同浏览器（你没有猜错，就是那个与众不同的浏览器）对宽度的诠释不一样。符合 W3C 标准的浏览器认为一个元素的宽度只等于其 content 的宽度，其余都要额外算。于是你规定一个元素：

```
.example {
    width: 200px;
    padding: 10px;
    border: 5px solid #000;
    margin: 20px;
}
```

则他最终的宽度应为：

宽度 = width(200px) + padding(10px * 2) + border(5px * 2) + margin(20px * 2) = 270px;
而在 IE（低于IE9） 下，最终宽度为：

宽度 = width(200px) + margin(20px * 2) = 240px;
我个人觉得 IE 的更符合人类思维，毕竟 padding 叫内边距，边框算作额外的宽度也说不下去。W3C 最后为了解决这个问题，在 CSS3 中加了 box-sizing 这个属性。当我们设置 box-sizing: border-box; 时，border 和 padding 就被包含在了宽高之内，和 IE 之前的标准是一样的。所以，为了避免你同一份 css 在不同浏览器下表现不同，最好加上：

```
*, *:before, *:after {
  -moz-box-sizing: border-box;
  -webkit-box-sizing: border-box;
  box-sizing: border-box;
}
```

这里还有两种特殊情况：

- 无宽度 —— 绝对定位（position: absolute;） 元素
- 无宽度 —— 浮动（float） 元素

它们在页面上的表现均不占据空间（脱离普通流，感觉像浮在页面上层一样，移动它们不影响其他元素的定位）。这就涉及到另外两个核心概念 position 和 float。

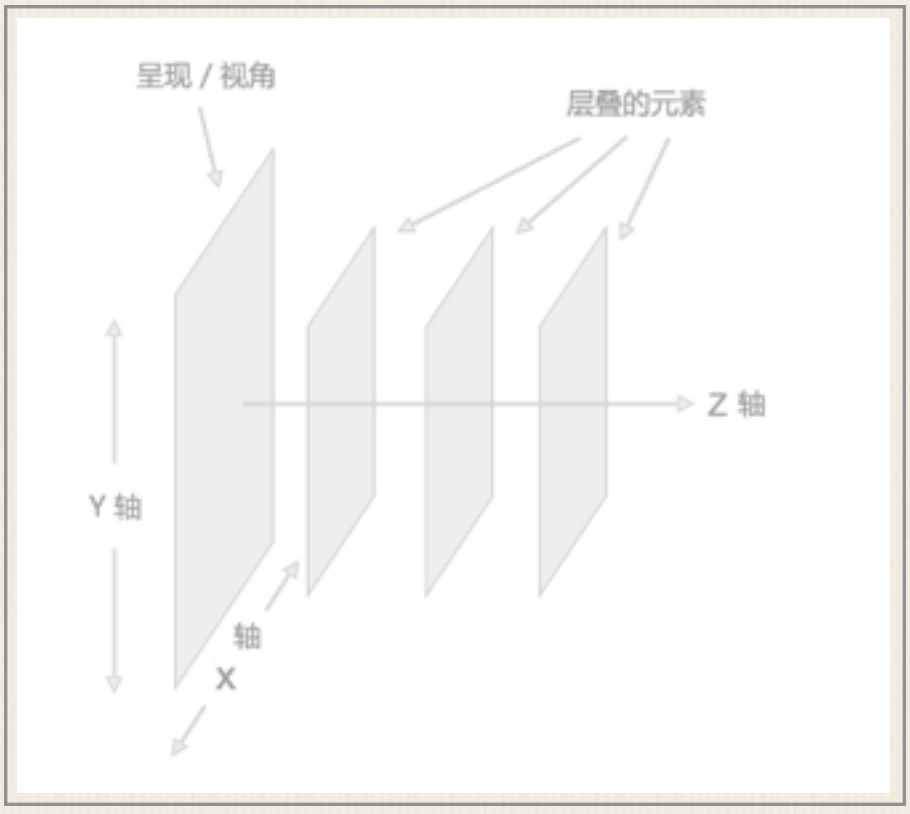
position

position 这个属性决定了元素将如何定位。它的值大概有以下五种：

position 值	如何定位
static	position的**默认值**。元素将定位到它的正常位置（上文提到过），其实也就相当于没有定位。元素在页面上**占据**位置。**不能**使用 top right bottom left 移动元素位置。
relative	相对定位，相对于元素的正常位置来进行定位。元素在页面**占据**位置。**可以**使用 top right bottom left 移动元素位置。
absolute	绝对定位，相对于*最近一级的* **定位不是 static 的**父元素来进行定位。元素在页面**不占据**位置。 **可以**使用 top right bottom left 移动元素位置。
fixed	绝对定位，相对于**浏览器窗口**来进行定位。其余和 absolute 一样，相当于一种特殊的 absolute。
inherit	从父元素继承 position 属性的值。

具体效果可以参考[w3school的实例]，或者自己写一下就明白了。

每个网页都可以看成是由一层一层页面堆叠起来的，如下图所示。



position 设置为 relative 的时候，元素依然在普通流中，位置是正常位置，你可以通过 left right 等移动元素。会影响其他元素的位置。

而当一个元素的 position 值为 absolute 或 fixed 的时候，会发生三件事：

1. 把该元素往 Z 轴方向移了一层，元素脱离了普通流，所以不再占据原来那层的空间，还会覆盖下层的元素。
2. 该元素将变为块级元素，相当于给该元素设置了 display: block;（给一个内联元素，如 ，设置 absolute 之后发现它可以设置宽高了）。
3. 如果该元素是块级元素，元素的宽度由原来的 width: 100%（占据一行），变为了 auto。

由此观之，当 position 设置为 absolute 或 fixed，就没必要设置 display 为 block 了。而且如果你不想覆盖下层的元素，可以设置 z-index 值 达到效果。

float

float 顾名思义，就是把元素浮动，它的取值一共有四个：left right none inherit，光看名字就懂了，无需多言。

最初的 float 只是用来实现文字环绕图片的效果，仅此而已。而现在 float 的应用已不止这个，前辈们也是写了无数博文来深入浅出的讲解它。

浅如：

经验分享：CSS浮动(float,clear)通俗讲解 篇幅不长，通俗易懂，可以看完这篇文章再回过头来看本文。

深如：

CSS float浮动的深入研究、详解及拓展(一)

CSS float浮动的深入研究、详解及拓展(二)

从本质上讲解了 float 的原理。

我就不班门弄斧写原理了，只说说 float 的几个要点就行了：

1. 只有左右浮动，没有上下浮动。
2. 元素设置 float 之后，它会脱离普通流（和 position: absolute; 一样），不再占据原来那层的空间，还会覆盖下一层的元素。
3. 浮动不会对该元素的上一个兄弟元素有任何影响。
4. 浮动之后，该元素的下一个兄弟元素会紧贴到该元素之前没有设置 float 的元素之后（很好理解，因为该元素脱离普通流了，或者说不在这一层了，所以它的下一个元素当然要补上它的位置）。
5. 如果该元素的下一个兄弟元素中有内联元素（通常是文字），则会围绕该元素显示，形成类似「文字围绕图片」的效果。（可参考CSS

float浮动的深入研究、详解及拓展(一)中的讲解）。这个我还是实践了一下的，点这个JSfiddle看看吧。

6. 下一个兄弟元素如果也设置了同一方向的 float，则会紧随该元素之后显示。

7. 该元素将变为块级元素，相当于给该元素设置了 display: block;（和position: absolute; 一样）。

这里还有个东西，就是广为人知的——清除浮动。具体的方法五花八门，可以看这篇：那些年我们一起清除过的浮动，我就不多说了。

写完本文后，脑子中又出现了一系列问题，假如 position 和 float 同时设置会出现什么问题？兼容性如何？哪个属性会被覆盖？还没来得及实践，改天以排列组合的方式看看到底是什么效果……如果有人实践过可以偷偷告诉我 ^_^

原文链接：http://jianshu.io/p/3a18fcd9fcda

CSS3实现鸡蛋饼饼状图loading等待转转转

作者：@张鑫旭
网站：http://www.zhangxinxu.com
我其实本想做个生物学者的

一、我又来了，你在哪里

hello, 时隔一日，我又露面了，是否有如隔三秋之感啊~

本文为之前“CSS3 animation渐进实现点点点等待提示效果”一文姐妹篇，虽说姐妹，但差异较大。

前文带有实验性质，有些自娱自乐；这里的实现就是真刀真枪项目要使用的，且难度上了好几层楼。

//zxx: 前文part3藏有对animation各个参数的简单释义，好学的盆友可以瞅瞅。还有伪进度条动画模拟哦~

二、山东杂粮煎饼转鸡蛋效果

据说，山东杂粮煎饼是IT屌丝男必备早餐佳品。

话说，以前我也老吃了，尤其师傅那个把鸡蛋转转转平铺在饼上的手法，让我看得啧啧称赞！

今天，要实现一个长任务等待提示效果。然后设计师就把做好的gif效果图给我，就是下面这个：

按照大众做法，我应该是把图片直接按图索骥，调调布局，然后早早回家抱老婆。

但是，我这个人，天生不安分守己。想到是用在客户端，客户端又是用的webkit内核，于是，立马决定使用CSS3来折腾一番。

外面的光环很好实现，360度转转转就OK. 但是，中间那个鸡蛋转转转的可不是好啃的骨头啊。人家师傅饼前一分钟，饼下十年功啊。

我生小辈想要习得这转饼的精髓，可得要好生琢磨一番啊！

三、看我转的煎饼效果

如果您手头的是IE10+这类支持animation的现代浏览器，您可以狠狠地点击这里：[CSS3饼状图loading旋转动画demo]

截图是死的，de-
mo是活的。建议点上面的地址去仔细对比CSS3实现和gif动画效果。

不难发现，这个gif尺寸又大，效果也不流畅，还烧性能。相比之下，立马被CSS3实现甩出了2条南京路。

CSS3效果更佳、性能更高、资源占用更少大家都认同了。关键是，这个大饼它是怎么转起来的?

略叼略复杂。

四、蛋饼旋转技能传授

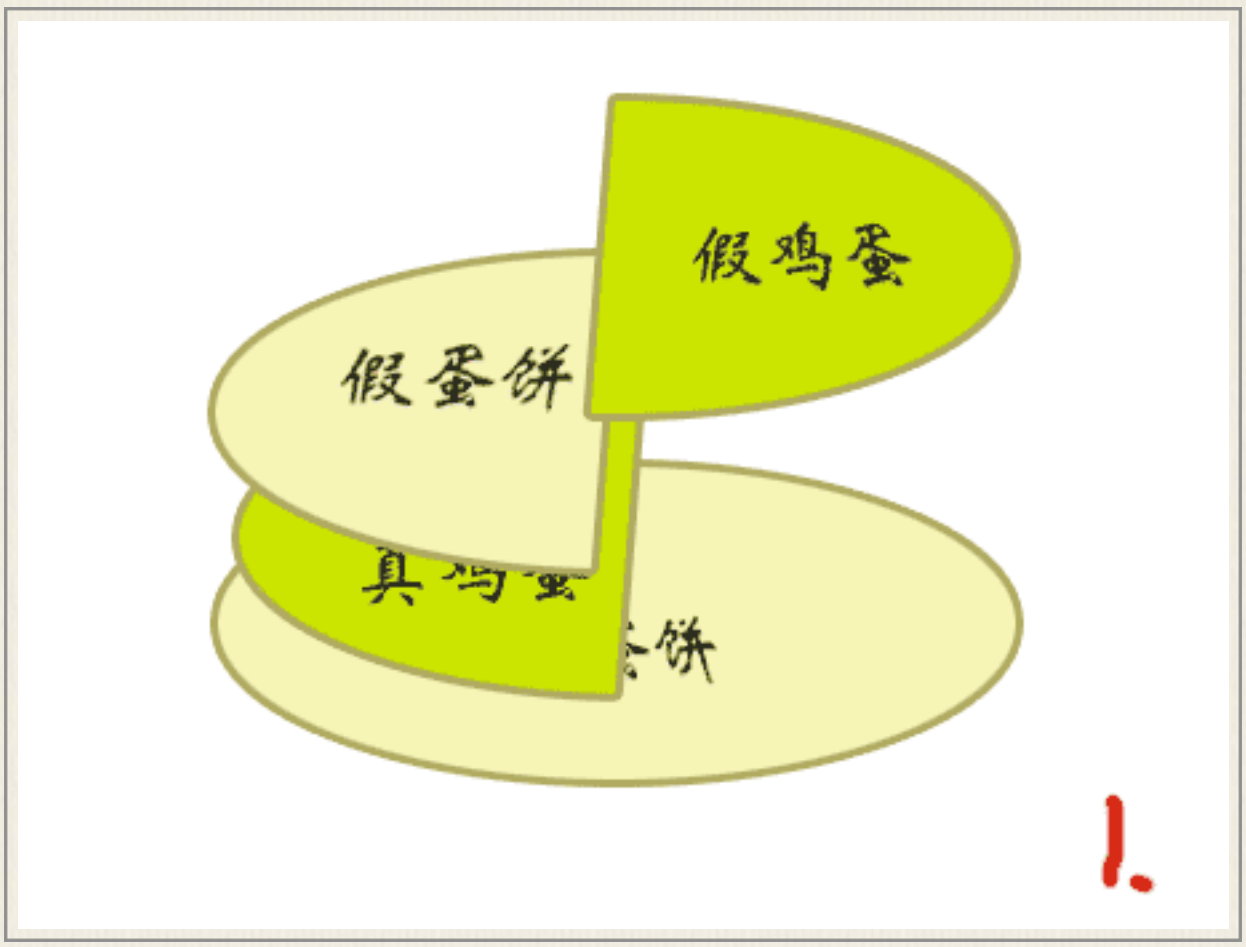
师父领进门修行在个人，听不懂我也没办法啦~~

我们肉眼看上去是一个鸡蛋被摊在了整个饼上，实际上，这只是个障眼法。

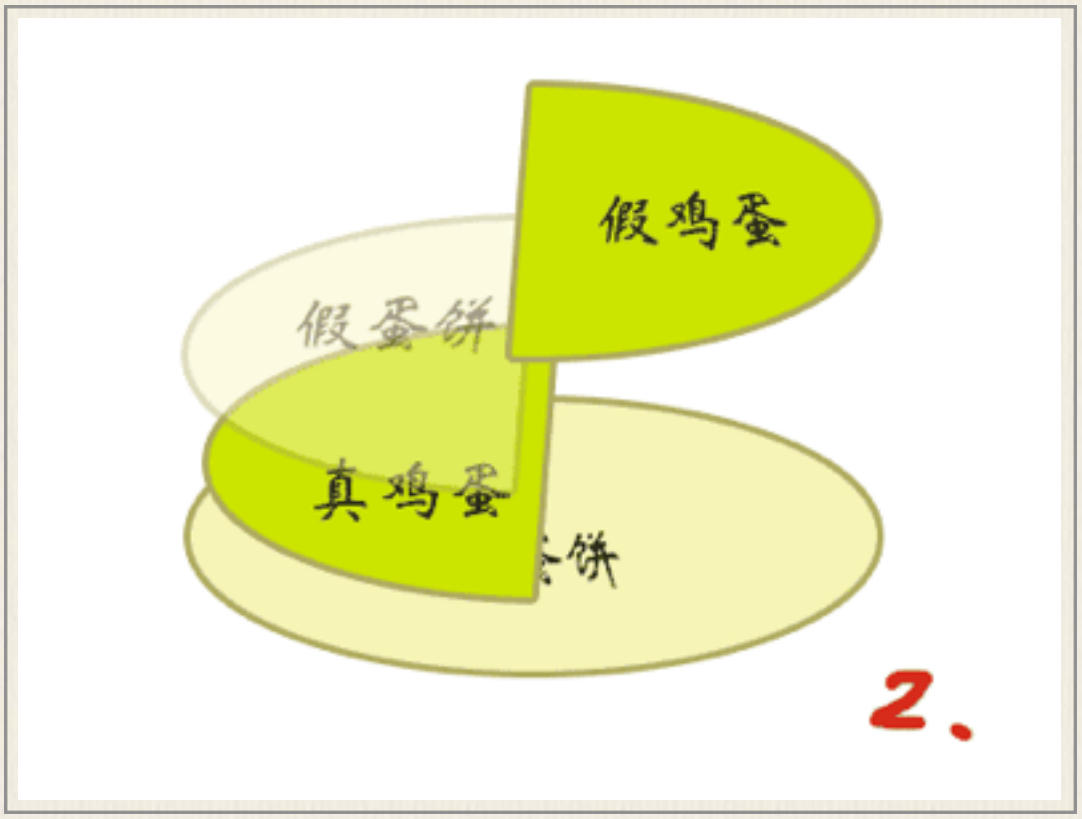
实际摊的鸡蛋，只有半个饼那么大。还有半个饼位置是长得像鸡蛋的假鸡蛋和长得像大饼的假大饼。显然这句话你听不懂，因为我自己都没听懂，哈哈~~所谓一图胜千言，示意图走起~

