第二部分

企业级应用架构

	原则与特性		结构
√	功能多样性		模块
√	概念完整性	√	依赖关系
	修改独立性		进程
	自动传播	√	数据访问
	可构建性		
√	增长适应性		
	熵增抵抗力		

第 3 章

伸缩性架构设计

Jim Waldo

3.1 简介

在设计系统架构时，一个比较有趣的问题就是确保系统在伸缩时的弹性。随着越来越多的系统运行在网络上或在互联网上提供访问，伸缩性正变得越来越重要。对于这样的系统，如果你希望误差的范围在几个数量级以内，那么容量规划的想法显然是荒谬的。如果你架起一个网站，然后它火了，你可能会突然发现有几百万的用户访问你的站点。同样容易出现的情况是，你架起了一个网站，却发现没有人感兴趣，你投入的所有设备都闲置着，消耗着能源和管理成本，浪费钱财（这同样是一种灾难）。在网络世界里，一个站点可以在几分钟内从其中一种状态转变成另一种状态。

只要是将系统连接到网络上，每个人都会遇到伸缩性问题，但是“大型多人在线游戏”（MMO）和虚拟世界特别关注这一点。这些系统必须具备伸缩性，以满足大量的用户。Web服务器的用户常常读取的是静态的内容，而且彼此之间没有交互，但MMO中的玩家或虚拟世界中的居民则不同，他们既需要与所处的世界进行交互（这改变了世界的基本信息），也需要彼此之间的交互。这些交互行为使得这类系统基础设施的伸缩性问题变得更复杂，因为用户与系统的交互几乎是独立的（除了那些不独立的情况），而且不会让世界的状态改变太多。对于一个世界里的任意两个参与者，他们在某个时刻进行交互的几率是非常小的。但是，几乎所有玩家在所有时候都在与他人交互。结果是这种系统并行程度非常高，但只有少数的交互是互相依赖的。

由于这些系统所培育起来的文化，MMO和虚拟世界的伸缩性问题进一步复杂化了。MMO和虚拟世界都源自于视频游戏产品。这是一种从PC游戏和游戏机游戏传统中成长起来的文化，在这种传统中，程序员会假定游戏运行在一台独立的机器或游戏机上。在这样的环境中，机器所有的资源都受游戏程序控制，程序的问题也限于单个用户玩游戏的情况（实际上，缺陷或奇怪的行为常常会被认为是游戏本身逻辑的一部分）。

这些游戏和编写、出品、扩展它们的公司，都属于娱乐行业。编写游戏的团队由一个出品人领导，有剧本和背景故事。游戏的目标是刺激、打动人，最重要的一点，要好玩。可靠性很好，但不一定必需。可扩展性是游戏的特性，让游戏在升级时能够加入新的故事情节和主题，但可扩展性不是代码的特性，不必让代码能以新的、不同的方式使用。

在线游戏和虚拟世界的兴起将这种文化带入了一个新的环境，在这个环境中，需求与企业应用开发者所面对的类似。多个用户通过网络在服务器上交互时，由于一个玩家的意外动作而导致的服务器崩溃将影响许多其他玩家。随着这些世界发展出自己的经济（有些经济与现实世界的经济有关系），在线世界的稳定性和一致性就超出了一个游戏的要求。随着这些世界中玩家或居民的人数达到百万级，伸缩的能力就成为了任何架构的首要需求。

Darkstar项目（本章后面将简称为Darkstar）是对这些游戏和虚拟世界创建者的需求挑战的回答。这个项目由Sun公司实验室的一个研究小组承担，它将在架构的伸缩性领域不断探索。这个项目特别有趣之处在于，它是针对MMO和虚拟世界的创建者，这些程序员与我们（作为系统设计者）所熟悉的那些程序员相比，有着非常不同的需求。得到的架构看起来似乎很眼熟，但如果你仔细查看，会发现它的不同之处，它与你的经验有所不同。得到的架构有着属于它自己的美丽，同时它也是一堂实践课，说明了不同的需求如何改变你所想到的构建系统的方式。

3.2 背景

像一座建筑或一个城市的物理架构一样，系统的架构必须适应环境，利用该架构创建的工件将存在于该环境之中。对于物理架构来说，这个环境包括工作的历史环境、它所处位置的气候、本地工人的技能、可以获得的建筑材料，以及建筑的使用意图。对于软件架构，这个环境不仅包括使用该架构的应用程序，也包括那些要使用该架构的程序员，以及由此受到的系统约束。

在创建Darkstar架构时，我们（注1）意识到的第一件事就是所有针对伸缩性而设计的架构都需要包含多台机器。我们不清楚，就算是最大的大主机系统是否能够满足今天的一

注1： 在提到Darkstar项目的架构开发时，我通常会说“我们”做了什么，而不是“我”做了什么。这不仅是编辑意义上的“我们”。该架构的设计是一个协作项目，由Jeffrey Kesselman、Seth Proctor和James Megquier发起，并经过Seth、James、Tim Blackman、Ann Wollrath、Jane Loizeaux和我的努力发展到现在的状态。

