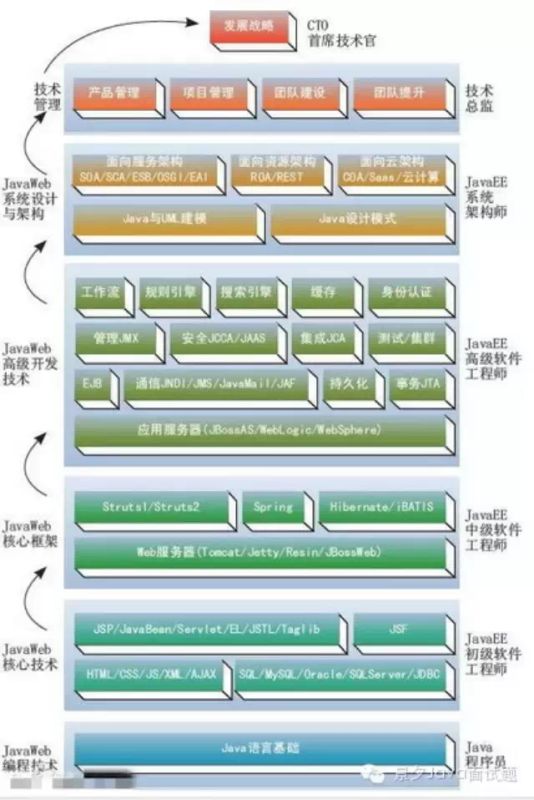
**框架架构**

## 一、架构师职位

**1 对于大多数开发人员，理想的职位都是软件架构师**，觉得这是一个很体面而又很酷的工作。但是大多数人对软件架构师的职责没有一个清晰的认识，包括我自己。今天我就抛砖引玉，简单描述一下软件架构师是做什么的。java知识平台已经开通原创保护功能，这跟小伙伴们的支持是分不开的，bert君在这里谢谢大家了。大家可以在文章底部留言，表达一下自己对架构师的看法。架构一词大概是源于建筑学，也常指建筑物在其尺度上是如何依靠内部的支撑物相互结合而稳固构造的方式。而架构师则是为满足某种架构设计目标而进行在较大尺度上进行整体构思的角色。在软件工程和企业信息系统领域，又有很多细分，如所谓的系统架构师、应用架构师、企业架构师等等。

**2 一般来说，应用架构师负责构建一个以解决特定问题为目标的软件应用的内部结合结构，一般以满足各种功能性需求以及维护性需求为设计考虑目标；系统架构师则提供运营支撑软件应用的信息系统的结构设计，一般以满足各种非功能性需求或运营性需求为设计目标（如安全性、可伸缩性、可互操作性等等）；企业架构师，就不光只顾IT系统的架构了，他应以企业的持续经营目标为考虑要素来构建企业所需要的内在结构设计。**



**3 架构师职责**

**1） 必须要了解需求和业务，架构师必须接地气。**

不了解需求的架构师，就去设计架构，指挥大家干活，纯粹是耍流氓。架构虽然不是产品经理，但是产品最终都是服务客户的。如果架构师只懂技术，不考虑需求，做出的系统可能是高大上，但是用户不一定买账，那就是一个失败的产品了。

**2） 架构师需要定义架构。**

搞清楚需求后，下一步就是思考，如何解决这些需求并定义它的架构。架构定义过程让你想清楚你打算怎么在兼顾需求和限制的情况下把问题解决好。架构定义是将结构，方针，原则和领导力引入软件项目的技术层面。

**3） 架构师还需要进行技术选型，选择哪种技术来支持业务的实现。**

这里注意的是，不一定要选择那些潮流技术，选择什么技术，要看自己的场景。有时候，选择一些最原始的技术，没准更适合自己的业务场景。技术选型是很重要的，选择正确的技术，就能事半功倍。如果选择不好，那么就会欠技术债务，不仅影响产品的开发周期，还会影响产品的稳定性。

**4） 架构师需要进行架构协作。**

这里的架构协作，不仅仅指技术方面了。一个好的产品，牵涉到很多人即“项目干系人”，要想让一个软件项目成功，需要所有的系统干系人紧密协作来保证架构和所在的环境很好的集成。

**5） 架构师还可以参与一些设计，开发，测试的工作。**

还是那句话，架构师要接地气。真正去使用自己的产品，不仅能对产品进度很好地把握，还能在使用中发现产品架构存在的问题，及时修补和调整。另外，这样做的另外一个好处，就是架构师能够真正融入团队，并不是高高在上的。这不仅不会削弱你的权威，相反，能够让下属更尊重你，这也是领导力养成的一个重要途径。

**4 架构师技术要求**

1） 熟悉分布式、高性能架构和开发技术，如分布式应用开发、数据分布式管理和同步等；

2） 精通J2EE系统架构，深刻理解J2EE架构的优缺点，具有大型基于J2EE体系结构的项目规划、系统架构设计、开发经验，精通j2ee设计模式；

3） 深刻理解软件系统架构，精通面向对象分析设计方法，逻辑能力佳，具有丰富的OOA、OOD、OOP、UML及SOA经验，精通RationalRose、PowerDesigner等设计工具；

4） 技术视野广阔，具有良好的前瞻性，思路清晰、逻辑性强，对移动支付和互联网支付的相关技术具有优秀的领悟力和前瞻性，有较强的业务分析能力；

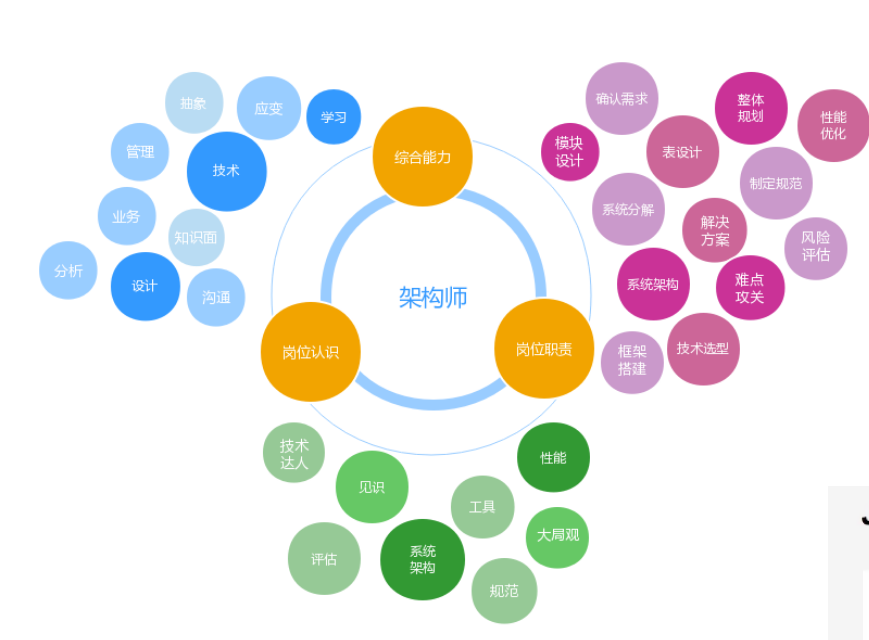
5） 良好的沟通能力、团队合作精神和服务意识；认真负责、具有高度责任感和敬业精神；

6） 对于性能瓶颈可以给出最优的切片,集群和分布式服务器搭建解决方案

7） 理解面向对象分析和设计的基本原则，熟悉常用的设计模式，熟悉ＵＭＬ；

8） 熟悉Java的多线程，线程与线程，进程与进程的通信机制；

9） 精通系统优化，对系统优化原理有深入的理解。对系统端到端性能优化有丰富的实践经验，熟悉各种远程本地Cache组件(尤其是Memcached,Redis)，对Cache服务器集群架构有丰富的经验；



## 二、架构搭建

**1 软件架构（Software Architecture）**

软件架构是指在一定的设计原则基础上，从不同角度对组成系统的各部分进行搭配和安排，形成系统的多个结构而组成架构，它包括该系统的各个组件，组件的外部可见属性及组件之间的相互关系。组件的外部可见属性是指其他组件对该组件所做的假设。软件架构设计就是从宏观上说明一套软件系统的组成与特性。

**软件架构设计是一系列有层次的决策 ，比如：功能与展现的决策；技术架构的决策；自主研发还是合作；商业软件还是开源软件** 。

**2 架构设计**

业务需求层出不穷；软件系统越来越复杂；参与的人越来越多；共性和特殊性的问题越来越多；技术发展日异月新；……

**3 软件架构师**



**4 架构师的素质**

**1） 全局思维**

从业务、市场，到技术实现；

从软件的过去、现在，到将来；

从外部客户，到内部研发；

从软件研发，到硬件部署；

从功能实现，到运行效率

**2） 战略思维**

在所在行业的发展战略；

在业务领域的发展战略；

在技术方向的发展战略；

在潜在市场的发展战略。

**3） 前瞻思维**

市场趋势的发展动向；

前沿技术的发展动向；

竞争对手的发展动向；

合作伙伴的发展动向。

**4） 抽象思维**

各项业务需求：抽象成功能模块；

各项功能的实现：抽象成软件架构。

**5）逆向思维**

假如不实现会怎样？

假如没搞定会怎样？

假如没有它会怎样？

假如被延期会怎样？

**5 架构师的分类**

随着行业和社会的发展，架构师的定义和分类越来越广泛和细分，广泛和细分其实并不矛盾，如果“广泛”是x轴，“细分”是y轴，则二维坐标系x和y轴中间的任一点就是一种架构师类别。但总体来说，或目前来说，集合业界的大致认知，总结如下

|  |  |  |
| --- | --- | --- |
| **No.** | **分类** | **描述** |
| 1 | 解决方案架构师 | 与客户探讨业务需求，将业务、市场，与技术、产品结合起来，为客户提供解决他们需求的方案。 |
| 2 | 系统架构师 | 也称应用架构师。最终确认和评估系统需求，并将业务转换为技术，为研发人员制订核心框架与技术规范 为研发工作澄清技术细节并扫清技术障碍 。 |
| 3 | 平台架构师 | 这里的平台其实包括两个平台，一个是系统平台，也就是负责搭建多个系统整合的系统应用平台；另外一个其实是基础平台，是专门负责搭建基础技术平台；两者其 实区别蛮大，也经常容易被从业人员混乱。举个简单例子，金蝶有平台架构师一职，但是金蝶BOSS应用和金蝶中间件两者招聘的对象和技术要求是截然不同的。 |
| 4 | 业务架构师 | 业务架构其实已经开始脱离技术层面了，但是它要求架构师有跨越多系统的大局观，去整合和组织不同系统的技术平台与交互模式。其实这个职位的未来也就是CIO了。 |
| 5 | 网络架构师 | 过去，我们可能听的最多的是网络工程师。不错，一个优秀的网络架构师必须有足够的网络技术基底，并且它的关注点也是系统的基础架构。比如说如果搭建并优化集群环境，如果构建基于云计算的系统应用与部署等等。它对于像淘宝、腾讯这样的互联网公司是极其重要的。 |
| 6 | 移动架构师 | 移动互联网的迅猛发展横向和纵向都细分出了很多新的职责和岗位，移动架构师的职责和作用日益重要，既要整体和全局考虑整个前后端的软件系统架构，又要重点深入移动客户端的架构设计的方方面面，既要有跨平台思维，又要拿捏好原生和混合开发的尺度，另外移动应用的特点，导致移动架构师必须要比传统系统架构师更加注重非功能性的质量属性。 |
| 7 | 前端架构师 | 这也是移动互联网的迅猛发展而细分出来的新的职责和岗位，这里的前端特指网站开发中的前端，主要考虑前端呈现层的设计（HTML/CSS/JS/AJAX/RIA/…），跨浏览器设计等等。 |
| 8 | ... | ... |

