2. 架构风格与参考架构

REST架构风格

理解REST的五个关键词

REST是所有Web应用都应该遵守的架构设计指导原则。当然，REST并不是法律，违反了REST的指导原则，仍然能够实现应用的功能。但是违反了REST的指导原则，会付出很多代价，特别是对于大流量的网站而言。

要深入理解REST，需要理解REST的五个关键词：

- 资源（Resource）
- 资源的表述（Representation）
- 状态转移（State Transfer）
- 统一接口（Uniform Interface）
- 超文本驱动（Hypertext Driven）

什么是资源?

资源是一种看待服务器的方式，即，将服务器看作是由很多离散的资源组成。每个资源是服务器上一个可命名的抽象概念。因为资源是一个抽象的概念，所以它不仅仅能代表服务器文件系统中的一个文件、数据库中的一张表等等具体的东西，可以将资源设计的要多抽象有多抽象，只要想象力允许而且客户端应用开发者能够理解。与面向对象设计类似，资源是以名词为核心来组织的，首先关注的是名词。一个资源可以由一个或多个URI来标识。URI既是资源的名称，也是资源在Web上的地址。对某个资源感兴趣的客户端应用，可以通过资源的URI与其进行交互。

什么是资源的表述?

资源的表述是一段对于资源在某个特定时刻的状态的描述。可以在客户端-服务器端之间转移（交换）。资源的表述可以有多种格式，例如HTML/XML/JSON/纯文本/图片/视频/音频等等。资源的表述格式可以通过协商机制来确定。请求-响应方向的表述通常使用不同的格式。

什么是状态转移?

状态转移（state transfer）与状态机中的状态迁移（state transition）的含义是不同的。状态转移说的是：在客户端和服务器端之间转移（transfer）代表资源状态的表述。通过转移和操作资源的表述，来间接实现操作资源的目的。

什么是统一接口?

REST要求，必须通过统一的接口来对资源执行各种操作。对于每个资源只能执行一组有限的操作。以HTTP/1.1协议为例，HTTP/1.1协议定义了一个操作资源的统一接口，主要包括以下内容：

- 7个HTTP方法：GET/POST/PUT/DELETE/PATCH/HEAD/OPTIONS
- HTTP头信息（可自定义）
- HTTP响应状态代码（可自定义）
- 一套标准的内容协商机制
- 一套标准的缓存机制
- 一套标准的客户端身份认证机制

REST还要求，对于资源执行的操作，其操作语义必须由HTTP消息体之前的部分完全表达，不能将操作语义封装在HTTP消息体内部。这样做是为了提高交互的可见性，以便于通信链的中间组件实现缓存、安全审计等等功能。

什么是超文本驱动?

“超文本驱动”又名“将超媒体作为应用状态的引擎”（Hypermedia As The Engine Of Application State，来自Fielding博士论文中的一句话，缩写为HATEOAS）。将Web应用看作是一个由很多状态（应用状态）组成的有限状态机。资源之间通过超链接相互关联，超链接既代表资源之间的关系，也代表可执行的状态迁移。在超媒体之中不仅仅包含数据，还包含了状态迁移的语义。以超媒体作为引擎，驱动Web应用的状态迁移。通过超媒体暴露出服务器所提供的资源，服务器提供了哪些资源是在运行时通过解析超媒体发现的，而不是事先定义的。从面向服务的角度看，超媒体定义了服务器所提供服务的协议。客户端应该依赖的是超媒体的状态迁移语义，而不应该对于是否存在某个URI或URI的某种特殊构造方式作出假设。一切都有可能变化，只有超媒体的状态迁移语义能够长期保持稳定。

> Web 是旨在成为一个 Internet 规模的分布式超媒体系统,这意味着它的内涵远远不只仅仅是地理上的分布。Internet 是跨越组织边界互相连接的信息网络。信息服务的提供商必须有能力应对无法控制(anarchic)的可伸缩性的需求和软件组件的独立部署。通过将动作控制(action controls)内嵌在从远程站点获取到的信息的表述之中,分布式超媒体为访问服务提供了一种统一的方法。因此 Web 的架构必须在如下环境中进行设计,即跨越高延迟的网络和多个可信任的边界,以大粒度的(large-grain)数据对象进行通信。 ——Roy Fielding

REST的主要特征

REST风格的架构所具有的6个的主要特征：

1. 面向资源（Resource Oriented）
2. 可寻址（Addressability）
3. 连通性（Connectedness）
4. 无状态（Statelessness）
5. 统一接口（Uniform Interface）
6. 超文本驱动（Hypertext Driven）

这6个特征是REST架构设计优秀程度的判断标准。其中，面向资源是REST最明显的特征，即，REST架构设

计是以资源抽象为核心展开的。可寻址说的是：每一个资源在Web之上都有自己的地址。连通性说的是：应该尽量避免设计孤立的资源，除了设计资源本身，还需要设计资源之间的关联关系，并且通过超链接将资源关联起来。无状态、统一接口是REST的两种架构约束，超文本驱动是REST的一个关键词，在前面都已经解释过，就不再赘述了。

REST与各种分布式应用架构风格

常见的分布式应用架构风格

从架构风格的抽象高度来看，常见的分布式应用架构风格有三种：

- 分布式对象（Distributed Objects，简称DO）：架构实例有CORBA/RMI/EJB/DCOM/.NET Remoting等等
- 远程过程调用（Remote Procedure Call，简称RPC）：架构实例有SOAP/XML-RPC/Hessian/Flash AMF/DWR等等
- 表述性状态转移（Representational State Transfer，简称REST）：架构实例有HTTP/WebDAV

DO和RPC这两种架构风格在企业应用中非常普遍，而REST则是Web应用的架构风格，它们之间有非常大的差别。

REST与DO的差别

REST支持抽象（即建模）的工具是资源，DO支持抽象的工具是对象。在不同的编程语言中，对象的定义有很大差别，所以DO风格的架构通常都是与某种编程语言绑定的。跨语言交互即使能实现，实现起来也会非常复杂。而REST中的资源，则完全中立于开发平台和编程语言，可以使用任何编程语言来实现。

DO中没有统一接口的概念。不同的API，接口设计风格可以完全不同。DO也不支持操作语义对于中间组件的可见性。

DO中没有使用超文本，响应的内容中只包含对象本身。REST使用了超文本，可以实现更大粒度的交互，交互的效率比DO更高。

REST支持数据流和管道，DO不支持数据流和管道。

DO风格通常会带来客户端与服务器端的紧耦合。在三种架构风格之中，DO风格的耦合度是最大的，而REST的风格耦合度是最小的。REST松耦合的源泉来自于统一接口+超文本驱动。

REST与RPC的差别

REST支持抽象的工具是资源，RPC支持抽象的工具是过程。REST风格的架构建模是以名词为核心的，RPC风格的架构建模是以动词为核心的。简单类比一下，REST是面向对象编程，RPC则是面向过程编程。

RPC中没有统一接口的概念。不同的API，接口设计风格可以完全不同。RPC也不支持操作语义对于中间组件的可见性。

RPC中没有使用超文本，响应的内容中只包含消息本身。REST使用了超文本，可以实现更大粒度的交互，交互的效率比RPC更高。

REST支持数据流和管道，RPC不支持数据流和管道。

因为使用了平台中立的消息，RPC风格的耦合度比DO风格要小一些，但是RPC风格也常常会带来客户端与服务器端的紧耦合。支持统一接口+超文本驱动的REST风格，可以达到最小的耦合度。

REST带来的利益

比较了三种架构风格之间的差别之后，从面向实用的角度来看，REST架构风格可以为Web开发者带来三方面的利益：

简单性

采用REST架构风格，对于开发、测试、运维人员来说，都会更简单。可以充分利用大量HTTP服务器端和客户端开发库、Web功能测试/性能测试工具、HTTP缓存、HTTP代理服务器、防火墙。这些开发库和基础设施早已成为了日常用品，不需要什么火箭科技（例如神奇昂贵的应用服务器、中间件）就能解决大多数可伸缩性方面的问题。

可伸缩性

充分利用好通信链各个位置的HTTP缓存组件，可以带来更好的可伸缩性。其实很多时候，在Web前端做性能优化，产生的效果不亚于仅仅在服务器端做性能优化，但是HTTP协议层面的缓存常常被一些资深的架构师完全忽略掉。