些在线游戏的要求（例如《魔兽世界》，据报道它有500万注册用户，几十万的同时在线用户）。就算单台机器能够处理这种负载，我们也不能在一开始就假定游戏会取得如此成功，需要这样的硬件投资。这在经济上是不可行的。这种应用需要能够从很小的系统开始，然后随着用户数的增长而增加处理能力，最后随着大家对游戏兴趣的衰退而降低处理能力。这与分布系统的特点相符，在分布式系统中，我们可以随着请求增长而添加（合理的小）机器，当请求下降时移走机器。所以我们从一开始就知道，总体架构必须是一个分布式系统。

我们也知道系统利用了芯片架构的当前趋势。MMO和虚拟世界（在较小程度上）曾经针对伸缩性利用过摩尔定律。随着处理器的速度倍增，可以创造的世界会在复杂度、丰富程度和互动性方面倍增。没有其他计算领域像游戏世界这样探索过处理器速度的增长所带来的好处。为游戏而设计的个人计算机总是将CPU速度、内存和图形性能推向极致。游戏机更激进地将这些方面推向极致，它们包含的图形系统远远超过了高端工作站上的图形系统，整个机器完全是为了游戏玩家的特殊需要而打造的。

芯片演进方面最近的变化是从不断增加的时钟速度转为实现多核处理器，这已经对游戏中能做的事情产生了影响。新芯片的设计目标不是将一件事做得更快，而是同时做多件事。如果在芯片上运行的多项任务实际上可以同时执行，那么在芯片层面上引入并发执行将带来更好的总体性能。在不改变时钟速度的情况下，4核的芯片应该能比单核的芯片多做3倍的事情。实际上，这种增长不是呈线性的，因为系统的其他一些部分没有以同样的方式实现并发。但是可以通过并发来实现系统总体性能的增长，而且制造这种并发的芯片比制造增加时钟速度的芯片要容易得多。

基于这一事实，MMO和虚拟世界应该是多核芯片和分布式系统的理想候选者。在MMO或虚拟世界中发生的大多数事情就像在真实世界中发生的大多数事情一样，与该世界中发生的其他事情是无关的。玩家继续他们的搜索或装饰他们的房间。他们与怪物交战或设计衣服。即使他们与该世界中的另一个玩家或居民发生交互，也只是与该世界的很少一部分居民发生交互。这正是令人为难的并行计算任务的特点，也正是多核和多机系统应该擅长处理的那种任务。

尽管这些系统中的任务的并行度让人为难，但为这样的系统编程的程序员却没有接受过分布式计算或并发编程方面的训练，也没有这方面的经验。这是极为微妙的领域，即使是在这个领域接受过训练的人和对这些技术相当有经验的人也会感到困难。要让大多数游戏程序员来开发高度并发的、分布式游戏服务器，就是要求他们做超出自己的专长和经验的事。

3.2.1 首要目标

这样的背景为我们确立了该架构的首要目标。对伸缩性的需求表明，系统应该是分布式

的、并发的，但我们需要为游戏开发者提供简单得多的编程模型。简而言之，目标就是游戏程序员应该把该系统视为一台单机，运行着一个线程，所有允许部署到多线程和多计算机上的机制都应该由Darkstar项目的基础设施来考虑。

在一般的情况下，对应用程序隐藏分布式和并发是不可能的。但MMO和虚拟世界不是一般的情况。我们试图实现的这种隐藏，其代价就是必需一种非常特别的、严格限制的编程模型。幸运的是，这种模型恰好非常适合游戏服务器和虚拟世界已经采用的编程方式。

Darkstar项目要求的一般编程模型是反应式的，在这种编程模型中，游戏的服务器端写成了事件监听器，监听客户端生成的事件（客户端就是游戏玩家使用的机器，通常是一台PC或游戏机）。如果检测到事件，游戏服务器就应该生成一项任务，这个任务是一个短期的计算序列，包括操作虚拟世界中的信息，并与最初生成事件的客户端或其他一些客户端进行通信。任务也可以由游戏服务器自己生成，要么是响应某些内部的变化，要么是周期性地根据时间来生成任务。在这种情况下，游戏服务器可以在游戏或虚拟世界中生成一些角色，这些角色是不受外部玩家控制的。

这种编程模型非常适合游戏和虚拟世界，但它也应用于一些企业级的架构中，如J2EE和Web服务。之所以需要创建一个不同于这些企业计算机制的架构，是因为MMO和虚拟世界存在的环境非常不一样。这种环境几乎刚好和经典企业环境相反，这意味着如果你接受过企业环境方面的训练，你知道的所有事情在这个新世界中几乎都是错的。

经典的企业环境可以描述为一个“瘦”客户端连接到一个“胖”服务器（这个服务器又常常连接到一个更“胖”的数据库服务器）。服务器将保存客户端需要的绝大部分信息，在最理想的情况下，客户端内存不多，没有自己的硬盘，它是服务器的一个称职的显示设备，绝大多数真正的工作在服务器上完成。

