可伸缩性架构设计

谢新华

一、项目概念

目标客户是历史上具有悠久的传统和成就的大公司，原来的管理和设计模式上采用层次化垂直结构，利用这种结构打造了一个具有固定业务、确定交互、可以重复执行的高效企业模型，这个模型曾经是高效率的也是正确的。

但是近年以来，随着用户对产品更新换代的要求越来越快、质量要求越来越高、竞争日益剧烈、外部压力日益增长，迫使这个企业在管理模型上重新定位，打碎长久以来形成的垂直结构，保证产品需要具有多样性、弹性和专业性，逐步相成一种趋向于水平集成的业务模型，也就形成了企业重构的趋势。

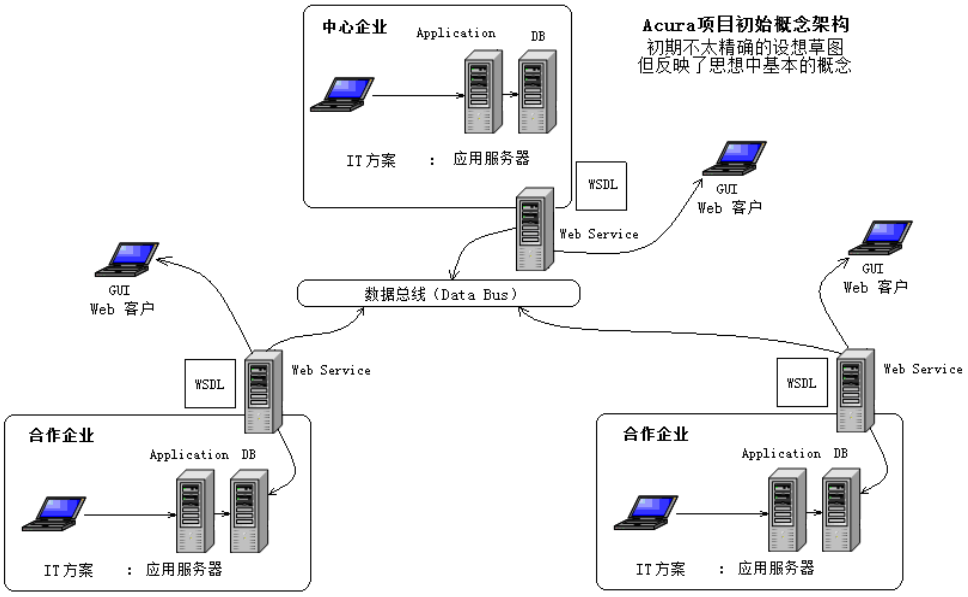
这种企业业务模型的解构趋势，也就是各个企业将根据自己的能力进行业务上的专业化分工。让一个企业主要专注于自己的核心业务，核心业务以外的任务将外包给第三方来提供实际的支援和交付，每个第三方再将重点放在提高自身的核心服务上。他们认为，这种由业务伙伴、供应商和客户形成的更多合作与互动，将会使他们向新的业务空间发展，并以更快的速度和更高的质量，为客户提供更加先进的设计和产品。

于是，当初封闭的企业内设计管理模式，无法应用于异地合作方协调设计环境，这就需要搭建基于互联网的合作方协同沟通平台，让部件设计合作方在早期就介入产品的研发过程，以缩短主要设计部门和合作方的沟通时间，提高合作方在新产品设计中的响应能力，实现各方共赢的局面。

可见这个软件产品对于用户具有相当大的价值，是值得认真对待的。

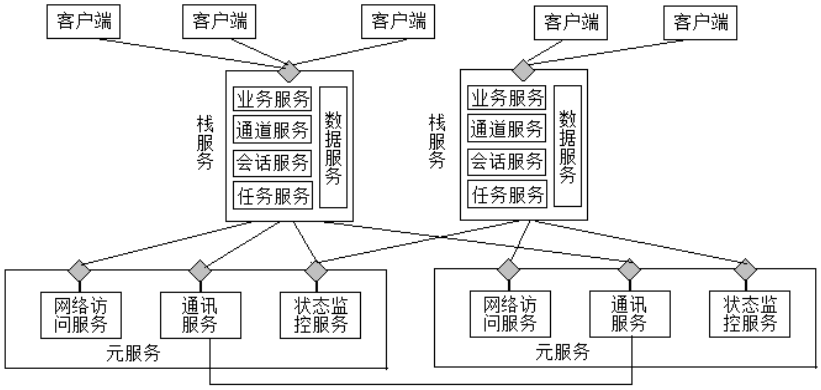
为此，项目组为项目起了一个好听的名字：Acura，

给项目起合理的名字非常重要，因为人类就是一个喜欢命名的生物。如果你认真给项目起名字，就意味着你非常在意这个项目，并且会仔细思考认真呵护这个项目。后续的所有人员包括编码员、测试人员等都会认真地给变量或者方法命名，使项目的可读性增加而且整洁，清洁的产品将从命名开始。项目的初步架构概念如下图所示。



系统中各独立部分的位置关系体现为传统的分层结构，而整体上用服务的特征来分布，

而服务又分为栈服务与元服务两个层次，如下图所示。



注意，前期并不需要做庞大的前端设计，而只需要一个简单的概念模型。图中的这些大块，可以随着功能模块的添加而轻松的增长。这种基于服务的架构使得每一部分保持高的内聚，并尽可能减少相互间的耦合，开发每一部分的程序人员并不需要过多的关注其他部分，增强了系统的可理解性，而且开发的专业性很集中，也保持了概念上的一致性。