下图示意的就是鸡蛋饼上的鸡蛋从100%变小成0的过程。

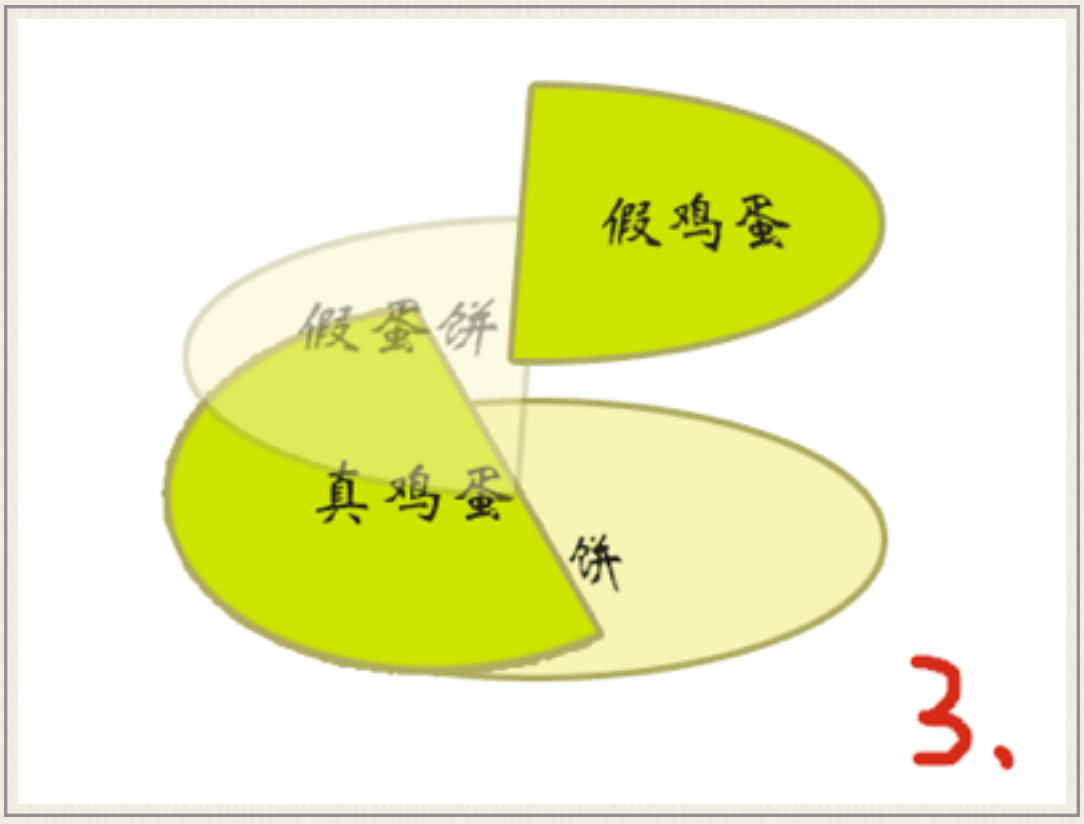
1. 这是没有干扰的蛋饼，你看到的是半个假饼和半个假蛋。



2. 当我们煎饼动画转起的一瞬间，我们让假的饼子隐藏回家打酱油去。于是，从上面看，我们看到的就是满满一层的鸡蛋。

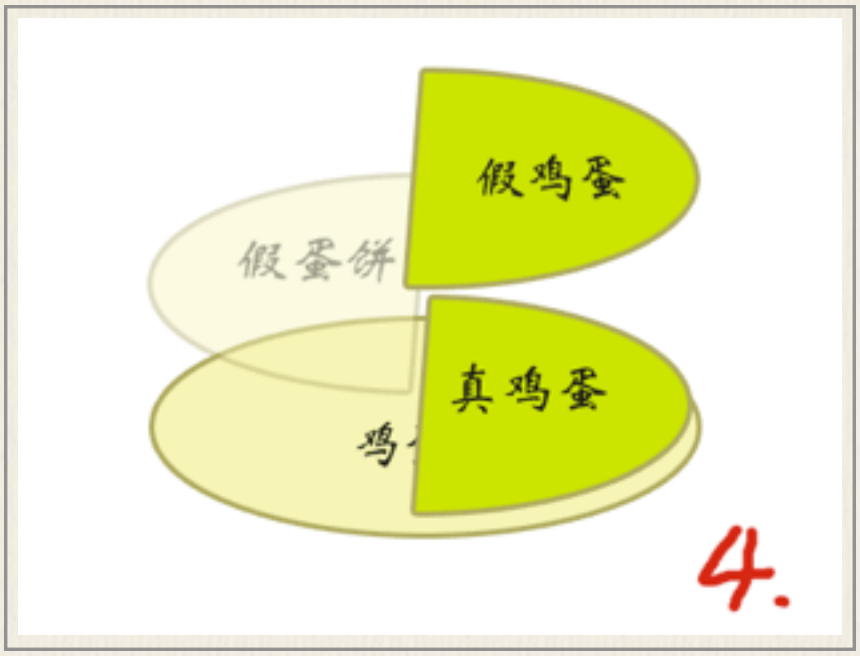


3. 真鸡蛋转起，你会发现，半个真鸡蛋，由于逆时针旋转，露出了点空（左侧上部）。



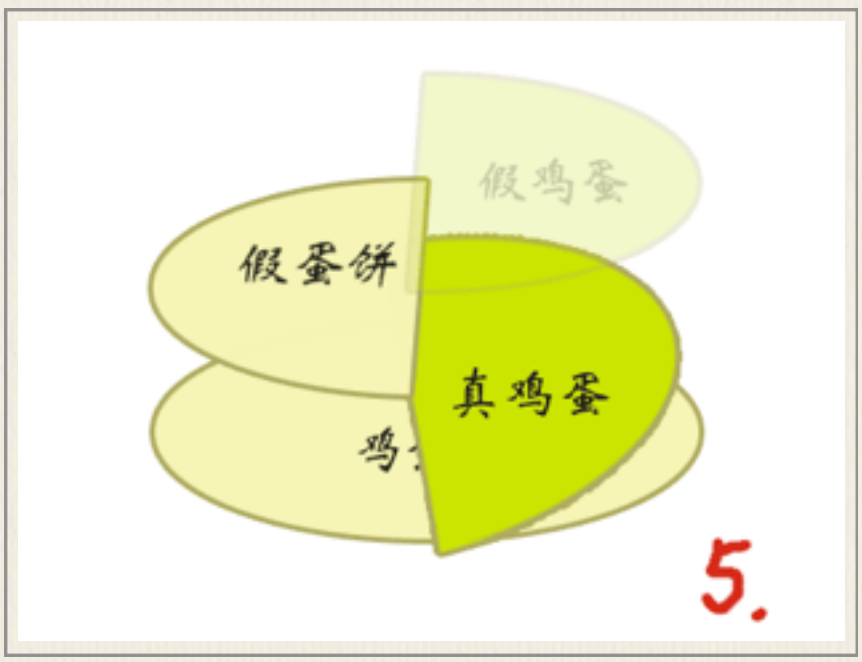
demo对应效果类似(浅色看成鸡蛋)：

4. 当真鸡蛋旋转了180度（半圈）的时候，真假鸡蛋正好重合在了一起，于是就是看到的就是蛋饼上半面鸡蛋。



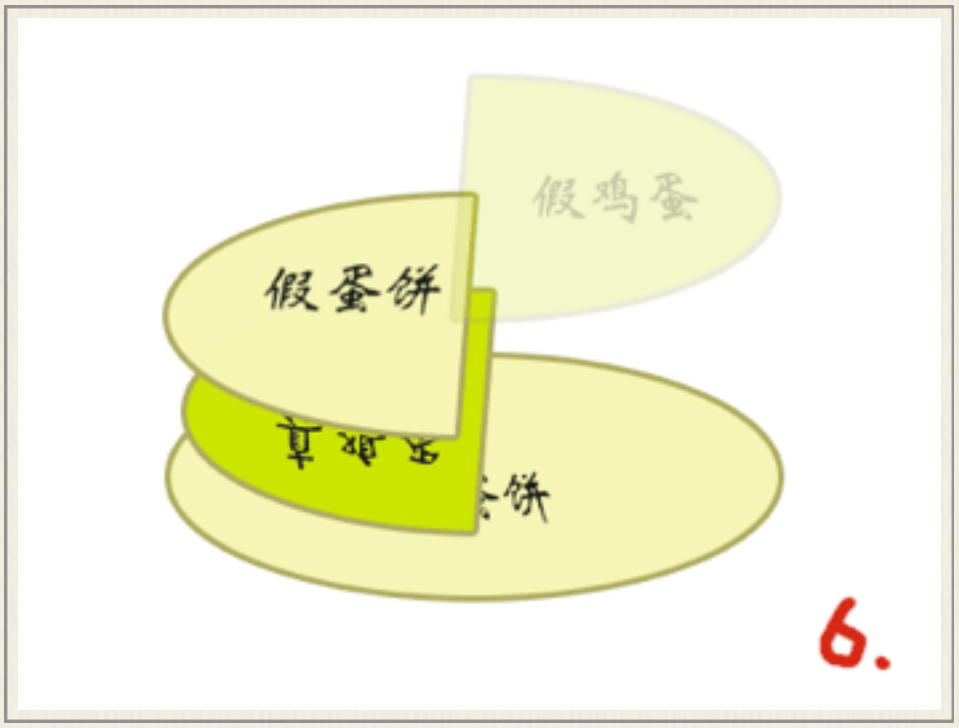
demo对应效果类似：

5. 此时，再继续旋转。我们让假鸡蛋隐藏，让之前打酱油的~~加饼子~~假饼子出现，覆盖部分真鸡蛋。我们就会看到，真鸡蛋面积越来越小了~



demo对应效果类似：

6. 一直旋转到360度，其完全被假的饼子遮盖，一点鸡蛋都看不到了。完成了从全部都0的动画过程。这就是蛋饼转转转的基本原理。



CSS3表示
可见，要实现我们想要的蛋饼效果，我们需要这些东西：

1. 圆形的蛋饼子 – 对应下面类名为inner元素
2. 旋转的半面真鸡蛋 – 对应下面类名为spiner的元素
3. 不动的半面蛋饼子，前半程隐藏，后半程出现 – 对应下面类名为masker的元素
4. 不动的半面假鸡蛋，前半程出现，后半程隐藏 – 对应下面类名为filler的元素

```
<div class="inner">
    <div class="spiner"></div>
    <div class="filler"></div>
    <div class="masker"></div>
</div>
```

1. inner主要实现圆以及背景色；
2. spiner主要实现半圆的360度逆时针旋转，其背景色有别于父元素的背景色；
3. filler半圆，定位在右侧，与旋转元素同样背景色；后面的180度隐藏；
4. masker半圆，定位在左侧，与大背景色色值相同；旋转前180度隐藏，之后显示遮盖；

其中，360度旋转CSS代码如下：

```
@keyframes spin {
  0%   { transform: rotate(360deg); }
  100% { transform: rotate(0deg); }
}
```

因为是逆时针，所以是从360deg到0deg.

前半程出现，后半程隐藏，可以借助animation step相关的timing function实现，代码如下：

```
@keyframes second-half-hide {
  0%         { opacity: 1; }
  50%, 100% { opacity: 0; }
}
```

后半程显示则是：

```
@keyframes second-half-show {
  0%         { opacity: 0; }
  50%, 100% { opacity: 1; }
}
```

于是，我们只要加个动画时间，以及无限执行就OK啦~~

```
.spiner { transform-origin: right center; animation: spin .8s infinite linear; }
.filler { animation: second-half-hide .8s steps(1, end) infinite; }
.masker { animation: second-half-show .8s steps(1, end) infinite; }
```

其他细节都是定位什么的，很基础的知识，就不啰嗦啦~~

饼其实还没有做好
啊，捣鼓了这么久还没有结束啊?

没错。仔细查看gif动画，你会发现，蛋饼它是从全盘都0再到整个360度覆盖的。

而，上午捣鼓的动画只是从360度无死角覆盖到0覆盖。一旦覆盖结束，就又要走360度开始，不连贯，怎么破?

我是这么处理的：
再覆盖一个蛋饼从0度到360度展示的动画。与一直捣鼓的动画前后半程分别展示就可以了。

于是，最终有如下HTML:

```
<div class="inner">
    <div class="spiner"></div>
    <div class="filler"></div>
```

```
    <div class="masker"></div>
</div>
<div class="inner2">
    <div class="spiner"></div>
    <div class="filler"></div>
    <div class="masker"></div>
</div>
```

inner和inner2也使用的前后半程隐藏的动画，动画时间正好是一个周期的2倍。

```
.inner { opacity: 1; animation: second-half-hide 1.6s steps(1, end)
infinite; }
.inner2 { opacity: 0; animation: second-half-show 1.6s steps(1, end)
infinite; }
```

于是，就有了完美的做蛋饼效果了。

五、结束语

靠，这么晚了，还结语呢，结你个蛋饼球吧~~

欢迎指正不准确之处，欢迎交流，感谢阅读，就这些，睡了~~晚安思密达~~

原文链接：http://www.zhangxinxu.com/wordpress/2014/04/css3-pie-loading-waiting-animation/

Google NewSQL之F1

作者：@刀尖红叶
DBA(MySQL&MongoDB;) 热爱开源,喜欢Linux,
偶尔码码Python、Golang

上一篇说 到了Google Spanner--全球级分布式数据库,但我觉得Spanner不能称作NewSQL,因为Spanner虽然有外部一致性、类SQL接口和常见事务支 持,但和传统关系型数据库相比功能仍不够丰富,对此,Google在Spanner上开发了F1,扩展了Spanner已有特性成为名副其实的 NewSQL数据库;F1目前支撑着谷歌的AdWords核心业务。

简介

AdWords业务本来使用MySQL+手工Sharding来支撑,但在扩展和可靠性上不能满足要求,加上Spanner已经在同步复制、外部一致性等 方面做的很好,AdWords团队就在Spanner上层开发了混合型数据库F1(F1意味着:Filial 1 hybrid,混合着关系数据的丰富特性和NoSQL的扩展性),并于2012年在生产环境下替代MySQL支撑着AdWords业务(注意： F1和 Spanner之间关系并不是MariaDB基于MySQL再开发的关系,而是类似InnoDB存储引擎和MySQL Server层的关系--F1本身不存储数据,数据都放在Spanner上)。

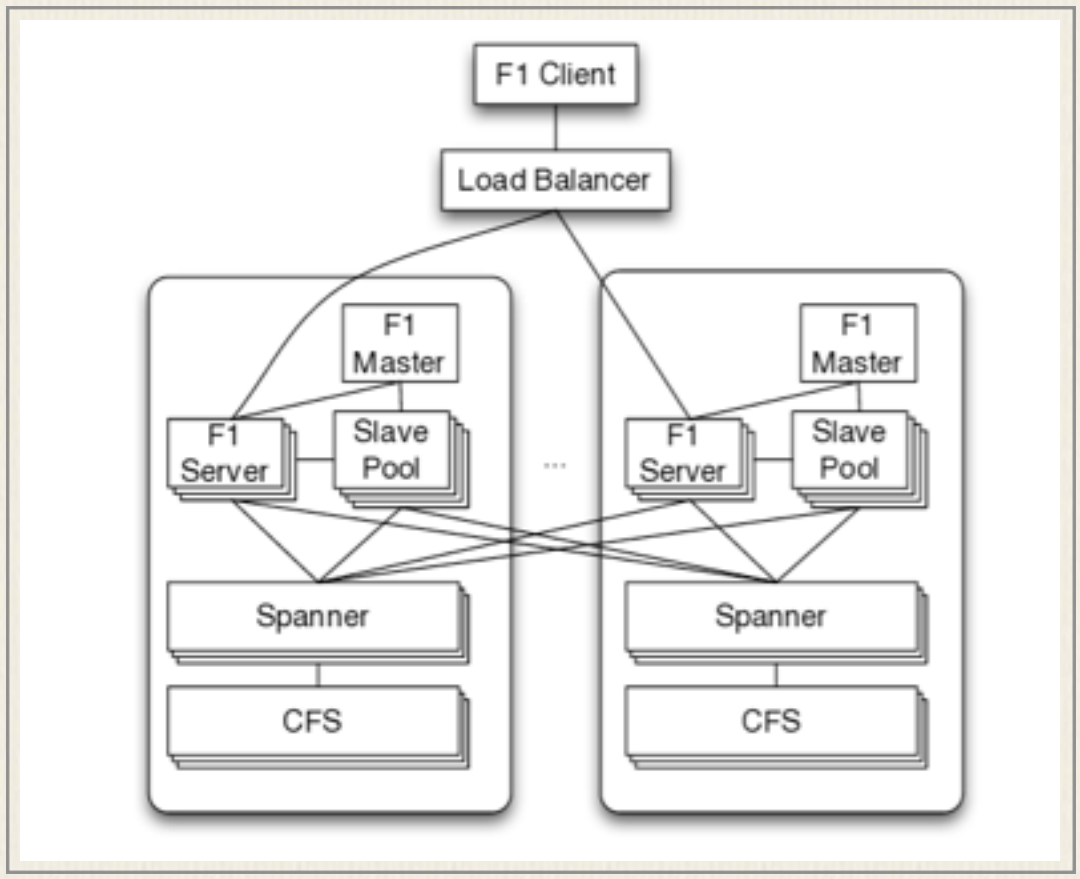
F1设计目标

1.可扩展性:F1的用户经常有复杂查询和join,F1提供了增加机器就能自动resharding和负载均衡。
2.高可用性:由于支撑着核心业务,分分钟都是money,不能因为任何事故(数据中心损坏、例行维护、变更schema等)而出现不可用状况。
3.事务:Spanner的外部一致性的是不够的,标准ACID必须支持。
4.易用性:友好的SQL接口、索引支持、即席查询等特性能大大提高开发人员的效率。

F1规模

截止这篇论文发表之时(2013年)F1支撑着100个应用,数据量大概100TB,每秒处理成百上千的请求,每天扫描数十万亿行记录,和以前MySQL相比,延迟还没增加;在这样规模下,可用性竟达到了惊人的99.999%!

F1架构总览



使用F1的用户通过client library来连接,请求再负载均衡到一个F1节点,F1再从远Spanner读写数据,最底层是分布式文件系统Colossus.负载均衡器会优先把 请求连到就近的数据中心里的F1,除非这数据中心不可用或高负载.通常F1和对应Spanner都在同一个数据中心里,这样能加快访问Spanner的速 度,假如同处的Spanner不可用,F1还能访问远程的Spanner,因为Spanner是同步复制的嘛,随便访问哪个都行。

大部分的F1是无状态的,意味着一个客户端可以发送不同请求到不同F1 server,只有一种状况例外:客户端的事务使用了悲观锁,这样就不能分散请求了,只能在这台F1 server处理剩余的事务。

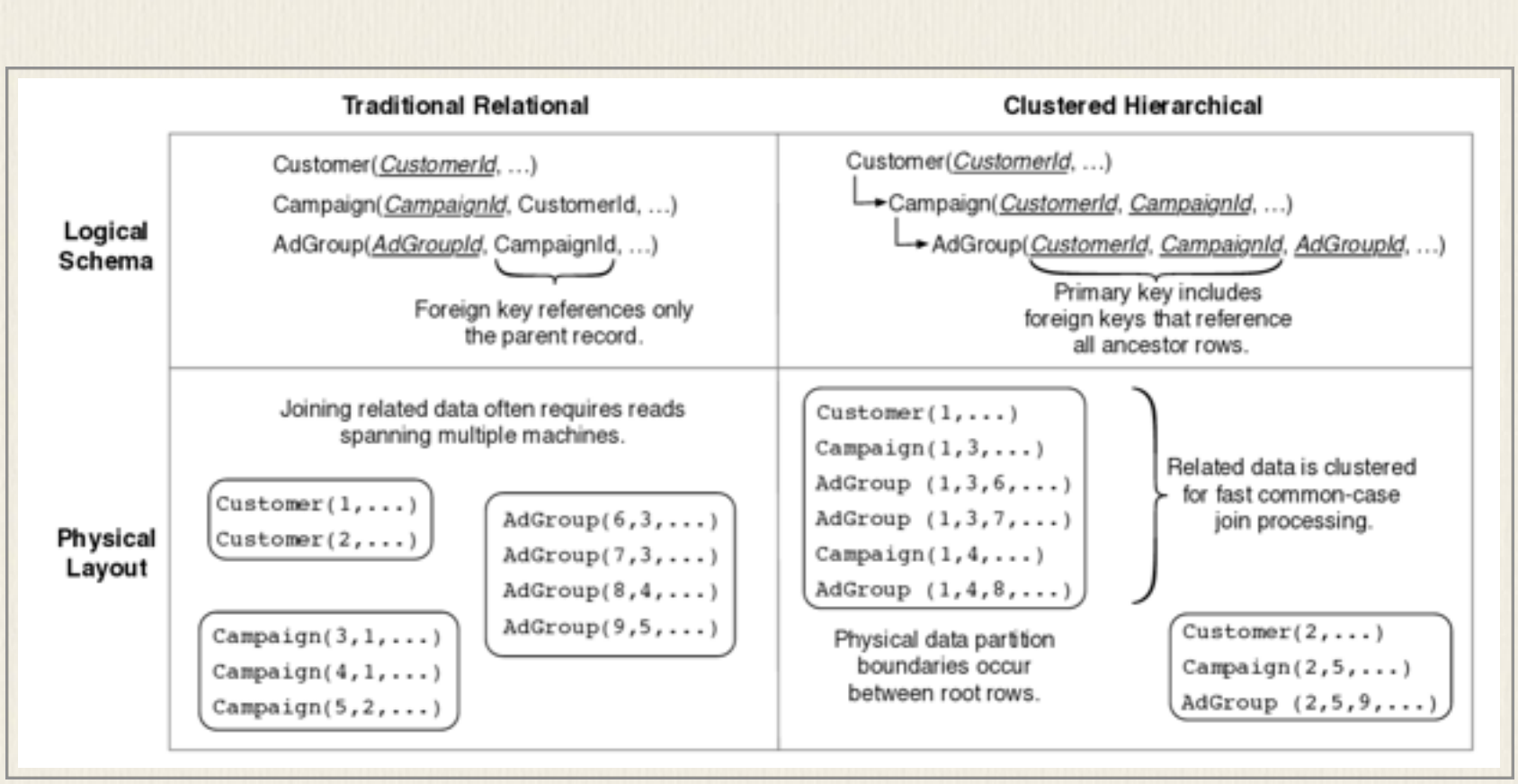
你可能会问上图的F1 Master和Slave Pool干嘛的?它们其实是并行计算用的,当query planner发现一个SQL可以并行加速执行的话,就会将一个SQL分拆好多并行执行单位交到slave pool里去执行,F1 Master就是监控slave

pool健康情况和剩余可用slave数目的.除了SQL,MapReduce任务也可用F1来分布式执行。

F1数据模型

F1的数据模型和Spanner很像--早期Spanner的数据模型很像Bigtable,但后来Spanner还是采用了F1的数据模型.