3.2.2 游戏世界

MMO和虚拟世界的环境始于一个非常胖的客户端：它通常是顶级的PC、具有最强劲的CPU、很大的内存，以及本身计算能力就很强的显卡。它也可以是一台游戏机，专门为图形密集的、高度交互的任务而设计。只要有可能，数据就会存放在这些客户端，特别是那些不会改变的信息，如地理信息、材质贴图和规则集。服务器保持尽可能的简单，通常只保存非常抽象的世界表示和其世界中的实体的表示。而且，服务器的设计目标是尽可能少地进行计算。绝大部分的计算留给了客户端。服务器的真正工作是保存共享的世界真实状态，确保不同客户端对世界的看法差异可以根据需要得到纠正。真实状态需要由服务器来保存，因为控制客户端的玩家很有兴趣让他们的表现变成最强，所以可能会受到诱惑，根据他们的喜好修改共享的真实（如果他们可以）。在一般情况下，如果有可能，玩家就会作弊，所以服务器必须是共享真实的最终来源。

MMO和虚拟世界的数据访问模式也和企业中看到的情况有着很大的区别。企业中的一般经验法则是90%的数据访问都是只读的，大多数任务会读取大量数据，然后再改写少量数据。在MMO和虚拟世界的环境中，大多数任务只访问服务器上少量的状态数据，但在它们访问的数据中，大约一半会被改写。

3.2.3 延迟是敌人

但是，这两种环境中最大的不同要追溯到用户所做的事情的不同。在企业环境中，目标是管理业务，如果总吞吐量得到改进，在处理中有一点延迟是可以接受的。在MMO和虚拟世界的环境中，目标是开心，而延迟是开心的敌人。所以MMO或虚拟世界的基础设施需要围绕着尽可能限定延迟的需求来设计，即便以吞吐量为代价也在所不惜。

在线游戏和虚拟世界显然已经找到了办法来实现伸缩性，以应对数量巨大的用户。目前的方法可以分成两大类。第一类实质上是基于地理位置来实现的。游戏设计成包含一组不同的区域，每个区域运行在一台服务器上。它可能是虚拟世界的一个岛或房间，也可能是在线游戏中的一个小镇或山谷。游戏设计试图让每个区域无关，限定地理区域的大小，这样服务器不会因太多用户进入这个区域而拥塞。在实践中，这样的区域常常能实现自我限制，因为当服务拥塞时，游戏就会变得响应比较慢，趣味性下降。因此，玩家就会转向更有趣的区域，这使得以前拥塞的区域人数减少，响应时间得到改进。

将不同地理区域分配给不同服务器来实现伸缩性的方法有一个问题，即必须在游戏编写时决定哪些区域应该放到一台服务器上。虽然在游戏或虚拟世界中添加新的区域相当容易，但是改变已经分配给服务器的区域却可能需要改动代码。决定哪些区域组成一个伸缩性单位，这必须是开发工作的一部分。

第二种处理游戏或虚拟世界中拥塞区域的方法叫做分区（sharding）。一个分区是该区域的一份副本，运行在它自己的服务器上，独立于其他的分区，它代表了游戏中相同的部分，即原来的区域。这样，分区可能代表了某个房间或村庄的不同副本，允许成倍的玩家进入到世界的这个部分中。分区的缺点是它们不允许处于不同分区的玩家彼此之间进行交互。随着游戏和虚拟世界变成更多的社交体验，而不仅仅是玩游戏，这种缺点就明显了。玩家的目标不只是要出现在虚拟世界中，而是要和他们的朋友（真实的和虚拟的）一起进入虚拟世界。分区阻碍了这个目标的实现。

因此，Darkstar架构的另一个主要目标就是支持随时伸缩，同时不要求游戏逻辑受到伸缩的影响。这个架构应该支持游戏动态地响应负载，而不是让这种响应成为游戏设计工作的一部分。

3.3 架构

Darkstar由一组独立的服务构成，这些服务可以在游戏或虚拟世界的服务器端的地址空间内获得。每个服务都定义为一个小的编程接口。尽管不是出于本意，但Darkstar项目提供的一些基本服务很像经典操作系统的服务，它们支持游戏或虚拟世界的服务器端访问持久存储，调度并执行任务，与游戏或虚拟世界的客户端进行通信。

用一组相互联系的服务来构建这个系统，显然是开始了“分而治之”的过程，分而治之是设计所有大型计算机系统的基本方法。每种服务都可以用一个接口来描述，这让使用该服务的程序不会受到底层实现变更的影响，同时也支持这些实现可以独立地完成。对一个服务的实现进行变更不应该影响到另一个服务的实现，即使其他的服务会利用到这个变更的服务（假定接口和接口的语义没有变更）。

我们采用服务分解的方法还有其他的原因。从一开始，Darkstar项目就设计成一个开放源代码的项目，希望我们能够放大核心团队的工作，支持其他社区成员创建更多的服务，丰富核心的功能。维护一个开源社区在任何情况下都是复杂的，我们相信在组成架构的服务之间拥有最大程度的隔离，将支持在不同服务实现层次之间的更高级的隔离。此外，当时并不清楚是否存在单一一组服务能够适合所有的MMO和虚拟世界。将基础设施设计为一组独立的服务，使得这些服务的不同组合可以在不同的情况下使用，这由使用该基础设施的具体项目的需求来决定。Darkstar栈中具体包含哪些服务可以由一个配置文件来设置。