松耦合

统一接口+超文本驱动，带来了最大限度的松耦合。允许服务器端和客户端程序在很大范围内，相对独立地进化。对于设计面向企业内网的API来说，松耦合并不是一个很重要的设计关注点。但是对于设计面向互联网的API来说，松耦合变成了一个必选项，不仅在设计时应该关注，而且应该放在最优先位置。

REST是一种为运行在互联网环境中的Web应用量身定制的架构风格。REST在互联网这个运行环境之中已经占据了统治地位，然而，在企业内网运行环境之中，REST还会面临DO、RPC的巨大挑战。特别是一些对实时性要求很高的应用，REST的表现不如DO和RPC。所以需要针对具体的运行环境来具体问题具体分析。但是，REST可以带来的上述三方面的利益即使在开发企业应用时，仍然是非常有价值的。所以REST在企业应用开发，特别是在SOA架构的开发中，已经得到了越来越大的重视。

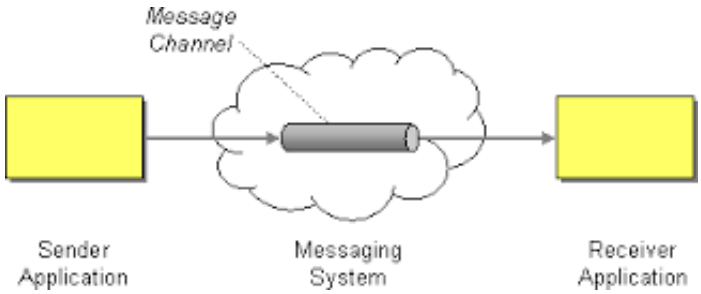
基于消息的分布式架构风格

在我参与过的所有企业应用系统中，无一例外地都采用（或在某些子系统与模块中部分采用）了基于消息的分布式架构。但是不同之处在于，让我们做出架构决策的证据却迥然而异，这也直接影响我们所要应用的消息模式。

常见的消息模式

消息通道（Message Channel）模式

我们常常运用的消息模式是Message Channel（消息通道）模式，如下图所示：



△ Message Channel模式

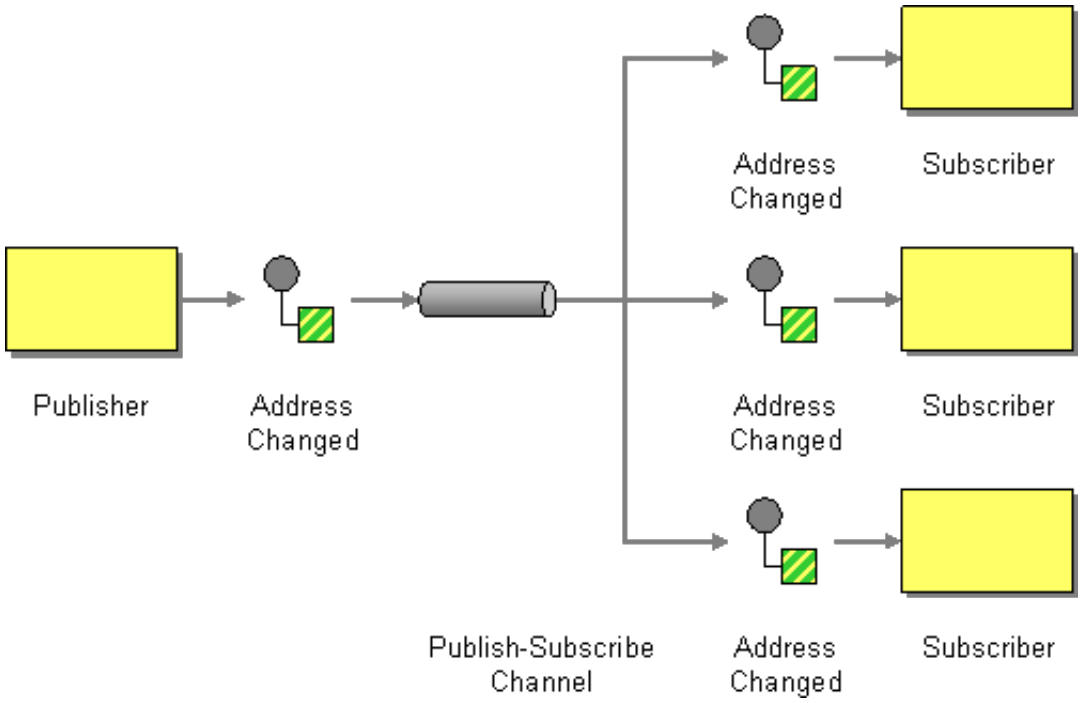
消息通道作为在客户端（消费者，Consumer）与服务（生产者，Producer）之间引入的间接层，可以有效地解除二者之间的耦合。只要实现规定双方需要通信的消息格式，以及处理消息的机制与时机，就可以做到消费者对生产者的“无知”。事实上，该模式可以支持多个生产者与消费者。例如，我们可以让多个生产者向消息通道发送消息，因为消费者对生产者的无知性，它不必考虑究竟是哪个生产者发来的消息。

虽然消息通道解除了生产者与消费者之间的耦合，使得我们可以任意地对生产者与消费者进行扩展，但它又同时引入了各自对消息通道的依赖，因为它们必须知道通道资源的位置。要解除这种对通道的依赖，可以考虑引入Lookup服务来查找该通道资源。例如，在JMS中就可以通过JNDI来获取消息通道Queue。若要做到充分的灵活性，可以将与通道相关的信息存储到配置文件中，Lookup服务首先通过读取配置文件来获得通道。

消息通道通常以队列的形式存在，这种先进先出的数据结构无疑最为适合这种处理消息的场景。微软的MSMQ、IBM MQ、JBoss MQ以及开源的RabbitMQ、Apache ActiveMQ都通过队列实现了Message Channel模式。

发布者-订阅者模式

这种模式实际上就是对Observer模式的体现，消费者扮演了观察者的角色，如下图所示：



△ Publisher Subscriber模式

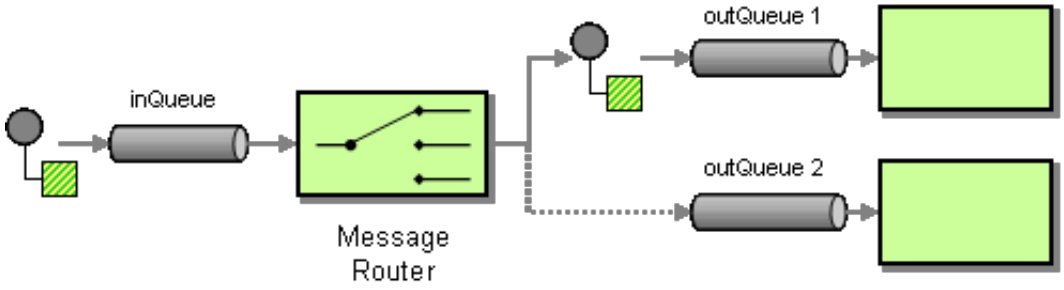
通常情况下，发布者和订阅者都会被注册到用于传播变更的基础设施（即消息通道）上。发布者会主动地了解消息通道，使其能够将消息发送到通道中；消息通道一旦接收到消息，会主动地调用注册在通道中的订阅者，进而完成对消息内容的消费。

对于订阅者而言，有两种处理消息的方式。一种是广播机制，这时消息通道中的消息在出列的同时，还需要复制消息对象，将消息传递给多个订阅者。例如，有多个子系统都需要获取从CRM系统传来的客户信息，并根据传递过来的客户信息，进行相应的处理。此时的消息通道又被称为Propagation通道。另一种方式则属于抢占机制，它遵循同步方式，在同一时间只能有一个订阅者能够处理该消息。实现Publisher-Subscriber模式的消息通道会选择当前空闲的唯一订阅者，并将消息出列，并传递给订阅者的消息处理方法。

消息路由（Message Router）模式

无论是Message Channel模式，还是Publisher-Subscriber模式，队列在其中都扮演了举足轻重的角色。然而，在企业应用系统中，当系统变得越来越复杂时，对性能的要求也会越来越高，此时对于系统而言，可能就需要支持同时部署多个队列，并可能要求分布式部署不同的队列。这些队列可以根据定义接收不同的消息，例如订单处理的消息，日志信息，查询任务消息等。这时，对于消息的生产者和消费者而言，并不适宜承担决定消息传递路径的职责。事实上，根据S单一职责原则，这种职责分配也是不合理的，它既不利于业务逻辑的重用，也会造成生产者、消费者与消息队列之间的耦合，从而影响系统的扩展。