逻辑层面上,F1有个和RDBMS很像的关系型schema,额外多了表之间层级性和支持Protocol Buffer这种数据类型.那表之间层级性怎么理解呢?其实就是子表的主键一部分必须引用父表,譬如:表Customer有主键 (CustomerId)，它的子表Campaign的主键必须也包含(CustomerId),就变成这种形式(CustomerId, CampaignId);以此可推假如Campaign也有子表AdGroup,则AdGroup的主键必须这种形式 --(CustomerId,CampaignId, AdGroupId).最外面父表Customer的主键CustomerId就叫一个root row,所有引用这个root row的子表相关row组成了Spanner里的directory(这个解释比Spanner那篇论文反而清楚,囧),下图比较了RDBMS和F1、 Spanner的层级结构区别:



那为什么F1、Spanner要使用这种层级存储方式呢?有多方面的好处:1.可以并行化:想象一下假如要取Campaign和AdGroup表里所有CustomerId=1的记录,在传统RDBMS里,因为Adgroup不直接包含有CustomerId字段,所以这种情况下只好顺序化先取Campaign表里满足条件的行再 处理AdGroup表;

而在层级存储方式下,Campaign和AdGroup表都包含CustomerId字段,就能并行处理了.2.可以加速 update操作:update一般都有where 字段=XX这样的条件,在层级存储方式下相同row值的都在一个directory里,就省的跨Paxos Group去分布式2PC了.

Protocol Buffer

不要听名字感觉Protocol Buffer是个缓存什么之类的,其实Protocal Buffer是个数据类型,就像XML、Json之类,一个数据类型而已;因为在谷歌内部的广泛使用,所以F1也支持它,但是是当做blob来对待 的.Protocol Buffer语义简单自然,还支持重复使用字段来提高性能(免得建子表)。

索引

所有索引在F1里都是单独用个表存起来的,而且都为复合索引,因为除了被索引字段,被索引表的主键也必须一并包含.除了对常见数据类型的字段索引,也支持对Buffer Protocol里的字段进行索引.

索引分两种类型:

Local:包含root row主键的索引为local索引,因为索引和root row在同一个directory里;同时,这些索引文件也和被索引row放在同一个spanserver里,所以索引更新的效率会比较高.

global:同理可推global索引不包含root row,也不和被索引row在同一个spanserver里.这种索引一般被shard在多个spanserver上;当有事务需要更新一行数据时,因为 索引的分布式,必须要2PC了.当需要更新很多行时,就是个灾难了,每插入一行都需要更新可能分布在多台机器上的索引,开销很大;所以建议插入行数少量多 次.

无阻塞schema变更

考虑F1的全球分布式,要想无阻塞变更schema几乎是mission impossible,不阻塞那就只好异步变更schema,但又有schema不一致问题(譬如某一时刻数据中心A是schema1,而数据中心B是对应确是schema2),聪明的谷歌工程师想出了不破坏一致性的异步变更算法,详情请参见--Online, Asynchronous Schema Change in F1

ACID事务

像BigTable的这样最终一致性数据库,给开发人员带来无休止的折磨--数据库层面的一致性缺失必然导致上层应用使用额外的代码来检测数据一致性;所以F1提供了数据库层面的事务一致性,F1支持三种事务:

1.快照事务:就是只读事务,得益于Spanner的全局timestamp,只要提供时间戳,就能从本地或远程调用RPC取得对应数据.

2.悲观事务:这个和Spanner的标准事务支持是一样的,参见Google NewSQL之Spanner.

3.乐观事务:分为两阶段:第一阶段是读,无锁,可持续任意长时间;第二阶段是写,持续很短.但在写阶段可能会有row记录值冲突（可 能多个事务会写同一行）,为此,每行都有个隐藏的lock列,包含最后修改的timestamp.任何事务只要commit,都会更新这个lock列,发 出请求的客户端收集这些timestamp并发送给F1 Server,一旦F1 Server发现更新的timestamp与自己事务冲突,就会中断本身的事务.

可以看出乐观事务的无锁读和短暂写很有优势:

1.即使一个客户端运行长时间事务也不会阻塞其他客户端.

2.有些需要交互性输入的事务可能会运行很长时间,但在乐观事务下能长时间运行而不被超时机制中断.

3.一个悲观事务一旦失败重新开始也需要上层业务逻辑重新处理,而乐观事务自包含的--即使失败了重来一次对客户端也是透明的.

4.乐观事务的状态值都在client端,即使F1 Server处理事务失败了,client也能很好转移到另一台F1 Server继续运行.

但乐观事务也是有缺点的,譬如幻读和低吞吐率.

灵活的锁颗粒度

F1默认使用行级锁,且只会锁这行的一个默认列;但客户端也能显示制定锁其他栏;值得注意的是F1支持层级锁,即:锁定父表的几列,这样子表的对应列也会相应锁上,这种层级锁能避免幻读.

变更历史记录

早前AdWords还使用MySQL那会,他们自己写java程序来记录数据库的变更

历史,但却不总是能记录下来:譬如一些Python脚本或手工SQL 的改变.但在F1里,每个事务的变更都会用Protocol Buffer记录下来,包含变更前后的列值、对应主键和事务提交的时间戳,root表下会产生单独的ChangeBatch子表存放这些子表;当一个事物 变更多个root表,还是在每个root表下子表写入对应变更值,只是多了一些连接到其他root表的指针,以表明是在同一个事务里的.由于 Change-
Batch表的记录是按事务提交时间戳顺序来存放,所以很方便之后用SQL查看.
记录数据库的变更操作有什么用呢?谷歌广告业务有个批准系统,用于批准新进来的广告;这样当指定的root row发生改变,根据改变时间戳值,批准系统就能受到要批准的新广告了.同样,页面展示要实时显示最新内容,传统方法是抓取所有需要展示内容再和当前页面 展示内容对比,有了ChangeBatch表,变化的内容和时间一目了然,只要替换需要改变的部分就行了.

F1客户端
F1客户端是通过ORM来与F1 Server打交道了,以前的基于MySQL实现的ORM有些常见缺点:
1.对开发人员隐藏数据库层面的执行
2.循环操作效率的低下--譬如for循环这种是每迭代一次就一个SQL
3.增加额外的操作(譬如不必要的join)
对此,谷歌重新设计了个新ORM层,这个新ORM不会增加join或者其他操作,对相关object的负载情况也是直接显示出来,同时还将并行和异步API直接提供开发人员调用.这样似乎开发人员要写更多复杂的代码,但带来整体效率的提升,还是值得的.
F1提供NoSQL和SQL两种接口,不但支持OLTP还支持OLAP;当两表Join时,数据源可以不同,譬如存储在Spanner上表可以和BigTable乃至CSV文件做join操作.

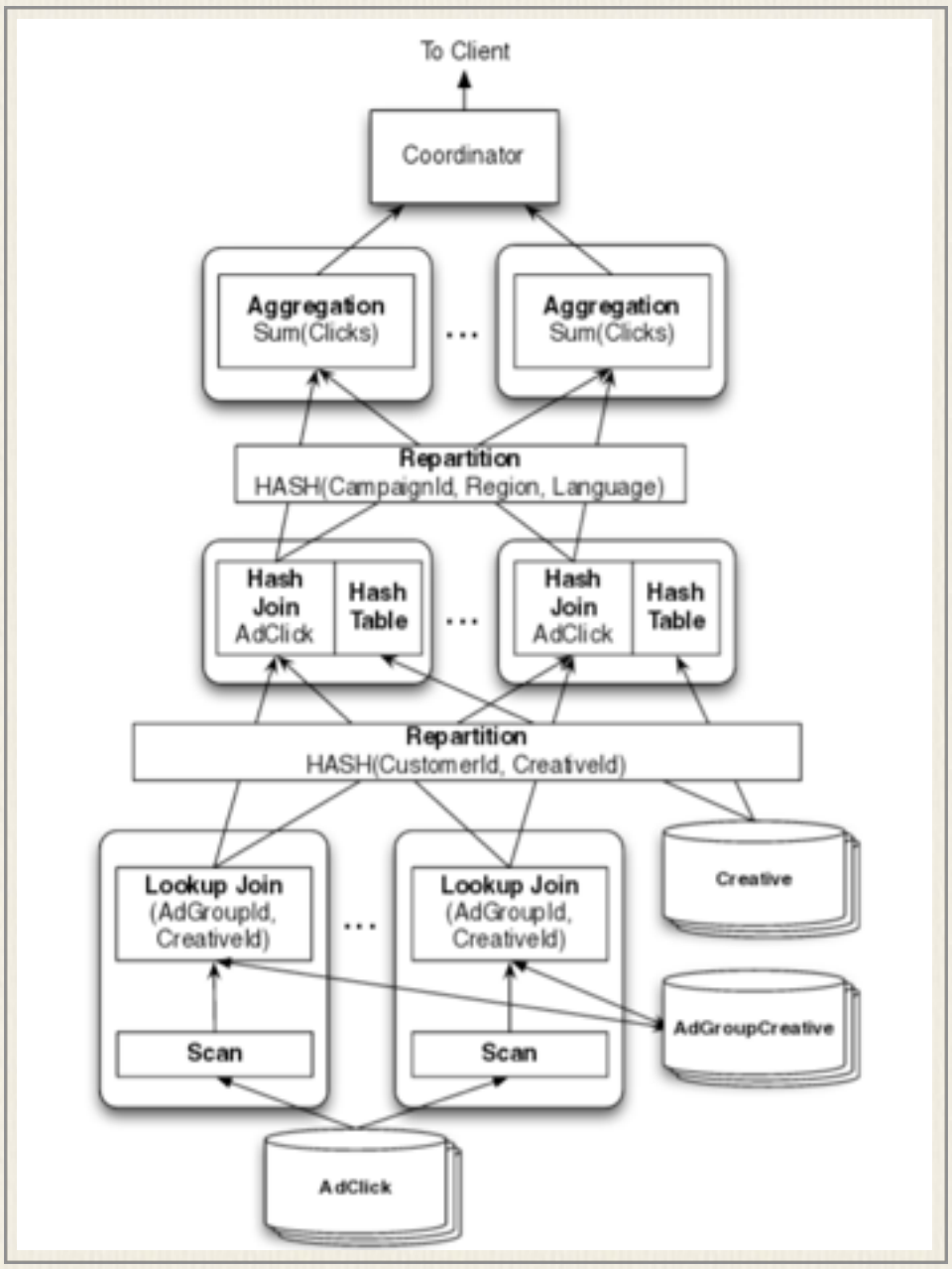
分布式SQL

既然号称NewSQL,不支持分布式SQL怎么行呢.F1支持集中式SQL执行或分布式SQL执行,集中式顾名思义就是在一个F1 Server节点上执行,常用于短小

的OLTP处理,分布式SQL一般适用OLAP查询,需要用到F1 slave pool.因为的OLAP特性,一般快照事务就ok了.
请看如下SQL例子:

```
SELECT agcr.CampaignId, click.Region,cr.Language,
SUM(click.Clicks)FROM AdClick clickJOIN AdGroupCreative agcrUSING (Ad-
GroupId, CreativeId)JOIN Creative crUSING (CustomerId, Crea-
tiveId)WHERE click.Date ='2013-03-23'GROUP BY agcr.CampaignId,
click.Region,cr.Language
```

可能的执行计划如下:



上图只是一个最可能的执行计划,其实一个SQL通常都会产生数十个更小的执行单位,每个单位使用有向无闭环图((DAG)来表示并合并到最顶端;你可能也 发现了在上图中都是hash partition而没有range partition这种方式,因为F1为使partition更加高效所以时刻动态调整partition,而range partition需要相关统计才好partition,所以未被采用.同时partition是很随机的,并经常调整,所

以也未被均匀分配到每个 spanserver,但partition数目也足够多了.repartition一般会占用大量带宽,但得益于交换机等网络设备这几年高速发展,带宽 已经不是问题了.除了join是hash方式,聚集也通过hash来分布式--在单个节点上通过hash分配一小段任务在内存中执行,再最后汇总.
可能受到H-Store的影响,F1的运算都在内存中执行,再加上管道的运用,没有中间数据会存放在磁盘上,这样在内存中的运算就是脱缰的马--速度飞 快;但缺点也是所有内存数据库都有的--一个节点挂就会导致整个SQL失败要重新再来.F1的实际运行经验表明执行时间不远超过1小时的SQL一般足够稳 定.
前面已经说过,F1本身不存储数据,由Spanner远程提供给它,所以网络和磁盘就影响重大了;为了减少网络延迟,F1使用批处理和管道技术--譬如当 做一个等值join时,一次性在外表取50M或10w个记录缓存在内存里,再与内表进行hash连接;而管道技术(或者叫流水线技术)其实也是大部分数据 库使用的,没必要等前一个结果全部出来再传递整个结果集给后一个操作,而是第一个运算出来多少就立刻传递给下个处理.对磁盘延迟,F1没上死贵的SSD, 仍然使用机械硬盘,而机械硬盘因为就一个磁头并发访问速度很慢,但谷歌就是谷歌,这世上除了磁盘供应商恐怕没其他厂商敢说对机械磁盘运用的比谷歌还牛--谷歌以很好的颗粒度分散数据存储在CFS上,再加上良好的磁盘调度策略,使得并发访问性能几乎是线性增长的.

高效的层级表之间join

当两个join的表是层级父子关系时,这时的join就非常高效了;请看下例:
SELECT * FROM Customer JOIN Campaign USING (CustomerId)
这这种类型的join,F1能一次性取出满足条件的所有行,如下面这样:

```
Customer(3)
  Campaign(3,5)
  Campaign(3,6)
Customer(4)
  Campaign(4,2)
  Campaign(4,4)
```

F1使用种类似merge join的cluster join算法来处理.
值得注意的是,一个父表即使包含多个子表,也只能与它的一个子表cluster

join;假如两个子表A和B要join呢,那得分成两步走了:第一步就是上述的父表先和子表A做cluster join,第二步再将子表B也第一步的结果做join.

客户端的并行

SQL并行化后大大加速处理速度,但也以惊人速度产生产生运算结果,这对最后的总汇聚节点和接受客户端都是个大挑战.F1支持客户端多进程并发接受数据,每个进程都会收到自己的那部分结果,为避免多接受或少接受,会有一个endpoint标示.

Protocol Buffer的处理

Protocol Buffer既然在谷歌广泛使用着,F1岂有不支持之理?其实Protocol Buffer在F1里是被当做一等公民优先对待的,下面是个Protocol Buffer和普通数据混合查询的例子:

```
SELECT c.CustomerId, c.Info
FROM Customer AS c
WHERE c.Info.country_code ='US'
```

其中c是个表,c.Info是个Protocol Buffer数据,F1不但支持将c.Info全部取出,而且也能单独提取country_code这个字段; Protocol Buffer支持一种叫repeated fields的东东,可以看做是父表的子表,唯一区别是显式创建子表也就显式制定外键了,而repeated fields是默认含有外键;这么说还是太抽象,举个例子:

```
SELECT c.CustomerId, f.feature
FROM Customer AS c
  PROTO JOIN c.Whitelist.feature AS f
WHERE f.status ='STATUS_ENABLED'
```

c是个表,c.Whitelist是Protocol Buffer数据,c.Whitelist.feature就是repeated fields了(当做c的子表);虽然看起来很奇怪c与它的Whitelist.feature做join,但把c.Whitelist.feature 当做一个子表也就解释的通了.
Protocol Buffer还可以在子查询中使用,如下面例子:

```
SELECT c.CustomerId, c.Info,
 (SELECT COUNT(*) FROM c.Whitelist.feature) nf
FROM Customer AS c
WHERE EXISTS (SELECT * FROM c.Whitelist.feature f WHERE f.status
!='ENABLED')
```

Protocol Buffer虽然省了建子表的麻烦,却带来额外性能开销:1.即使只需要取一小段,也要获取整个Protocol Buffer,因为Protocol Buffer是被当做blob存储的;2.取得数据要经过解压、解析后才能用,这又是一笔不菲开销.

在谷歌使用规模

F1和Spanner目前部署在美国的5个数据中心里支撑Adwords业务,东西海岸各两个,中间一个;直觉上感觉3个数据中心就已经很高可用了,为什 么谷歌要选择5个呢?假如只部署在3个数据中心,一个挂了后剩余两个必须不能挂,因为commit成功在Paxos协议里需要至少2节点;但如果第二个节 点又挂了此时就真的无法访问了,为了高可用,谷歌选择了5个数据中心节点.

东海岸有个数据中心叫做preferred leader location,因为replica的Paxos leader优先在这个数据中心里挑选.而在Paxos协议里,read和commit都要经过leader,这就必须2个来回,所以客户端和F1 Server最好都在leader location这里，有助降低网络延迟.

性能

F1的读延迟通常在5~10ms,事务提交延迟高些50~150ms,这些主要是网络延迟;看起来是很高,但对于客户端的总体延迟,也才200ms左右,和之前MySQL环境下不相上下,主要是新ORM避免了以前很多不必要的其他开销.

有些非交互式应用对延迟没太高要求,这时就该优化吞吐量了.尽量使用不跨directory的小事务有助于分散从而实现并行化,所以F1和Spanner 的写效率是很高的.其实不光写效率高,查询也很迅速,单条简单查询通常都能在10ms内返回结果,而对于非常复杂的查询,得益于更好的并发度,也通常比之 前MySQL的时候快.

之前MySQL的查询数据都是不压缩存储在磁盘上,磁盘是瓶颈,而在F1里数据都是压缩的,虽然压缩解压过程带来额外CPU消耗,但磁盘却不在是瓶颈,CPU利用率也更高,以CPU换空间总体还是值得的.

结尾

可以看到F1完善了Spanner已有的一些分布式特性,成为了分布式的关系型数据库,也即炒的火热的NewSQL;但毕竟谷歌没公布具体实现代码,希望尽早有对应开源产品面世可以实际把玩下.