3.3.1 宏观结构

图3-1展示了基于Darkstar项目基础设施的游戏或虚拟世界的基本结构。一些服务器构成了游戏或虚拟世界的后端。每个服务器运行着一组选定服务的副本（称为Darkstar栈）和游戏逻辑的副本。客户端将连接到其中一个服务器，与服务器保存的该世界的抽象表示进行交互。

与大多数的复制策略不同，游戏逻辑的不同副本不需要处理相同的事件。每个副本可以独立地与客户端进行交互。在这个设计中，复制主要用于支持伸缩性，而不是确保容错（虽然我们后面会看到，容错也实现了）。而且，游戏逻辑本身不知道、也不需要知道在其他机器上运行着服务器的其他副本。游戏程序员编写的代码就像在一台机器上执行一样，不同副本之间的协作由Darkstar项目的基础设施来完成。实际上，如果游戏的容量只需要一台服务器，基于Darkstar的游戏就能够在一台服务器上运行。

客户端连接到游戏逻辑使用的通信机制是基础设施的一部分。这些机制支持客户端到服务器的直接通信，也支持一种“发布 - 订阅”通道，任何发往通道的消息都会送达该通道的所有订阅者。

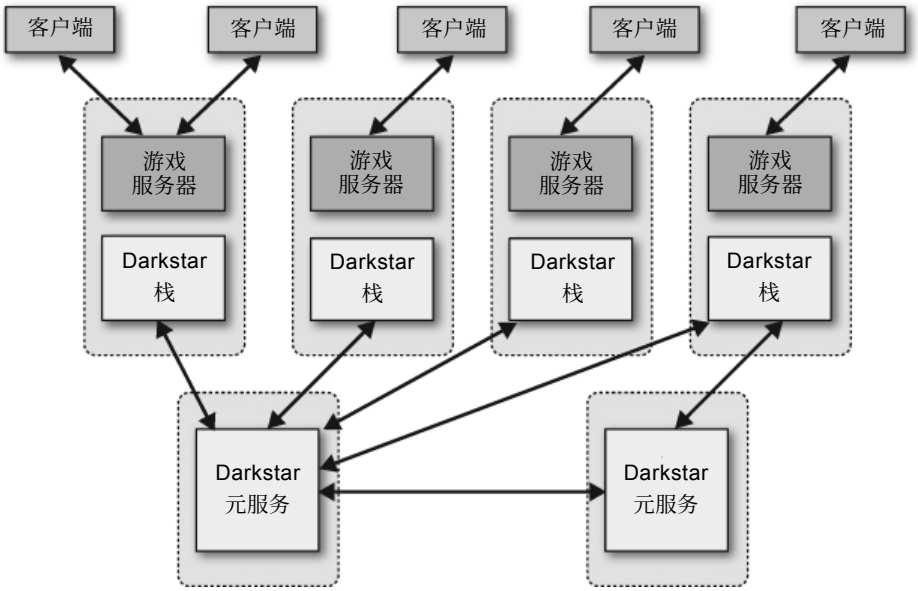

图3-1：Darkstar项目的高层架构

Darkstar栈由一组元服务来协调，这是一组网络访问服务，游戏或虚拟世界的程序员是不可见的。这些元服务支持栈的各个副本之间进行协作，共同运营整个游戏。例如，这些元服务将所有独立的副本持续工作，如果某个副本失效，就会发起失效恢复。这些元服务还会跟踪各副本的负载，在需要的时候重新分配负载，或者随时添加新的服务器，增加总体容量。由于这些服务对于Darkstar项目的用户来说是完全隐藏的，所以它们可以随时改变或移除，或者添加新的服务，这都不需要修改游戏或虚拟世界的代码。

对于在Darkstar项目环境中创建游戏或虚拟世界的程序员来说，可见的架构就是栈中包含的一组服务。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，它们构成了运营环境的核心，如图3-2所示。

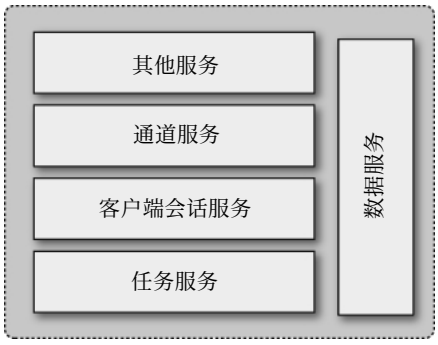

图3-2：Darkstar栈

3.3.2 基本服务

在这些栈层面的服务中，最基本的服务就是“数据服务”（Data Service），游戏或虚拟世界用它来保存、读取和操作所有持久数据。这里的持久概念可能比其他系统中的持久概念更宽泛。对于在Darkstar项目环境中编写的游戏或虚拟世界，任何存在时间超过一个任务的数据都被视为持久的，必须在“数据服务”中保存。我们曾假定在这种编程模型中任务的时间是短暂的（这也是需求），所以几乎所有用于表示游戏或虚拟世界的服务器端的数据都需要持久。“数据服务”也将运行在不同服务器上的游戏或虚拟世界的副本联系在一起，因为所有这些副本都共享同一个（概念上的）“数据服务”实例。所有的副本都会访问相同的数据，所有的副本都可以根据需要读取或改变存储在“数据服务”中的数据。