这个架构虽然很基本，但是却为增长提供了坚实的基础。回想前面的例子，那个系统没有总体规划，只是在方便的地方嫁接和修补功能。这里之所以称之为规划，是因为初始设计需要一定的抽象性，而不是面面俱到。

二、可伸缩性架构设计问题

1，为什么要考虑可伸缩型问题

在进行 Acura 系统初始概念确定之后，我们需要对某些宏观架构概念作更细腻的分析。一个需要集中思考的问题，就是如何确保系统在伸缩的时候的弹性。随着越来越多的系统在互联网上提供访问，伸缩性变得越来越重要。因此在这个项目中，我们对系统的可伸缩性问题作了更加深入的讨论，从而改变了初期确定的架构。为什么要研究可伸缩性问题呢？

1）系统处于极其容易变化的环境中，难以进行初期的容量规划

当一个系统处于极其容易变化的环境中时，初期的容量规划有时显得比较荒谬，因为到底系统运行后会发生什么？谁也无法预料。

2）系统规模和特征可能会发生变化

对于一个水平模型大型离岸合作方协同设计系统，初期的合作方可能比较少，关系也比较简单。到后来可能有相当多的合作方形成巨大的商业设计网络。在商业运营进一步开展的过程中，这些合作方在数量和业务范围上还会发生许多变化。与普通 Web 系统只是提供静态内容不同，这些合作方的每一个相关用户，不但需要与中央企业进行交互，也需要彼此之间进行交互（例如在三维空间中修改设计、相互探讨）。这类交互使得这类系统伸缩性问题变得更加复杂。

3）系统的应用特点是并行性

对于任意两个参与者，在某个时间进行交互的可能性是比较小的，但几乎所有的人员在所有的时刻都在与他人进行交互。因此这样的系统并行程度将会相当高，而且只有少数交互是相互依赖的。另一方面，一个员工的意外动作使服务器崩溃，将可能影响相当多的人的工作。如果相关人员的数目达到了上万级，伸缩的能力就成了任何架构的首要需求。

2，系统设计的背景

1）伸缩性架构设计需要包含多台服务器

正是因为上述考虑，在创建 Acura 项目的时候，我们意识到的第一件事情就是凡是伸缩性架构设计都需要包含多台服务器。即使单台大型服务器能够处理这个负载，一开始就架设大型服务器在经济上是不可行的。因为我们不能断定，在建立水平商业模型的时候，是不是一开始就那么成功，万一规模不够，就会造成巨大经济损失。另一方面，也不清楚在将来商业模型发生变化的时候，这样的大型服务器是不是还能发挥作用。

2）我们赌的是未来发展方向是并发而不是速度

另一方面，我们也要看到了芯片架构发展的趋势。客户个人电脑中 CPU、内存与图形处理能力大幅度提升，图形能力甚至已经超过了一些大型工作站。还需要注意到，我们认为芯片演进的趋势，已经从不断增加速度转为实现多核处理器（2～16 核），例如，在时钟速度不变的情况下，4 核处理器能比单核芯片多做 3 倍的事情。这就需要在设计的时候考虑到并行性，如果没有考虑到这种并行性，要提高性能的空间是很小的，好在制造这种并发比增加 CPU 速度要容易得多。

3）系统架构应该是多核芯片与分布式系统的结合

基于这一事实，设计中我们考虑，整个系统架构应该是多核芯片与分布式系统的结合。在这个系统中所发生的情况与现实世界是相似的，也就是每个客户在系统中相互讨论设计的时候，只是与这个体系中很少的一部分人发生交互，这正是并行计算任务的特点，也是多核多机系统应该擅长处理的那种任务。

4）架构设计还需要考虑开发人员的特点

另外还有一个情况，也就是参与 Acura 项目开发的程序员，大多数擅长于图像系统编码，但很少接受过分布式计算和并发编程方面的训练，也不太有这方面的经验。即使少数在这个领域接受过一些训练的人，对于并发处理也会感觉到困难。这样一来，要大多数程序员来做这种并行并发设计，就是要求他们做超出自己专长和经验的事情。

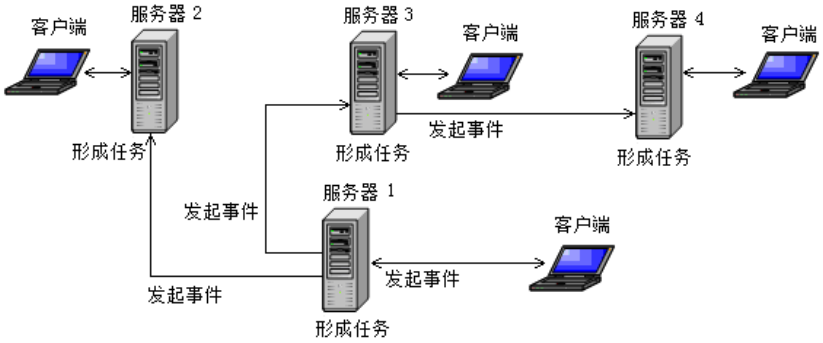
3，确定设计的首要目标

1）系统设计的首要目标是可伸缩性

为了保证设计的指向性，我们考虑的下一个问题是：什么样的情况下设计才被认为是成功的？这就需要确定设计的首要目标。总结上面所讨论过的背景，Acura 项目所面对的的情况是：为了实现整个架构的伸缩性，系统应该是分布式的、并发的。但是又要为开发者提供简单得多的编程模型。换句话说，程序员应该把系统看成一台单机，运行着一个线程。把它部署到多线程和多计算机上的工作，应该由 Acura 的基础设施来达到。