参考资料

[F1 - The Fault-Tolerant Distributed RDBMS Supporting Google's Ad Business]

[F1: A Distributed SQL Database That Scales]

原文链接：http://www.leafonsword.org/google-f1/

SQL，NoSQL 以及数据库的实质

作者：@_王垠_
网站：http://www.yinwang.org/
当然我在扯淡

在之前的一些博文里（比如这篇），我多次提到关系式数据库和 SQL 的缺陷。我觉得它们是制造了问题又自己来解决，而且永远没法解决好。可是由于时间原因，一直没有来得及解释我的观点，以至于很多人不理解我在说什么，还以 为是信口开河。所以现在有了点闲暇时间，我就把这里面的细节稍微说一下。也许你会发现，得出这些结论所需要的背景知识，比你想象的要多得多。

描述性语言的局限性

当我指出 SQL 的问题时，总是避免不了有人批驳说：“SQL 是描述性的语言。你只告诉它 What，而不是告诉它 How。”我发现每一次有人批驳我的观点，总是拿一些我多年前就听腻了，看透了的“广告词”，而现在这同样的事又发生在 SQL 身上。他们没有发现，我不但能实现 SQL，而且已经实现过比 SQL 强大很多的语言（逻辑式语言），所以我其实早已看透了所有这些语言的实质，我知道那些广告词在很大程度上是误导。

现在我就来分析一下 SQL 与逻辑式语言之间的关系，并且找出这类“描述性语言”共同的弱点。

Prolog 与人工智能的没落

可以说，“只告诉它 What，而不是告诉它 How”，只是一个不切实际的妄想，而且它并不是 SQL 首创的口头禅。在 SQL 诞生两年以前，有人发明了 Prolog，著名的“逻辑式语言运动”的先锋。Prolog 使用了与 SQL 非常类似的广告词，声称自己能够一劳永逸的解决人工智能和自动编程的问题，这样

人们不需要写程序，只需要告诉电脑“想要什么”，然后电脑就能自动生成算法，自动生成代码来解决这问题。

世界上总是有很多这种类似“减肥药”的东西，每一个都声称自己是“不需运动，不需节食，一个星期瘦 20 斤！”然而由于人类的智力和经验参差不齐，总会有人上当。Prolog 当年的风头之大，以至于它被日本政府采用并且大力推广，作为他们所谓的“第五代计算机”的编程语言。可惜的是，减肥药毕竟是减肥药，科学道理决定了 Prolog 必定失败，以及“人工智能冬天”（AI winter）的到来。

为什么 Prolog 会失败呢？这是因为 Prolog 虽然“终究”有可能自动解决某些问题，然而由于它的算法复杂度太高，所以没法在我们有生之年完成。说白了，Prolog 采用的“计算”方式就是“穷举法”。为了得到用户“描述”的问题的答案，而不需要用户指定具体的数据结构和算法，Prolog 必须对非常大的图状解空间进行完全的遍历（Prolog 采用深度优先搜索）。而这种解空间的“状态”数量往往是与程序运行时路过的分支数目成指数关系，这就决定了 Prolog 虽然“最终”可能找到问题的答案，却很有可能在地球毁灭之前都没法完成它的搜索。而且由于 Prolog 无法表达真正意义上的“逻辑否”操作，所以对于很多问题它永远无法得到正确的答案（这是一个非常深入的问题，30 多年的研究，仍然没有结果）。

过于具体的细节我不想在这里解释，你只需要明白的是我绝不是在信口开河。我在描述性语言方面的造诣超乎绝大多数人的想象。普通人使用这些语言只是作 为一个用户，而我是作为一个设计者和实现者。而我的设计和实现能力，又在所有实现者里面处于领先的地位。所以我看到了太多普通使用者所看不到的东西，我看 到了他们头脑里的局限性。在 Indiana 的日子里，我重新实现，并且扩展了一种与 Prolog 类似的逻辑式语言，叫做 mini-Kanren。它也就是 Dan Friedman 的新书《The Reasoned Schemer》 的主题。我不但完全重写了 miniKanren 的代码，而且为它加入了一种技术叫做 constraint logic programming，并且在那之上实现了一个非常干净利落的“逻辑否”操作。经过这番动手操作，我对 miniKanren 和逻辑式语言的工作方式可以说是了如指掌。虽然 miniKanren 比起 Prolog 更加优雅，而且

在搜索算法上有所改进（广度优先而非深度优先），它本质上采用的计算方式也是一样的：穷举法。所以在很多时候它的效率很低，用法不灵活。像 Prolog 一样，miniKanren 并不能用来解决很多实际的问题。有些很简单的 Scheme 代码，你却不能把它翻译成等价的 miniKanren。然而就是这样一种语言，比起 SQL 的表达能力，其实也是天上地下。所以在实现并且扩展了 miniKanren 这样的“智能语言”之后再来看 SQL，对于我来说就像是在解剖一只青蛙。

这里只举一个例子，说明我所看到的所谓“描述性语言”的局限，看不懂的人可以暂时跳过。在 IU 的时候总有一些人喜欢用 miniKanren 来实现 Haskell 和 OCaml 的 Hindley-Milner 类型系统（HM 系统）。HM 那种最基本的基于 unification 的类型推导，miniKanren 确实能做到，因为 miniKanren，Prolog 和 HM 系统一样，都是基于 Robinson unification 算法。 这种算法虽然精巧，表达能力却是非常有限的。如果遇到一些必要的扩展，比如 HM 系统所需要的 let-polymorphism，你就需要对 miniKanren 语言本身进行扩展。也就是说，你不再是用 miniKanren 实现你的算法，而是用一种过程式或者函数式语言（比如 Scheme）把你的算法加到 miniKanren 里面作为“语言特性”，然后再利用这个你刚实现的新特性来“实现”你的算法。于是你就发现，其实 miniKanren 本身并没有足够的表达力表示完整的 HM 类型推导算法。如果 miniKanren 必须经过 Scheme 扩展才能表达 HM 算法，那么比起直接的 Scheme 实现，miniKanren 并没有任何优势。一个新的语言特性要有价值，它必须能够在很多地方使用。然而这些 miniKanren 的扩展，每一个只能用到一个地方。所以它们其实完全失去了作为语言特性的价值，还不如直接写一段 Scheme 代码。

这也就是为什么虽然我很感谢 miniKanren 教会了我逻辑编程的原理，然而我实现过的所有强大的类型系统（有些的能力大大超过 HM 系统），全都是用最普通的过程式或者函数式语言。“描述性语言”声称的好处，其实在这种关键时刻总是微乎其微，还不如调用普通语言的库代码。

从 Prolog 到 SQL

扯了这么多 Prolog 和 miniKanren 的事情，这跟关系式数据库和 SQL 的讨论到底有何关系呢？其实，这些东西是有非常深层次的内在联系的。一个有趣的事情是，miniKanren 里面的“Kan-
ren”一词并不是一个英语国家的人名，而是日语“関連”（かんれん，读作 kan-
ren）。而“逻辑式语言”的另一个名字，其实叫做“关系式语言”（relational language）。

在数学上，“关系”（relation）意味着“没有方向”，意味着“可逆”。然而可笑的是，所谓的“关系式数据库”并不具有这种可逆计算的能力。 Prolog 和 miniKanren 其实是比 SQL 强大很多的语言，是真正的“关系式语言”，它们能够在比较大的程度上完成可逆计算。比如在 miniKanren 里面，你可以使用这样“查询操作”：如果 x+y 等于 10，y 等于 2，那么 x 等于几？所以很多 Prolog 和 miniKanren 可以表达的查询，SQL 没法表示。SQL 其实只能用于非常简单的，有“明确方向”的查询操作。由于这些局限性，再加上很多其他的设计失误（比如语法像英语，组合能力弱），它只适合会计等人员使用，一旦遇到程序员需要的，稍微复杂一点的数据结构，它就没法表达了，而且会像 Prolog 一样引起诸多的性能问题。

由于 SQL 比起逻辑式语言有更多的限制，表达力弱很多，再加上 SQL 对于基本的数据结构进行了“索引”，逻辑式语言的性能问题在 SQL 里面得到了局部的缓解。比如，对于基本的算数操作 x < 10，SQL 能够通过对索引（B 树）的查找来进行“优化”，从而避免了对 x 所有可能的值（一个非常大的空间）进行完全的遍历。然而这种索引的能力是非常有限的，它几乎没有扩展能力，而且很难自动生成。所以一旦遇到更加复杂的情况，数据库自带的索引就没法满足需要了。除了极其简单的情况，SQL 的编译器无法自动生成高效的索引。

更要命的是，这种问题的来源是根本性的，不可解决的，而不只是因为某些数据库的 SQL 编译器不够“智能”。很多人不理解这一点，总是辩论说“我们为何需要 Java 而不是写汇编，也就是我们为何需要 SQL。”然而，把 Java 编译成高效的汇编，和把 SQL 编译成高效的汇编，是两种本质上不同的问

题。前者可以比较容易的解决，而后者是不可能的（除了非常个别的情况）。如果你理解“编译器优化”的本质就会发 现，这里面有一个拓扑学上的质的飞跃。把 Java 编译成高效的汇编，是一个非常简单的，线性的优化。这个过程就像改进一个已经连接好的电路，把里面太长的电线缩短一点，这样时延和电阻可以小一些。而把 SQL 优化成高效的汇编，是一个非线性的优化。这个过程不只是缩短电线那么简单的问题，它需要解开一些错综复杂的“结”。这种优化不但非常难以实现，需要大量的 “潜在数学知识”，而且有可能花费比执行代码还多的时间。

我只举一个例子来说明这个问题。如果你需要迅速地在地图上找到一个点附近的城市，SQL 无法自动在平面点集上建造像 KD-tree 那样的数据结构。这是很显然的，因为 SQL 根本就不知道你的数据所表示的是平面上的点集，也不理解平面几何的公理和定理。跟 B-tree 类似，知道什么时候需要这种特殊的索引结构，需要非常多的潜在数学知识（比如高等平面几何），所以你肯定需要手动的建立这种数据结构。你发现了吗，你其实 已经失去了所谓的“描述性”语言带来的好处，因为你完全可以用最普通的语言，加上一些构造 B-tree, KD-tree 的“库代码”，来实现你所需要的所有复杂查询操作。你的 SQL 代码并不会比直接的过程式代码更加清晰和简洁。再加上 SQL 本身的很多设计失误，你就发现使用 SQL 数据库其实比自己手工实现这些数据结构还要痛苦。你学会 SQL 是为了避免编程，结果你不得不做比编程还要苦逼的工作，还美其名曰“performance tuning”。

到这里也许有人仍然会说，这只是因为现在的 SQL 编译器不够智能，总有一天我们能够制造出能够“自动发明”像 B-tree, KD-tree 这样索引结构的“优化算法”。我对此持非常不乐观的态度。首先你要意识到，哪怕最基本的数学知识，也是经过了人类几千年的实践，研究和顿悟才得到的。计算 机虽然越来越快，它却缺乏对于世界最直接的观察和探索能力，所以在相当长的时间内，计算机是根本不可能自动“想到”这些数学和算法问题的，就不要谈解决它 们。其次，即使计算机有一天长了脚可以走路，有了眼睛可以看见东西，有了“自由意志”，可以自己去观察世界，它却不一定能够发现并且解决“人类关心的数学 问题”，因为它根本不知道人类需要什么。最后，我们需

要在有生之年解决这些迫切的问题，我们无法等待几十年几百年，就为了让计算机自己想出像 KD-tree 一类众所皆知的数据结构。

计算机不可能猜到人类到底想要什么，这就是为什么你几乎总是需要手动指定索引的原因，而且这种索引需要数据库“内部支持”。你一次又一次的希望 SQL 能够自动为你生成高效的索引和算法，却一次又一次的失望，也就是这个原因。当然，你永远可以使用所谓的 stored procedure 来扩展你的数据库，然而这就像是我的 IU 同学们用 miniKanren 来实现 HM 类型系统的方式——他们总是先使用一种过程式语言（Scheme）来添加这种描述性语言的“相关特性”，然后欢呼：“哇，miniKanren 解决了这个问题！”而其实呢，还不如直接使用过程式语言来得直接和容易。

这种总是需要扩展的显现也出现在数据库的语言里面。经验告诉我，如果想数据库处理大量数据时达到可以接受的性能，你几乎总是需要使用普通语言对手头 的数据库进行所谓的“扩展”，然后从 SQL 等查询语言“调用”它们。这种扩展代码往往是一次性的，只能用在一个地方，从而使得这些查询语言失去了存在的意义。因为如果经常如此，我们为何不直接发送 这种过程语言到数据库里面执行，从而完全取代 SQL?

另外有一种数据库查询语言叫 Datalog，它结合了 SQL 和 Prolog 的特点。然而以上对于 SQL 和 Prolog 的分析，同样也适用于 Datalog。

关系模型的实质

每当我批评 SQL，就有人说我其实不理解关系模型，说关系模型本身并没有问题，所以现在我就来分析一下什么是关系模型的实质。其实关系模型比起逻辑式语言，基本就是 个衍生产物，算不上什么发明。关系代数其实对应逻辑式语言里面的一个很小的部分——它的数据结构及其基本操作，只不过关系模型有更大的局限性而已。所以学 会了逻辑式语言的设计之后，你直接就可以把关系模型这种东西想出来。

每当谈到关系模型，总是有人很古板的追究它与 SQL，Datalog 等“查询语言”的区别。然而如果你看透了逻辑式语言的本质就会发现，其实“语言”和“模型”这两者并没有本质区别和明确界限。人们总是喜欢制造这些概念

上 的壁垒，用以防止自己的理论受到攻击。追究语言和模型的差别，把过错推到 SQL 和 IBM 身上，是关系式数据库领域常见的托词，用以掩盖其本质上的空洞和设计上的失误。所以在下面的讨论里为了方便，我仍然会使用少量 SQL 来表示关系模型里面对应的概念，但这并不削弱我对关系模型的批评。

关系模型与逻辑式语言

我们先来具体探讨一下关系模型与逻辑式语言的强弱关系。之前我们已经提到了，关系式数据库所谓的“关系”，比起逻辑式语言来说，其实是小巫见大巫。 关系式数据库的表达能力，绝对不会超过逻辑式语言。关系式代数里面的“=”，join 等构造都是没有方向的。然而与逻辑式语言不同，这些“可逆操作符”在关系代数里的用法受到非常大的限制。比如，这些可逆操作都不能跨过程，而且关系模型并 不包含递归函数。所以你并不能真正利用这种“无方向的代码”来完成比“有方向代码”更加强大的功能，大部分时候它们本质上只是普通程序语言里面最普通的一 些表达式，只不过换了一种更“炫”的写法而已。

总是有人声称限制语言的表达力可以让语言更加容易优化，然而如果一个语言弱得不能用，优化做得再好又有什么用。关系模型的核心，其实是普通程序语言 里面最简单的部分：表达式。如果缺乏控制结构和递归，这些表达式的能力只相当于最简单的计算器。经验告诉我，就算表达力这么弱的语言，很多数据库的编译器 也不能把优化做好，所以这不过是为它的弱表达力找个借口。另外由于这种无方向的表达式让你在阅读的时候很难看清楚数据的“流向”，所以你很难理解这里面包 含的算法。这种问题也存在于逻辑式语言，但因为逻辑式语言的表达力在某些方面强于过程式语言，所以感觉还不算白费劲。然而，关系模型有着逻辑式语言的各种 缺点，却不能提供逻辑式语言最基本的长处，所以比起过程式语言来说其实是一无是处。

关系模型与数据结构

我们再来探讨一下关系模型与数据结构的关系。很多人认为关系式数据库比起数据结构是一个进步，然而经过仔细的思考之后我发现，它其实不但是一

个退 步，而且是故弄玄虚，是狗皮膏药。在 IU 的时候，我做过好几个学期数据库理论课程的助教。当时我的感受就是，很多计算机系学生上了“数据结构”课程之后，再来上“数据库理论”课程，却像是被洗脑 了一样，仿佛根本没有理解数据结构。经过一段时间的接触之后，我发现其实他们大部分人只是被数据库领域的诸多所谓“理论”，“模型”或者“哲学”给迷惑 了。本来是已经理解的数据结构和算法，却被数据库理论给换成了等价却又吓人的新名词，所以他们忽然搞不明白了。我是很负责的老师，所以我努力地思索，想让 他们找回自我，最后我成功了。经过我如下的分析，他们大多数后来都茅塞顿开，对关系式数据库应用自如，最后取得了优异的成绩。

其实，关系模型的每一个“关系”或者“行”（row），表示的不过是一个普通语言里的“结构”（就像 C 的 struct）。一个表（table），其实不过是一个装着结构的数组。举个例子，以下 SQL 语句构造的数据库表：

```
CREATE TABLE Students ( sid CHAR(20),
                        name CHAR(20),
                        login CHAR(20),
                        age INTEGER,
                        gpa REAL )
```

其实相当于以下 C 代码构造的结构的数组：

```
struct student {
  char* sid;
  char* name;
  char* login;
  int age;
  double gpa;
}
```

每一个 join，本质上就是沿着行里的“指针”（foreign key）进行“寻址”，找到它所指向的东西。在实现上，join 跟指针引用有一定区别，因为 join 需要查软件哈希表，所以比指针引用要慢。指针引用本质上是在查硬件哈希表，

所以快很多。当然，这些操作都是基于“集合”的，但其实普通语言也可以表示集合 操作。

所谓的查询（query），其实就是普通的函数式语言里面的 filter, map 等抽象操作，只不过具体的数据结构有所不同。关系式代数更加笨拙一些，组合能力弱一些。比如，以下的 SQL 语句

```
SELECT Book.title
 FROM Book
 WHERE price > 100
```

本质其实相当于以下的 Lisp 代码（但不使用链表，执行机制有所不同而已）：

```
(map book-title
     (filter (lambda (b) (> (book-price b) 100)) Book)
```

所以关系模型所能表达的东西，其实不会超过普通过程式（函数式）语言所用的数据结构，然而关系模型却有过程式数据结构所不具有的局限性。由于经典的 关系“行”只能有固定的宽度，所以导致了你没法在结构里面放进任何“变长”的东西。比如，如果你有一个变长的数组需要放进结构，你就需要把它单独拿出来， 旋转 90 度，做成另外一个表，然后在原来的表里用一个“key”指向它们。在这个“中间表”的每一行，这个 key 都要被重复一次，产生大量冗余。这种做法通常被叫做 normalization。这种方法虽然可行，然而我不得不说这是一个“变通”。它的存在是为了绕过关系模型里面的无须有的限制，终究导致了关系式数据 库使用的繁琐。说白了，normalization 就是让你手动做一些比 C 语言的“手动内存管理”还要低级的工作，因为连 C 这么低级的语言都允许你在结构里面嵌套数组！然而，很多人宝贵的时间，就是在构造，释放，调试这些“中间表格”的工作中消磨掉了。

这些就是关系模型所有的秘密。如果你深刻的理解了数据结构的用法，那么通过反复推敲，深入理解以上这番“补充知识”，你就能把已知的数据结构常识应用到所谓的“关系模型”上面，从而对关系式数据库应用自如，甚至可以使用 SQL 写出非常复杂和高效的算法。

另外有一些人（比如这篇文章） 通过关系模型与其它数据模型（比如网状模型之类）的对比，以支持关系模型存在的必要性，然而如果你理解了这小节的所有细节就会发现，使用基本的数据结构， 其实可以完全的表示关系模型以及被它所“超越”的那些数据模型。说实话，我觉得这些所谓“数据模型”全都是故弄玄虚，无中生有。数据模型可以完全被普通的 数据结构所表示，然而它们却不可能表达数据结构带有的所有信息。这些模型之所以流行，是因为它们让人误以为知道了所谓的“一对一”，“一对多”等肤浅的概 念就可以取代设计数据结构所需要的技能。所以我认为它们其实也属于技术上的“减肥药”。