虽然“数据服务”看起来像是使用一个数据库的好地方，但是存储的需求实际上与通常条件下对标准数据库的需求有着很大的差别。存储的对象之间静态的关系很少，游戏中也不需要对存储的内容进行复杂的查询。相反，简单的命名策略就足够了，包括在编程语言层面上对对象的引用。“数据服务”也必须针对延迟进行优化，而不是针对吞吐量来优化。特定服务要访问的对象个数可能很少（我们初步的测算基于一些游戏和虚拟世界的原型，这些测算表明每个任务大约访问一打对象），在这些访问的对象中，大约一半会在任务执行中改变。

第二个栈层面的服务是“任务服务”（Task Service），它用于调度或执行任务。这些任务要么是响应从客户端收到的某个事件，要么是由游戏或虚拟世界服务器本身的内部逻辑发起的。绝大部分任务是一次性事件，是由于客户端的某种动作产生的，它们从“数据服务”中读取一些数据，操作这些数据，可能还进行一些通信，然后结束。任务也可能生成其他的任务，或者生成定期任务，在特定的时间执行或以特定的时间间隔执行。所有任务的执行时间必须很短，执行一项任务的最大时间是一个可配置的值，但默认值是100毫秒。

游戏或虚拟世界的程序员会看到因事件或服务器逻辑本身而生成的单个任务，但在底层，Darkstar的基础设施正尽其所能调度最多的任务。特别地，由服务器逻辑生成的任务与响应客户发起的事件而生成的任务是并行执行的，就像响应不同客户端而生成的任务一样。

这样的并发执行可能导致数据竞争。要处理这种竞争，就需要“任务服务”和“数据服务”协作。在底层，在服务器程序员不可见的地方，“任务服务”调度的每个任务都包装在一个事务中。这个事务确保了任务中的所有操作要么全部完成，要么都不完成。此外，所有改变“数据服务”中对象的值的操作都由该服务作为中介。如果有多个任务试图改变相同的数据对象，只有一个任务会执行，其他任务都会中止，并安排在稍后执行。执行的那个任务会运行到结束。当执行的任务结束时，其他的任务就可以执行了。虽然

服务器程序员可以说明访问的数据将被修改，但这不是必需的。如果数据对象先被读取，然后被修改，“数据服务”会在任务提交之前检测到这种修改。在读取时就说明打算进行修改，这是一种优化，能够更早地检测到冲突，但是不事先说明修改的意图也不会影响程序的正确性。

将任务包装到一个事务中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时，消息才会发出。这是通过Darkstar栈中余下两项核心服务来完成的。

3.3.3 通信服务

第一个服务是“会话服务”（Session Service），它是客户端和游戏或虚拟世界服务器之间通信的中介。在登录和认证后，客户端与服务器之间就会建立起一个会话。服务器通过会话监听客户端发出的消息，解析消息的内容，确定生成怎样的任务来响应该消息。客户端通过会话接收来自服务器的响应。这些会话隐藏了客户端和服务器的真实端点，这一点对于Darkstar的多机伸缩性策略是很重要的。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所引发的任务还没有完成，后一条消息就不会提交。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。对来自特定客户端的消息排序是Darkstar框架中唯一的消息排序保证机制，外部观察者看到的多个客户端之间的消息顺序，与游戏或虚拟世界内看到的顺序有很大不同。

Darkstar栈中总可以得到的第二种通信服务是“通道服务”（Channel Service）。通道是一种一对多的通信机制。在概念上，通道可以由任何数目的客户端加入，任何发往该通道的消息都会送达所有与通道相关的客户端。这里似乎是应用端到端技术的好地方，可以让客户端之间直接通信，不会增加对服务器的负载。但是，这种通信需要由一些受信任的代码来监控，确保玩家不会利用不同的客户端实现来发送不正确的消息或欺骗消息。既然客户端假定是在用户或玩家的控制之下，那么客户端的代码就不能信任，因为很容易把原来的客户端代码换成另外的“定制过的”客户端代码。所以，实际上，所有通道消息都必须经过服务器，（可能）在经过服务器逻辑检查之后。

会话和通道的复杂性有多种原因，其中之一就是它们必须遵守任务的事务语义。因此，会话连接或通道上的实际消息传送不能够在调用相应的`send()`方法时发生，它只能够在该方法所处的任务提交时才能发生。

这些通信机制为我们实现伸缩性机制的第二部分奠定了基础。既然所有通信都必须通过Darkstar会话或通道的抽象层，而这些抽象层又不暴露客户端或服务器通信的真实端点，那么在实体通信和通信起止端的实际位置之间就存在着一个抽象层。这意味着我们可以在Darkstar系统中将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

端对这次通信的感觉。从游戏或虚拟世界的视角来看，通信也是经过一个会话或通道。但是底层的基础设施可以随着时间的推移和负载的变化，根据负载平衡的需要，将会话或通道从一台机器移到另一台机器。