2）事件驱动的任务系统

Acura 项目要求程序设计是事件驱动的，其服务器写成了事件监听器，以监听客户端发生的事件：如果客户发生了一个事件，服务器则生成一项对应的任务，这个任务是一个短期的计算序列，包括操作网络上的信息，并以生成事件的客户端为端点，与其它客户端进行通信。并且也完成一些该服务必须完成的一些共享计算，如下图所示。



3）“胖”客户端有利于提升整个系统的性能

还要注意到这个系统概念中客户端是相当“胖”的，客户端应该具有强劲的 CPU、很大的内存以及能力很强的显卡。大部分的计算都在客户本地完成，包括图形图像处理等。只要有可能，一些短期不会改变的数据也可以放在客户端，包括图形信息、材料信息、设计规则以及某些基本的业务流程。为了保证系统的实时性，服务器的设计目标就是尽可能的减少计算、尽可能简单，绝大多数计算任务都留给了客户端。这在目前客户端 PC 能力越来越强的情况下是合理的，它可以极大地提升整个系统的性能。

4）在网络上传输的数据要精心选择

而网络上传输的数据是经过精心选择的，服务器真正的工作就是保证整个共享业务的真实状态，而且没有权限的人员也不能自行修改这些状态。这就保证了网络数据的有效和敏捷。尽力减少网络上传递的信息，可以确保系统的延时尽可能控制在可以接受的范围内。

5）与典型企业级“瘦”客户端的概念的区别

这种设计方案与典型的企业级“瘦”客户端的概念是不同的，在普通企业级系统中，一个“胖”服务器处理企业绝大多数业务，而数据库可能更“胖”。这在很多情况下是合理的，但是对于本系统客户端需要处理复杂的图形和图像，交互的性能也要求很高，所以采用胖客户端是合理的。

另一方面，普通企业级系统 90%的信息是只读的，大多数任务都会读取大量数据，然后修改少量数据，而在在这个系统中，大多数任务是访问服务器上少量的状态数据，而且在访问的数据中，大约一半以上会改写。不过，在这个合作方协同设计系统中，有一部分业务是符合瘦客户端的要求，例如管理信息等，对于这一部分业务，设计的时候也采用 B/S 架构。

4，关注整个系统设计的主要敌人

1）关注主要敌人可以使指向更加清晰

在 Acura 项目初始概念设计的时候，除了关注成功的目标以外，还关注了致使项目失败的主要敌人是什么。实践告诉我们，避免错误比处处都做得对容易，这样就更加能够使设计具有指向性，更加清晰，并且避免风险。我们注意到了，致使 Acura 项目失败的最大敌人是“延迟”。

在大多数企业环境中，目标是管理业务，只要吞吐量得到改进，在处理中有些延迟也是可以接受的。但是对于强调实时交互的协同设计，延迟会大大影响这种协同设计的有效性，所以基础设施应该围绕着减少延迟来设计，即使以吞吐量为代价也在所不惜。

2）选择击败敌人的策略

为了应对巨大的用户以及可伸缩性，目前大多数情况采取的办法主要有两种：

第一种：把客户设计成不同的区域，每个区域一台服务器，也就是以地理位置为中心进行设计，由于限定了区域的大小，就可以防止客户进入太多而发生阻塞。但是这样的方法有个问题，就是设计的时候必须考虑，哪些区域的客户应该放在同一台服务器上，哪些区域组成一个伸缩单位，这些考虑必须是开发工作的一部分。

第二种：称之为“分区”（sharding），也就是说把工作相同的一部分做成一个分区，也是一个区域的副本，运行在区域中的每个服务器上。这样对于同一个类型的工作，客户数目就可以大大增加。但是它不允许不同分区的客户进行交互，这就使它的应用受到了限制。

Acura 项目的设计主要目标是支持随时伸缩性，同时业务逻辑不能够受到这种伸缩性的影响。这个架构应该支持业务随时动态响应负载，但又不能使这种响应成为业务逻辑设计的一部分，为此我们该怎么来做呢？

5，项目的宏观架构思想

1）每个服务都用一个接口描述实现“分而治之”

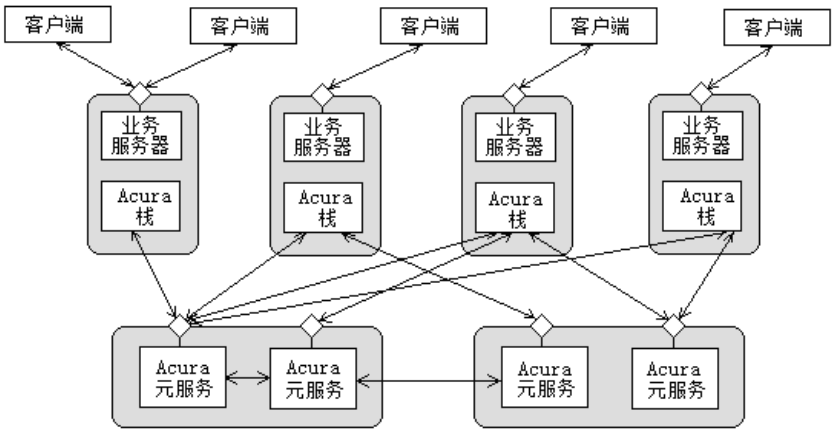
Acura 项目的技术基础是 Web Service，它是由一组独立的服务所组成，这些服务可以在系统服务器端的地址空间内获得。每个服务都定义了一个小的编程接口（很像经典操作系统的服务），这些接口可以用 WSDL 来描述。它支持对虚拟服务器端访问持久存储，调度并执行任务，以及与其它客户端通信。