NoSQL 的“革命”

SQL 和关系模型所引起的这一系列无须有的问题，终究引发了所谓 NoSQL 的诞生。很多人认为 NoSQL 是划时代的革命，然而在我看来它很难被称为是一次“革命”，最多可以被称为“不再愚蠢”。而且大多数 NoSQL 数据库的设计者们并没有看到以上所述的问题，所以他们的设计并没有完全摆脱关系模型以及 SQL 带来的思维枷锁。

最早试图冲破关系模型和 SQL 限制的一种数据库叫做“列模式数据库”（column-based database），其代表包括 Vertica, HBase 等产品。这种数据库其实就是针对了我刚刚提到的，关系模型无法保存可变长度数组的问题。它们所谓的“列压缩”，其实不过是在“行结构”里面增加了对“数 组”的表示和实现。很显然，每一个数组需要一个字段来表示它的长度，剩下的空间用来依次保存每一个元素。所以在这种数据库里，你大部分时候不再需要进行 normalization，也不需要重复存储很多 key。这种对数组的表示，是一开始就应该有的，却被关系模型排除在外。然而，很多列模式数据库并没有看到这一实质。它们经常设定一些无端的限制，比如你 的变长数组只能有非常有限的嵌套层数之类，所以它们其实没能完全逃脱关系式数据库带来的思想枷锁。让我很惊讶的是，如此明显的事情，数据库专家们最开头恁是看不到。到后来改来改去改得六成对，还美其名曰“优化”和“压缩”。

最新的一些 NoSQL 数据库，比如 Neo4j, MongoDB 等，部分的针对了 SQL 的表达力问题。Neo4j 设计了个古怪又不中用的查询语言叫 Cy-

pher，MongoDB 使用冗长繁琐的 JSON 来直接表示对数据的查询，就像是在手写编译器里的 AST 数据结构。Neo4j 的 Cypher 语言不但语法古怪，表达力弱，而且效率非常低，以至于几乎任何有用的操作你都必须使用 Java 写扩展来完成（参考这篇博文）。所以到现在看来，数据库的主要问题已经转移到了语言设计的问题，而且它们会在很长一段时间之内处于混沌之中。

其实数据库的问题哪有那么困难。只要你有一个好的程序语言，你就可以发送这种语言的代码到“数据库服务器”，这个服务器可以远程执行你的代码，调用 服务器上的“库代码”对数据进行索引，查询和重构，然后返回代码指定的结果。如果你看清了 SQL 的实质，就会发现这样的“过程式设计”其实并不会损失 SQL 的“描述性语言”的表达能力。反而由于过程式语言使用的简单性，直接性和普遍性，会使得开发效率大大提高。NoSQL 数据库比起 SQL 和关系式数据库存在一些优势，也就是因为它们在朦胧中朝着这个方向发展。

然而，NoSQL 并不总是朝着正确的方向发展。因为设计它们的人往往没有专业的修习过程序语言的设计，缺乏对历史教训的认识，所以他们经常犯下不应有的设计错误。我经历过 好些 NoSQL 数据库之后发现，它们的查询语言设计几乎完全没有章法。而且由于具体的实现质量以及商业动机，这些数据库往往有各种各样恼人的问题。这是必然的现象，因为 这些数据库公司靠的就是咨询和服务作为收入，如果他们把这些数据库高质量又开源的实现，没有烦人的问题，谁会给他们付费呢?

所以，这些 NoSQL 数据库问题的存在，也许并不是因为人们都很笨，而是因为世界的经济体制仍然是资本主义，大家都需要骗钱糊口，大家都舍不得给“小费”。有人说我这是“阴谋 论”，可惜总有一天你会意识到，这的确是一个阴谋。如果你想知道跟 NoSQL 数据库公司打交道是什么感觉，可以参考一下我这篇博文里面关于 Neo4j 的部分。

总结

说了这么多，其实主要的只有几点:

1. 关系模型是一个无需有的东西。它严重束缚了人们的思想，其本质并不如普通的数据结构简单和高效。它比起逻辑式语言的理论来说是非常肤浅的。
2. SQL，Datalog，Prolog 等所谓“描述性语言”的价值被大大的高估了。使用它们的人往往有“避免编程”的心理，结果不得不做比编程还要痛苦的工作：数据库查询优化。
3. 数据库完全可以使用普通的程序语言（Java，Scheme 等）的“远程执行”来进行查询，而不需要专门的查询语言。这在某种程度上就是 NoSQL 数据库的实质和终极发展方向。

对数据库的问题有更多兴趣的人，可以参考我的一篇相关的英文博客，以及这篇《一种新的操作系统的设计》里面的相关部分。

习题

有些人评论说我其实不懂 SQL，现在我就来考考你的 SQL 编程能力，看看到底是谁理解 SQL 多一些 :)

题目：请用 SQL 实现 Dijkstra 的最短路径算法。

为了加深对数据库的认识，每个人都应该用 SQL 来实现这样稍微复杂的算法，而不只是写一些高中生都会的基础程序。如果你仍然相信“What，not How”的广告，反对使用 SQL 来写这样的过程式算法，那么就请你更进一步，使用你所谓的“What 查询方式”来高效的解决最短路径问题。

致谢

在这里我要感谢指导过我的数据库教授 Dirk Van Gucht。当年是他的作业让我们用 SQL 写出像 Dijkstra 最短路径算法，以及各种有一定难度的递归代码，引发了我对数据库本质的思索。当时我是全班唯一能正确完成所有这些代码的人，是助教的“标准答案”的来源。 在后来的几年里，我成为了 Dirk 的课程助教，在帮助学生们的同时，我继续思索数据库的本质，从而间接地导致了本文的诞生。

原文链接：http://www.yinwang.org/blog-cn/2014/04/24/sql-nosql/

磁盘满了MySQL会做什么?

作者：@billy鹏的足迹

新浪微博组DBA主管。新浪数据库平台长期招聘，推荐或自荐请私信。让我一起来挑战高并发和海量数据吧。

最近遇到一个故障和磁盘满有关系，并且同事也发现经常有磁盘满导致操作hang住无响应的情况，于是抽时间研究了一下这2种情况。

一、磁盘满了之后MySQL会做什么?

我们看下官方的说法

```
When a disk-full condition occurs, MySQL does the following:
    * It checks once every minute to see whether there is enough space
to write the current row. If there is enough space,it continues as if
nothing had happened.
* Every 10 minutes it writes an entry to the log file, warning about
  the disk-full condition.
```

其实MySQL本身并不会做任何操作，如官方文档说说，只会每分钟check一次是否有空闲空间，并且10分钟写一次错误日志。

但是再次期间由于磁盘满了，意味着binlog无法更新，redo log也无法更新，所有buffer pool中的数据无法被flush上，如果不幸的服务器重启，或者实例被kill了，那必然会造成数据丢失，这几乎是一定的。所以，处理磁盘满的问题最好 是先释放出来一定空间让dirty数据刷新下来。

二、磁盘满了为什么会导致操作hang住?

1、select

首先经过经验和实际测试，select操作不会由于磁盘满导致问题，也就是所有select操作都会正常运行。

2、insert

经过不通的测试发现，当磁盘满了之后，并不是第一个insert就卡住，而是会在n个之后出现卡住的情况。

通过查看error日志，发现卡住现象和刷磁盘的操作有关系。

```
[ERROR] /usr/local/mysql-5.1.42/libexec/mysqld: Disk is full writing
'./test/cj_webex.MYD'

[ERROR] /usr/local/mysql-5.1.42/libexec/mysqld: Disk is full writing
'./mysql-bin.000017'
```

为了验证推论是否正确，我们将sync_binlog设置为1，在这种情况下，insert第一条就卡住了，并且error log中直接报错提示写bin-log失败。看来卡住确实和刷磁盘有关系。

目前已知和刷磁盘有关系的参数有3个，分别是sync_binlog，innodb_flush_log_tr_commit，和duoblewrite。

3、show slave status

在从库经过测试，操作会被卡住，这主要是由于执行show slave status需要获得LOCK_active_mi锁，然后锁上mi->data_lock，但是由于磁盘满了无法将io_thread中的数据写入到relay log中，导致io_thread持有mi->data_lock锁，这就导致了死锁。

所以，这就导致在磁盘满的情况下，执行show slave status操作会卡住。

4、show status

测试可以正常操作，但是如果先执行了show slave status操作的情况下，show status也会被卡住。这是因为执行show status需要锁上LOCK_status， 而由于status状态中包含slave status，所以还需要锁上LOCK_active_mi。如果限制性了show slave status，这时候由于mi->data_lock死锁问题，导致io_thread不会释放LOCK_active_mi锁。这时候就导致 show status和show slave status争抢同一把LOCK_active_mi锁，也形成了死锁。

所以，在磁盘满的情况下，如果先执行show slave status，后执行show status，连个操作都会卡住。

原文链接：http://www.cnblogs.com/billyxp/p/3675820.html

不做运维，不懂日志——日志编码

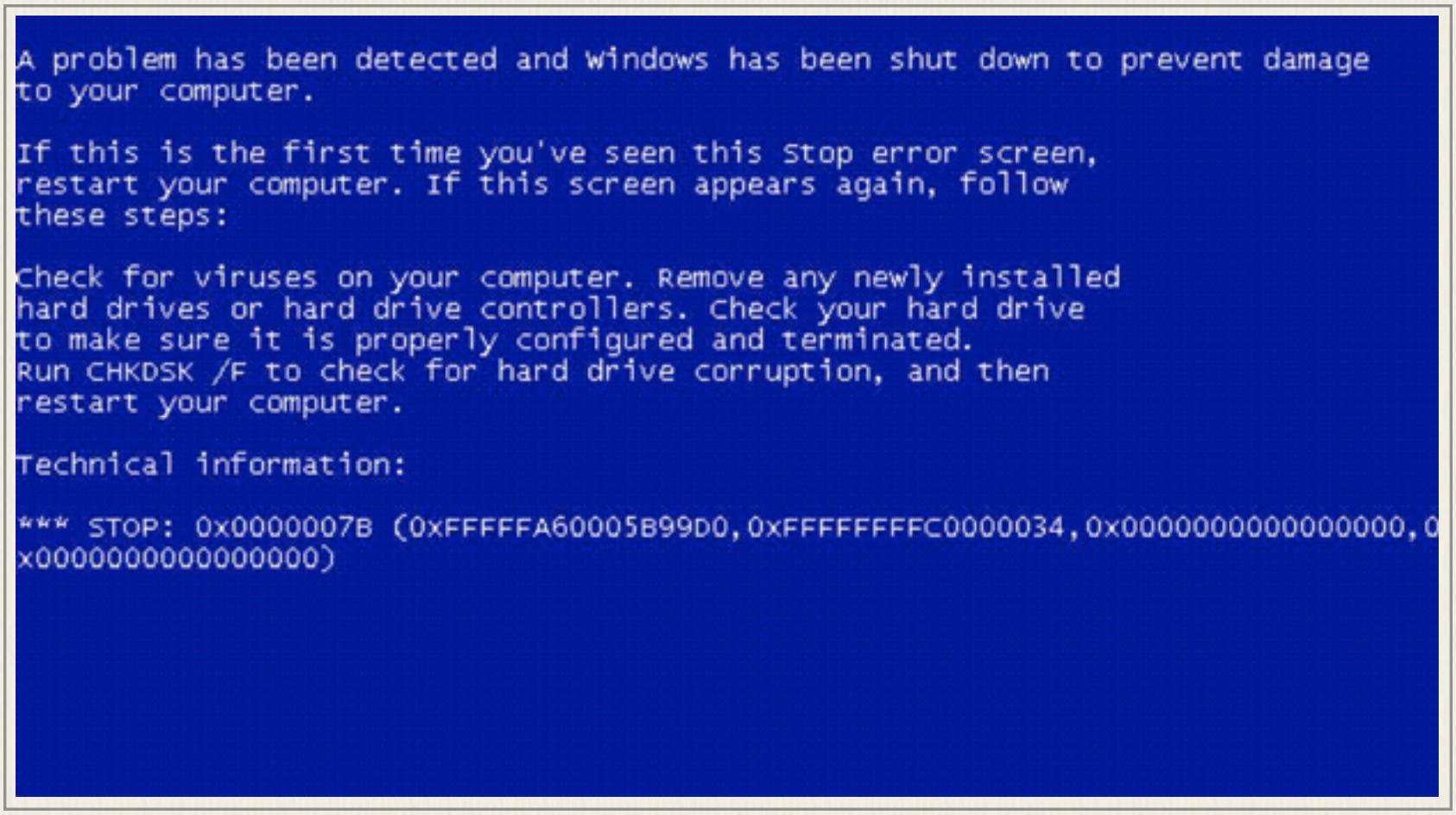
作者：@juvenxu
网站：juvenxu.com
程序员，敏捷教练，《Maven实战》作者

一个成功的软件，全力开发的时间可能占其整个生命周期的1/4还不到，软件发布后要运维（Operation），运维的视角和开发的视角是很不一样 的，但是有一点，运维的数据能反哺开发，同时，开发的时候也得考虑可运维性，其中非常重要的一点是日志，没有日志，运维就瞎了大半。怎么写日志，就得从运 维的需求来看了，通常会有以下一些常见的场景（已典型互联网应用为例）：

1. 访问来源，包括访问量，访问者数据，如用户名、IP等等。
2. 基于上一点细化，访问的接口，读、写、删……
3. 软件系统内部的核心链路，比如我这有个系统要在中美直接同步文件，那同步的情况运维的时候就要掌握。
4. 软件系统对其他所依赖系统的访问情况，比如我这个系统依赖一个分布式缓存，那访问缓存的量、是否超时等情况需要了解。
5. 系统异常，比如磁盘满了。

记录这些信息的目的大抵有：帮助分析系统容量方便扩容；在系统某些部分工作不正常的时候及早发现；发生严重故障后方面定位问题原因。认识到这些需求后，下一步就是怎么实现的问题了。

前面提到的5点，有些可以通过抛异常实现，例如访问分布式缓存超时，有些则显然不是异常，例如就是正常的缓存访问。我觉得可以用一种统一、规范的方 式记录，这种方法就是打码。我记得以前用Windows 98/2000的时候，经常会遇到蓝屏，蓝屏上会有一堆我看不懂的英文，并且总是伴随着一个错误码。



虽然我看这玩意儿没一点好心情，但我相信微软的工程师肯定能从那个奇怪的状态码上判断出是哪里出了问题，硬盘坏道？光驱卡死？诸如此类……其实类似的做法数据库也用，比如MySQL。

用统一的代码表示错误（也可以表示正常但核心的业务点）最大的好处就是便于搜索、统计和分析，在动辄数以万行记的日志文件中寻找感兴趣的信息，一页 一页翻看是不现实的，稍微做过点运维的必然会用上 grep,awk,wc 等工具，这个时候如果信息都有代码标识，那真是再方便不过了！例如，我用代码 FS_DOWN_200 表示对系统的正常下载访问，日志是写在 monitor.log 文件中的，我就可以使用一行shell统计4月22号5点到6点之间的正常访问量：

```
$ grep FS_DOWN_200 monitor.log | grep "2014-04-22 05:" | wc -l
1
$ grep FS_DOWN_200 monitor.log | grep "2014-04-22 05:" | wc -l
```

具体每条日志记录什么，那就是更详细的了，基本就是时间、日志编码、额外的有用信息，如：

```
2014-04-22 05:06:18,561 - FS_DOWN_200 216 UT8TFSDXc8XXXagOFbXj.jpg
1
2014-04-22 05:06:18,561 - FS_DOWN_200 216 UT8TFSDXc8XXXagOFbXj.jpg
```

除了时间和日志编码外，还有响应时间（216ms）和具体访问的文件名。

当然如果你有日志监控和分析系统就更棒了！你就可以在系统中录入关键字监控，比如每分钟统计次数，然后看一天、一周的访问量趋势图。进一步的，如果这个量发生异常，让系统发出报警。如果没有关键字，从海量日志中分析纷繁复杂形态各异的信息，再监控，是非常难的一件事情。

为什么要把日志代码设计成 FS_DOWN_200 这样子的，下面稍微解释下，这个代码分成三段：

1. FS： 表示我们的系统，这是最高的级别，公司中有很多系统，那各自定义自己的标识。
2. DOWN： 表示我们系统中的一个核心业务点或者对其他依赖系统的访问，还可以是UP（上传），SYNC（同步），或者TAIR（对缓存系统访问）。
3. 200： 具体健康码，参考HTTP规范，200表示OK，其他包括404（不存在），504（超时）等等。

有了这些代码，再结合公司的监控系统，我们做统计分析就非常方便了，每天多少下载、多少上传、多少成功、多少失败、对其他依赖系统访问多少量、多少 失败率，一目了然。进一步的加上监控，当某些值突然发生变化，比如下载量/
上传量暴跌、访问其他系统依赖超时大量增多，就能及时响应。

日志对于运维实在太重要了，而如果不接触运维，又怎能理解其真正的需求，因此我说，**不做运维，不懂日志。**

本实践受 @linux_china 和 Release It! 17.4 小节启发，表示感谢。
原文链接：http://www.juvenxu.com/2014/04/22/log-with-code/

再见了NSLog

作者：**@随缘·Sunny**

我是前言

打Log是我们debug时最简单朴素的方法，NSLog对于objc开发就像printf对于c一样重要。但在使用NSLog打印大量Log，尤其是在游戏开发时（如每一帧都打印数据），NSLog会明显的拖慢程序的运行速度（游戏帧速严重下滑）。本文探究了一下NSLog如此之慢的原因，并尝试使用lldb断点调试器替代NSLog进行debug log。

小测试

测试下分别使用NSLog和printf打印10000次耗费的时间。

CFAbsoluteTimeGetCurrent()函数可以打印出当前的时间戳，精度还是很高的，于是乎测试代码如下：

```
CFAbsoluteTime startNSLog = CFAbsoluteTimeGetCurrent();
for (int i = 0; i < 10000; i++) {
    NSLog(@"%d", i);
}
CFAbsoluteTime endNSLog = CFAbsoluteTimeGetCurrent();
CFAbsoluteTime startPrintf = CFAbsoluteTimeGetCurrent();
for (int i = 0; i < 10000; i++) {
    printf("%d\n", i);
}
CFAbsoluteTime endPrintf = CFAbsoluteTimeGetCurrent();
NSLog(@"NSLog time: %lf, printf time: %lf", endNSLog - startNSLog,
endPrintf - startPrintf);
```

这个时间和机器肯定有关系，只看它们的差别就好。为了全面性，尝试了三种平台：

```
NSLog time: 4.985445, printf time: 0.084193 // mac
NSLog time: 5.562460, printf time: 0.019408 // 模拟器
NSLog time: 10.471490, printf time: 0.090503 // 真机调试(iphone5)
```

可以发现，在mac上（模拟器其实也算是mac吧）速度差别达到了60倍左右，而真机调试甚至达到了离谱的100多倍。

探究原因

基本上这种事情一定可以在Apple文档中找到，看NSLog的文档，第一句话就说：Logs an error message to the Apple System Log facility.，所以首先，NSLog就不是设计作为普通的debug log的，而是error log；其次，NSLog也并非是printf的简单封装，而是Apple System Log(后面简称ASL)的封装。

ASL

ASL是个啥？从官方手册上，或者从终端执行man 3 asl都可以看到说明：

```
These routines provide an interface to the Apple System Log facility.
They are intended to be a
replacement for the syslog(3) API, which will continue to be supported for backwards compatibility.
```

大概就是个系统级别的log工具吧，syslog的替代版，提供了一系列强大的log功能。不过一般我们接触不到，NSLog就对它提供了高层次的封装，如这篇文档所提到的：

You can use two interfaces in OS X to log messages: ASL and Syslog. You can also use a number of higher-level approaches such as NSLog. However, because most daemons are not linked against Foundation or the Application Kit, the low-level APIs are often more appropriate

一些底层相关的守护进程(deamons)不会link如Foundation等高层框架，所以asl用在这儿正合适；而对于应用层的用NSLog。

在CocoaLumberjack的文档中也说了NSLog效率低下的问题：

NSLog does 2 things:

- It writes log messages to the Apple System Logging (asl) facility. This allows log messages to show up in Console.app.
- It also checks to see if the application's stderr stream is going to a terminal (such as when the application is being run via Xcode). If so it writes the log message to stderr (so that it shows up in the Xcode console).