3.3.4 任务的可移动性

要实现负载均衡的能力，其关键之处在于，对于我们要求的编程模型和必须使用的基本栈服务，响应客户端事件或游戏内部事件的任务可以从任何一台运行着Darkstar栈和游戏或虚拟世界副本的机器上移动到另一台同样的机器上。任务本身是用Java编写的（注2），这意味着只要（物理）机器的运行时栈中包含相同的Java虚拟机，任务就能够运行。任务读取和操作的所有数据必须从“数据服务”获得，“数据服务”是所有机器上的游戏或虚拟世界的实例和Darkstar栈所共享的。通信由“会话服务”或“通道”来实现中介，它们抽象了通信的真实端点，而且支持特定的会话和通道从一个服务器移动到另一个服务器上。因此，所有任务都可以运行在任意一个游戏服务器的实例上，同时不改变任务的语义。

这使得Darkstar的基本伸缩机制看起来很简单。如果有一台机器超载了，只要将一些任务从这台机器移到另一台负载较小的机器就行了。如果所有的机器都超载了，就向运行着Darkstar栈和游戏/虚拟世界的集群中添加新的机器。底层的负载平衡软件会将负载分发给新的机器。

对单台机器的负载进行监控并在需要时重新分配负载，这是元服务的工作。这些元服务是网络层面的服务，对于游戏或虚拟世界的程序员是不可见的，但是它们对Darkstar栈中的服务是相互可见的。例如，这些元服务会监控哪些机器正在运行（并监控是否有机器失效），哪些用户与某台机器有关，不同的机器当前的负载。由于元服务对于游戏或虚拟世界的程序员是不可见的，所以它们在任何时候都可以改变，不会影响到游戏逻辑的正确性。这让我们能够尝试不同的策略和方法，实现系统的动态负载平衡，也让我们能够丰富基础设施所需的元服务集合。

同样，我们使用实现多机伸缩机制来实现系统的高容错。由于任务和通信机制所使用的数据是与具体机器无关的，所以很明显，我们可以将任务从一台机器移到另一台机器上。但是如果机器失效，我们如何恢复在那台机器上执行的任务呢？答案很简单：任务本身也是持久对象，保存在系统的“数据服务”中。因此，如果一台机器失效，该机器上正在执行的所有任务都被视为中断的事务，会重新调度在不同的机器上执行。尽管这种重

注2： 更准确地说，所有的任务都由字节码的序列组成，可以在Java虚拟机上执行。我们不关心源语言是什么，我们只关心从源语言编译出来的结果可以运行在所有支持游戏或虚拟世界的分布式环境中。

新调度比在一台机器上重新调度中断的任务的延迟要长，但系统的正确性是不变的。系统的用户（游戏玩家或虚拟世界的居民）顶多会注意到响应时间暂时有点延长。这样的延迟让人有点不舒服，但总好过现在其他游戏或虚拟世界环境中服务器崩溃所造成的影响。在那些环境中，会导致玩家掉线，还可能导致相当一部分游戏状态的丢失。

3.4 关于架构的思考

也许所有人关于架构及其实现的第一个问题就是它的性能。虽然未经深思熟虑就对架构进行优化是诸多罪恶之源，但是我们也可能设计出一个架构，而它的实现根本不可能达到好的性能。由于Darkstar架构的一项基本选择，这种担心是真实的。而且由于游戏行业的特点，确定服务器基础设施的性能是很难的。

要确定游戏或虚拟世界服务器基础设施的性能，其难度源自一个简单的事实：没有针对大规模MMO或虚拟世界的性能测试标准或共同接受的例子。缺少性能测试标准并不奇怪，因为绝大多数游戏或虚拟世界的服务器组件都是针对特定的游戏或虚拟世界从头开发的。只有少数的通用基础设施可以作为可复用的构建块，这些组件一般是事后从特定的游戏或虚拟世界中提取出来的，提供给其他构建类似游戏的人使用。也许是因为游戏行业还比较年轻，或是因为娱乐业中出现重要技术的偶然性，结果是没有共同接受的性能测试标准用于测试新的基础设施，也无法对不同的基础设施进行比较。

关于游戏或虚拟世界的预期计算、数据操作和通信负载也基本上没有什么资料，所以很难创建性能测试标准程序。部分原因是已有的服务器都是定制的。每台服务器都是为特定的游戏或虚拟世界设计的，所以考虑的是那款游戏或虚拟世界的具体负载特征。而且，这种状况也是因为游戏业的强烈的保密性。在游戏业中，关于一款游戏开发的信息是严格保密的，而且发布游戏的实现方式也是严格保密的。同时，行业中的许多人对这些信息是不感兴趣的。大量的思考和讨论集中在艺术设计、故事情节或玩家交互模式上，这些令新游戏更有趣、更好玩，而不是游戏服务器的设计方式和游戏为支持并发玩家（这个数字也是严格保密的）所采取的伸缩性技术。所以，获得现有的游戏和虚拟世界的这种服务器负载信息都很困难。

根据我们的经验，即使我们能够找来开发者，谈论他们的游戏或虚拟世界加在服务器上的负载，他们也常常会报告错误的信息。这不是因为他们想保持商业优势而故意错误报告他们服务器的情况，而是因为他们真的自己不也了解。在游戏服务器上基本不会加入一些手段，让他们收集有关服务器真实性能或完成事务的信息。对于这种服务器的分析一般最多是经验性的。程序员在服务器上工作，直到它让游戏玩起来有趣，这是一种迭代式的工作方式，而不是仔细对代码本身进行测量。在这些系统中，更多的是手工技术活，而不是科学测定。