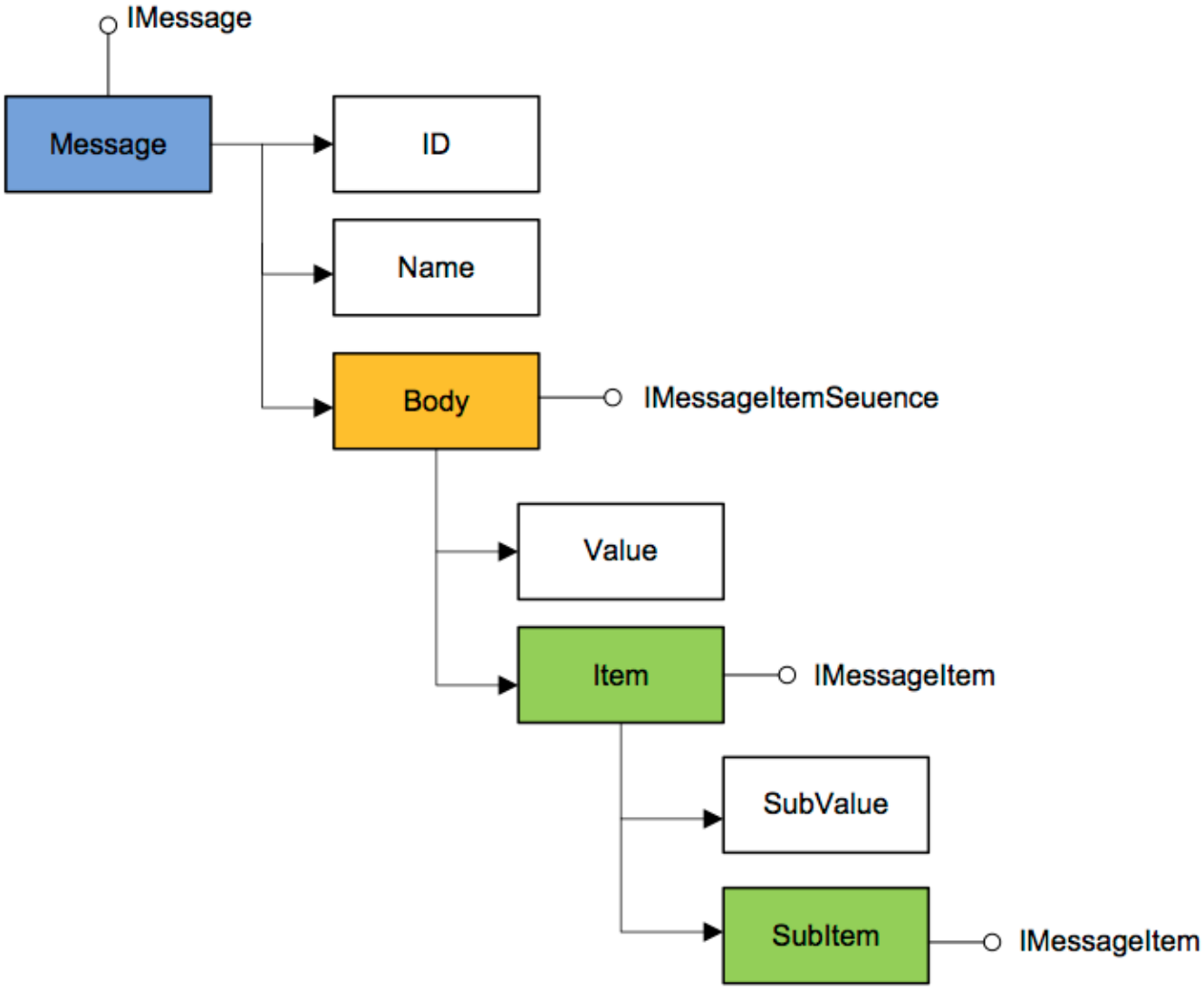
既然这三种对象（组件）都不宜承担这样的职责，就有必要引入一个新的对象专门负责传递路径选择的功能，这就是所谓的Message Router模式：



△ Message Router模式

通过消息路由，我们可以配置路由规则指定消息传递的路径，以及指定具体的消费者消费对应的生产者。例如指定路由的关键字，并由它来绑定具体的队列与指定的生产者（或消费者）。路由的支持提供了消息传递与处理的灵活性，也有利于提高整个系统的消息处理能力。同时，路由对象有效地封装了寻找与匹配消息路径的逻辑，就好似一个调停者（Meditator），负责协调消息、队列与路径寻址之间关系。

> **案例：制造工业的CIMS系统**
>
> 在CIMS系统中，我们尝试将各种业务以服务的形式公开给客户端的调用者，我们将消息划分为ID，Name和Body，并定义了IMessage接口。
>
> 消息的定义结构如下图所示：



△ 消息的结构

在实现服务进程通信之前，我们必须定义好各个服务或各个业务的消息格式。通过消息体的方法在服务的一端设置消息的值，然后发送，并在服务的另一端获得这些值。

我们在客户端引入了一个 `ServiceLocator` 对象，它通过 `MessageQueueListener` 对消息队列进行侦听，一旦接收到消息，就获取该消息中的name去定位它所对应的服务，然后调用服务的 `Execute(aMessage)` 方法，执行相关的业务。

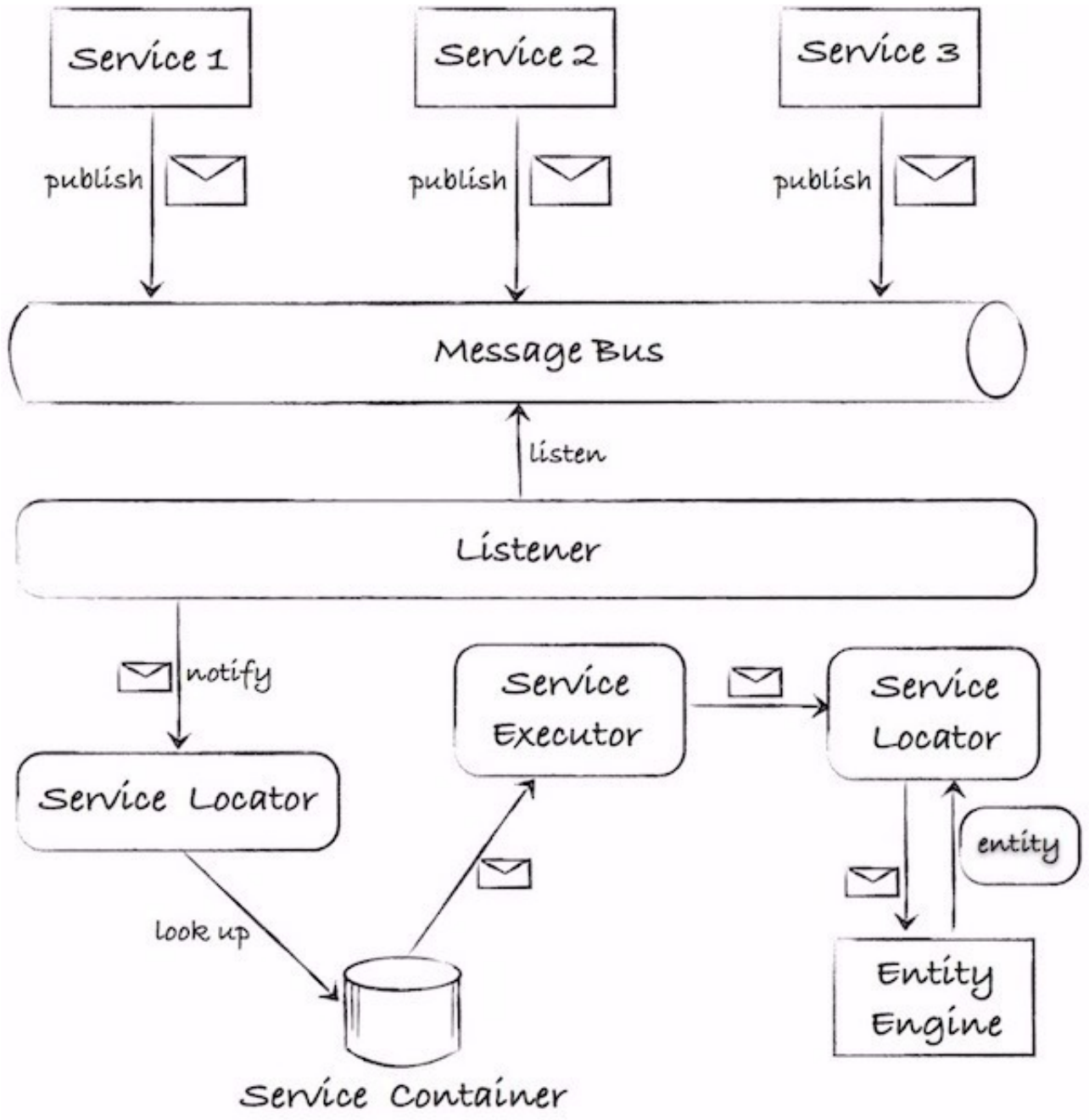
`ServiceLocator` 承担的定位职责其实是对存储在ServiceContainer容器中的服务进行查询。ServiceContainer容器可以读取配置文件，在启动服务的时候初始化所有的分布式服务（注意，这些服务都是无状态的），并对这些服务进行管理。它封装了服务的基本信息，诸如服务所在的位置，服务的部署方式等，从而避免服务的调用者直接依赖于服务的细节，既减轻了调用者的负担，还能够较好地实现服务的扩展与迁移。