To send a log message to the ASL facility, you basically open a client connection to the ASL daemon and send the message. BUT - each thread must use a separate client connection. So, to be thread safe, every time NSLog is called it opens a new asl client connection, sends the message, and then closes the connection.

意识大概是说，NSLog会向ASL写log，同时向Terminal写log，而且同时会出现在Console.app中（Mac自带软件，用NSLog打出的log在其中全部可见）；不仅如此，每一次NSLog都会新建一个ASL client并向ASL守护进程发起连接，log之后再关闭连接。所以说，当这个过程出现N次时，消耗大量资源导致程序变慢也就不奇怪了。

时间和进程信息

主要原因已经找到，还有个值得注意的问题是NSLog每次会将当前的系统时间，进程和线程信息等作为前缀也打印出来，如：

```
2012-34-56 12:34:56.789 XXXXXXXX[36818:303] xxxxxx
```

当然这些也可能是作为ASL的参数创建的，但不论如何，一定是有消耗的（虽然这个prefix十有八九不是我们需要的看到的）

如何是好

NSLog有这样的消耗问题，那该怎么办呢?

1. 拒绝残留的Log。现在项目都是多人共同开发，我们应该只把Log作为错误日志或者重要信息的日志使用，commit前请把自己调试的log去掉（尤其是在循环里写log的小伙伴，简直不能一起快乐的玩耍了）
2. release版本中消除Log。debug归debug，再慢也不能波及到release版本，用预编译宏过滤下就好。
3. 是时候换个Log系统了，如CocoaLumberjack，自建一个简单的当然也挺好（其实为了项目需要自己也写了个小log系统，实现可以按名字和级别显示log和一些扩展功能，以后有机会分享下）

不过个人认为de-
bug时最好还是用调试器进行调试（尤其是只需要知道某个变量值的时候）

尝试使用断点+lldb调试器打Log

关于强大的lldb调试器用一个专题来讲都是应该，现在只了解一些皮毛，不过就算皮毛的功能也可以替代NSLog这种方法进行调试了，重要的一点是:**使用断点log不需要重新编译工程**，况且和Xcode已经结合的很好，在此先只说打Log这件事。

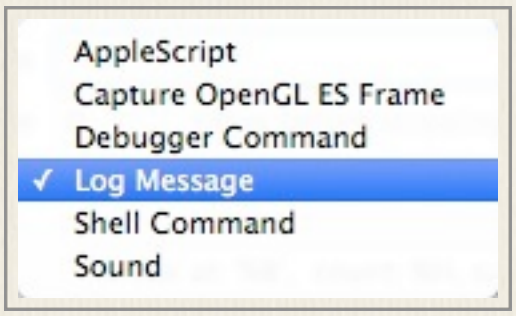
简单断点+po(p)

断点时可以在xcode的lldb调试区使用po或p命令打印对象或变量，对于当前栈帧中引用到的变量都是可见的，所以说假如只是看一眼某个对象运行到这儿是不是存在，是什么值的话，设个断点就够了，况且IDE已经把这个功能集成，鼠标放变量上就可以了。

lldb一些常用调试技巧可以这篇入门教程

Condition和Action断点

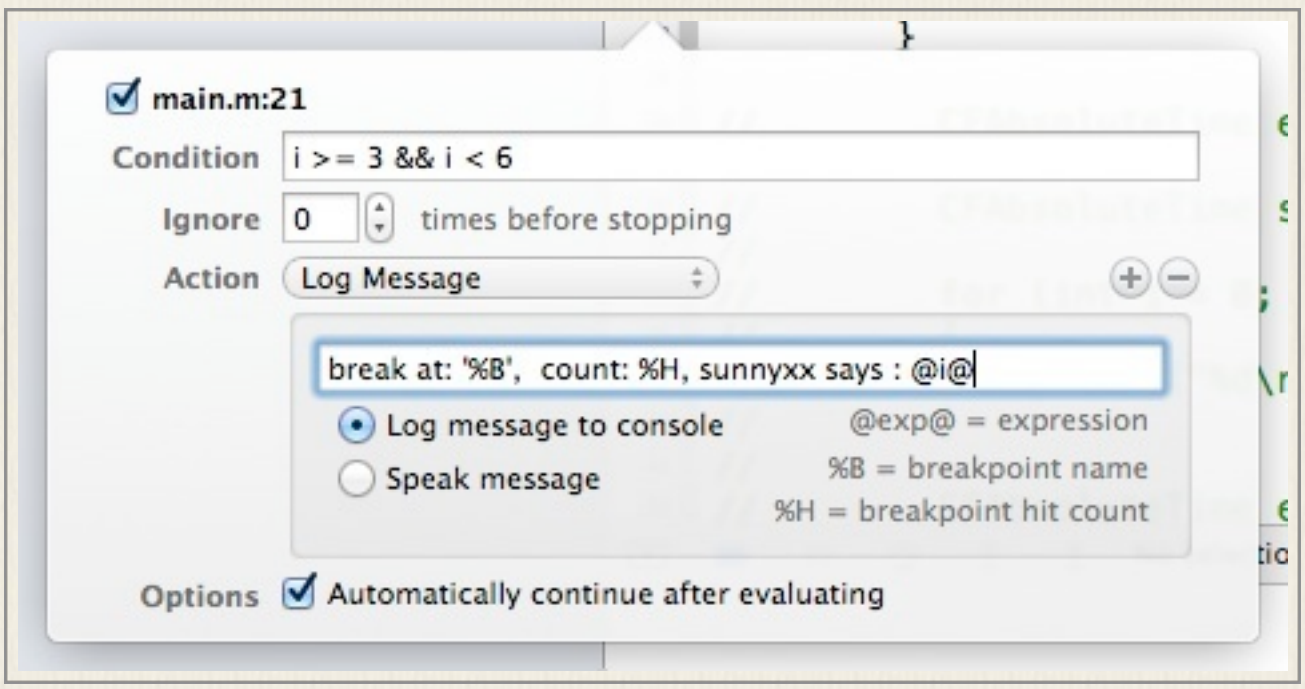
断点不止能把程序断住，触发时也按一定条件，而且可以执行（一个或多个）Action，在断点上右键选择Edit Break-
point，弹出的断点设置中可以添加一些Action：



其中专门有一项就是Log Message，做个小测试：

```
for (int i = 0; i < 10; i++)
{
    // break point here
}
```

设置断点后编辑断点：



输入框下面就有支持的格式，表达式(或变量)可以使用@exp@这种格式包起来。于是乎输出：

```
break at: 'main()',  count: 4, sunnyxx says : 3
break at: 'main()',  count: 5, sunnyxx says : 4
break at: 'main()',  count: 6, sunnyxx says : 5
```

正如所料。

更多的调试技巧还需要深入研究，不过可以肯定的是，比起单纯的使用NSLog，使用好的工具可以让我们debug的效率更高

总结

- NSLog耗费比较大的资源
- NSLog被设计为error log，是ASL的高层封装
- 在项目中避免提交commit自己的Debug log，release版本更要注意去除NSLog，可以使用自建的log系统或好用的log系统来替代NSLog
- debug不应只局限于log满天飞，lldb断点调试是一个优秀的debug方法，需要再深入研究下

References

https://developer.apple.com/library/mac/documentation/Darwin/Reference/ManPages/man3/asl.3.html

http://theonlylars.com/blog/2012/07/03/ditching-nslog-advanced-ios-logging-part-1/

https://github.com/CocoaLumberjack/CocoaLumberjack/wiki/Performance

https://developer.apple.com/library/mac/documentation/MacOSX/Conceptual/BPSystemStartup/Chapters/LoggingErrorsAndWarnings.html#//apple_ref/doc/uid/10000172i-SW8-SW1

http://www.cimgf.com/2012/12/13/xcode-lldb-tutorial/

原文链接：http://blog.sunnyxx.com/2014/04/22/objc_dig_nslog/

微博平台的RPC服务化实践

作者：@唐福林
微博开放平台，伪架构师，兼职客服，负责微博底层性能优化，也负责 t.cn 短链，关注粉丝，分组和计数器。短链，关注，粉丝，分组，计数错误都可以找我

2014年第一分钟，新浪微博的发布量以808298条再次刷新记录，第一秒微博发布量相较去年提升55%。（数据来源：新浪科技）这是微博平台 RPC 框架 “Motan” 上线后第一次抗峰值，整体表现平稳，基本达到最初的“应用方无感知”的目标。

在RPC服务化这个事情上，微博平台不是第一个吃螃蟹的：早的有亚马逊和eBay等国外先驱，近的有Twitter的finagle，淘宝的 dubbo等等，网上各种公开的资料铺天盖地。另一方面，单纯的RPC调用功能实现，从技术上看其实并不复杂：client 发起调用，框架拦截调用信息，序列化，传输，server端收到调用信息，反序列化，根据调用信息发起实际调用获取结果，再原路返回。实现这些功能可能也 就三五天的事情，但在一个复杂的业务环境下，稳定可靠的应用它，才是最大的挑战。

微博平台的 RPC 服务化拆分历程始于2013年7月。在此之前，我们花了很长的时间讨论服务化的目标，主要是项目的范围：哪些问题不属于服务化项目需要解决的问题。 实际的框架代码开发花了三个工程师（王喆 @wangzhe_asdf9 陈波@fishermen 麦俊生@麦俊生）大约一个月时间，然后花了将近两个月的时间推动在第一个业务上线：调整工程师的开发模式，调整测试流程，修改上线系统，添加监控和报警， 小流量测试，灰度发布，最后才是全量上线。然后又花了一个月，在微博平台主要业务中全部上线。

微博RPC的一些基本的数据指标：

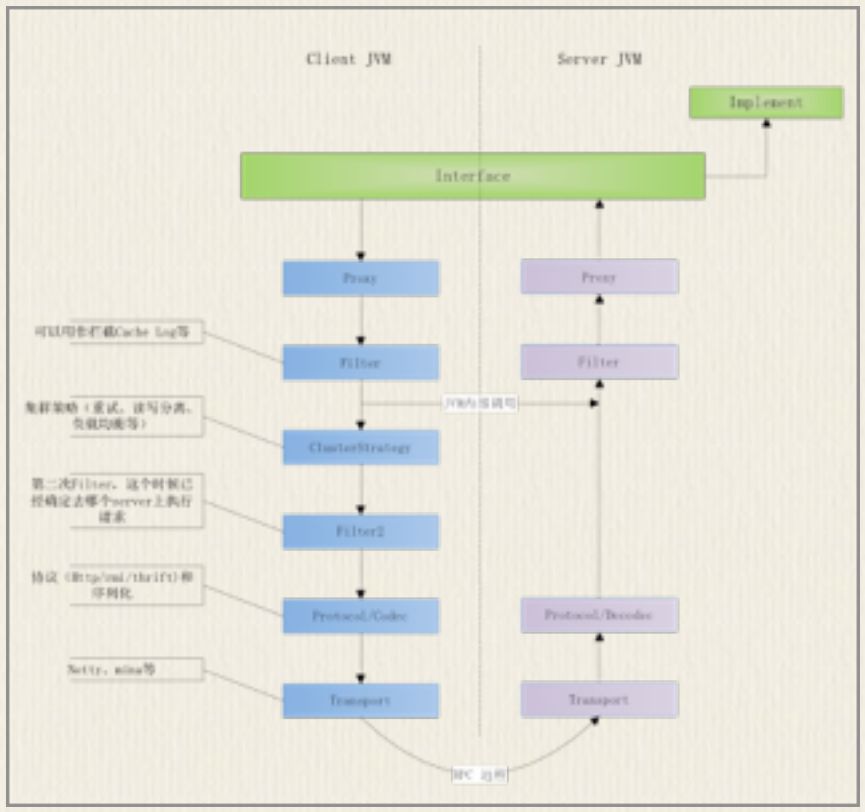
- Motan 框架：2w+ 行 Java 代码，1w 行 test 代码，UT行覆盖率超过70%（当前 Motan 实现中，与微博平台内部多个系统都有功能绑定，还不具备开源条件，但开源是我们从一开始就设立好的目标之一）
- 支持 2 种调用方式：inJVM 和 TCP远程调用。inJVM 方式类似 loop-back 网卡：数据经过了协议栈流程处理，但没有流经真正的网络设备。inJVM方式主要用来支持开发调试和测试，以及在RPC服务上线初期作为Fail-Back降级使用
- 典型业务场景下单实例 tps 极限 20k，微博平台一般采用单机双实例，即单机极限 40k
- 典型业务场景下平均响应时间 <3 ms，框架层额外消耗 < 0.01 ms
- 最大的单个核心业务日调用量超过 800亿次

RPC服务化的目的大约有两种：将一个大一统的应用拆分成多个小的RPC服务，那么目的就是为了解耦和，提升开发效率；如果是将传统的Http或其它方式远程调用改造成高效的RPC调用，那么就是为了提升运行效率。不幸的是，微博平台的RPC框架，需要同时达到这两个目的：既要在平台内部将一个大一 统的应用拆分，又要考虑到后续向开放平台的大客户们提供RPC接入的可能。因此，微博平台在技术选型和方案设计上做了很多的权衡和妥协：

- 首先，是选择已有开源方案，还是自己开发一个新方案？选择的依据按重要程度排序：是否满足自己的核心需求，方案成熟度，认知成本（即二次 开发难度）。由于是拆分一个已有的复杂应用，微博平台的一个核心需求是：应用开发方希望尽可能的平滑迁移，最好能做到应用方无感知。我们评估的多个开源方 案没有一个能满足，所以只能自己做一个了
- 灵活性与误用的可能性：框架开发方总是有一个偏见，觉得我这个框架越灵活越好，最好每个步骤每个环节都是可以由使用方自己配置或定义。但 对于一个内部强制使用的框架来说，使用方式的统一性也同样重要，换句话说，对于大部分的环节步骤，都需要保证团队内部各使用方都按同样的方式进行配置，防 止误用，并降低学习和沟通成本。我

们的经验是，框架开发完成后，还需要有“框架使用方”角色，将所有的灵活性限制在框架使用方的层面，避免直接暴露所有细 节到最终的业务开发方

- 序列化方式选择：微博平台从2011年引入了 PB 序列化方式，以替代 cache 和 db 中的 json 文本。但在 RPC 框架上线过程中，我们选择了对 Java 对象更为友好的 Hessian2 。因为之前的 PB 序列化需要定义 proto 文件和生成代码，平台只对必要的 model 类做了支持，而 rpc 可能涉及到更多的 wrap 类，业务逻辑类等，为所有的类提供 pb 支持的工作量太大，而且后期维护困难。当然了，Motan 框架支持各种不同的序列化方式配置。
- 通讯协议选择：在评估了几个开源RPC框架的协议设计后，我们最终选择了在 TCP 链接基础上设计自己的 RPC 通讯协议，一个简单的二进制协议：定长 header 中包含一个 length 字段，然后就是二进制的 body payload，即序列化之后的 rpc request 或 response 。
- 集群管理：微博平台Motan框架当前依赖于内部开发的Config Service（Code name Vintage，based on ZooKeeper）来进行服务注册，服务发现和变更通知。
- Trace系统：微博平台Motan框架当前依赖于内部开发的类似Twitter Zipkin的Trace系统（Code name Watchman）来对RPC请求做抽样及全量trace。



Motan 架构图（RPC调用数据流图）

一套新的架构在大规模的推广使用过程中总会遇到各种问题，微博平台RPC服务化也不例外。总结起来，我们遇到的问题包括：

1. 整体服务的SLA水平降低。由于前端接口依赖多个后端RPC服务，每个RPC服务风吹草动都会直接影响接口成功率。初期改造过程中，对RPC服务并没有做明确的SLA要求，加上前端有些地方对调用超时异常兼容不够，导致前端调用失败几率增加，接口成功率降低。后来我们对每一个RPC服务 设立了具体的SLA要求，并优化了前端重试及失败处理机制，从而保证了整体服务的SLA。
2. 必须提供稳定的测试环境。前期改造过程中，服务调用方在进行线下测试时，由于被调用方同时也在调整，导致经常出现测试环境服务不可用的情 况，严重影响了调用方的测试使用。因此快速搭建一个独立的以RPC服务为单位的测试环境，在整个服务化过程中还是十分重要的。当前微博平台以 OpenStack 为基础搭建了一套快速测试环境分配系统，支持RPC服务粒度的测试环境分配。
3. 统一的开发、测试、上线、监控流程。在初期改造中，由于涉及多个部门和开发团队，各RPC服务的开发测试上线流程都沿用原来团队的做法， 不统一，导致多个互相依赖的RPC服务同时上线新版本的时候，经常出现衔接不上，无法线下集成测试，只能线上测试的情况，严重影响业务迭代速度。后来平台 通过搭建一套统一的CI流程以及优化运维上线系统，初步解决了这个问题。

微博平台 RPC 服务化拆分的故事，2014年依然继续：当前 Motan 框架完成了在微博平台的核心业务上线，接下来，我们的工作重点方向包括：

- 多语言支持：RPC多语言支持很难，如果没有特别的理由，建议绕过。RPC多语言有两种不同的思路：一种是类似 Thrift/PB 定义语言无关的 proto，自动生成对应语言的代码；另一种是没有 proto 定义，只有文档规范说明，业务方自己实现，或者使用框架自带的lib实现，类似 Hessian/MsgPack 。微博平台选择的是第二种，当前已经支持 PHP client （兼容 Yar 协议），以及 C Server 。

- RPC Service 运行时环境支持：为 Java 和 C service 提供统一的运行时环境支持。对于 Java RPC Service，目标是让业务方像写 Servlet 一样写 Service，统一打成 war 包后在 Tomcat 中运行；对于 C Service，将业务代码抽离成 so，由框架进行加载。
- 标准化 RPC 接口改造，并推广到其它部门及开放平台使用：将当前平台内部的 Service 改造成类似微博 OpenApi 那样的标准化接口，推广给其它部门使用，并最终通过微博开放平台，开放给外部开发者使用。

微博RPC团队招聘进行中，欢迎感兴趣有想法的同学们一起来参与!