这并不是说，这些游戏或虚拟世界后面的服务器是一些粗制滥造的代码，也不是说它们做得不好。实际上，许多代码是效率杰作，展示了聪明的编程技巧，也展示了针对高要求的应用构造一次性、专门目的服务器的优势。但是，为每个游戏或虚拟世界构建一个新服务器的习惯意味着人们不太注意积累构建这种服务器所需的知识，也没有共同接受的机制来比较不同的基础设施。

3.4.1 并行与延迟

缺少有关服务器可接受的性能的资料，这一点引起了Darkstar团队的特别关注，因为我们所做的一些关键决定，都是围绕着如何能够从游戏或虚拟世界服务器中获得好的性能。也许Darkstar架构和一般实践之间的最大区别就在于，Darkstar架构拒绝在服务器的主内存中存放任何重要的信息。所有生存周期超过一次具体任务的数据都需要持久在“数据服务”中，这是实现Darkstar基础设施功能的核心。这让基础设施能够检测到并发问题，反过来又让系统能够对程序员隐藏这些问题，同时让服务器能够利用多核架构。它也是实现整体伸缩性的关键组件，支持任务从一台服务器移动到另一台服务器，从而在一组机器上实现负载平衡。

在游戏和虚拟世界服务器领域，任何时候都持久保存游戏状态是一种异端邪说，因为人们普遍很关心延迟。在编写这种服务器时，大家的观点是只有将所有信息都放在内存中才能让延迟足够小，达到要求的响应时间。可以偶尔保存状态的快照，但对交互速度的要求表明，这种长时间的操作只能够偶尔进行，而且要在后台进行。所以，从表面上看，我们的架构似乎绝不可能达到足够好的性能，从而服务于它的目标应用。

虽然要求数据持久肯定是这个架构的主要不同之处，而且要求通过“数据服务”来访问数据会在架构中引入一定的延迟，但我们相信我们所采取的方式更具有竞争力，原因有几点。首先，我们相信能够让访问内存数据和访问“数据服务”中的数据之间的差异远远小于一般人们的看法。虽然在概念上每个生命周期超出一次任务的对象都需要从持久存储中读出，并写入持久存储，但实现这种存储可以利用人们在数据库缓存和一致性方面多年的研究成果，从而减少因这种方式而导致的数据访问延迟。

如果我们能够将访问局限在一个特定服务器上的几组特定对象，就更是如此了。如果用到一组特定对象的那些任务都运行在一台服务器上，那么就可以利用该服务器的缓存，达到接近内存的对象读写速度（受到需要满足的持久性约束的影响）。我们可以识别任务属于哪些玩家或虚拟世界的哪些用户。这样，我们就可以利用基础设施中服务所接收到的数据访问和通信请求，来收集特定时刻游戏或虚拟世界中数据访问模式和通信模式的信息。有了这些信息，我们相信能够非常准确地估计哪些玩家应该与另一些玩家放在一起。因为我们可以根据需要将玩家移动到任何服务器上，所以能够根据观察到的运行

时行为，主动地将相关的玩家尽可能地放在同一台服务器上。这让我们能够运用数据库领域熟悉的标准缓存技术，尽量减少访问和保存持久信息的延迟。

这听起来非常像目前在大规模游戏和虚拟世界中为实现伸缩性而采用的地理区域分解技术。在这种技术中，服务器开发者将世界分解成一些区域，将它们指派给一些服务器，不同的区域就成为用户分区的机制。同一区域的玩家比不同区域的玩家进行交互的可能性更大，所以这种集中在一个服务器上的优势就体现出来了。不同之处在于，目前的地理区域分解是在游戏开发过程中进行的，被编入源代码，放到服务器上。而我们的位置集中基于运行时的信息，可以根据游戏中发生的实际玩法和交互模式来实现动态调整。这类似于编译时优化和运行时优化之间的区别。前一种方法试图针对程序所有可能的运行进行优化，而后一种方法试图针对当前的运行进行优化。

我们不相信我们能够消除内存访问和持久访问之间的差别，而且我们也不认为有必要这样做，最后让这种架构比使用内存的架构性能更好。要知道，通过让所有的数据持久，我们可以支持在服务器上使用多线程（从而也支持多核）。尽管我们不相信并发是完美的（即对于每个添加的核，我们都能充分利用），但我们确实相信在游戏和虚拟世界中可以使用大量的并行运算（初步的结果也证实了这种看法）。如果可使用的并行运算超过我们可能引入的延迟，那么游戏或虚拟世界的总体性能就会更好。

3.4.2 赌未来

我们对多核处理器中多线程的信心基本上是在赌处理器将来的发展方向。目前服务器的处理器提供2～32个核，我们相信将来的芯片设计将集中向更多的核发展，而不是让现有的核以更高的时钟频率运行。当我们在几年前开始这个项目时，这种赌博似乎比现在更具投机性。那时候，我们在做展示时常常说这种设计是一种“假定”的演练，说我们正尝试一种架构，如果芯片性能更好地支持多线程而不是单线程的时钟速度，这种架构将是可行的。这就是在研究实验室中进行这类项目的好处之一，可以接受设计方法中存在很高的风险，探索一个将来也许在商业上可行的领域。我们决定构建一个以多线程为中心的架构，与这个决定做出时相比，目前芯片设计的趋势让我们看得更清楚。（注3）