在这个系统中，我们主要引入了Messaging模式，通过定义的IMessage接口，使得我们更好地对服务进行抽象，并以一种扁平的格式存储数据信息，从而解除服务之间的耦合。只要各个服务就共用的消息格式达成一致，请求者就可以不依赖于接收者的具体接口。通过引入的Message对象，我们就可以建立一种在行业中通用的消息模型与分布式服务模型。事实上，基于这样的一个框架与平台，在对制造行业的

业务进行开发时，开发人员最主要的活动是与领域专家就各种业务的消息格式进行讨论，这样一种面向领域的消息语言，很好地扫清了技术人员与业务人员的沟通障碍；同时在各个子系统之间，我们也只需要维护服务间相互传递的消息接口表。每个服务的实现都是完全隔离的，有效地做到了对业务知识与基础设施的合理封装与隔离。

对于消息的格式和内容，我们考虑引入了Message Translator模式，负责对前面定义的消息结构进行翻译和解析。为了进一步减轻开发人员的负担，我们还可以基于该平台搭建一个消息-对象-关系的映射框架，引入实体引擎（Entity Engine）将消息转换为领域实体，使得服务的开发者能够以完全面向对象的思想开发各个服务组件，并通过调用持久层实现消息数据的持久化。同时，利用消息总线（此时的消息总线可以看做是各个服务组件的连接器）连接不同的服务，并允许异步地传递消息，对消息进行编码。

这样一个基于消息的分布式架构如下图所示：



△ 分布式架构

对于分布式系统架构而言，消息会作为最主要的通信方式，因此选择符合项目要求的消息中间件就成为了架构师的职责。

案例：运用RabbitQ

技术选型

架构师需要重点考虑的是应该选择哪种消息中间件来处理此等问题？这就需要我们必须结合具体的业务场景，来识别这种异步处理方式的风险，然后再根据这些风险去比较各种技术，以求寻找到最适合的方案。

在北美医疗系统中，需要实现功能是对整个系统中的医疗术语（Term）进行查找和替换。我们通过分析业务场景以及客户性质，我们发现该业务场景具有如下特征：

- 在一些特定情形下，可能会集中发生批量的替换删除操作，使得操作的并发量达到高峰；例如FDA要求召回一些违规药品时，就需要删除药品库中该药品的信息；
- 操作结果不要求实时性，但需要保证操作的可靠性，不能因为异常失败而导致某些操作无法进行；
- 自动操作的过程是不可逆转的，因此需要记录操作历史；
- 基于性能考虑，大多数操作需要调用数据库的存储过程；
- 操作的数据需要具备一定的安全性，避免被非法用户对数据造成破坏；
- 与操作相关的功能以组件形式封装，保证组件的可重用性、可扩展性与可测试性；
- 数据量可能随着最终用户的增多而逐渐增大；

针对如上的业务需求，我们决定从以下几个方面对各种技术方案进行横向的比较与考量。如下图所示：

	Concurrency	Monitoring	Maintenance	Lincense	Cost	Security	Distributed	Deployment	Failover
Microsoft Message Queue	Yes	Tools for monitoring, such as Activexperts Network Monitor	Microsoft provided, we have commercial support.	Combined with Windows	No	MSMQ supports access control, auditing, authentication, and encryption to secure MSMQ enterprise.	Not Supported	Nothing	Not directly support.
Apache ActiveMQ	Yes	A lot of tools and third party tools. http://activemq.apache.org/how-can-i-monitor-activemq.html	Open source project, commercial support not guaranteed.	Apache License	No	Strong. Provides pluggable security through various different providers. Easy to config.	Supported	Need install Java	HA via master/slave failover built-in. It's not great.
RabbitMQ	Yes	A lot of tools which support web api, REST api and other plugins etc. http://www.rabbitmq.com/how.html#management	Open source project, commercial support not guaranteed.	MOZILLA PUBLIC LICENSE	No	Not very strong. RabbitMQ has pluggable support for various SASL(Simple Authentication and Security Layer) authentication mechanisms.	Supported	Need install Erlang	Weak. If clustered, the cluster behaves poorly when nodes go away, until their data is restored; see.

△ 技术矩阵

- 并发：选择的消息队列一定要很好地支持用户访问的并发性；
- 安全：消息队列是否提供了足够的安全机制；
- 性能伸缩：不能让消息队列成为整个系统的单一性能瓶颈；
- 部署：尽可能让消息队列的部署更为容易；

> - 灾备：不能因为意外的错误、故障或其他因素导致处理数据的丢失；
> - API易用性：处理消息的API必须足够简单、并能够很好地支持测试与扩展；
>
> 在项目中，通过对各种消息中间件的综合评价与质量评估，选择了RabbitMQ。原因如下：
> * 我们选择放弃MSMQ，是因为它严重依赖Windows操作系统；它虽然提供了易用的GUI方便管理人员对其进行安装和部署，但若要编写自动化部署脚本，却非常困难。同时，MSMQ的队列容量不能查过4M字节，这也是我们无法接收的。
> * Resque的问题是目前仅支持Ruby的客户端调用，不能很好地与.NET平台集成。此外，Resque对消息持久化的处理方式是写入到Redis中，因而需要在已有RDBMS的前提下，引入新的Storage。
> * 我们比较倾心于ActiveMQ与RabbitMQ，但通过编写测试代码，采用循环发送大数据消息以验证消息中间件的性能与稳定性时，我们发现ActiveMQ的表现并不太让人满意。至少，在我们的询证调研过程中，ActiveMQ会因为频繁发送大数据消息而偶尔出现崩溃的情况。相对而言，RabbitMQ在各个方面都比较适合我们的架构要求。
>
> 例如在灾备与稳定性方面，RabbitMQ提供了可持久化的队列，能够在队列服务崩溃的时候，将未处理的消息持久化到磁盘上。为了避免因为发送消息到写入消息之间的延迟导致信息丢失，RabbitMQ引入了`Publisher Confirm`机制以确保消息被真正地写入到磁盘中。
>
> 在并发处理方面，RabbitMQ本身是基于erlang编写的消息中间件，作为一门面向并发处理的编程语言，erlang对并发处理的天生优势使得我们对RabbitMQ的并发特性抱有信心。RabbitMQ可以非常容易地部署到Windows、Linux等操作系统下，同时，它也可以很好地部署到服务器集群中。它的队列容量是没有限制的（取决于安装RabbitMQ的磁盘容量），发送与接收信息的性能表现也非常好。RabbitMQ提供了Java、.NET、Erlang以及C语言的客户端API，调用非常简单，并且不会给整个系统引入太多第三方库的依赖。
>
> 即使我们选择了RabbitMQ，但仍有必要对系统与具体的消息中间件进行解耦，这就要求我们对消息的生产者与消费者进行抽象，这有利于未来消息中间件的扩展。
>
> 我们用Quartz.Net来实现Batch Job。通过定义一个实现了IStatefulJob接口的Job类，在Execute()方法中完成对队列的侦听。Job中RabbitMQSubscriber类的ListenTo()方法会调用Queue的Dequeue()方法，当接收的消息到达队列时，Job会侦听到消息达到的事件，然后以同步的方式使得消息弹出队列，并将消息作为参数传递给Action委托。
>
> 队列的相关信息例如队列名都存储在配置文件中。Execute()方法调用了request对象的MakeRequest()方法，并将获得的消息（即JobId）传递给该方法。它会根据JobId到数据库中查询该Job对应的信息，并执行真正的业务处理。