关于作者

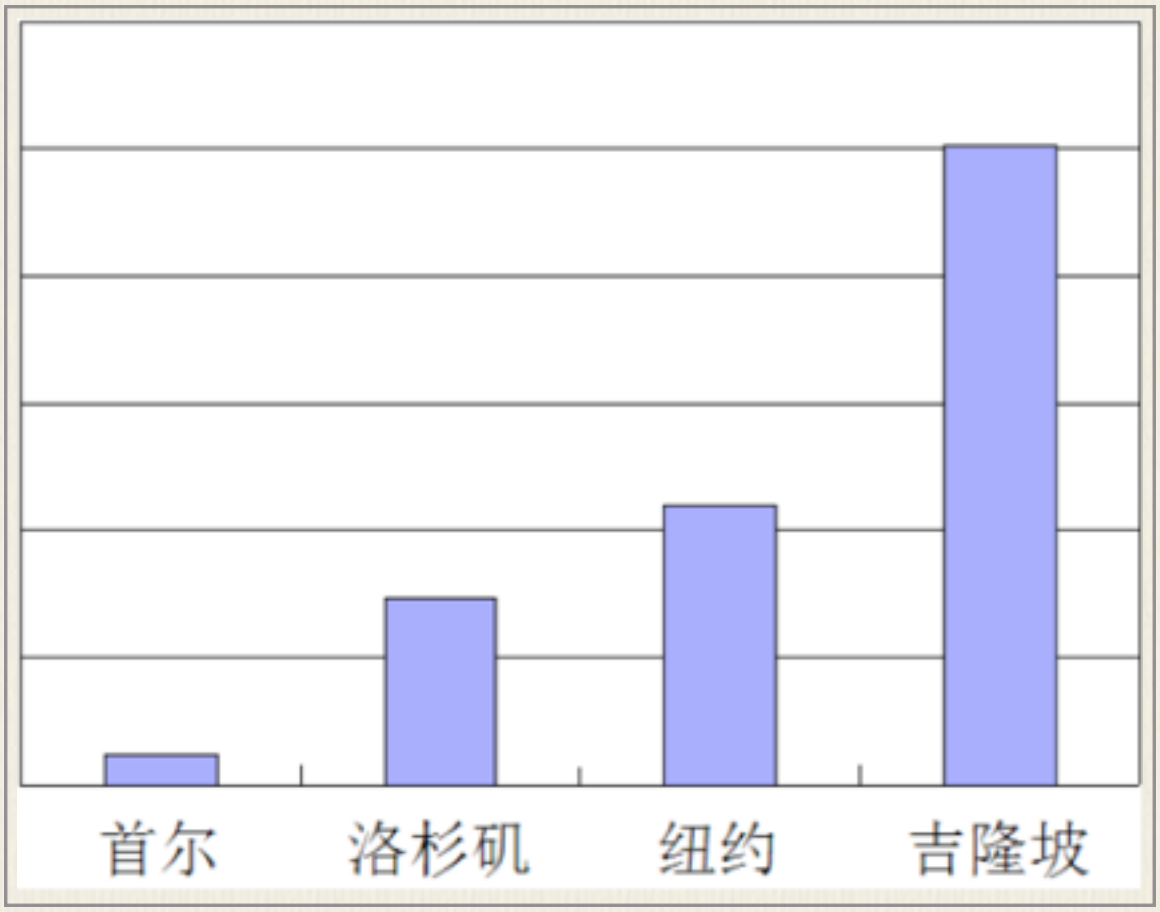
唐福林（@唐福林），微博技术委员会成员，微博平台资深架 构师，致力于高性能高可用互联网服务开发，及高效率团队建设。从2010年开始深度参与微博平台的建设，目前工作重心为微博服务在无线环境下 的端到端全链路优化。业余时间他是一个一岁女孩的爸爸，最擅长以45°凉开水冲泡奶粉。

原文链接：http://www.infoq.com/cn/articles/weibo-rpc-practice

微博多机房体系

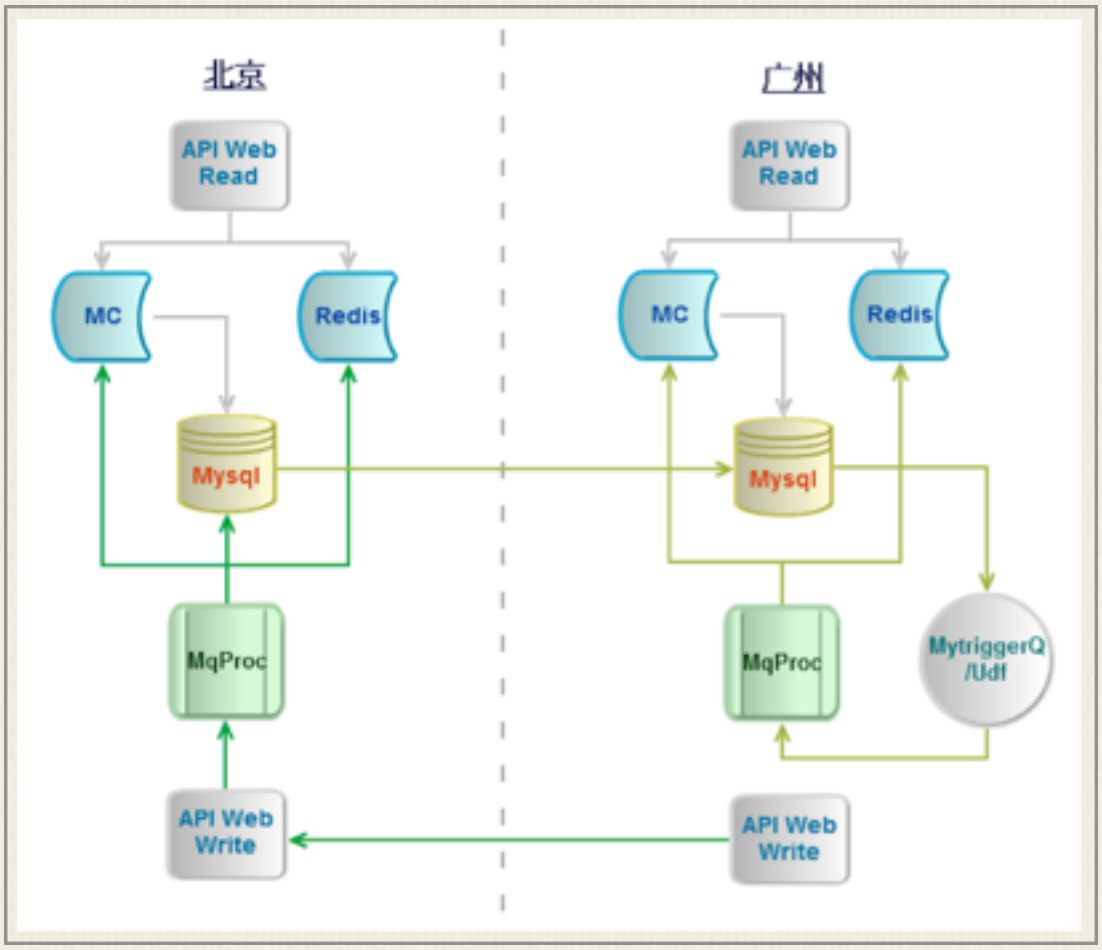
作者：陈飞

在国内网络环境下，单机房的可靠性无法满足大型互联网服务的要求，如机房掉电，光缆被挖的情况也发生过。微信就曾发生大面积故障，包括微信信息无法 发出、无法刷新朋友圈、无法连接微信网页版，或接收到的图片无法打开等。同时，微信公众平台也出现了503报错，范围影响北京、上海、广东、浙江等近20 个省市。故障的原因，微信团队指出是由于“市政道路施工导致通信光缆被挖断，影响了微信服务器的正常连接”。单机房除了单点风险之外，另外一个问题是不适 应复杂的网络环境，下图是调研国外各地区用户访问微博的延迟情况，可以看到南方海外用户的访问延迟比较大。



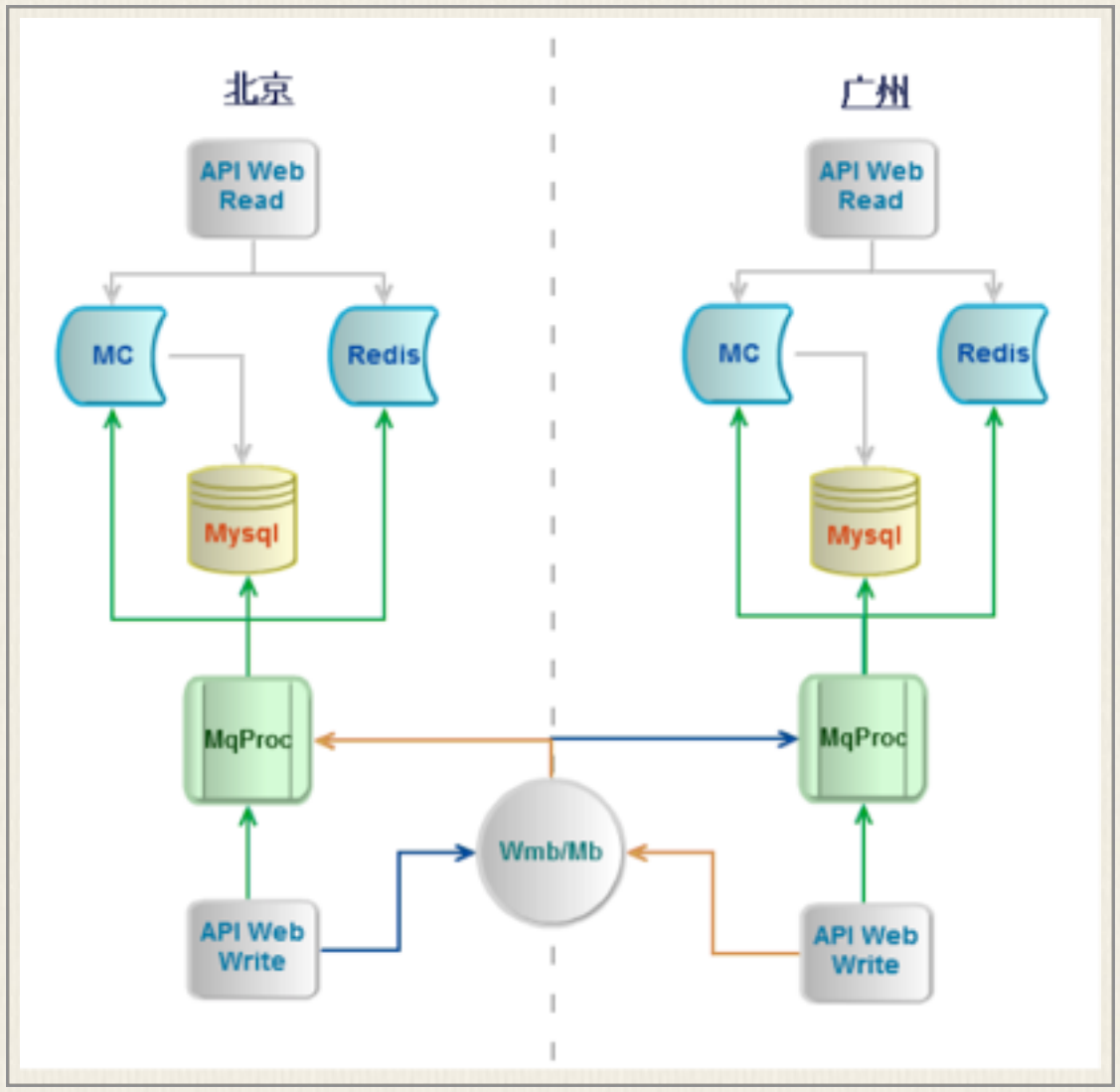
为了解决上述问题，微博在2010年启动了多机房部署的架构升级。微博不同于静态内容，静态内容CDN基本上大的互联网公司都会做，已经非常成熟。 动态内容CDN是业内的难点，国内很少有公司能够做到非常成熟的多机房动态内容发布的成熟方案。同时根据微博业务特点，又要求多机房方案能够支持海量规 模、可扩展、高性能、低延迟、高可用，所以面临很多技术挑战。 微博的多机房架构V1版本，解决了海量动态数据的CDN同步问题。一般业界的数据同步架构可以归纳为三种：Master-Slave，Multi-Master，Paxos。考虑到微博的特点是海量数据，低延迟，弱一致性，所以

Paxos并不适合微博，而Multi-Master在当时并没有成熟的 产品，所以微博开始采用的是Master-Slave方案，如下图：



由于Memcached服务端是无状态的，分布式是在客户端实现，所以需要解决两个机房Memcached数据同步的问题。微博研发了MytriggerQ，通过解析Mysql的binlog，还原更新操作实现Memcached数据同步。

数据库方面Mysql自身的Master-Slave同步实现是比较成熟的。但是在微博的海量数据情况下，广州两从的结构就导致同步的数据量翻倍， 导致带宽被大量占用。针对这个问题微博是通过Relay的方式来解决，即北京到广州仅需要同步一份数据，到广州后再由Relay服务器同步两份数据给从 库。由于Relay服务器可以代理多个从库，所以在基本没有增加资源的情况下，我们把同步带宽降低了一倍。 而Redis的同步实现就不太成熟了，由于不支持断点续传，一旦网络抖动导致主从不一致后，导致大量的带宽被占用，甚至出现过专线100%被占用的情况， 严重影响正常的机房间通信，同步恢复时间需要几个小时甚至几天。所以微博对Redis的同步机制进行了改造，利用AOF特性支持断点续传。改造后即使在专 线中断的情况下，同步也可以在几秒钟内恢复正常。 V1版本实现了微博多机房从无到有，V2版本重点解决了多机房的可靠性和可扩展性。V2版本实现了Master-Master架构，通过消息总线同步用户 操作行为，而不再依赖底层存储系统的同步，每个机房都独立完成读写操作。

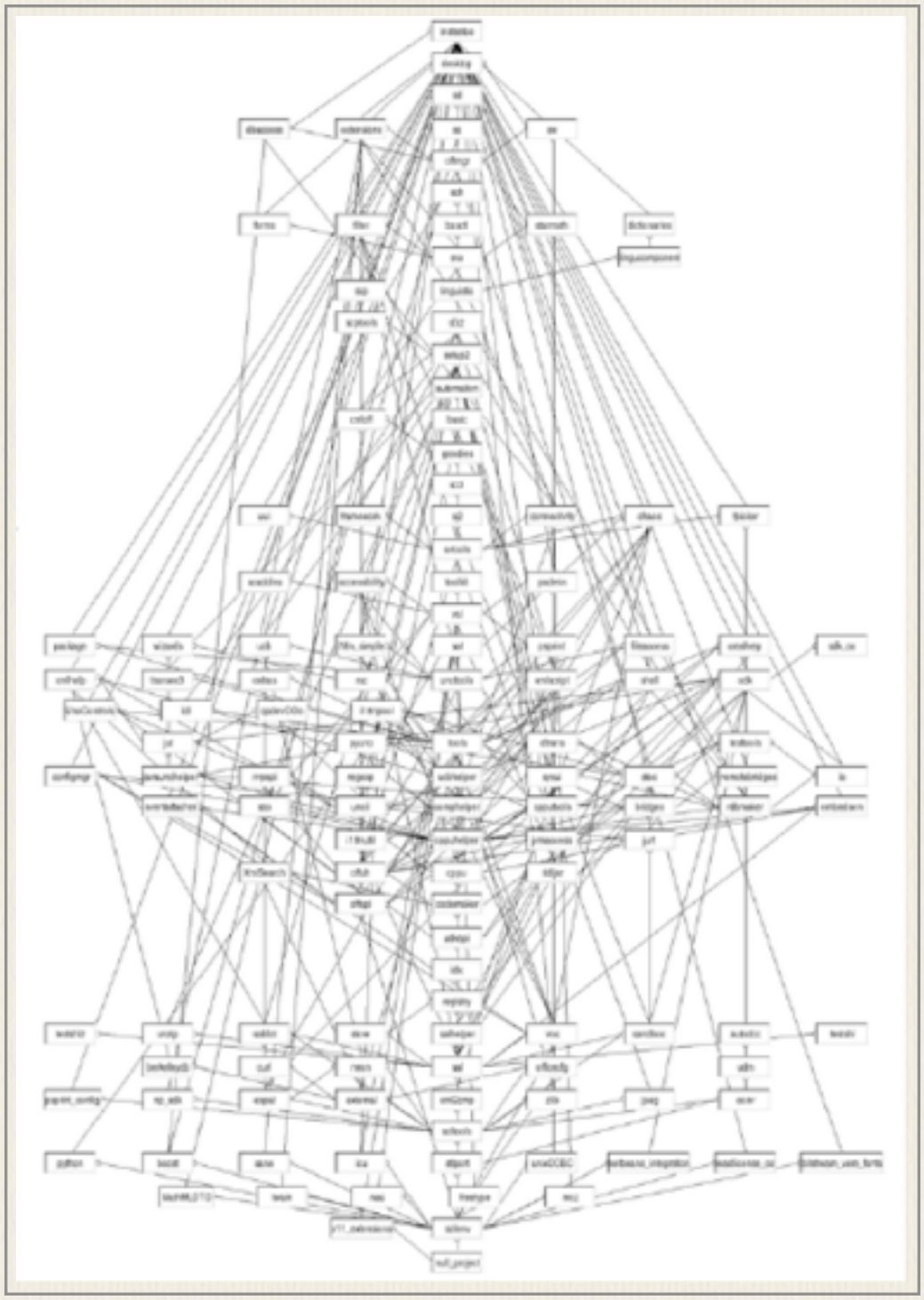


面对微博海量实时数据，业界通用的消息总线产品无法满足性能要求，所以我们自己基于MemcacheQ实现了一套消息总线WMB，它与普通消息总线产品最大的差别是采用一写一读的方式实现消息同步。这种方式最大的好处是消除了并发锁消耗，单机性能可以发挥到极致，而吞吐量可以通过增加机器线性扩容。 目前这套消息总线同步性能单机极限达到每秒10万消息同步性能。

可靠性方面，由于各机房仅通过消息总线进行同步，不依赖任何底层资源，所以各个机房都可以独立对外提供服务，任何一个机房出现问题都可以实现流量快 速切换。可扩展性方面，增加一个机房仅需线性扩展消息总线即可完成，机房的部署结构与数据同步对业务完全透明。微博多机房已经实现从北京、广州两个机房的 结构升级到广州亚太、北京电信、北京联通三个核心机房的部署结构。

Master-Master架构非常依赖消息总线的一致性，而在网络延迟比较验证的多机房环境下，MemcacheQ存在消息丢失的隐患，即而服务 端完成消息读取，但在传输过程中超时，客户端无法再次获取这条消息。为了解决这个问题，我们在WMB的升级版WeiBus消息总线中实现了消息同步序号的 功能，支持客户端在超时情况下，可重复获取消息。

但是随着微博业务的蓬勃发展，业务依赖关系越来越复杂，多机房部署成本压力越来越大，而且运维成本也不断攀升，下图是一个产品的服务依赖关系图。微博多机房V3版本实现了业务灵活多机房部署架构，支持业务自定义机房部署个数，及部署区域。