用一组互相联系的服务来构建这个系统，显然开始了“分而治之”的过程，每个服务都用一个接口描述，这就可以让使用这个服务的程序不受底层实现变更的影响。

这种方法还有一个附带的好处，由于系统很庞大而且应用区域广泛，我们希望不同地区的开发团队创建更多的独立服务，使得这些服务的不同组合可以在不同的情况下使用。这个概念称之为面向服务的架构（SOA）。

2）基本宏观架构的描述

根据上面的概念，我们得出了这种大型公司离岸合作方协同设计系统基本架构。如下图所示。



具体地说，一些服务器构成了不同业务系统的后端，每个服务器针对不同类型的客户，运行一组选定的副本（我们称之为 Acura 栈），以及业务逻辑的副本。客户端将连接到其中的一个服务器，与服务器上所保存的这个大型公司世界的抽象表示进行交互。

每个副本都是独立的与客户端交互，同时每个服务其中的副本并不需要知道在其它机器上运行着的其它副本。程序员编写程序的时候，就像在一台机器上编写本地代码一样。

事实上如果所处理的内容只需要一台服务器（比如本地修改设计，不需要多人交互性设计），那么它就是在一台服务器中运行。而从客户端到服务器的通信机制，是基础设施的一部分，这个机制支持客户端到服务器的直接通信。

3）元服务

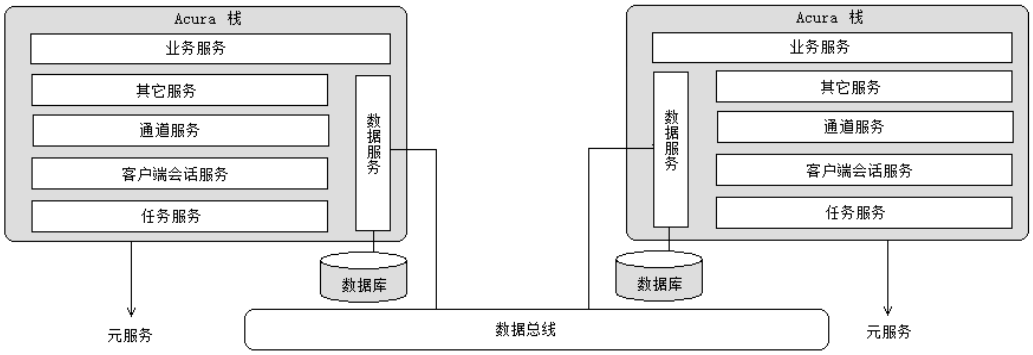
Acura 栈由一组元服务来协调，这是一组网络访问服务，对于业务的程序员是不可见的，他看到的只是由 WSDL 所表达的接口。这些元服务支持栈的各个副本之间进行协作，共同运营整个系统。例如，这些元服务来监控每个副本的工作，一旦某个副本失效，就会发起失效恢复。再比如，这些元服务可以跟踪每个服务器的负载，在需要的时候重新分配负载，或者提请增加服务器。

这些元服务包括：网络访问服务、通讯服务以及服务器状态监控服务等。由于这些元服务对于用户和业务程序员来说是完全隐蔽的，因此它们可以随时改变或者移除，或者添加新服务，这都不需要改变业务代码。

4）Acura 栈

对于业务程序员来说，可见的架构就是栈中间所包含的一组服务。服务的全集是可以改变和配置的，但 4 个基本服务必须存在，他们构成了整个系统运营环境的核心，如下图所

示。



6，基本服务的描述

在 Acura 栈基础架构有两个最基本的服务：也就是数据服务和任务服务。

1）数据服务

在这些栈层面的服务中，最基本的服务是数据服务，主要用来保存、读取和操作持久数据。在 Acura 项目中，这些数据是分布的保存在网络各个节点上的，有一部分也保存在与本服务相联的轻型数据库中，通过一套预先定义的标准方法来调用。任何存在时间超过一个任务的数据都被认为是“持久”的，必须在“数据服务”中保存。

由于不同服务器上的副本都共享同一个（概念上的）“数据服务”实例，所以“数据服务”也把这些副本相联系，所有的副本都可以根据需要读取和改变存储在“数据服务”中的数据。这样的数据称之为数据总线。在设计的时候，“数据服务”必须针对延迟优化，而不是针对吞吐量优化。很多情况下并不需要对数据进行复杂的查询，因此使用简单的命名策略也就足够了。

2）任务服务

它用于调度和执行任务，这些任务主要用于响应客户事件，也响应内部事件。大部分任务是一次性事件，是由于客户的某种动作所产生的。它们可能是从“数据服务”中读取一些数据，操作这些数据。也可能是进行一些通信。任务也可能会生成其它的任务。所以的任务执行时间必须很短，例如默认为 100 毫秒。

在上层程序员所看到的可能是因为事件（或者是服务器逻辑生成的）生成的单个任务，但底层基础设施则是尽可能调度尽可能多的任务，并且是并发执行的。

要注意到这样的并发可能会导致数据竞争，这就要求在底层，在任务程序员看不到的地方，实现“任务服务”和“数据服务”的协作，所有对于数据的改变都以“数据服务”为中介，把“任务服务”调度的每个任务都包装在一个事务里面。这些事务保证任务中的所有写操作，要么全部完成，要么都不完成。如果多个服务试图改变相同的数据对象，只有一个任务会执行，其他任务会中止，并安排在稍后执行。