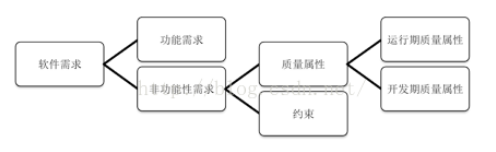
**6 软件需求**

软件架构设计中常说需求驱动架构设计，可见需求在整个架构设计中起到了关键指引和方向的作用，如果以目标导向为原则，则需求的满足和实现就是架构设计的终极目标。

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **阶段** | **需求阶段** | **说明** | **什么人做什么事** | | **可能产生的文档** | |
| **客户方** | **实施方** | **客户方** | **实施方** |
| 1 | 业务需求 | 来自客户的原始需求，背景描述，业务诉求和期望目标。 | 项目负责人或接口人描述需求或提供客户方需求文档。 | 销售或售前接触客户，听取和记录需求。 | 工作说明书（SOW），或邀标书 | 售前的方案建议书 |
| 2 | 用户需求 | 通常是在问题定义的基础上进行用户访谈、调查，对用户使用的场景进行整理，从而建立从用户角度的需求。 | 项目负责人或接口人接受访谈和调研。 | 需求分析师或项目经理等进行需求调研。 |  | 调研计划、用户需求调研提问库、调研日志、用户需求说明书 |
| 3 | 软件需求 | 从系统实现的角度描述的需求。开发人员（设计和分析人员）在业务需求、用户需求的基础上形成的。 |  | 需求分析师或项目经理、架构师等讨论和细化需求，编写需求文档 |  | SRS（Software Requirement Specification）软件需求规格说明书 |

**7 软件需求分类**

**注：在架构设计中，对非功能性需求的重视程度，也会影响架构设计的好与劣；但也要平衡过渡设计和适可而止的关系。**



**8 软件架构的过程**

业界软件架构设计的方法论很多，各有各自的应用场景和特点，下文结合ADMEMS（Architecture Design Method has been Extended to Method System）架构设计方法论说明软件架构的过程：

|  |  |  |  |
| --- | --- | --- | --- |
| **架构阶段** | **目标** | **方式方法** | **现实工作场景** |
| 预架构阶段 | 全面理解需求；需求结构化，摒弃“需求列表”，建立二维需求观（ADMEMS矩阵）。 | 使用ADMEMS矩阵方法，捋清需求间关系和发现衍生需求。 | 1、与人：与项目经理、需求分析师等内部需求人员了解需求；与客户了解需求（不建议架构师做需求分析师角色）。 2、与物：了解《需求规格说明书》等需求文档。" 3、对需求有什么问题，反馈给售前或销售，可能会参与拜访客户或电话会议。 4、销售或售前有时会要求提供一个大致的工作量，以便他们初步评估项目可行性。 |
| 概念架构 | 高层组件及其关系 | 1、初步设计，基于关键功能，借助鲁棒图进行以发现职责为目的的初步设计（不是必须）。 2、高层分割，将复杂系统切分为多个二级系统或多个子系统。 3、考虑非功能需求，采用ADMEMS推荐的目标-场景-决策表。 | 1、参与内部讨论：项目可行性分析、讨论，从需求、技术、人力、风险等角度提供建议。 2、项目投标准备：参与投标团队的技术方案编写，编写系统架构章节，解决招标书上技术问题的问答。 3、参与项目讲标：作为讲标团队成员参与项目讲标，负责技术问答环节的应对。 |
| 细化架构 |  | 5视图法 | 在项目概要设计阶段，进行架构设计，制定规范和约定，为详细设计提供指导。 |
| 实现 | 详细设计 编码实现 | 架构设计形成详细设计文档 | 在项目实现阶段，对开发人员提供规范指引和技术支持。 |

**注：架构设计的过程和内容的不是固定不变的，现实中，比如售前提供解决方案中，很多时候需要架构师提供细化架构中才会深思的逻辑架构、物理架构等，这时候，架构师就需要有螺旋思维和跳跃思维的方式，就像武功中，招式是死的，人是活的，要学会灵活运用。**

**9 软件架构设计的方式方法**

**（1）多视图法**  
多视图方法是业界广泛认同的一种架构设计思路，包括：  
● SEI的3视图法：  
模块视图、组件-连接器视图、分配视图。  
● 西门子的4视图法：  
概念视图、模块视图、代码视图、执行视图。  
● RUP的4+1视图法：  
用例视图、逻辑视图、开发视图、进程视图、物理视图。  
● 联邦企业架构框架：  
技术架构视图、信息架构视图、应用架构视图、业务架构视图。  
● ……

**（2） 5视图法**



**（3）视图法分析的意义：**

● 全面分析软件系统方方面面的问题  
● 尽早地发现和排除项目风险与不确定因素  
● 从不同角度去展现要设计的软件系统  
● 为项目进行中不同的干系人提供指导：  
    -- 逻辑架构描述系统功能，并指导系统测试  
    -- 开发架构规范软件的层次及代码风格  
    -- 数据架构指导数据库的设计  
    -- 运行架构定义了一些关键过程的设计  
    -- 物理架构明确软件如何部署与实施

**（4） 5视图法设计步骤**

两种观念：

|  |  |
| --- | --- |
| 观念 | 设计步骤 |
| 观念一 | 顺序进行：  1、逻辑架构。  2、开发架构  3、数据架构  4、运行架构  5、物理架构 |
| 观念二 | 5个视图是穿插进行设计的，对复杂系统而言，根本不可能将逻辑视图设计完了后再考虑其它视图。 |
| 笔者观念 | 观念一和观念二的情况在现实状况中都可能存在，要根据具体情况具体分析，但总体而言，5视图穿插进行思考更有利于思考的全局性和完整性。 |

**10 如何进行5视图法的设计**

**（1） 逻辑架构**

逻辑架构的重点是考虑软件功能性需求。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| No. | 考虑的方面 | 产出物 | 工具 | 说明 |
| 1 | 系统功能划分为几个子系统与功能模块？ | 系统功能树 | 树型结构图 |  |
| 2 | 向什么用户提供什么样的功能？ | 用例模型 | UML用例图 | 体现用户和行为 |
| 3 | 每个功能都是怎样的操作流程与分支？ | 用例描述 | 用例描述表 | 含输入输出、事件流分析；  不要有界面描述 |
| UML活动图 | 进行业务流程分析，包括泳道图 |
| 4 | 如何通过界面与用户交互？怎样交互？ | 鲁棒分析 | 鲁棒图 | 通过对“用例描述表”进行原文分析法拣出名词和动词 |
| 5 | 应当设计哪些类与界面？怎样设计？ | 领域模型 | UML类图 |  |
| 6 | 与哪些外部系统接口？怎样接口？ | 接口描述 | UML类图 |  |

**（2）开发架构**

开发架构重点关注的是开发编码实现方面的问题。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| No. | 考虑的方面 | 产出物 | 工具 | 说明 |
| 1 | 分层结构设计 | 分层架构图（开发架构图） | 各种绘图工具 | 好的分层结构支持自动化测试 |
| 2 | 开发技术选项 | 开发语言  开发框架  开发工具 |  | 考虑商用产品、开源框架、自研框架 |
| 3 | 模块划分 | 源码工程；Project目录结构；  分包(分库) |  |  |
| 4 | 开发规范 | 开发/编码规范文档； |  |  |
| 5 | 软件质量属性 | 分析和决策结果 |  | 考虑运行期和开发期软件质量属性，并权衡利弊进行决策。 |

**（3）数据架构**

数据架构不仅仅要考虑开发中涉及到的数据库，实体模型，也要考虑物理架构中数据存储的设计。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| No. | 考虑的方面 | 产出物 | 工具 | 说明 |
| 1 | 数据是集中还是分布存储的？如何考虑分布式存储？ | 数据架构图 |  |  |
| 2 | 领域模型到数据库表的转换？表结构关系的设计？ | 逻辑模型  物理模型  ER图 | Power Designer  Visio |  |
| 3 | 实体如何设计？充血模型和贫血模型？ | UML类图 |  |  |
| 4 | 使用什么数据库？关系型还是非关系型？ | 选型结果 |  |  |

|  |  |
| --- | --- |
| 关系型数据库 | 非关系型数据库（NoSQL） |
| Oracle（首次发行：1980年）  MySQL（首次发行：1995）  MS SQL Server（首次发行：1989）  PostgreSQL（首次发行：1989）  IBM DB2（首次发行：1983）  Microsoft Access（首次发行：1992）  Sybase ASE（首次发行：1987）  SQLite（首次发行：2000）  …… | MongoDB（首次发行：2009）  Cassandra（首次发行：2008）  Apache CouchDB  Hbase  Redis  db4o  BaseX  …… |