即使我们只能得到50%的完美并发，如果我们能把使用持久存储的延迟控制在使用内存的延迟的2～16倍，就能够在性能上持平。我们相信在并发方面以及减少访问持久状态和全内存方案之间的差异方面都可以做得更好。但是结果主要取决于构建于这个基础设施之上的应用的使用模式（我们曾提到，这一点很难发现）。

我们也不应认为减少延迟就是这个基础设施的唯一目标。通过将游戏或虚拟世界服务器端的对象全部保存在“数据服务”中，我们把因服务器失效而导致的数据丢失减到了最

注3： 再一次说明，在早期的设计决定中少许的运气是很重要的。

小。实际上，在大多数情况下，服务器失效时用户只会注意到延时有一点增加，因为任务（它们本身也是持久对象）从失效的服务器移到了另一台服务器上。没有数据会丢失。一些缓存机制可能导致丢失几秒钟的游戏成果，但即使是这样，也比在线游戏和虚拟世界目前使用的机制好得多，它们只是将偶尔进行内存快照作为主要的持久方式。在它们的基础设施中，如果服务器在不巧的时间崩溃，可能会造成数小时的游戏成果丢失。只要延迟是可以接受的，Darkstar所使用持久机制的可靠性更高，这对于在这个基础设施上构建系统的开发者和系统的用户来说都是优点。

3.4.3 简化程序员的工作

实际上，如果在支持伸缩性的同时减少延迟是服务器开发者的唯一目标，那么开发者最好的方法就是专门针对特定的游戏，编写自己的分布式和多线程基础设施。但这要求服务器开发者处理复杂的分布式和并发编程。在为速度需求而过度烦恼之前，我们应该想到Darkstar的第二个同样重要的目标，即在支持多线程、分布式游戏产品的同时，为程序员提供一个单机单线程的开发模型。

在相当大的程度上，我们已经实现了这一目标。通过将所有任务封装到事务中，并在“数据服务”中检测数据冲突，程序员就能够享受到多线程的好处，又不必在他们的代码中引入锁协议、同步和信号量。程序员不必担心如何将玩家从一台服务器移到另一台服务器，因为Darkstar为他们提供了透明的负载平衡。编程模型虽然有自己的风格和限制，但社区中的早期成员认为，这对他们开发的那种游戏和虚拟世界来说是比较自然的。

不幸的是，我们发现我们不能够向程序员隐藏所有的东西。当在Darkstar上编写的第一个游戏表现出极少的并行度（以及意料之外的糟糕性能）时，这一点就明确了。通过检查源代码，我们很快就发现了原因。游戏中的数据结构设计导致了游戏中所有的状态改变都只涉及一个对象，并由它来协调所有的工作。使用这个对象实际上使得游戏中所有动作序列化执行，这使得基础设施不能够发现并利用并发计算。

当我们发现这一点时，我们与游戏开发者进行了长时间的讨论，主题是在设计对象时需要考虑到并发访问。通过对游戏中数据对象进行审查，我们发现了一些类似的情况：数据设计方案排除了并发的可能性（并非出自设计者本意）。当这些对象重新设计之后，系统的整体性能增加了好几个数量级。

这告诉我们，不可能让使用Darkstar的开发者完全不知道系统底层的并发和分布式实质。但是，他们对系统这方面特点的知识不需要包括并发控制、锁，以及在系统的各个分布式部分之间的通信。实际上，他们只需要在设计活动中确保他们的数据对象定义能够充分利用并发。这种设计一般只需要确保对象定义是自包含的，它们的操作不需要依赖其他对象的属性。这一点对于任何系统来说，都不是不好的设计原则。

关于Darkstar架构，我们还有许多方面没有测试，或者说我们并未完全理解。虽然我们已经得到了一个系统，使得多台机器能够利用多线程运行一个游戏或虚拟世界，同时对服务器程序员（几乎）保持透明，但是我们还没有通过添加核心服务之外的其他服务，来检验该架构的能力。由于Darkstar任务的事务本质，这可能比我们开始设想的要复杂得多，但我们希望这些添加的服务不需要参与到核心服务的事务中。我们已经开始试验通过不同的方式来收集系统负载的信息和实现负载平衡。幸运的是，因为实现这种负载平衡的机制对于使用系统的程序员是完全不可见的，所以我们可以移除老的方式，引入新的方式，同时又不影响Darkstar的用户。

作为一个架构，Darkstar展示了一些创新的方法，这使它变得很有趣。它试图构造一个游戏或虚拟世界的基础设施，使其具有企业级软件一样的可靠性，同时又满足游戏行业对延迟、通信和伸缩性的要求。它是目前为数不多的这类尝试之一。通过利用更多机器和更多线程来实现效率，我们希望能够抵消因使用持久存储机制而导致的延迟增加。最后，游戏和虚拟世界环境中极为不同的情况，即客户端的处理很多而服务器端的处理很少，与我们常见的高并发、分布式系统环境形成了鲜明的对比。现在说这个架构是否成功还为时尚早，但我们相信它已经很有趣了。