如果数据对象先被读取，而且读取时说明要进行修改。其间数据被其他客户修改了，那么在任务提交之前，“任务服务”应该能检测到这种修改了，这样一来，就有可能更早的检测到冲突。

把任务包装到一个事务里面意味着通信机制也必须支持事务，只有当包装了消息发送任务的事务提交的时候，消息才会发出。这是通过 Acura 栈余下两项核心服务来完成的。

7，通信服务的描述

在 Acura 栈基础架构的通信服务包括会话服务与通道服务。

1）会话服务

这是客户端与系统环境之间通信的中介，在登录认证之后，在客户端和服务器之间就会建立一个会话（Session）。服务器通过会话监听客户端发出的消息，解析消息的内容，并确定生成怎样的任务来响应这个会话。客户端也通过会话来接受来自服务的响应。

这些会话隐藏了客户端和服务器的真实端点，这对于 Acura 架构多机伸缩性策略是十分必要的。会话也负责确保维持消息的顺序。如果来自某个客户端的前一条消息还没有完成，后一条消息也就不会提交。在“会话服务”对任务进行这样的排序之后，“任务服务”就得到了极大的简化。“任务服务”可以假定它在任何时候收到的任务，在本质上都是并发的。

2）通道服务

通道是一种一对多的通讯机制。从概念上说，通道可以有任何数量的客户端参与，任何送达该通道的消息，都回送到与这个通道有关的客户端。似乎这种点对点的通信并不会加重服务器负担，但是通信需要受到一些受信任的代码控制，防止某些客户发送不正确的消息或者欺骗消息。对于“胖”客户端来说，原则上所有客户都是不受信任的（因为客户可以自己订制性能），所以通道消息必须经过服务器，并经过服务器检查以后才能被发出。

通道服务必须遵行任务的“事务”语义，所以不能使用简单的 Send()来实现。

这些通信机制为我们实现伸缩机制奠定了基础。由于在实体通信和通信起止端的实际位置之间存在一个抽象层，所有的通信都必须通过会话或者通道的抽象层，而这些抽象层又不会暴露客户端或者服务器通信的真实端点。这就意味着在这个系统中我们可以很容易的把服务器通信的端点从一台机器移到另一台机器，同时不会改变客户对这次会话的感觉。

8、任务的可移动性

Acura 项目要实现负载均衡的能力，关键之处在于：对于 java 编写的任务而言，只需要这些机器包含相同的虚拟机，那么所有的栈服务、响应客户端事件或内部事件的任务，都可以从一台机器移动到另一台机器上。任务读取和操作的所有数据，都必须从数据服务中获得，而数据服务是所有服务器栈所共享的。通信由“会话服务”和“通道服务”来实现中介，它抽象了通信的真实端点。因此所有的任务都可以运行在任何一台服务器的实例上，同时不改变任务的语义。

这就使得 Acura 的基本伸缩机制看起来很简单。如果一台机器超载了，只需要把这台机器的任务迁移到另一台负载较小的机器上就可以了。如果所有的机器都超载了，那就要在群集中增加新的机器。底层的负载平衡软件，会自动把负载分发给新的机器。

对单台机器的负载进行监控，并且在需要的时候重新分配负载，这是元服务的工作。这种元服务是网络层面的工作，对于业务开发的程序员是不可见的，但对于系统中的服务相互之间是可见的。例如，这些元服务会监控哪些机器正在运行，哪些用户与某台机器有关，不同的机器当前的负载情况。但是这不会影响业务逻辑的正确性，这就可以让我们尝试不同的策略和方法，实现系统的动态平衡。

9，关于并行与延迟的考虑

1）数据存放在持久服务而不是内存

在整个 Acura 项目系统设计中，由于系统最大的敌人是延迟，一般人认为，把所有的信息都放在内存中是合适的，因为这样才可能使延迟最小。把数据都存放在“持久服务”中肯定会带来延迟，但是我们相信访问内存和访问“持久服务”的差异并不像一般人的看法那么大。我们需要在延迟和并发中寻求平衡。

系统架构概念拒绝在服务器的内存中存放任何重要信息，所有生命周期超过一个任务的的数据都存放在“持久服务”中。这样就可以让服务可以使用多核架构，也便于支持把任务从一台机器迁移到另一台机器，从而在一组机器上实现负载平衡。以此来达到所需要的效

率，而不是仅仅依靠机器性能。让所有的数据持久，我们就可以支持在服务器上使用多线程（从而支持多核），而在系统中大量使用并行计算，就会使系统整体性能更好。

2）利用数据库缓存

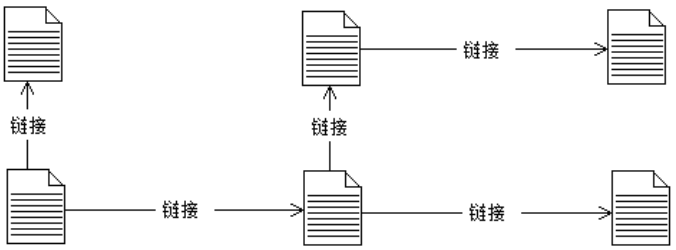
虽然概念上每个生命周期超出一个任务的数据都需要从持久存储中读出，并且写入存储，但我们还是可以利用数据库缓存，从而减少这种方法导致的数据库延迟。特别是如果我们能够把访问局限在特定服务器上的几组对象，就可以利用这个服务器上的缓存，以达到接近内存对象的读写速度。我们可以利用基础设施来识别任务属于哪些用户，收集特定时刻数据访问模式和通信模式，这样就可以准确估计那些客户应该和另一些客户放在一起，根据需要把客户移动到相应的服务器上，这样就可以利用缓存有效的减少延迟。当然这种方法也并不是必须的，因为实践表明这种延迟影响并不是那么大。