**（4） 运行架构**

运行架构关注的不再是全局而是局部，着重关注那些关键点与难点，常常需要技术攻关与预研。主要考虑控制流、通讯机制、资源争用、锁机制、同步异步、并发、串行，同时也要考虑质量属性。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| No. | 考虑的方面 | 产出物 | 工具 | 说明 |
| 1 | 运行：同步vs.异步；并发vs.串行 | 考虑开发架构中代码的实现。 |  |  |
| 2 | 交互：对象间交互；状态转换 | 考虑开发架构的合理性，到类、到接口、到代码。 |  |  |
| 3 | 质量：安全；可靠；可伸缩 | 考虑开发架构的合理性 |  |  |
| 4 | 性能：响应时间；吞吐量 | 估算：  在线人数、并发人数；  每秒事务量；  响应时间。 |  |  |

**（5）物理架构**

物理架构主要考虑硬件选择和拓扑结构，软件到硬件的映射，软硬件的相互影响。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 考虑的方面 | 产出物 | 工具 | 说明 |
| 1 | 网络方面：网络拓扑；网络设备；安全机制 | 拓扑图  安全规范 |  |  |
| 2 | 性能方面：可靠性、可伸缩性 | 需要什么样设备性能 |  |  |
| 3 | 部署方面：集中式还是分布式；组件部署 | 部署图 |  |  |

**11 一个思考，谁驱动了架构设计？**

需求驱动了架构设计？

质量属性了驱动了架构设计（ADD）？

领域驱动了架构设计（DDD）？

风险驱动了架构设计？

质疑驱动了架构设计？

……

到底是谁驱动了架构设计？我们以船舶设计建造为例，来看这些问题：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **目标和结果 à** | **问题 à** | **回答 à** | **小结** | |
| 设计和制造一艘远洋货轮，能经历数月海上颠簸和长途跋涉，并保证货物的安全和完整，最后能顺利抵达目标港口。 | 我们为什么要设计和制造这样的大船？ | 市场有这个需求；  获利很丰厚；  解决就业；  …… | 不管是外部需求还是内部的需求，都是需求。不正是需求驱动吗？ | |
| 如何保证货船能长途航行并经受波涛和风雨？ | 船要造的结实； | 鲁棒性 | 这些不正是质量属性驱动设计吗？ |
| 引擎设计的强劲； | 性能 |
| 船的燃料充足； | 持续可用性 |
| 装备卫星电话、GPS、雷达等设备 | 互操作性 |
| 货物和人员要安全 | 安全性 |
| 船舶设计师怎么设计这样的船舶？ | 现在都会通过计算机进行船舶的CAD和3D模型设计。 | 是不是佷似领域模型驱动了设计？ | |
| 造船一定有很多风险吧？ | 那是，比如订货方客户有时提出改装船舶的意见（需求变化的风险）；有时某些工艺成品率不稳定（质量风险）；等等。 | 所有可能的风险点在设计时都要考虑到，做好预案，才能保证架构设计的可行性和灵活性，风险驱动了架构设计。 | |
| 上面怎么提了那么多问题，其实还有很多问题，比如…… | 嗯，你现在有不少疑问了。 | 不断的质疑架构设计中可能存在各种问题，有质疑才有思考，才有解决方案，从而推动架构设计的不断完善。 | |

**12 软件架构设计的误区**

● 高开高走落不到实处

● 理想与现实需要折中

● 遗漏关键性约束与非功能需求

● 为虚无的未来埋单而过度设计

● 过早做出关键性决策

● 客户说啥就是啥成为酱油哥

● 埋头干活儿缺乏前瞻性

● 架构设计还要考虑系统可测性

● 架构设计不要企图一步到位

**13 总结：**

**以上几个方面都能驱动架构设计，并不是零和的规则，而是一个立方体从不同方向看的问题。以上方面有些是指导思想，有些是行动方式，有的兼而有之，阐述方式看似不同**

**，终极目标还是造出大船（实现需求），造出好船（质量属性），怎么造（领域模型），造的顺利（风险控制），挑不出毛病（架构师自己先质疑问题并解决了）。**

## 三、架构设计

**1 基本理论**

1） 扩展性设计

X轴扩展 —— 关注水平的数据和服务克隆，也就是前文提到的“加机器解决问题”

Y轴扩展 —— 关注应用中职责的划分，比如数据类型，交易执行类型的划分

Z轴扩展 —— 关注服务和数据的优先级划分，如分地域划分

1. 可用性设计

3） 可靠性设计，高可用

4） 一致性设计,分布式事物

5） 负载均衡设计，nginx.f5,lvs

6） 过载保护设计，熔断，隔离

**2 协议设计**

1. 二进层协议

2） 本文协议

**3 接入层架构设计**

1） DNS轮询，nginx

2） 动静态分离,nginx

3） 静态化,静态化是解决减轻网站压力,提高网站访问速度的常用方案.

4） 反向代理，nginx

5） LVS，负载均衡

6） FS，分布式文件系统

7 ）CDN，将这些缓存服务器分布到用户访问相对集中的地区或网络中，在用户访问网站时，利用全局负载技术将用户的访问指

向距离最近的工作正常的缓存服务器上，由缓存服务器直接响应用户请求。

**4 逻辑层架构设计**

1） 连接池，Durid

2） 串行化技术,Serializable

3） master架构,Master-Master和Master-Slave

Master-Master

解决的问题是: 数据双写,但是避免数据正确性打破.双写的不同的表.或者说不同的行.有 proxy 控制.

数据的写拆分. 数据完整性保持一致. 按表整体移. 按行移动比较难.

Master-slave 区别:

解决数据的可靠性问题.

半同步. commit 时半同步,不是每次 sql 提交

4） 批量写入

5） 配置中心,zookeeper分布式配置中心

6） 去中心化

在一个分布有众多节点的系统中，每个节点都具有高度自治的特征。节点之间可以自有链接，形成新的连接单元。任何一个节点都可能成为阶段性的中心，但不具备强制性的中心控制模式。节点与节点之间的影响，会通过网络而形成非线性因果关系。这种开放式、扁平化、平等性的系统现象或结构，被称为去中心化。

7） 通讯机制PRC同步

RPC：它是一种规范，采用请求和响应的方式，优势是语言无关性，劣势是没有类型校验这种编译期的检查能力，必须做大量的运行时检查

8） 通讯机制RMI同步

RMI：远程方法调用，这是代码上提供的支持，依赖java语言，优势是具备类型校验的能力

9） 通讯机制MQ异步,Kafka

10） 通讯机制Cron异步,Quartz

11） 双工架构,允许二台设备间同时进行双向资料传输。一般的电话、手机就是全双工的系统，因为在讲话时同时也可以听到对方的声音。

12） 主从同步,Slave将master的binary log复制到其中继日志。首先slave开始一个工作线程（I/O），I/O线程在master上打开一个普通的连接，然后开始binlog dump process。

13） 读写分离,读写分离就是在主服务器上修改，数据会同步到从服务器，从服务器只能提供读取数据，不能写入，实现备份的同时也实现了数据库性能的优化，以及提升了服务器安全

14） 高可用缓存优化,Redis

**5 六大原则**

**1）. 单一职责原则（Single Responsibility Principle - SRP）**

原文：There should never be more than one reason for a class to change.

译文：永远不应该有多于一个原因来改变某个类。

理解：对于一个类而言，应该仅有一个引起它变化的原因。说白了就是，不同的类具备不同的职责，各施其责。这就好比一个团队，大家分工协作，互不影响，各做各的事情。

应用：当我们做系统设计时，如果发现有一个类拥有了两种的职责，那就问自己一个问题：可以将这个类分成两个类吗？如果真的有必要，那就分吧。千万不要让一个类干的事情太多！

**2）. 开放封闭原则（Open Closed Principle - OCP）**

原文：Software entities like classes, modules and functions should be open for extension but closed for modifications.

译文：软件实体，如：类、模块与函数，对于扩展应该是开放的，但对于修改应该是封闭的。

理解：简言之，对扩展开放，对修改封闭。换句话说，可以去扩展类，但不要去修改类。

应用：当需求有改动，要修改代码了，此时您要做的是，尽量用继承或组合的方式来扩展类的功能，而不是直接修改类的代码。当然，如果能够确保对整体架构不会产生任何影响，那么也没必要搞得那么复杂了，直接改这个类吧。

**3）. 里氏替换原则（Liskov Substitution Principle - LSP）**

原文：Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it.

译文：使用基类的指针或引用的函数，必须是在不知情的情况下，能够使用派生类的对象。

理解：父类能够替换子类，但子类不一定能替换父类。也就是说，在代码中可以将父类全部替换为子类，程序不会报错，也不会在运行时出现任何异常，但反过来却不一定成立。

应用：在继承类时，务必重写（Override）父类中所有的方法，尤其需要注意父类的 protected 方法（它们往往是让您重写的），子类尽量不要暴露自己的 public 方法供外界调用。

**4）. 最少知识原则（Least Knowledge Principle - LKP）**

原文：Only talk to you immediate friends.

译文：只与你最直接的朋友交流。

理解：尽量减少对象之间的交互，从而减小类之间的耦合。简言之，一定要做到：低耦合，高内聚。

应用：在做系统设计时，不要让一个类依赖于太多的其他类，需尽量减小依赖关系，否则，您死都不知道自己怎么死的。

**5）. 接口隔离原则（Interface Segregation Principle - ISP）**

原文：The dependency of one class to another one should depend on the smallest possible interface.

译文：一个类与另一个类之间的依赖性，应该依赖于尽可能小的接口。

理解：不要对外暴露没有实际意义的接口。也就是说，接口是给别人调用的，那就不要去为难别人了，尽可能保证接口的实用性吧。她好，我也好。

应用：当需要对外暴露接口时，需要再三斟酌，如果真的没有必要对外提供的，就删了吧。一旦您提供了，就意味着，您将来要多做一件事情，何苦要给自己找事做呢。

**6）. 依赖倒置原则（Dependence Inversion Principle - DIP）**

原文：High level modules should not depends upon low level modules. Both should depend upon abstractions. Abstractions should not depend upon details. Details should depend upon abstractions.