选择的时机

究竟在什么时候，我们应该选择基于消息处理的分布式架构？根据我参与的多个企业应用系统的经验，窃以为需要满足如下几个条件：

- 对操作的实时性要求不高，而需要执行的任务极为耗时；

- 存在企业内部的异构系统间的整合；
- 服务器资源需要合理分配与利用；

对于第一种情况，我们常常会选择消息队列来处理执行时间较长的任务。此时引入的消息队列就成了消息处理的缓冲区。消息队列引入的异步通信机制，使得发送方和接收方都不用等待对方返回成功消息，就可以继续执行下面的代码，从而提高了数据处理的能力。尤其是当访问量和数据流量较大的情况下，就可以结合消息队列与后台任务，通过避开高峰期对大数据进行处理，就可以有效降低数据库处理数据的负荷。

对于不同系统乃至于异构系统的整合，恰恰是消息模式善于处理的场景。只要规定了消息的格式与传递方式，就可以有效地实现不同系统之间的通信。

SOA架构风格

面向服务体系架构（SOA）的风格在过去10年中已经成为设计大型分布式系统的主流模式。SOA背后的核心思想是设计一个作为交互服务网络的系统。服务通过定义良好的接口提供清晰具体的功能。

利用服务（Web Service或者REST），可以很好地隔离服务的提供者与调用者之间的依赖，实现系统的松散耦合。设计时需遵循“服务是自治的”原则，并且能够较好地定义服务的边界，即使服务的实现发生了变化，对于整个系统而言，也不会严重到伤筋动骨的地步。重要的是，我们必须改变系统设计的思路与精神，利用面向服务而非面向对象的方式来考虑架构的整体设计。

> ### 案例分析：瑞士信贷的SOA架构
>
> 瑞士信贷很早就开始了内部系统的IT化，高度IT化有效地支撑了瑞士信贷的全球业务，然而随着时间的推移，那些还在采用传统CS架构的遗留系统已经不能满足越来越复杂的业务需求，而庞大的代码库、复杂的数据结构、异构系统以及紧耦合的系统平台，使得整个系统的主要精力都用在了应用集成上。应对需求变化的成本变得越来越高，维护成本与日俱增的同时，带来的是对IT系统的反思。瑞士信贷自然而然地走上了SOA之路。
>
> 服务管理
>
> 一个最基本的设计原则是客户端只能通过瑞士信贷信息总线上的服务接口访问主机上的数据。
>
> 各个项目在需要时才会构建服务。初期，当需要数据而又没有足够的接口存在时，各个项目会申请新的接口或扩展原有接口。集成架构师在初步质量检查步骤中评审这一“基本请求”并从如下三种可能的结论中选择其一：
>
> - 当能够满足需求的服务已经存在时，当前项目可以直接使用该服务。例如，股票交易系统可能计划构建一个股票代码服务，但并不知道其他应用已经可以提供这一信息。
> - 如果申请的服务并没有重用的潜力，当前项目将构建一个私有服务，独占式地在该应用内部使用。这种情况经常发生在原本就是面向服务的较大型应用上。
> - 如果申请的服务有重用潜力，就需要一个有扩展性的设计以便更广泛地应用。这种情况下，股票代码服务可能会被扩展成在提供标准的ISIN代码的同时还为每个币种提供内部编码。

在第二次质量检查时，一组设计专家会评审这一扩展后的设计并让设计方案更加清晰可执行。最后一次质量检查将确保服务的实现能够满足非功能性的质量指标，如性能。

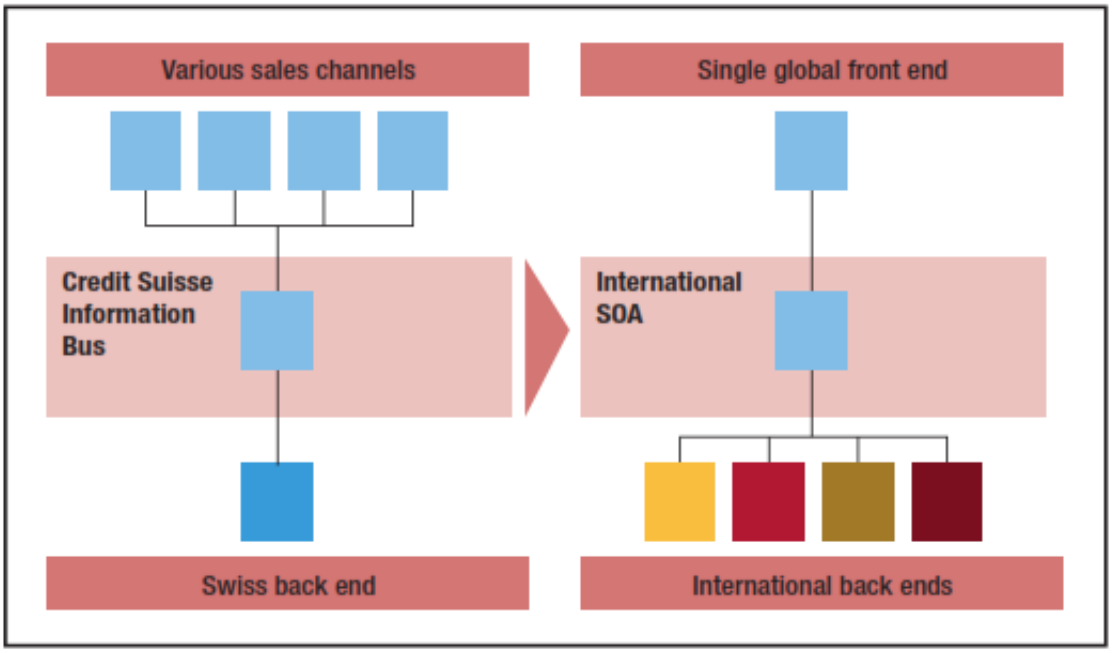
尽管从1998年开始就已经可以使用信息总线，不过项目经理们并不愿意构建和使用服务。在几个大型的性能和稳定性十分关键的应用采用了瑞士信贷信息总线后，开发团体的信任度才逐渐增加。从2002年开始，总线上暴露的功能不断增加，直到2007年，大部分的服务已经可用时，增长开始放缓。在客户端方面，采用量从2003年开始有了飞跃。当暴露在总线上的功能超过一半时，大型客户端（调用几百个服务并且每天产生上百万个服务调用的大型程序包）最终全面采用了该服务架构。此后，直到大约2009年，基本上所有客户端都会通过服务层访问主机。

架构设计的战略目标是通过服务暴露主机上所有的数据和功能。目前为止构建的绝大多数服务用于让分布式应用访问和操作数据——可能是主数据（客户信息）、参考数据（货币代码）或业务数据（账户信息）。一小部分服务提供对业务功能的访问，例如发起一次支付、提交一个交易订单或者税额计算。

一个重要的目标是鼓励在多个应用中重用服务。平均的重用因子是4，也就是每个服务被4个不同的应用使用。重用的情况很不平衡。某些服务会被大概100个应用重用，而大概有一半的服务只有一个消费者。意料之中的是，提供了参考数据和客户数据的服务具有最高的重用率。