三、可伸缩性数据总线与面向资源的架构

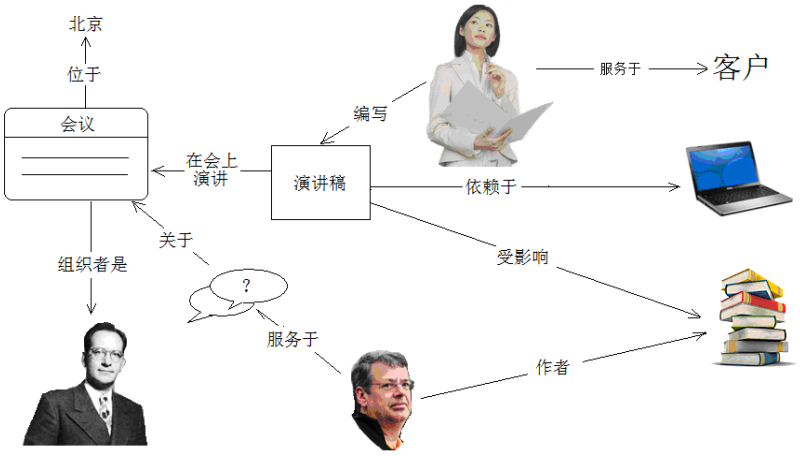
现在我们把眼光转移到数据总线。由于分布式合作方协同设计的业务特点，在 Acura 项目需要集成互联网上的分布式数据，把它们变成一个统一的数据总线，并以一致的方式向使用者提供。这里需要考虑的问题是：在系统将来演化的过程中，怎么样才能使数据的应用变得容易和易于理解呢？如何才能简化这些数据的应用呢？我们认为，遵从人们的习惯就可以使事物变得简单。为此，在这一部分的设计中，我们考虑以资源为核心的架构设计风格。

1，Web 思想的演化

在目前流行的概念中，Web 是以文档为中心的，文档的真正神奇之处在于链接，这使我们能够通过方便的建立新窗口来访问链接的内容。没有起点，也没有终点。只要我们知道要找的是什么，我们通常就能够找到它。目前常用的搜索引擎，可以帮助我们来找到所要找的内容。Web 中的文档是用名称来区别的，名称成为一种重要的沟通工具。



但是，Web 应用不论提不提供具体数据，基本上它都是用内部数据来驱动的，在所谓“Web 2.0”的网络技术流行以后，数据在系统中的核心地位就变得更加明显了。因为“Web 2.0”展现的核心主题就是数据驱动的。基于数据的 Web 连接了人、文档、数据、服务和概念。如下图所示。



如何使这样具体的东西被抽象和简化？我们可以这样设想：在这个环境中，基本的交互是逻辑上的客户端-服务器请求。我们必须有感兴趣的信息的地址。利用一种标识符“统一资源定位符（URL）”，无歧义的确定了在全球地址空间的一个引用，并且告诉我们如何解释这个请求。这已经成为了一种习惯。

人们愿意使用 URL，因为在这个过程中，不需要事先懂得实现这个请求的技术。这样，就有可能使请求变得简单，并且在面对后端变化的时候有弹性。但是，资源比文档所需要的操作更多，为了在相同名字的地址中可以取回不同的数据格式，这就需要一些属性来控制细节。

2，以资源为核心的 REST 设计风格

1）REST 以资源为核心的模型和相应的设计风格：

与以远程业务服务为核心的 SOA 相比，以资源为核心的架构让我们可以从崭新的视角审视互联网应用。面向服务的架构（SOA）核心概念是服务（Service），服务所描绘的是业务。比如：我们要提供整数加法 Web 服务，我们会很自然地想到通过类似下面的 URL 来表达服务接口：

http://www.example.com/add?a=1&b=2

并通过 xml 结构表达结果：3。

在面向资源的架构（Resource-Oriented Architecture，ROA）”中主要应用被称为 REST（Representational State Transfer 表述性状态转移）的描述方法。REST 是 HTTP 协议的作者 Roy Fielding 博士在其博士论文中提出的一种互联网应用架构风格。REST 是为互联网应用量身定做了一个简洁模型，也是与 HTTP 协议的完美结合，它使得架构具有高扩展性，为互联网应用构架设计和异构系统集成设计带来了一套新的方法论。

REST 是一种轻量级的 Web Service 架构风格，它的实现和操作明显比 SOAP 和 XML-RPC 更为简洁，可以完全通过 HTTP 协议实现。REST 的核心概念是资源（Resource）。在面向资源的架构中并没有服务的概念，同样是上面的例子，在 REST 的世界中，我们写：

http://www.example.com/add?a=1&b=2

表达的是一个 XML 网页资源的 ID，而非服务的接口。所以，REST 让我们从资源的角度来审视互联网应用并指导我们的设计，这是它与 SOA 最本质的区别。

2）REST 的设计概念和准则

REST 在互联网应用中可以降低开发的复杂性，提高系统的可伸缩性。REST 提出了一些设计概念和准则：

- 网络上的所有事物都被抽象为资源（resource）；
- 每个资源对应一个唯一的资源标识（resource identifier）；

- 通过通用的连接器接口（generic connector interface）对资源进行操作；
- 对资源的各种操作不会改变资源标识；
- 所有的操作都是无状态的（stateless）。