译文：高层模块不应该依赖于低层模块，它们应该依赖于抽象。抽象不应该依赖于细节，细节应该依赖于抽象。

理解：应该面向接口编程，不应该面向实现类编程。面向实现类编程，相当于就是论事，那是正向依赖（正常人思维）；面向接口编程，相当于通过事物表象来看本质，那是反向依赖，即依赖倒置（程序员思维）。

应用：并不是说，所有的类都要有一个对应的接口，而是说，如果有接口，那就尽量使用接口来编程吧。

将以上六大原则的英文首字母拼在一起就是 SOLID（稳定的），所以也称之为 SOLID 原则。只有满足了这六大原则，才能设计出稳定的软件架构！但它们毕竟只是原则，只是四人帮给我们的建议，有些时候我们还是要学会灵活应变，千万不要生搬硬套，否则只会把简单问题复杂化

**6 补充原则**

1）. 组合/聚合复用原则（Composition/Aggregation Reuse Principle - CARP）

当要扩展类的功能时，优先考虑使用组合，而不是继承。这条原则在 23 种经典设计模式中频繁使用，如：代理模式、装饰模式、适配器模式等。可见江湖地位非常之高！

2）. 无环依赖原则（Acyclic Dependencies Principle - ADP）

当 A 模块依赖于 B 模块，B 模块依赖于 C 模块，C 依赖于 A 模块，此时将出现循环依赖。在设计中应该避免这个问题，可通过引入“中介者模式”解决该问题。

3）. 共同封装原则（Common Closure Principle - CCP）

应该将易变的类放在同一个包里，将变化隔离出来。该原则是“开放-封闭原则”的延生。

4.） 共同重用原则（Common Reuse Principle - CRP）

如果重用了包中的一个类，那么也就相当于重用了包中的所有类，我们要尽可能减小包的大小。

5）. 好莱坞原则（Hollywood Principle - HP）

好莱坞明星的经纪人一般都很忙，他们不想被打扰，往往会说：Don't call me, I'll call you. 翻译为：不要联系我，我会联系你。对应于软件设计而言，最著名的就是“控制反转”（或称为“依赖注入”），我们不需要在代码中主动的创建对象，而是由容器帮我们来创建并管理这些对象。

6）. 不要重复你自己（Don't repeat yourself - DRY）

不要让重复的代码到处都是，要让它们足够的重用，所以要尽可能地封装。

7）. 保持它简单与傻瓜（Keep it simple and stupid - KISS）

不要让系统变得复杂，界面简洁，功能实用，操作方便，要让它足够的简单，足够的傻瓜。

8）. 高内聚与低耦合（High Cohesion and Low Coupling - HCLC）

模块内部需要做到内聚度高，模块之间需要做到耦合度低。

9）. 惯例优于配置（Convention over Configuration - COC）

尽量让惯例来减少配置，这样才能提高开发效率，尽量做到“零配置”。很多开发框架都是这样做的。

10）. 命令查询分离（Command Query Separation - CQS）

在定义接口时，要做到哪些是命令，哪些是查询，要将它们分离，而不要揉到一起。

11）. 关注点分离（Separation of Concerns - SOC）

将一个复杂的问题分离为多个简单的问题，然后逐个解决这些简单的问题，那么这个复杂的问题就解决了。难就难在如何进行分离。

12）. 契约式设计（Design by Contract - DBC）

模块或系统之间的交互，都是基于契约（接口或抽象）的，而不要依赖于具体实现。该原则建议我们要面向契约编程。

**7 互联网架构设计原则**

**一，可（异地）部署和就近路由接入，破除单点故障；**

（可分布，可调度的原则）

**二，数据上报和监控平台；**

（用户行为数据，系统性能监控数据，系统异常和业务相关数据等的上报）

**三，数据分级存储原则：**单内存cache存储，内存cache+异步更新，内存cache+同步更新；

（从三个纬度分析用户行为模型，决定相关数据的存储策略：1，能忍受用户数据的丢失吗？2，能忍受数据的非及时性吗？

3，数据的读写比例分布如何？）

**四，动静分离原则；**

（能静态化尽量静态化，在代码和进程部署上，在DNS层上做好动静分离的系统设计准备）

**五，轻重分离原则；**

（保持接入和业务处理的分离，接入尽量轻量化，使得系统具有很好的吞吐量，处理尽量异步化，使得可以平滑扩展）

**六，破除服务依赖原则：**同一IDC的其他服务对系统的影响，第三方调用系统接口的隔离和过载保护，依赖第三方服务的

监控和安全保护原则等。

**七，柔性可用原则；**

（处理好异常情况下的灰度体验，区分好关键处理路径和非关键路径，而系统设计要尽量把关键路径转换成非关键路径）

**八，能异步的尽量异步原则；**

（通过内存管道，操作流水等技术进行拼接各个处理模块）

**九，灰度原则；**

（灰度发布策略是根据用户号码段，用户ip段，还是用户vip等级，用户所在城市等进行灰度升级，保证系统的平滑迭代）

**十，异常的快速响应和一键切换原则**；

（IDC断电？系统切换到正常的成本是多少？时间呢？需要几个人操作？牛的系统可以一个人在管理后台按一个按钮就可以切换，再按一下就可以切换回来）

**十一，有损服务原则；**

（用低成本提供海量的服务原则）

**十二，一切简单化处理的原则，**真真假假，假假真真！

（方案的简单化，允许的误差评估等）

**十三，充分利用DNS层做好系统的可分布设计。**

**十四，区分系统行为和用户行为并分别进行设计**，甚至在关键时刻可以进行转换。

**十五，坚持app\_server设计的无状态设计原则**，转变用户行为为系统行为，使得app\_server具有无状态的特点。

**十六，负载均衡原则和平滑扩容。**

**十七，多级cache设计以及各个cache的路由设计。**

**十八，“大系统小做”原则。**

**十九，面向接口编程**，面向服务编程设计原则，解耦业务模块的依赖关系。

**二十，强事务模型到最终一致性事务模型的转换原则。**

## 四、架构优化

**1、一台“全能”的服务器**

把应用系统网站、数据库、文件系统等都在一台服务器下，这样形成了最初级的服务器，一般是非常简单的应用，使用的用户量相当有限。一些企业的门户网站或刚上线使用的系统会采用这样的方式进行部署。

**2、系统网站、数据库和文件系统不同的服务器进行部署**

这个有先考虑的是把数据库和系统网站分离出来部署到不同的服务器。很多的系统网站很长一段时间都是把系统网站和文件服务器放在一起，把数据库分离出去后发现网站的性能没有多大的提升时，才考虑把文件系统从系统站点分离出去，减少读取文件带来了网络开销和IO读取。在配置服务是需要根据服务器所承载的职责用途分配不同性能的硬件设备，如文件服务器更需要考虑的是硬盘。

**3、集群的方式进行部署**

当随着业务发展和数据的不断积累，大量的数据和高并发用户的集中访问，在高性能的一台服务器也无法承受进行处理，即使是把系统网站、数据库和文件系统分离到不同的服务器中。作为web服务器是直接面向用户，所有的用户都需要从web服务器作为入口进入系统，承担着大量的用户请求，此时需要考虑的是web服务器的压力问题，一般的处理方式在web服务器前面部署负载均衡服务器调用用户请求，根据设置不同的调度策略把用户的请求分发到各个web服务器中，通常负载均衡和反向代理服务器部署在同一台服务器，到后续才会把他们分开。常见的负载均衡的硬件有F5,软件产品如HAProxy、LVS、Nginx。当然，这种情况下需要考虑的是用户的请求用可能会分发到不同的服务器中(负载均衡可以进行配置)，保证不同请求间的服务端的使用状态（Session、Application、ViewState、Cache）一致性成为需要解决的问题，因此保证是系统是无状态的这个问题就不是问题了。我们要考虑Cache一致性的问题，则把Cache部署到独立的一台或多台缓存服务器中，现在流行的是Redis，也可考虑使用memerycache、AppFabric。可以参考是Docker。

上面的集中式部署，如果系统可以分布在各个地区或区域，则可以考虑分布式部署，不同的地区地域各部署一套系统，定义一个时间点把数据同步到主服务器中，在通过主服务器分发到各个地区服务器，各地区的系统相当于一个中转站。

**4、根据业务拆分成多个应用服务器，web服务与应用进行分离**

一般Web服务和应用都会部署在同一台服务器上，此时则需要把应用从web服务器中独立出来，并且根据业务模块把各个应用部署到不同的服务器中，如数据查询应用、用户管理应用、报表管理应用等。web服务器只处理用户请求和用户数据交互的呈现，应用服务器处理业务逻辑处理和数据读写。应用分布到多个服务器中遵循的规则是每个应用可以承担起相当独立的运作，减少相互间的依赖。不同应用间的通讯可以使用消息队列(MQ)的方式实现，也可以是SOA服务、微服务的方式提供。数据库也可以按照此方式进行拆分。同时把web服务器中的所有js、CSS、image都分离出来部署到静态文件服务器中。

**5、海量数据的查询**

数据的不断积累，到了海量数据时，则会造成数据集查询的性能瓶颈。通常使用的方式是nosql数据库和搜索引擎来解决查询的问题。系统的数据不是一定要使用我们常规的方式放到关系型数据库中，把一下常使用查询的数据或数据库的索引数据放到nosql中。现在使用比较多的是Redis和Mongodb，搜索引擎有lucene和支持.net版本的lucene.net。也有基于lucene内核实现的Solr、ElasticSearch等搜索引擎，这些都可支持集群的方式部署。

**6、系统之间的交互**

随着业务的不断深入和拓展，会出现系统群的存在，而系统系统之间的通讯和数据交互也会频繁的进行，在此时就会出现服务层。服务层是为各个客户端(调用方)提供数据服务，在应用层之上。.net一般服务器层使用的是WCF、WebService和Restful方式的WebAPI，数据传输实体一般都定义成DTO。在我们服务器设计层面上更多的考虑保证数据事务的一致性，例如事务补偿的机制。对每个服务器记录在特定时间段中负荷、资源使用情况、访问量和连接数等参数数据，随时可以掌握每台服务器的运行情况，根据每个业务模块的服务器使用情况进行调度，保证所有的服务器的压力达到相对的均衡。