大量的服务（和相对较低的重用率）是这种按需驱动方法所导致的。在理想的设计中，可能只需要大概一半的服务就能够提供同样的功能。

下图是瑞士信贷的SOA架构：

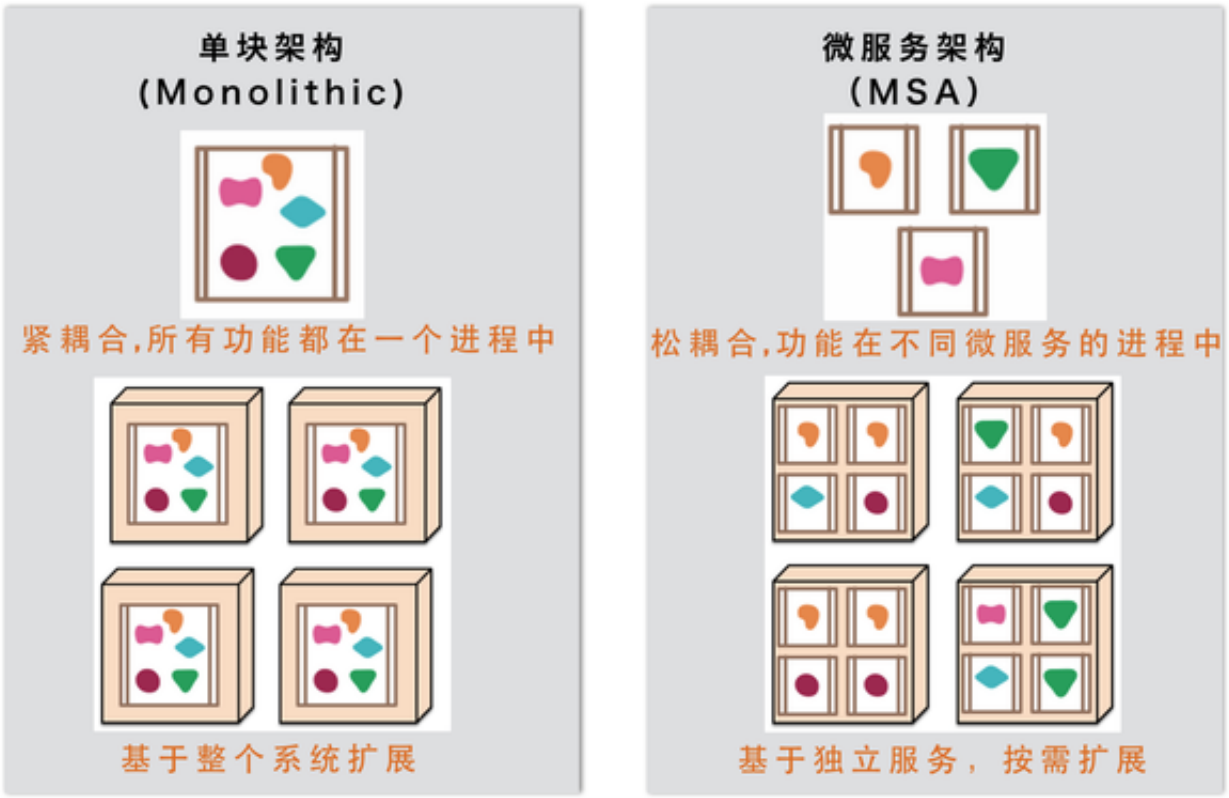


△ SOA架构

总结：首先，在大型公司中，深层次的架构变更所花费的时间要比大多数人们认为的时间更长。导致这一情况的原因是大部分的项目都是厌恶风险的，只希望采用经过验证的方法。新方法的验证加上从设计决策到完成实施的时滞，大概需要增加三到四年的时间。在此之后，则取决于不同的推广策略，完全实现某个策略可能要花费数年时间。如果想在这个领域取得成功，耐心和毅力是必不可少的。如果你们公司的首席信息官希望在一个季度内就看到结果，就不要推进SOA或其他的企业架构方案。其次，在考虑

> SOA时，在企业范围内所采用的技术是一个非常重要的先决条件，不过这一部分比较容易。在我们看来，更加困难的部分在于如何围绕SOA安排整个组织，如何为整个组织提供可以创建通用语言的恰当的语义框架，以及如何实现必要的管理过程。

SOA与Micro Service架构



△ 单块架构 vs. 微服务架构

微服务通常有如下几个特征：

小，且专注于做一件事情

每个服务都是很小的应用，至于有多小，是一个非常有趣的话题。有人喜欢100行以内，有人赞成1000行以内。数字并不是最重要的。仁者见仁，智者见智，只要团队觉得合适就好。只关注一个业务功能，这一点和我们平常谈论的面向对象原则中的”单一职责原则”类似，每个服务只做一件事情，并且把它做好。

运行在独立的进程中

每个服务都运行在一个独立的操作系统进程中，这意味着不同的服务能被部署到不同的主机上。

轻量级的通信机制

服务和服务之间通过轻量级的机制实现彼此间的通信。所谓轻量级通信机制，通常指基于语言无关、平台无关的这类协议，例如XML、JSON，而不是传统我们熟知的Java RMI或者.Net Remoting等。

松耦合

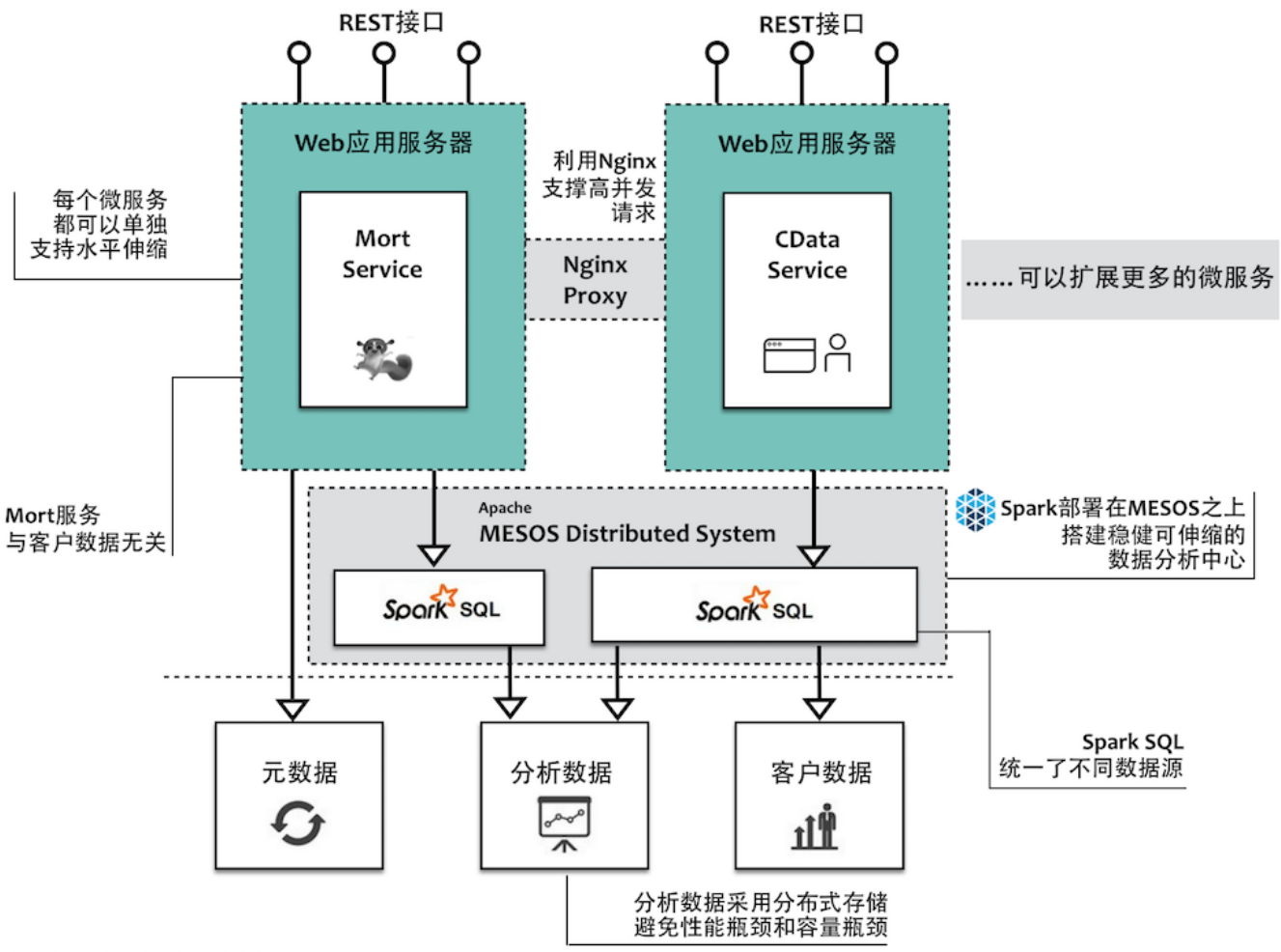
不需要改变依赖，只更改当前服务本身，就可以独立部署。这意味着该服务和其他服务之间在部署和运行上呈现相互独立的状态。