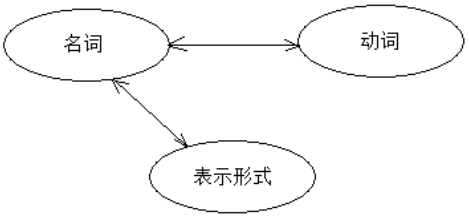
对于当今最常见的网络应用来说：

- “唯一的资源标识”就是 URL；
- “通用的连接器接口”是 HTTP；
- “对资源的各种操作不会改变资源标识”就是我们常说的 URL 不变性。

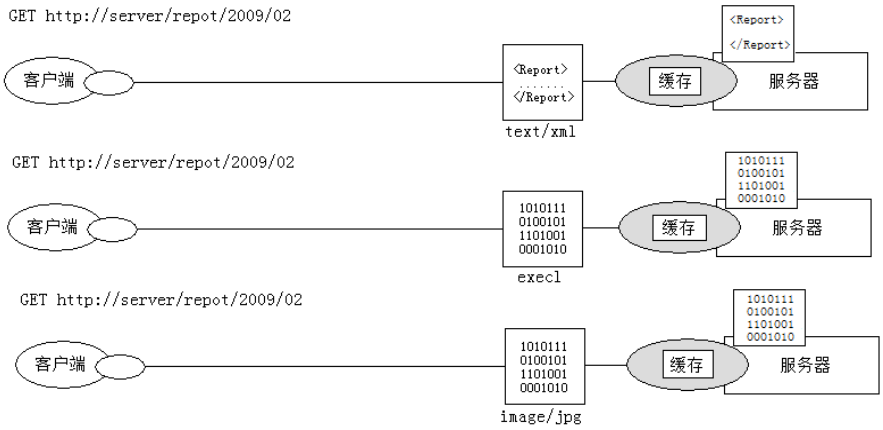
这些概念中的“资源（resouce）”是最容易使人产生误解的。resouce 所指的并不是数据，而是“数据+特定的表现形式（representation）”，这也就是为什么 REST 的全名是 Representational State Transfer 的原因。举个例子来说，“本月卖得最好的 10 本书”和“你最喜欢的 10 本书”在数据上可能有重叠（有一本书即卖得好，你又喜欢），甚至完全相同。但是它们的表现形式不同，因此是不同的资源。

3）REST 的关注点分离基本结构

REST 的一个很大的特点是关注点分离，它的基本结构如下图所示。



通过分离事务的名称和表现形式，我们就可以用同样的名称来获取不同的表现形式。



REST 之所以能够简化开发，是因为它所引入的架构约束，遵循了 CRUD 原则。CRUD 原则认为，对于资源只需要四种最基本的行为：Create（创建）、Read（读取）、Update（更新）和 Delete（删除）。这四个操作是一种原子操作，也就是一种无法再分的操作，通过它们可以构造更复杂的操作过程。

更进一步讲，REST 架构让人们真正理解 HTTP 网络协议的本来面貌，如果使用 HTTP 作为通用的连接器接口，把一个 URL 的操作限制在了 4 个之内：GET、POST、PUT 和 DELETE。这正好对应于资源的四个原子操作：获取、创建、修改和删除（CRUD）。这种针对资源的网络应用设计和开发方式，可以降低开发的复杂性，提高系统的可伸缩性。也使得资源的调用更加简洁与易于理解。

4）REST 与 SOA 的区别

应该注意，我们不能混淆 REST 与 SOAP 的区别，REST 是关于信息管理的，而不一定是通过 URL 来调用任意的行为，如果我们在设计的时候，冥思苦想动词的数量是不是够用?

那也许我们想的并不是调用数据而是调用行为，这种情况可以使用 SOAP。如果需要把重要的业务概念变成可以寻址的信息资源，并且在不同的环境下需要有不同的表现形式，那么我们就可以利用 REST 的长处。

REST 方案在后台可以使用 SOAP 来实现请求，但是对外利用 REST 接口带来的好处，使得上层业务的开发者不需要过多地考虑数据的物理位置、物理形式、甚至它们的格式（是 Oracle 还是 Sql Server？），同时还可以使用缓存，同一个数据可以用不同的方法表现，这就给系统带来了可扩展性和可伸缩性。

5）REST 的无状态性

Acura 项目概念架构中，可伸缩性是一个重要的要求。为了保证 Acura 项目数据资源的可伸缩型，我们在数据总线的设计中采用了 REST 风格。在 REST 之所以能够提高系统的可伸缩性，是因为它的规则强制所有操作都是无状态的，这样就没有上下文的约束。这样一来，在我们强调要做分布式、做集群的时候，就不需要考虑上下文的问题了。同时，它也可以令系统可以有效地使用“池”。REST 对性能的另一个提升来自其对客户和服务器任务的分配：服务器只负责提供资源，以及操作资源的服务，而客户要根据资源中的数据和表示法。自己做业务应用。这就减少了服务器的开销。

REST 风格应用可以实现交互，并且天然地具有服务器无状态的特征。在状态迁移的过程中，服务器不需要记录任何 Session，所有的状态都通过 URL 的形式记录在了客户端。换句话说，这里的无状态服务器，是指的服务器并不保存会话状态（Session），而资源本身则是天然的状态，通常是需要被保存的。