**7、CDN和数据仓库**

CDN需要考虑两个方面的内容存储问题，一个是内容源的集中存储，另一个是内容在Cache节点中的分布式存储。它借助于建立索引、缓存、流分裂、组播（Multicast）等技术，将内容发布或投递到距离用户最近的远程服务点（POP）处。整体性的网络负载均衡技术，通过内容路由器中的重定向（DNS）机制，在多个远程POP上均衡用户的请求，以使用户请求得到最近内容源的响应。根据内容的可用性、服务器的可用性以及用户的背景，在POP的缓存服务器上，利用应用层交换、流分裂、重定向（ICP、WCCP）等技术，智能地平衡负载流量。通过内部和外部监控系统，获取网络部件的状况信息，测量内容发布的端到端性能（如包丢失、延时、平均带宽、启动时间、帧速率等），保证网络处于最佳的运行状态。可以搭建几个中心，中心之间的数据需要保持同步。

数据库仓库的数据可用通过ETL/ELT将数据定期的从生产环境数据库导入数据。大型的系统主要是使用按列存储、并行数据库的方式。

**8、云**

一般企业可以考虑使用云来部署系统，现在比较流行的是Azure、阿里云、亚马逊等多会提供服务，如阿里云，ECS提供虚拟服务器、SLB提供负载均衡、RDS提供数据库服务、OSS提供存储服务、DRDS提供分布式数据服务MaxCompute提供大数据服务、RocketMQ提供MQ、OCS提供分布式缓存服务，还有很多的服务。这样企业可以根据自身的情况可以购买对应的服务，从而节省成本。

**对应数据库层面性能优化的，可以了解一下Hadoop、Spark、ZooKeeper、Dubbo，但这些在.net应用比较麻烦。**

**一 代码级别**

1 关联代码优化

2 cache对齐

3 设计模式

4 分支预测

5 copy on write

6 内联优化

**二 工具优化**

1 Oprofile

2 Gprof

3 JDK工具

**三 系统优化**

1 cache

2 延迟计算

3 数据预读

4 异步

5 轮询与通知

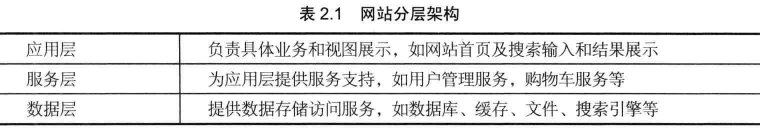
6 内存池

7 模块化

**五、架构模式**

**1. 分层**

分层是企业应用中最常见的一种架构模式，将系统在横向维度上切分成几个部分，每个部分负责一部分相对单一的职责，然后通过上下层的依赖和调用组成一个完整的系统。分层在计算机世界里无处不在，网络的七层结构就是一种分层架构。计算机的硬件，操作系统，应用软件也可以看成一种分层。大型网站架构一般将系统分为应用层，服务层，数据层（这和MVC有异曲同工之意）



通过分层，我们可以将庞大的系统分割成不同的部分，便于分工合作开发与维护。而且各层之间具有一定的独立性，只要维护调用接口不变，各层内部可以独立演化发展只要保证接口不变

**2.分割**

 分割就是在系统纵向上进行切分，将同一层的服务进行纵向分割，包装成高内聚低耦合的模块单元。比如购物、论坛、搜索、广告、卖家、卖家分割成不同的应用，交给不同的部门独立开发

**3.分布式**

对于大型网站，分割和分层的一个主要目的就是便于分布式部署，及不同的模块部署到不同的服务器上，通过远程调用协同工作（这些服务器在同一机房内网时，速度会很快，网络延迟率很低）。分布式意味着使用更多的机器，CPU以及系统资源也就更多，类似于分而治之，将一项工作分给多个人共同完成。另外分布式也存在的一些问题，服务器越多，服务宕机的概率也就越大，一台服务器不可用可能导致整个系统不可用。另外分布式环境中保证数据一致性也比较困难，分布式事务也难以保证。分布式还导致网站依赖错综复杂，开发维护难度大。常用的分布式，分布式应用和服务：是将分层分割后的应用和服务模块分布式部署，改善网站性能。分布式静态资源：将网站的讲台资源如JS,CSS,LOGO图片等资源独立部署，采用独立域名，即人们常说的动静分离

**4.集群**

使用分布式虽然已经将模块分开，然而对于用户集中访问的模块（比如网站首页）效果依然不好，所以还需要独立部署服务器集群，即多态服务器都不熟相同应用，通过负载均衡设备共同对外服务。

**5.缓存**

 缓存是解决系统性能的首选方案，在在之前的[《大型网站架构演化》](http://www.cnblogs.com/xyxxs/p/4733464.html" \t "view_window)中，我们也有提到缓存的形式及特点，这里不做过多累述，主要分为以下几种：本地缓存 ，分布式缓存，CDN,反向代理

**6.异步**

异步消息队列可用性抢，一台宕机另一台可以继续工作（消息产生的服务器和消息处理的服务器）可用性好，另一方面也大大减少了响应时间，消除了并发访问高峰的问题，比如新浪微博就是一个很好的例子，大V用户比如姚晨发了一条微博，我们默认采用退的模式的时候，她发完微博我们就要推2000万次，不可能等待执行完了才给姚晨返回，所以将任务扔到消息队列中，先返回用户体验会更好

## 六、架构素质

在软件开发这个行业走的久了，不免都会考虑将来的发展方向。很多人都已经分析了，基本总结一下就两条路：技术型道路、非技术型道路。其中技术型道路中普遍看好的有系统架构师和培训讲师，而非技术型道路中普通看好的有项目管理和产品销售。今天主要探讨的是系统架构师这条路应该如何走（虽然我个人不太看好，但却一直从事着）。

**要走系统架构师这条路，首先要明确到底什么是架构师，其实似乎没有很明确的定义，而且各个公司的衡量标准也不太一致。我们暂且这样认为：**

（1）架构师的主要责任是提供开发人员和项目经理之间的共用沟通媒体。他们负责让业务规则及需求与工程实践及限制相适应，以确保成功；

（2）系统架构师负责设计系统整体架构，从需求到设计的每个细节都要考虑到，把握整个项目，使设计的项目尽量效率高，开发容易，维护方便，升级简单。

**1 具备实际编程的经验，创新能力**

最少三年吧，多了就不说了，这个年限虽然不能过于武断，但没有这个年限肯定是不行的，毕竟没有积累就没有经验。而且要有成功经验和失败经验，只有成功没有失败，那不是一个完整的经验积累。

**2 具备抽象思维和总结的能力，项目管理能力**

    架构师，顾名思义，在系统未搭建好之前，就要能够有一个草图在心。而如果是对现有系统的改造，那么能在看过系统的文档（如果有的话）和代码后，就能总结出系统的架构特点。

架构师必须能够理解表述模糊的概念并将其变成相关各方能够理解的项目构件。他们必须能够理解抽象概念，并以具体的语言对其进行沟通。开发人员中好的候选者经常要求或自己主动解释开发生命周期中容易混淆的问题。他们能迅速评估各种想法并将其纳入后续工作的操作建议中。 好的开发人员经常具有很强的数学能力，而好的架构师则倾向于表现出更强的口头表达能力，具有很强的解决技术问题的能力，但还必须能够准确获知更为全面的人员如何与技术交互的信息。这要求具有某种形式的抽象思维(而不再是代码的细节)，这种思维能力可能较难形成。

**3 具备书面表达和口头交流的能力，沟通能力**

 综合利用架构图，UML图，文字和代码片断，表达自己设计思想，至于是Word还是ppt，应该通吃。写作技能也非常重要，还需要具有制作草图的技能或使用制图软件的能力。在开发人员中发现架构师的最有价值标准是有效的沟通。架构师必须对理解方面的差距进行预计，然后才能有所贡献。架构师不必对各方意见进行协调，因为这是项目经理的工作。他们的任务是确定表述系统设计时的最佳工具和构件，以促进有效的意见交换。

**4 具备技术资讯吸收和鉴别的能力，学习能力**

作为开发人员出身，对于某一个具体问题的研究能力（虽然很多人总结为google能力），已经相当具备了。但是对技术资讯的全面接受和选择性深入了解能力，并且做出正确的判断，那些技术无非是厂家的噱头，而那些技术是真正可以用到项目，提高项目质量的好技术，这种能力确实至关重要的。任何一个架构师，都会像是一个吸血鬼一样，每天不停的在网络中、书籍中、同事中、...吸取技术和其他。这种能力和毅力是重要的，也是比较难于保持的。

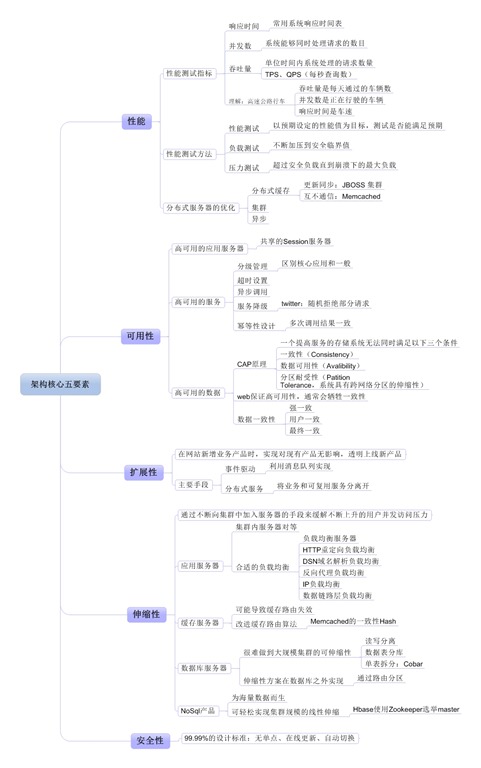
**5 具备积极解决问题的主动性**

    很多时候，架构师的日常工作目标并不十分明确。开发人员直接参考功能规范来列出任务清单，而不是向架构师索取和求助，毕竟开发人员是有独立工作的能力和义务的。但在必要的时候，还是需要架构师出马的，往往在最关键的问题上，架构师亲自出手一小时，要强于开发人员讨论一天。毕竟架构师师总揽全局需求和全局架构的。这就要求架构师在需要的时候，可以主动出手，并且，一个好的架构师可以预知在何时何处开发人员是需要帮助或者协助的。

**除了以上几点以外，还需要很多很多，比如完整的人格魅力、健康的身体等等。总之，这个职业具有一定的特殊性，属于透支体力和精力的职业。这个职业，需要从业者具备太多的素质，却给以不成比例的报酬，绝对是一个付出和收获不成正比的职业。**

## 七、架构核心五要素

**性能、可用性、扩展性、伸缩性、安全性**

[](http://images.cnitblog.com/blog/90573/201404/131651286223938.jpg)