Mort BI的微服务架构

Mort使用Spray作为REST框架。它能够很好地支持REST服务，并与应用服务器完全解耦，可以根据需要将其部署到Jetty、Tomcat上。我们可以根据业务场景划分自治服务的边界，并将不同的服务（或微服务）部署到不同的应用服务器上，然后通过Nginx提供反向代理，并支撑高并发请求。

目前，Mort采用了微服务架构，将整个系统划分为两个自治的服务：Mort Service与CData Service。CData服务负责与客户数据（库）的交互，包括读取客户数据，导入客户数据，获取分析维度元数据信息。Mort Service作为Mort的核心服务，负责元数据管理以及执行数据分析与查询。

整个微服务架构如下图所示：

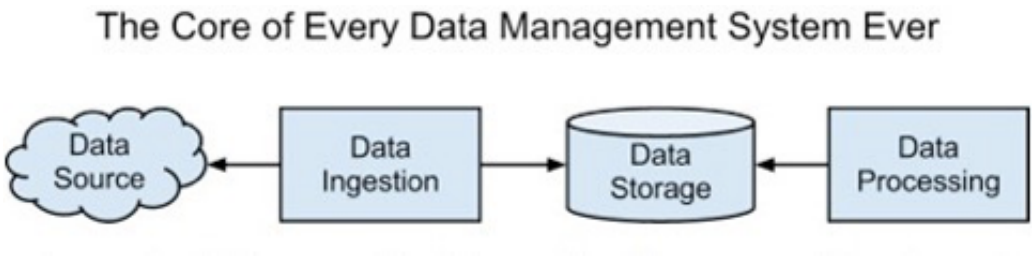


△ Mort BI微服务架构

数据为中心的软件架构

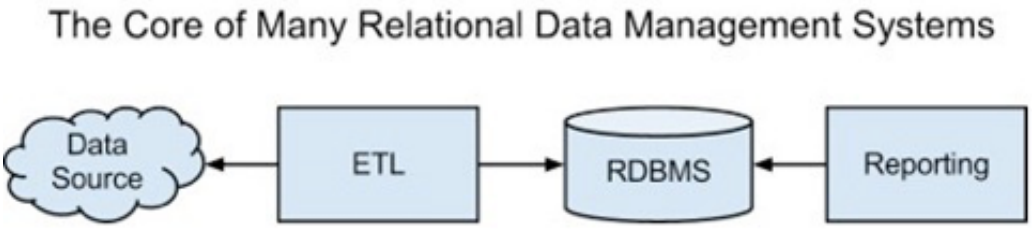
数据管理系统的核心

主要的数据管理系统（Data Management System）最少要包括三部分：data ingestion，data storage和data analysis。如下图所示：



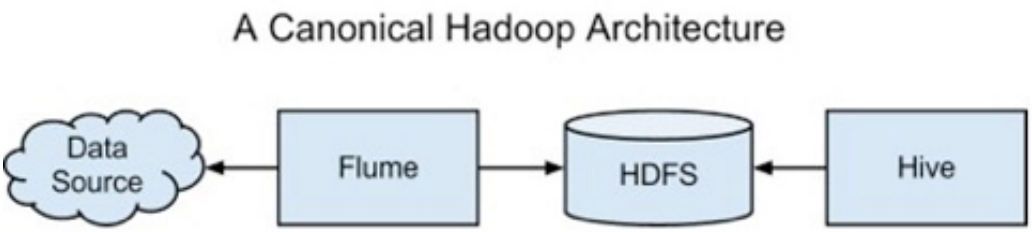
△ 数据管理系统的核心模型

对于关系型数据库，这个图可以理解为：



△ 关系数据库的核心模型

结合这个概念来理解Hadoop，如图所示：



△ Hadoop的核心模型

Lambda架构

大数据处理技术需要满足系统的可伸缩性与复杂性。首先要认识到这种分布式的本质，要很好地处理分区与复制，不会导致错误分区引起查询失败，而是要将这些逻辑内化到数据库中。当需要扩展系统时，可以非常方便地增加节点，系统也能够针对新节点进行rebalance。其次是要让数据成为不可变的。原始数据永远都不能被修改，这样即使犯了错误，写了错误数据，原来好的数据并不会受到破坏。

因此，一个大数据系统需要具备如下属性：

- 健壮性和容错性（Robustness和Fault Tolerance）
- 低延迟的读与更新（Low Latency reads and updates）
- 可伸缩性（Scalability）
- 通用性（Generalization）

- 可扩展性（Extensibility）
- 内置查询（Ad hoc queries）
- 维护最小（Minimal maintenance）
- 可调试性（Debuggability）

Lambda架构的主要思想就是将大数据系统构建为三个层次：Speed Layer、Serving Layer和Batch Layer。

Batch Layer

在Lambda架构中，实现batch view = function(all data)的部分被称之为batch layer。它承担了两个职责：

- 存储Master Dataset，这是一个不变的持续增长的数据集
- 针对这个Master Dataset进行预运算

显然，Batch Layer执行的是批量处理，例如Hadoop或者Spark支持的Map-Reduce方式。 它的执行方式可以用一段伪代码来表示：

```
function runBatchLayer():
  while (true):
    recomputeBatchViews()
```

例如这样一段代码：

```
Api.execute(Api.hfsSeqfile("/tmp/pageview-counts"),
     new Subquery("?url", "?count")
         .predicate(Api.hfsSeqfile("/data/pageviews"),
             "?url", "?user", "?timestamp")
         .predicate(new Count(), "?count");
```

代码并行地对hdfs文件夹下的page views进行统计（count），合并结果，并将最终结果保存在pageview-counts文件夹下。

利用Batch Layer进行预运算的作用实际上就是将大数据变小，从而有效地利用资源，改善实时查询的性能。但这里有一个前提，就是我们需要预先知道查询需要的数据，如此才能在Batch Layer中安排执行计划，定期对数据进行批量处理。此外，还要求这些预运算的统计数据是支持合并（merge）的。

Serving Layer

Batch Layer通过对master dataset执行查询获得了batch view，而Serving Layer就要负责对batch view进行操作，从而为最终的实时查询提供支撑。因此Serving Layer的职责包含：

- 对batch view的随机访问
- 更新batch view

Serving Layer应该是一个专用的分布式数据库，例如Elephant DB，以支持对batch view的加载、随机读取以

及更新。注意，它并不支持对batch view的随机写，因为随机写会为数据库引来许多复杂性。简单的特性才能使系统变得更健壮、可预测、易配置，也易于运维。

Speed Layer

只要batch layer完成对batch view的预计算，serving layer就会对其进行更新。这意味着在运行预计算时进入的数据不会马上呈现到batch view中。这对于要求完全实时的数据系统而言是不能接受的。要解决这个问题，就要通过speed layer。从对数据的处理来看，speed layer与batch layer非常相似，它们之间最大的区别是前者只处理最近的数据，后者则要处理所有的数据。另一个区别是为了满足最小的延迟，speed layer并不会在同一时间读取所有的新数据，相反，它会在接收到新数据时，更新realtime view，而不会像batch layer那样重新运算整个view。speed layer是一种增量的计算，而非重新运算（recomputation）。