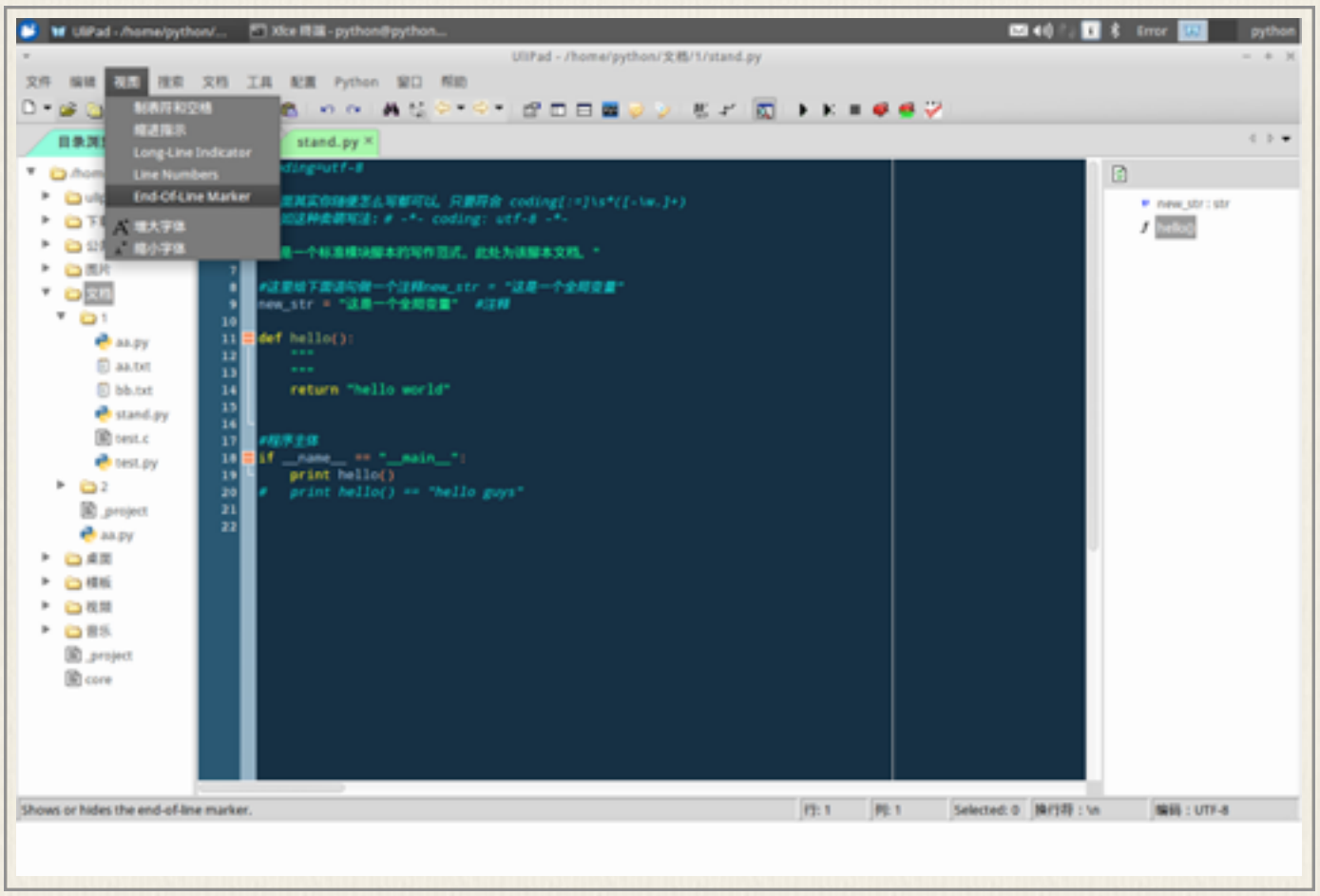


业务定制部署需要解决业务路由问题。在当前全国的网络环境下，南北网络专线延迟一般在30到40毫秒之间，而机房内延迟一般小于1ms。业务路由需 要支持尽可能路由到调用本地机房调用，对需要跨机房调用的请求进行打包以便减少网络延迟的影响。微博根据自身业务特点，实现了业务路由服务，支持将多个业 务请求进行打包，将多个请求打包成一个请求，自动识别本地业务部署，把需要跨机房调用的请求一次性请求到对应机房，并将返回的结果打包后一并返回。并且支 持自动识别业务部署结构变更，并对非核心业务异常自动隔离。

随着移动互联网的迅猛发展，3个机房的部署结构不能完全解决用户访问速度问题，一种解决方案是让机房更加靠近用户。但是社交网络由于数据的网状访 问，较难选择合适的切分维度，目前微博核心业务仍需要各机房同步全量数据，部署更多机房的成本压力比较大。QZone的SET化和Tumblr的Cell 化在解决社交网络拆分维度方面都值得参考，微博也在进行Cell化方面的尝试，相关信息也会在 @微博平台架构 微博帐号上与社区进行交流，希望感兴趣的同学积极与我们互动。 微博只是多机房之路上迈出了一小步，仍有很多难题有待攻克。希望对多机房系统，对微博的架构感兴趣的同学加入到我们微博的团队，共同打造一流的分布式系 统。

原文链接：http://www.infoq.com/cn/articles/weibo-multi-idc-architecture

【开源专访】Uliweb李迎辉：从自我需求出发做开源



Uliweb是一个开放源代码的Python Web框架。该项目基于一些成熟的项目，如werkzeug(其为Flask的底层核心库)、SQLAlchemy等，同时还借鉴了web2py的 Template模板模块、Django的一些设计思想和成果，进行了自己的改造和创新。同时还自行实现了其它的一些常用组件，如缓存/会话（Session）支持、国际化（I18N）、表格处理等。

李迎辉（新浪微博@limodou）

Uliweb项目创始人李迎辉（Limodou）是CPUG（中国Python用户组）核心成员之一、python-cn邮件列表创建者。最近，CSDN CODE采访了李迎辉，请他分享了多年从事开源工作的经验和体会。李迎辉表示，做开源项目最重要的是兴趣、坚持和创新，同时，做自己需要的项目，坚持使用自己的工具来做日常的事情，也很重要。

Uliweb代码托管地址： https://github.com/limodou/uliweb

李迎辉的Github 地址： https://github.com/limodou

以下是采访记录。

CSDN：首先请您简单介绍一下自己，包括您目前从事的工作、关注的领域，在开源社区都有哪些身份等。

李迎辉：我网名叫Limodou，真实姓名是李迎辉。目前在银行做开发，现在主要负责项目管理相关的应用开发，主要是Web方面。以前的业务领域主要在交易系统，做过C和Java。在开源社区主要就是python-cn邮件列表的创建者，CPUG的主要成员。

CSDN：请介绍一下您的开源项目Uliweb和Ulipad的一些近况?

李迎辉：我做的项目很多，UliPad和 Uliweb是最大的两个。现在UliPad因为精力问题更新不是太多，也有想法再优化一下，看机会和精力。Uliweb现在还在不断的优化 中，Uliweb从2008年开始开发，目前功能已经比较完善，我已经在公司内部使用它来开始我负责的项目管理的Web应用。

CSDN：您为什么要做Uliweb和Ulipad项目，而不选择已有的开源实现?

李迎辉：先说UliPad。在开发 Python时，我喜欢用完全使用Python开发的编辑器，这样自已可以在需要时进行修改。于是我开始参与DrPython项目，贡献了一些代码。在开 发过程中，发现增加一个新功能要在原来的代码基础之上进行修改，不是特别方便，于是我根据学到的mixin的技术想对原来的DrPython进行优化，改 进如菜单、参数管理等功能的生成，但是作者不是很理解，而且这种修改对原来的架构变化有些大，于是我想干脆我自已做。这样在2004年开始，我就利用 mixin技术开始搭建了UliPad编辑器，整个功能实现基本上都是以mixin方式来实现的，实现了功能的分布式开发。UliPad功能还是挺完善 的，在很多易用性、中文支持上做了不少的工作。

在2005年，我开始接触Web框架，在此之前我也用PHP、zope做过网站，所以Web对我来说并不是全新的东西。只不过，从2005年我的重点开始向Web方面偏移，因为我觉得未来的确需要越来越多的协作，而Web是最方便的方式。

当时，Django还没有发布，只是放出口风将要发布，它在Python社区还是受到了一些关注的。那 时，Python社区还是有一些Web框架，但是Python社区的重点并不在Web上，ROR在那个时候已经开始流行了。在Django发布出来之后， 我学习了它的教程，也读了不少的源码。在发现问题的时候

也提交问题和补丁，也提交过中文翻译的一些东西，但是在使用过程中发现一些不方便的地方，于是在邮 件列表、在问题中与Django团队的成员和用户进行交流，提出自己的想法和建议，有些在Django看来是无法接受的。于是我开始学习其它的框架，之后 也参与了web2py框架的一些开发、贡献代码等，但是也仍然与web2py的作者的一些理念产生冲突。于是在2008年，我打算自己写框架，不再受制于 人。

所以我不选择已有的开源项目自己做，主要原因是我想按自己的想法来实现，而这些想法不被这些项目接受。当然，随着项目的不断发展，我实现了更多独创的东西，在某些地方我认为超越了其它的项目。

CSDN：您从什么时候开始接触开源的？开源对您的工作和生活带来了哪些变化？

李迎辉：具体时间我已经记不清了，我在2000年接触Python之前就学过Delphi、PHP等，大概在97、98年吧，那时我写过一些Delphi的软件已经开始用开源的方式进行发布了。但是后来我转到Python上了，就一直以Python为主了。

刚开始我以开源的形式发布我的代码，其实就是放出源代码，版权什么的不是太清楚，并且不是太愿意，因为我好不容易写的软件就这么公开，真是有些舍不得。但是随着我使用Python越来越多，**我从别人那里免费并且自由地拿到了他们的代码，学到了越来越多的东西，我就越来越认为开源是伟大的，因为这体现了知识的分享，人与人的相互信任与欣赏，让我们可以更容易地学习别人的经验和知识**。特 别是互联网让获取知识越来越容易，可以让我们更方便地进行创新。所以我越来越支持开源，现在应该算是一个坚定的开源拥护者。在我做的项目中，我自己写的东 西基本上都是以开源的方式发布出来的。在我做的项目中，基本上也都是开源软件搭建的。如果没有开源，我想我现在的水平远远不是这样的。

CSDN：做开源仅凭一时热情是容易的，坚持下来却很难得。我们知道Uliweb推出已经5、6年了，是什么支撑您能够一直做下去？

李迎辉：对于我来说主要还是兴趣。在2010 年底前，我在公司主要是做交易系统，用的都是Java，所以Python对我来说主要就是个爱好。后来让我做项目管理相关的开发，那时还是个小工具，现在 已经发展得很大了。我当时选择了Uliweb作为底层的框架，功能和性能上完全可以满足目前的需要。因此现在已经算是专职在作Uliweb的项目开发，所 以也可以有更多的时间来维护和优化Uliweb项目。也算是兴趣和工作相结合了吧。

CSDN：您在维护开源项目方面有什么心得或经验与大家分享？是否有意识的对开源项目做一些运营工作?

李迎辉：**我最大的体会就是坚持**。很从开发人员都有自己的开源项目，但是很多都做不下去了，可能有各种各样的原因。但不管怎么样，其实就是没坚持下去。**还有就是要有自己的创新**。如果只是简单复制就没什么意思。**同时，做开源项目我觉得先考虑满足自己的需要**，比如我因为觉得需要一个纯Python的编辑器，就开发了UliPad，因为我觉得未来Web是发展的重点，我开发了Uliweb。在做这两个开源项目中，又因为各种各样的需求，我还创建了其它的开源项目，比如生成文档的工具等。**自己要坚持使用自己的工具来做日常的事情**。

我的开源项目都是个人的，所以也没什么运营，只是为了方便交流创建了邮件列表、QQ群什么的，用来与大家交流，回答问题。

CSDN：是否有人为您的项目做过捐助?您如何看待开源项目商业化的问题?

李迎辉：目前没人捐助，因为我也从来没要过。

开源与商业化并不是冲突的，而且我也听说有些公司因为项目是开源的，反而更容易获得别人的信认。只不过，不同的开源项目可能对别人的吸引程度不同，产品可能更容易商业化，而框架之类的基础类项目相比之下就困难得多。

CSDN：以您多年从事开源事业的经历来说，您认为对于一个开源项目来说最重要的是什么?

李迎辉：我说不太好，可能是有用和创新吧。

CSDN：谢谢。

本文转载自：**http://code.csdn.net/news/2819411**

Jon Skeet：我不想知道我在SO上到底花了多少时间（图灵访谈）

作者：@李盼ituring
「码农」电子刊编辑，负责图灵访谈。说什么都觉得自己怪可笑的。

Jon Skeet，谷歌软件工程师，微软资深C# MVP，拥有10余年C#项目开发经验。自2002年来，他一直是C#社区、新闻组、国际会议的活跃技术专家和Stack Overflow总排名第一的用户，回答了数以万计的C#和.NET相关问题，他还建立了Joda Time的.NET版本Noda Time。Jon著有《深入理解C#》，他常年维护博客Coding Blog。外国程序员们调侃他的段子"Jon Skeet Facts"也很有趣。

[+]查看原图

图灵访谈：毫无疑问你是一位C#方面的专家，撰写了3版的《深入理解C#》，你有计划为你熟知的其他语言或工具写书吗？

Jon：很遗憾，除此之外我很少有熟悉的工具或语言了。我在Java方面还算比较专业，但是我感觉讲Java的书已经很多了。

这几年，我一直想让自己学习一下F#，Go或者Ruby……我觉得我真应该从某个时候开始学起来。我觉得在我学习F#的时候截个屏感觉应该很棒，这时候要是再有一个具体的学习目标就更好了。我希望让其他人看到我的学习过程会对他们产生帮助。

图灵访谈：你每周花在回答问题上的时间有多少？对你来说最大的收获是什么？

Jon：要弄明白我每周花在Stack Overflow上的时间非常困难——这不是件我想弄明白的事儿，因为我可能并不喜欢这个答案。回答问题会为我带来各种各样的收获，我最喜欢的事儿就是在 回答一个问题的同时学到新的知识。另外，有时候我很多年前回答的问题会突然收到新的评论，这个评论来自于现在仍需要面对这个问题的人，他们用我的答案解决 了问题，这样的事我也很喜欢。

图灵访谈：很多你的读者很好奇你在Google的工作。Google内部有用C#写的项目吗？是什么项目？

Jon：对于我在Google的工作我不能说太多，但是我可以告诉大家我在Google已经写了两个C#的项目：我的针对C#的协议缓冲（Protocol Buffers）端口（20%的时间）以及Windows Phone 7上Google搜索应用的第一个版本。

还有一些关于访问Google API的C#项目，.NET的Google APIs Client Library是这些项目的根。

图灵访谈：你Google软件工程师的身份和微软MVP的身份之间有冲突吗？如果有，你是如何做到在Google提升你的技术能力的？

Jon：我只是个名义上的MVP，事实上——我并没有在保密协议（NDA）下，我也没有从微软那里得到什么礼物，比如MSDN（微软开发者网络）的

免费订阅之类的。但是作为MVP我还是很高兴的，这证明了微软已经认可了我对.NET社区做出的贡献。

要想在Google提升技术能力和在别的公司决然不同，在Google编程的挑战和在其他地方遇到的挑战有着天壤之别。为了要在C#方面跟得上，我都得在自己个人的时间里学习——比如我在Noda Time上花的时间也是我的业余时间。

图灵访谈：确实，C#是一种优雅、便捷、规范化的语言，但是，你认为C#最大的问题是什么？有什么已经存在的语言已经解决了这个问题?

Jon：还是那个问题，我对其他语言知之甚少——所以让我来说哪个语言解决了C#没有解决的问题还是挺难 的。比如，虽然以前也被告知过，但是在我习惯于在C# 2里以匿名方法闭包之前，我都不太喜欢这样的用法。这是一个关于Blub困境的例子。我很喜欢C#让处理数据（通过LINQ）和异步代码变得更简单，但 是，我不喜欢在C#里大量使用动态类型，虽然我很欣赏C#把这部分加入到语言中。我怀疑未来的语言可能会混合使用动态类型和静态类型，但是要从头开始就这 么做——这样就可以避免这两种类型偶尔在C#里产生的矛盾了。

C#曾经有个问题（虽然已经在C# 6中得以解决了）是执行不可变类型时所需要的样板代码的数量。长时间以来我一直都很喜欢不可变性，但是C#在几年来的众多改进中（比如对象初始化设置）都只是帮助提高了可变类型。能看到主构造函数和只读自动执行属性开始解决这方面的问题我很高兴。

图灵访谈：你希望看到C#在未来有什么样的新特性？Go的并行执行goroutines？或者是像F#或Scala那样的模式匹配?

Jon：我认为模式匹配是很有可能的，但是对于goroutines我并不确定……我感觉C#已经选择了异步这条路线，如果混搭上goroutines会显得很奇怪。

说到其他特性——C#最好的特性永远是那些让我产生惊喜的特性。我觉得这样的特性对于我来说就必须要有CLR改变，当然，我的意思并不是它们已经毫无可能。如果代码能明确显示出与之工作的数据类型的话，我就会比较欣赏，这样就可以很好地阻止你在上面执行不合适的操作了。在这样的一些

原则下，我更 愿意看到可以允许让单一实现分享到多种类型的类型系统，每个类型只能显示底层API的有限子集。在某些情况下，可能会出现在同一个API下的多重类型，但 是它们仍旧会是完全不同的类型——可能之间还会有一些显式转换。当然，F#的“测量单位（Units of Measure）”的概念可能也是出于这样的考虑。

图灵访谈：C#或者说.NET似乎只能运行在Windows或者Linux下的Mono上，C#的发展前景对于C#程序员来说似乎也不是很明朗。你认为.NET应该扩展跨平台的兼容性吗?

Jon：我非常高兴能看到像Xamarin这样的工具，它可以让C#代码运行在iOS和Android上，而且，我还买了一台iPad和一台Mac Mini，就是为了探索代码的运行情况。到时候你也可以让C#代码运行在所有主流的移动平台上，还包括Mac，Linux以及Windows，我感觉这还不算是很受限制吧。Mono的边边角角上还是有点糙，这是当我研究Noda Time国际化的极端案例时发现的——但是它的工作性能还是好得让人惊叹。Roslyn现在已经开源，而且已经能被Mono所用也是很振奋人心的事儿。

图灵访谈：微软现在将9寸以下wp授权费降为0。你认为这会增加C#开发者的数量吗？你对WP有什么样的期待?

Jon：因为Windows Phone和Android之间的利益冲突，我觉得我最好还是不要回答这个问题。

图灵访谈：作为一位C#高手，你对C#初学者们有什么建议吗？对于有编程经验的人来说，你想提醒他们在学习过程中要注意哪些事?

Jon：如果是我的话，我会首先关注语言的核心方面。很多写给初学者的书（在我看来）从开始的时候就用 GUI和数据库的沟通作为第一章的内容，读者们还不知道他们敲下的代码是什么意思呢！这样做就是欲速而不达。我认为应该先要确保你理解这个语言，之后再去 学习一堆关于函数库的知识……如果这样，就算你不知道具体某个调用是干什么的，至少你可以理解你面前代码的运行机制，然后你可以通过查看相关的文档来理解 剩下的。

最经常被人误解的C#（以及Java）的方面似乎就是各种变量、对象、以及索引的区别——如果你可以百分之百地确定你理解了其中的区别，那你就已经上路了。

图灵访谈：在中国很多程序员随着年龄增大会转换角色，变成经理，你对这些程序员有什么样的建议?

Jon：从软件工程师到经理的转换在西方也非常常见，我怀疑虽然这有时是个好事，但很多时候也不是。我经常 会看到很多经理在私底下都极想要做自己手下工作范围内的编程工作。我是想说，一个好的经理应该有能力用他们的经验来帮助初级工程师变得更有效率，他们可以 分享自己的经验，在管理的同时也能指导他们。

从个人角度上说，我很高兴这两样我都有所涉及——我不想把自己和编程隔离，但是肩负一定的管理责任也是一件好事。

原文链接：http://www.ituring.com.cn/article/76098