**第一要素~性能**

**1 性能的测试指标**

**1 响应时间**

应用执行一个操作需要的时间，包括从发出请求开始到收到最后响应数据所需要的时间。响应时间是系统最重要的性能指标，直观地反映了系统的“快慢”。下表列出了一些常用的系统操作需要的响应时间。



**2 并发数**

系统能够同时处理请求的数目

**3 吞吐量**

单位时间内系统处理的请求数量;

如：TPS、QPS（每秒查询数）

随着并发数的增大，吞吐量随着增大；

超过阈值后，并发数继续增大，吞吐量下降，直到吞吐量降为0，网站宕机；

理解上述3个指标：类比高速公路行车

吞吐量就是每天通过的车辆数

并发数是正在行驶的车辆

响应时间是车速

**2 性能测试方法**

**1 性能测试**

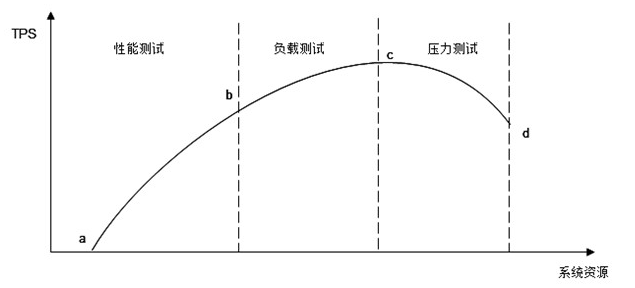
以预期设定的性能值为目标，测试是否能满足预期

**2 负载测试**

不断加压到安全临界值

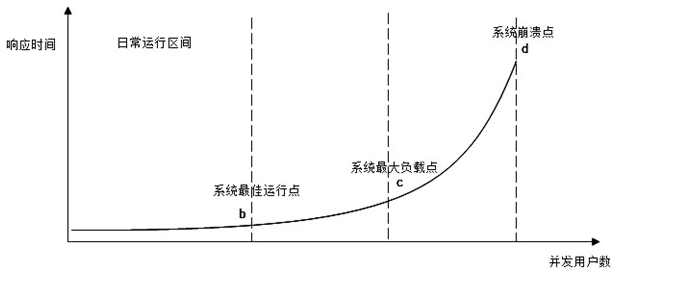
**3 压力测试**

超过安全负载直到崩溃下的最大负载



下图中的横坐标表示消耗的系统资源，纵坐标表示系统处理能力（吞吐量）。在开始阶段，随着并发请求数目的增加，系统使用较少的资源就达到较好的处理能力（a～b段），这一段是网站的日常运行区间，网站的绝大部分访问负载压力都集中在这一段区间，被称作性能测试，测试目标是评估系统性能是否符合需求及设计目标；随着压力的持续增加，系统处理能力增加变缓，直到达到一个最大值（c点），这是系统的最大负载点，这一段被称作负载测试。测试目标是评估当系统因为突发事件超出日常访问压力的情况下，保证系统正常运行情况下能够承受的最大访问负载压力；超过这个点后，再增加压力，系统的处理能力反而下降，而资源消耗却更多，直到资源消耗达到极限（d点），这个点可以看作是系统的崩溃点，超过这个点继续加大并发请求数目，系统不能再处理任何请求，这一段被称作压力测试，测试目标是评估可能导致系统崩溃的最大访问负载压力。

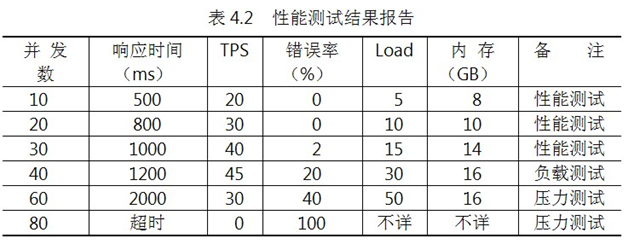
性能测试反应的是系统在实际生产环境中使用时，随着用户并发访问数量的增加，系统的处理能力。与性能曲线相对应的是用户访问的等待时间（系统响应时间），如图所示。



在日常运行区间，可以获得最好的用户响应时间，随着并发用户数的增加，响应延迟越来越大，直到系统崩溃，用户失去响应。

**3 性能测试报告**

测试结果报告应能够反映上述性能测试曲线的规律，阅读者可以得到系统性能是否满足设计目标和业务要求、系统最大负载能力、系统最大压力承受能力等重要信息，下表是一个简单示例。

[](http://images.cnitblog.com/blog/90573/201404/131651265759295.png)

**第二要素~可用性**

**1 高可用的应用服务器**

构建高可用的应用服务器关键是session的处理，如果能使得每天机器上处理的请求都无状态，即可实现应用服务器的线性伸缩；服务器宕机后也可随时将请求放到其它机器上再次处理； 而对于需要处理状态信息的应用，解决方案是让每台机器使用共享的Session服务器，本地上还是无状态特征；

**2 高可用的服务**

设计要点；

分级管理 ，区别核心应用和一般应用；在流量增加到超过网站的处理能力时，为了保证核心应用的可用性，可以将一般应用的服务停止，将资源让位给核心服务； 比如博客中的博文显示和博客的评论功能；

超时设置；

异步调用；

服务降级 ；eg:twitter：随机拒绝部分请求

幂等性设计；

保证多次调用结果一致；

**3 高可用的数据**

CAP原理

一个提高服务的存储系统无法同时满足以下三个条件

一致性（Consistency）

数据可用性（Avalibility）

分区耐受性（Patition Tolerance，系统具有跨网络分区的伸缩性）

WEB架构设计中，通常为了保证高可用性，会牺牲一致性；

数据一致性分为：强一致、用户一致、最终一致；

大型WEB站点一般只保证数据的最终一致性；

**第三要素~扩展性**

在网站新增业务产品时，实现对现有产品无影响，透明上线新产品。

主要手段

**1 事件驱动**

**2 利用消息队列实现**

**3 分布式服务**

**4 将业务和可复用服务分离开**

**第四要素~伸缩性**

通过不断向集群中加入服务器的手段来缓解不断上升的用户并发访问压力。

可能加入的服务器有以下4类，其伸缩性能各不相同；

**1 应用服务器**

通过设计实现集群内服务器对等，使用合适的负载均衡可保证良好的伸缩性；

**2 缓存服务器**

缓存服务器伸缩性良好，但新加入服务器后可能导致缓存路由失效；

通过改进缓存路由算法，比如Memcached的一致性Hash，可实现加入新的缓存服务器后对原有缓存的影响尽量减少；

**3 数据库服务器**

很难做到大规模集群的可伸缩性

实现方式有：读写分离、数据表分库

（单表拆分：Cobar）

也可考虑伸缩性方案在数据库之外实现（通过路由分区）

**4 NoSql产品**

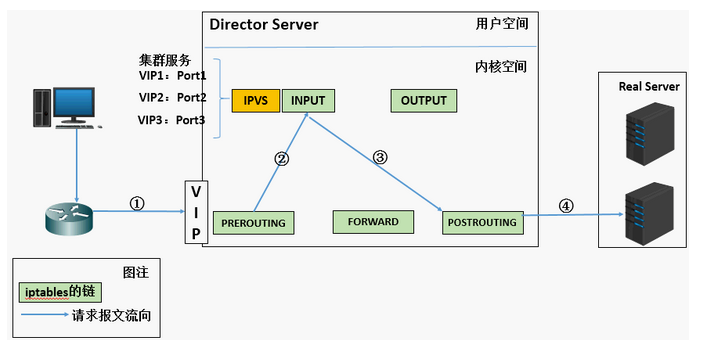
Nosql数据库就是为海量数据而生，可轻松实现集群规模的线性伸缩；

（Hbase使用Zookeeper选举master）

**第五要素~安全性**

**99.99%的设计标准：无单点、在线更新、自动切换**

## 八、LVS



**负载均衡集群是 load balance 集群的简写，翻译成中文就是负载均衡集群。常用的负载均衡开源软件有nginx、lvs、haproxy，商业的硬件负载均衡设备F5、Netscale。这里主要是学习 LVS 并对其进行了详细的总结记录。**

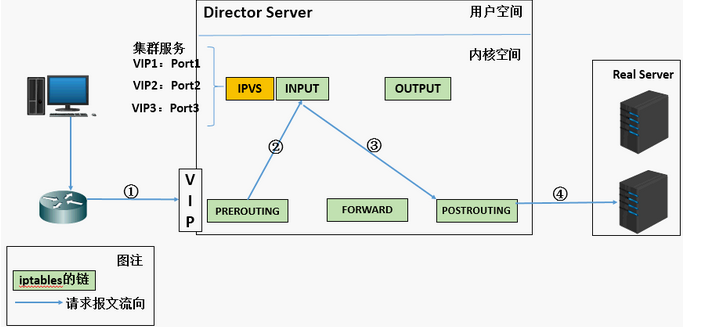
**1 负载均衡LVS基本介绍**

LB集群的架构和原理很简单，就是当用户的请求过来时，会直接分发到Director Server上，然后它把用户的请求根据设置好的调度算法，智能均衡地分发到后端真正服务器(real server)上。为了避免不同机器上用户请求得到的数据不一样，需要用到了共享存储，这样保证所有用户请求的数据是一样的。

LVS是 Linux Virtual Server 的简称，也就是Linux虚拟服务器。这是一个由章文嵩博士发起的一个开源项目，它的官方网站是 http://www.linuxvirtualserver.org 现在 LVS 已经是 Linux 内核标准的一部分。使用 LVS 可以达到的技术目标是：通过 LVS 达到的负载均衡技术和 Linux 操作系统实现一个高性能高可用的 Linux 服务器集群，它具有良好的可靠性、可扩展性和可操作性。从而以低廉的成本实现最优的性能。

LVS 是一个实现负载均衡集群的开源软件项目，LVS架构从逻辑上可分为调度层、Server集群层和共享存储。

**2 LVS的基本工作原理**



1）. 当用户向负载均衡调度器（Director Server）发起请求，调度器将请求发往至内核空间

2）. PREROUTING链首先会接收到用户请求，判断目标IP确定是本机IP将数据包发往INPUT链

3）. IPVS是工作在INPUT链上的，当用户请求到达INPUT时，IPVS会将用户请求和自己已定义好的集群服务进行比对，如果用户请求的就是定义的集群服务，那么此时IPVS会强行修改数据包里的目标IP地