如果我们按 SOA 的服务思维，很容易想到在服务器端保存 Session，每次选择以后修改 Session，根据 Session 产生相应的结果。但如果以 REST 的状态表述转移模型为指导，我们会自然地得出这样设计：

这样一来，每一个资源表示一个状态（存在于客户端），资源包含了到下一个资源的超链接，每当用户选择转移到下一个相应的状态的时候，所有的会话状态其实都是通过 URL 的形式保存在了客户端，在服务器端则实现了无状态。

与有状态服务设计相比，无状态服务更容易实现系统性能的横向扩展。便于通过增加硬件，部署多个无状态服务，来提升系统性能，这与 Acura 项目设计目标是一致的。而在有状态服务模式中，Session 的存储、共享都会带来必须解决的难题，也给增加并发通道和服务器带来了难度。换句话说，有状态服务的性能一般无法通过增加硬件来提升。

6）新的架构设计思维方式

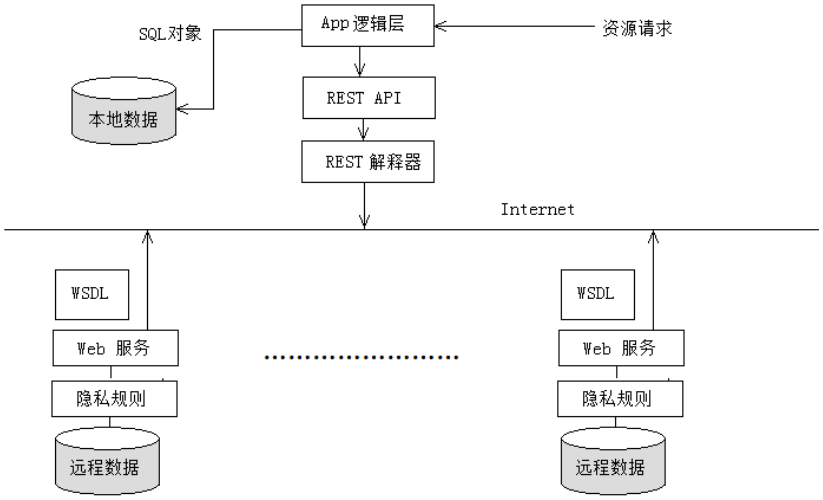
REST 除了给我们带来了一种崭新的架构以外，还有一个重要的贡献是在开发系统过程中的一种新的思维方式：也就是通过 URL 来设计系统的结构。根据 REST 的定义，每个 URL 都代表一个资源，而整个系统就是由这些资源组成的。因此，如果 URL 是设计良好的，那么系统的结构就也应该是设计良好的。

对于大量普通开发人员来说，考虑一个系统如何架构总是一个很抽象的问题。在 REST 架构中通过 URL 设计系统结构。虽然我们连一个功能都没有实现，但是我们可以先设计出我们认为合理的 URL，这些 URL 甚至不能连接到任何 page 或 action，但是它们直观地告诉

我们：系统对用户的访问接口就应该是这样（这个思路很象测试驱动的开发）。根据这些URL，我们可以很方便地设计系统的结构。如果每个用户需求都可以抽象为资源，那么就可以完全使用 REST。

由此看来，使用 REST 的关键是如何抽象资源，抽象得越精确，对 REST 的应用就越好。如果有朝一日我们可以把所有的用户需求都抽象为资源，那么可以预期，现时的 MVC 架构就可以退出历史的舞台了。当然，现时我们还无法找到一种可以把所有的用户需求都抽象为资源的方法，因此在 Acura 项目中，只是在数据总线的设计中采用了 REST 风格，这样也减轻了大部分程序员的负担。为了保证可实现性，在后台具体技术的应用上，我们还是采用了 XML/Web Service。

数据总线的概念架构如下图所示。



小结：

Acura 项目在架构设计的时候展现了一些创新方法。由于基本的目标非常清晰，所以架构展现了一种独特的风格。

- 由于对业务可扩展性的追求，我们使用了面向服务的架构；
- 由于对目标可伸缩性的追求，我们使用了分布式体系和并发通道；
- 由于对分布式数据一致性的追求，我们使用了面向资源的架构。

这样一个可伸缩性的系统需要考虑程序员、用户都处在广大的分布式空间，架构追求让开发者只遵守少量规则，就可以在这个平台上编写大规模多人交互的软件。

具体做法是试图创建一个基础架构，使它具备企业级系统的可靠性，同时又满足在这个环境下对于延迟、通信和伸缩性的要求。它通过更多的机器和更多的线程来实现效率（而不仅仅依靠 CPU 速度和内存），从而希望抵消因使用持久存储机制导致的延迟增加。

在这个系统中由于客户处理任务的复杂性，力争更多的处理任务放到客户端，而服务器端的处理将很少。这与我们常见的廋客户企业级系统形成鲜明对比，对于实时设计的环境，这个概念设计是有效的。

优秀的架构往往表现在关注点的合理分离与结合，让这些关注点尽可能分离，然后用简单的机制结合在一起，从而得到高内聚、低耦合的系统。每个好的架构都有一个考虑重心。关注点不是一成不变的，系统演化过程中，架构师会不断面对新的关注点需求，这就需要发展演进式架构，目前只考虑当前关注点，有些关注点会推迟到将来考虑，这就需要对架构的可扩展性作为一个关注点来研究，并提出明确的目标和概念，因为只有变化是永恒不变的。