址及端口，并将新的数据包发往POSTROUTING链

4）. POSTROUTING链接收数据包后发现目标IP地址刚好是自己的后端服务器，那么此时通过选路，将数据包最终发送给后端的服务器

**3 LVS的组成**

LVS 由2部分程序组成，包括 ipvs 和 ipvsadm。

1. ipvs(ip virtual server)：一段代码工作在内核空间，叫ipvs，是真正生效实现调度的代码。

2. ipvsadm：另外一段是工作在用户空间，叫ipvsadm，负责为ipvs内核框架编写规则，定义谁是集群服务，而谁是后端真实的服务器(Real Server)

**4 LVS相关术语**

1. DS：Director Server。指的是前端负载均衡器节点。

2. RS：Real Server。后端真实的工作服务器。

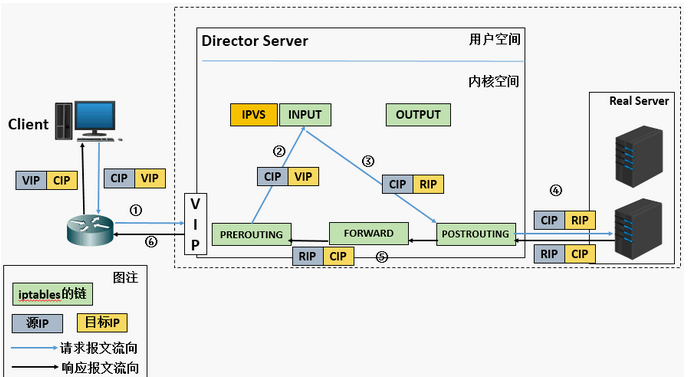
3. VIP：向外部直接面向用户请求，作为用户请求的目标的IP地址。

4. DIP：Director Server IP，主要用于和内部主机通讯的IP地址。

5. RIP：Real Server IP，后端服务器的IP地址。

6. CIP：Client IP，访问客户端的IP地址。

**5 LVS/NAT原理和特点**



(a). 当用户请求到达Director Server，此时请求的数据报文会先到内核空间的PREROUTING链。 此时报文的源IP为CIP，目标IP为VIP

(b). PREROUTING检查发现数据包的目标IP是本机，将数据包送至INPUT链

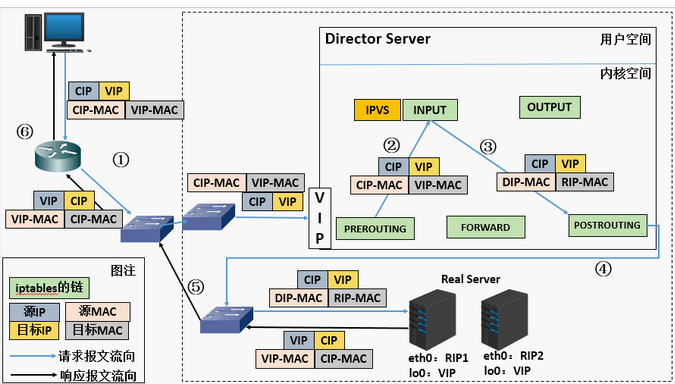
(c). IPVS比对数据包请求的服务是否为集群服务，若是，修改数据包的目标IP地址为后端服务器IP，然后将数据包发至POSTROUTING链。 此时报文的源IP为CIP，目标IP为RIP

(d). POSTROUTING链通过选路，将数据包发送给Real Server

(e). Real Server比对发现目标为自己的IP，开始构建响应报文发回给Director Server。 此时报文的源IP为RIP，目标IP为CIP

(f). Director Server在响应客户端前，此时会将源IP地址修改为自己的VIP地址，然后响应给客户端。 此时报文的源IP为VIP，目标IP为CIP

**6 LVS/DR原理和特点**



**1. 重将请求报文的目标MAC地址设定为挑选出的RS的MAC地址**

(a) 当用户请求到达Director Server，此时请求的数据报文会先到内核空间的PREROUTING链。 此时报文的源IP为CIP，目标IP为VIP

(b) PREROUTING检查发现数据包的目标IP是本机，将数据包送至INPUT链

(c) IPVS比对数据包请求的服务是否为集群服务，若是，将请求报文中的源MAC地址修改为DIP的MAC地址，将目标MAC地址修改RIP的MAC地址，然后将数据包发至POSTROUTING链。 此时的源IP和目的IP均未修改，仅修改了源MAC地址为DIP的MAC地址，目标MAC地址为RIP的MAC地址

(d) 由于DS和RS在同一个网络中，所以是通过二层来传输。POSTROUTING链检查目标MAC地址为RIP的MAC地址，那么此时数据包将会发至Real Server。

(e) RS发现请求报文的MAC地址是自己的MAC地址，就接收此报文。处理完成之后，将响应报文通过lo接口传送给eth0网卡然后向外发出。 此时的源IP地址为VIP，目标IP为CIP

(f) 响应报文最终送达至客户端

**2. LVS-DR模型的特性**

特点1：保证前端路由将目标地址为VIP报文统统发给Director Server，而不是RS

RS可以使用私有地址；也可以是公网地址，如果使用公网地址，此时可以通过互联网对RIP进行直接访问

RS跟Director Server必须在同一个物理网络中

所有的请求报文经由Director Server，但响应报文必须不能进过Director Server

不支持地址转换，也不支持端口映射

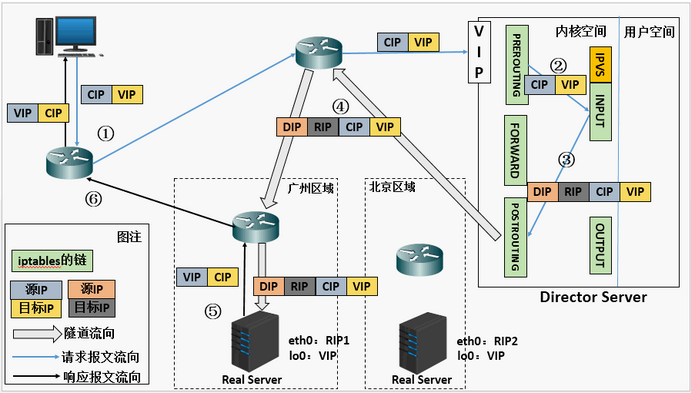
RS可以是大多数常见的操作系统

RS的网关绝不允许指向DIP(因为我们不允许他经过director)

RS上的lo接口配置VIP的IP地址

缺陷：RS和DS必须在同一机房中

**7 LVS/Tun原理和特点**



在原有的IP报文外再次封装多一层IP首部，内部IP首部(源地址为CIP，目标IIP为VIP)，外层IP首部(源地址为DIP，目标IP为RIP)

(a) 当用户请求到达Director Server，此时请求的数据报文会先到内核空间的PREROUTING链。 此时报文的源IP为CIP，目标IP为VIP 。

(b) PREROUTING检查发现数据包的目标IP是本机，将数据包送至INPUT链

(c) IPVS比对数据包请求的服务是否为集群服务，若是，在请求报文的首部再次封装一层IP报文，封装源IP为为DIP，目标IP为RIP。然后发至POSTROUTING链。 此时源IP为DIP，目标IP为RIP

(d) POSTROUTING链根据最新封装的IP报文，将数据包发至RS（因为在外层封装多了一层IP首部，所以可以理解为此时通过隧道传输）。 此时源IP为DIP，目标IP为RIP

(e) RS接收到报文后发现是自己的IP地址，就将报文接收下来，拆除掉最外层的IP后，会发现里面还有一层IP首部，而且目标是自己的lo接口VIP，那么此时RS开始处理此请求，处理完成之后，

通过lo接口送给eth0网卡，然后向外传递。 此时的源IP地址为VIP，目标IP为CIP

(f) 响应报文最终送达至客户端

**LVS-Tun模型特性**

RIP、VIP、DIP全是公网地址

RS的网关不会也不可能指向DIP

所有的请求报文经由Director Server，但响应报文必须不能进过Director Server

不支 RS的系统必须支持隧道

8 **LVS的八种调度算法**

**1. 轮叫调度 rr**

这种算法是最简单的，就是按依次循环的方式将请求调度到不同的服务器上，该算法最大的特点就是简单。轮询算法假设所有的服务器处理请求的能力都是一样的，调度器会将所有的请求平均分配给每个真实服务器，不管后端 RS 配置和处理能力，非常均衡地分发下去。

**2. 加权轮叫 wrr**

这种算法比 rr 的算法多了一个权重的概念，可以给 RS 设置权重，权重越高，那么分发的请求数越多，权重的取值范围 0 – 100。主要是对rr算法的一种优化和补充， LVS 会考虑每台服务器的性能，并给每台服务器添加要给权值，如果服务器A的权值为1，服务器B的权值为2，则调度到服务器B的请求会是服务器A的2倍。权值越高的服务器，处理的请求越多。

**3. 最少链接 lc**

这个算法会根据后端 RS 的连接数来决定把请求分发给谁，比如 RS1 连接数比 RS2 连接数少，那么请求就优先发给 RS1

**4. 加权最少链接 wlc**

这个算法比 lc 多了一个权重的概念。

**5. 基于局部性的最少连接调度算法 lblc**

这个算法是请求数据包的目标 IP 地址的一种调度算法，该算法先根据请求的目标 IP 地址寻找最近的该目标 IP 地址所有使用的服务器，如果这台服务器依然可用，并且有能力处理该请求，调度器会尽量选择相同的服务器，否则会继续选择其它可行的服务器

**6. 复杂的基于局部性最少的连接算法 lblcr**

记录的不是要给目标 IP 与一台服务器之间的连接记录，它会维护一个目标 IP 到一组服务器之间的映射关系，防止单点服务器负载过高。

**7. 目标地址散列调度算法 dh**

该算法是根据目标 IP 地址通过散列函数将目标 IP 与服务器建立映射关系，出现服务器不可用或负载过高的情况下，发往该目标 IP 的请求会固定发给该服务器。

**8. 源地址散列调度算法 sh**

与目标地址散列调度算法类似，但它是根据源地址散列算法进行静态分配固定的服务器资源。

**9 LVS结合keepalive**

LVS可以实现负载均衡，但是不能够进行健康检查，比如一个rs出现故障，LVS 仍然会把请求转发给故障的rs服务器，这样就会导致请求的无效性。keepalive 软件可以进行健康检查，而且能同时实现 LVS 的高可用性，解决 LVS 单点故障的问题，其实 keepalive 就是为 LVS 而生的。

**1、实验环境**

4台节点

Keepalived1 + lvs1(Director1)：192.168.0.48

Keepalived2 + lvs2(Director2)：192.168.0.58

Real server1：192.168.0.18

Real server2：192.168.0.28

IP: 192.168.0.38

**2、安装系统软件**

Lvs + keepalived的2个节点安装

# yum install ipvsadm keepalived -y

Real server + nginx服务的2个节点安装

# yum install epel-release -y

# yum install nginx -y

**3、设置配置脚本**

Real server节点2台配置脚本：

# vim /usr/local/sbin/lvs\_dr\_rs.sh

#! /bin/bash

vip=192.168.0.38

ifconfig lo:0 $vip broadcast $vip netmask 255.255.255.255 up

route add -host $vip lo:0

echo "1" >/proc/sys/net/ipv4/conf/lo/arp\_ignore

echo "2" >/proc/sys/net/ipv4/conf/lo/arp\_announce

echo "1" >/proc/sys/net/ipv4/conf/all/arp\_ignore

echo "2" >/proc/sys/net/ipv4/conf/all/arp\_announce

**2节点rs 上分别执行脚本：**

bash /usr/local/sbin/lvs\_dr\_rs.sh

keepalived节点配置(2节点)：

主节点( MASTER )配置文件

vim /etc/keepalived/keepalived.conf

vrrp\_instance VI\_1 {

state MASTER

interface eth0

virtual\_router\_id 51

priority 100

advert\_int 1

authentication {

auth\_type PASS

auth\_pass 1111

}

virtual\_ipaddress {

192.168.0.38

}

}

virtual\_server 192.168.0.38 80 {

delay\_loop 6

lb\_algo rr

lb\_kind DR

persistence\_timeout 0

protocol TCP

real\_server 192.168.0.18 80 {

weight 1

TCP\_CHECK {

connect\_timeout 10

nb\_get\_retry 3

delay\_before\_retry 3

connect\_port 80

}

}

real\_server 192.168.0.28 80 {

weight 1

TCP\_CHECK {

connect\_timeout 10

nb\_get\_retry 3

delay\_before\_retry 3

connect\_port 80

}

}

}

从节点( BACKUP )配置文件

拷贝主节点的配置文件keepalived.conf，然后修改如下内容：

state MASTER -> state BACKUP

priority 100 -> priority 90

keepalived的2个节点执行如下命令，开启转发功能：

# echo 1 > /proc/sys/net/ipv4/ip\_forward

**4、启动keepalive**

先主后从分别启动keepalive

service keepalived start

**5、验证结果**

实验1

手动关闭192.168.0.18节点的nginx，service nginx stop 在客户端上去测试访问 http://192.168.0.38 结果正常，不会出现访问18节点，一直访问的是28节点的内容。

实验2

手动重新开启 192.168.0.18 节点的nginx， service nginx start 在客户端上去测试访问 http://192.168.0.38 结果正常，按照 rr 调度算法访问18节点和28节点。

实验3

测试 keepalived 的HA特性，首先在master上执行命令 ip addr ，可以看到38的vip在master节点上的；这时如果在master上执行 service keepalived stop 命令，这时vip已经不再master上，在slave节点上执行 ip addr 命令可以看到 vip 已经正确漂到slave节点，这时客户端去访问 http://192.168.0.38 访问依然正常，验证了 keepalived的HA特性。

## 九、Thrift

Apache Thrift软件框架用于可扩展的跨语言服务开发，简单来说就是RPC远程调用，它是一个完整的 RPC 框架体系。

**1 Thrift支持的数据类型**

1.基本类型

bool：布尔值 (true or false), one byte

byte：有符号字节

i16：16位有符号整型

i32：32位有符号整型

i64：64位有符号整型

double：64位浮点型

string：编码或者二进制的字符串

**2 容器（Containers）**

Thrift容器与流行编程语言的容器类型相对应，采用Java泛型风格。它有3种可用容器类型：

list: 元素类型为t1的有序表，容许元素重复。

set:元素类型为t1的无序表，不容许元素重复。

Map

**3 RPC全称为Remote Procedure Call,意为远程过程调用.**

假设有两台服务器A,B.A服务器上部署着一个应用a,B服务器上部署着一个应用b,现在a希望能够调用b应用的某个函数(方法),但是二者不在同一个进程内,不能直接调用,就需要通过网络传输,在AB服务器之间建一条网络传输通道,a把参数传过去,b接收到参数调用自己的方法,得到结果,再通过网络传回给a,简单讲就是A通过网络来调用B的过程.这个过程要涉及的东西很多,比如多线程,Socket,序列化反序列化,网络I/O,很复杂,于是牛掰的程序员把这些封装起来做成一套框架,供大家使用,就是RPC框架.

**4 thrift的跨语言特型**

**thrift通过一个中间语言IDL(接口定义语言)来定义RPC的数据类型和接口,这些内容写在以.thrift结尾的文件中,然后通过特殊的编译器来生成不同语言的代码,以满足不同需要的开发者,比如java开发者,就可以生成java代码,c++开发者可以生成c++代码,生成的代码中不但包含目标语言的接口定义,方法,数据类型,还包含有RPC协议层和传输层的实现代码.**

**5 thrift的协议栈结构**

thrift是一种c/s的架构体系.在最上层是用户自行实现的业务逻辑代码.第二层是由thrift编译器自动生成的代码，主要用于结构化数据的解析，发送和接收。TServer主要任务是高效的接受客户端请求，并将请求转发给Processor处理。Processor负责对客户端的请求做出响应，包括RPC请求转发，调用参数解析和用户逻辑调用，返回值写回等处理。从TProtocol以下部分是thirft的传输协议和底层I/O通信。TProtocol是用于数据类型解析的，将结构化数据转化为字节流给TTransport进行传输。TTransport是与底层数据传输密切相关的传输层，负责以字节流方式接收和发送消息体，不关注是什么数据类型。底层IO负责实际的数据传输，包括socket、文件和压缩数据流等。

**6 thrift的下载与安装**

首先,在官网下载安装包http://thrift.apache.org/download 目前官网是0.10.0版本.下载完之后解压到想要安装的目录.

进入根目录:

step1:cd thrift-0.10.0

step2:./configure

step3:make

step4:make install

安装的时候,第二步出现了问题,提示:Bison version 2.5 or higher must be installed on the system!

原因是Bison版本过低,mac默认安装的版本是2.3,因此需要安装最新版的Bison,命令行输入:brew install bison安装最新版bison,如果你的命令行反馈:Command not found,很可能是因为你没装homebrew,命令行输入:

ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"

**7 java和thrift**

**创建一个服务Hello,创建文件Hello.thrift,代码如下:**

namespace java service.demo

service Hello{

string helloString(1:string para)

}

这里定义了一个名为helloString的方法,入参和返回值都是一个string类型的参数.

终端进入Hello.thrift所在目录,执行命令:

thrift -r -gen java Hello.thrift

发现在当前目录下多了一个gen-java的目录,里面的有一个Hello.java的文件.这个java文件包含Hello服务的接口定义Hello.Iface,以及服务调用的底层通信细节,

包括客户端的调用逻辑Hello.Client以及服务端的处理逻辑Hello.Processor,

**pom.xml中添加相关的依赖,并将Hello.java文件复制到项目中:**

<dependency>

<groupId>org.apache.thrift</groupId>

<artifactId>libthrift</artifactId>

<version>0.10.0</version>

</dependency>

<dependency>

<groupId>org.slf4j</groupId>

<artifactId>slf4j-log4j12</artifactId>

<version>1.7.5</version>

</dependency>

**创建HelloServiceImpl实现Hello.Iface接口:**

package service.demo;

import org.apache.thrift.TException;

public class HelloServiceImpl implements Hello.Iface {

public String helloString(String para) throws TException {

return "result:"+para;

}

}

**创建服务端实现代码HelloServiceServer,把HelloServiceImpl作为一个具体的处理器传递给Thrift服务器:**

package service.demo;

import org.apache.thrift.TProcessor;

import org.apache.thrift.protocol.TBinaryProtocol;

import org.apache.thrift.server.TServer;

import org.apache.thrift.server.TSimpleServer;

import org.apache.thrift.transport.TServerSocket;

import org.apache.thrift.transport.TTransportException;

public class HelloServiceServer {

public static void main(String[] args) {

try {

System.out.println("服务端开启....");

TProcessor tprocessor = new Hello.Processor<Hello.Iface>(new HelloServiceImpl());

// 简单的单线程服务模型

TServerSocket serverTransport = new TServerSocket(9898);

TServer.Args tArgs = new TServer.Args(serverTransport);

tArgs.processor(tprocessor);

tArgs.protocolFactory(new TBinaryProtocol.Factory());

TServer server = new TSimpleServer(tArgs);

server.serve();

}catch (TTransportException e) {

e.printStackTrace();

}

}

}

**创建客户端实现代码HelloServiceClient,调用Hello.client访问服务端的逻辑实现:**

package service.demo;

import org.apache.thrift.TException;

import org.apache.thrift.protocol.TBinaryProtocol;

import org.apache.thrift.protocol.TProtocol;

import org.apache.thrift.transport.TSocket;

import org.apache.thrift.transport.TTransport;

import org.apache.thrift.transport.TTransportException;

public class HelloServiceClient {

public static void main(String[] args) {

System.out.println("客户端启动....");

TTransport transport = null;

try {

transport = new TSocket("localhost", 9898, 30000);

// 协议要和服务端一致

TProtocol protocol = new TBinaryProtocol(transport);

Hello.Client client = new Hello.Client(protocol);

transport.open();

String result = client.helloString("哈哈");

System.out.println(result);

} catch (TTransportException e) {

e.printStackTrace();

} catch (TException e) {

e.printStackTrace();

} finally {

if (null != transport) {

transport.close();

}

}

}

}

全部工作完成后,下面来测试一下,先执行服务端main方法,在执行客户端main方法,会在客户端控制台打印出:result:哈哈.