因而，Speed Layer的作用包括：

- 对更新到serving layer带来的高延迟的一种补充
- 快速、增量的算法
- 最终Batch Layer会覆盖speed layer

Speed Layer的等式表达如下所示：

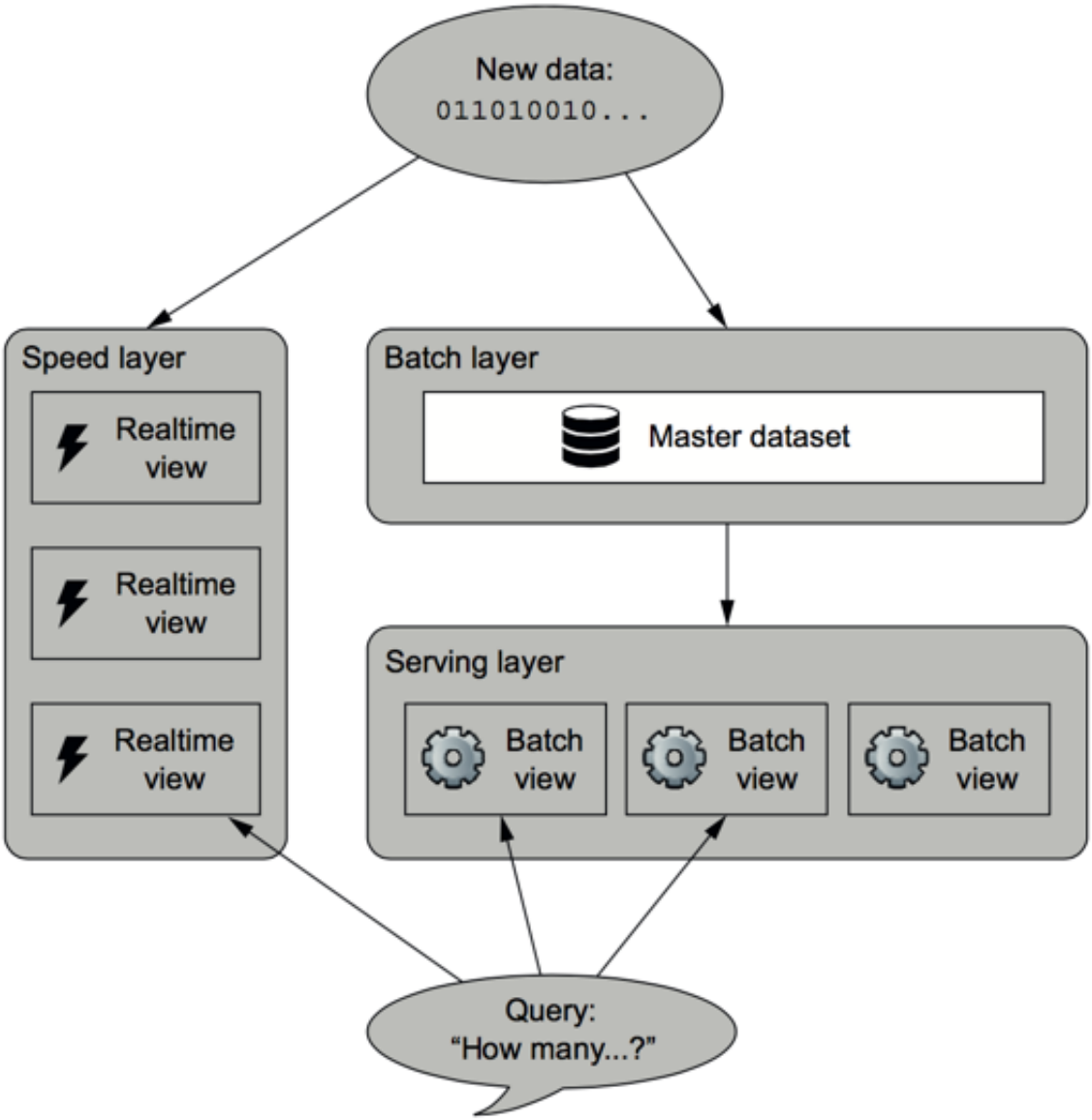
```
realtime view = function(realtime view, new data)
```

注意，realtime view是基于新数据和已有的realtime view。

总结下来，Lambda架构就是如下的三个等式：

```
batch view = function(all data)
realtime view = function(realtime view, new data)
query = function(batch view . realtime view)
```

整个Lambda架构如下图所示：

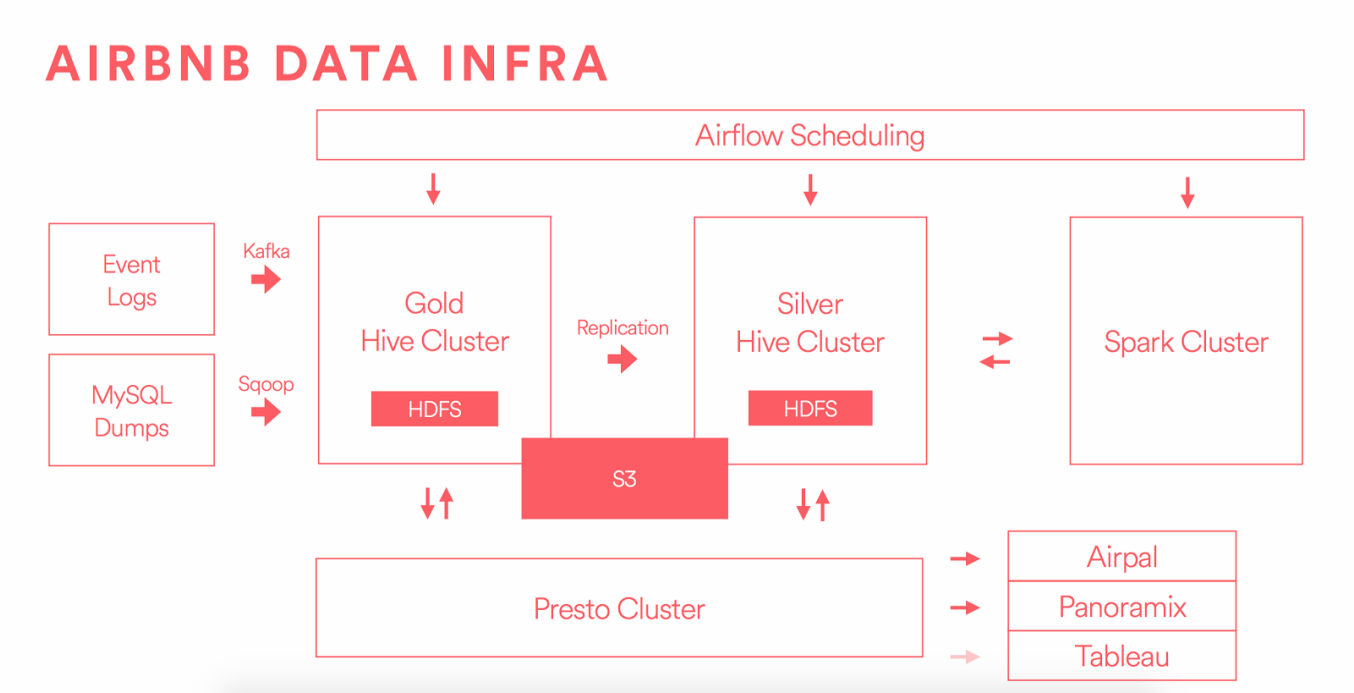


△ Lambda架构

---

airbnb的基础数据架构

下图是airbnb数据基础设施的主要架构：



源数据进入我们的系统有两个主要渠道：源代码发送Kafka事件和线上数据库将AWS RDS中的存储导出，再通过Sqoop传递。这里数据源包含用户的活动事件数据和快照源数据，发送到“金”集群存储，并开始运行我们的提取，转换和加载（ETL）。在此步骤中，我们针对业务逻辑，汇总表格，并执行数据质量检查。

在上面的图中，有“金”和“银”两个独立集群。分离原因是保证计算和存储资源的隔离，如果一个挂了可以做灾难恢复。这种架构提供了一个理想环境，最重要的工作严格保障SLA（服务保证协议），避免资源密集型即席查询的影响。我们把‘银’集群作为一个生产环境，但是放宽保证，可以承受资源密集型查询。

通过两个集群我们获得隔离力量，在管理大量的数据复制并维持动态系统之间有同步的成本。“金”是我们的真正来源，我们将复制“金”数据的每一位到“银”。“银”集群上生成的数据不会被复制回“金”，所以你可以认为这是“银”作为一个超集集群，是单向复制方案。因为我们的很多分析和报告从“银”簇发，当“金”有新数据产生，我们尽快复制它到“银”，去保证其他工作刻不容缓运行。更关键的是，如果我们更新预先存在的“金”集群上的数据，我们必须小心的更新并同步传播给“银”。

我们不跑Oracle，Teradata，Vertica，Redshift等，而是使用Presto对所有Hive管理的表做即席查询。我们都希望在不久的将来，联通Presto和Tableau。

其他的一些在图中要注意的东西，包括Airpal，使用Presto支持基于Web查询执行的工具，我们搭建并开源了。这是在数据仓库即席SQL查询，1/3以上的所有员工都使用该工具运行查询主界面。作业调度功能就是通Airflow，一个以编程方式编写，调度和监控的平台，可以运行在Hive，Presto，Spark，MySQL的数据管道等。我们在逻辑上跨集群共享Airflow，但物理作业运行在合适的集群机器上。Spark集群是工程师和数据科学家机器学习工作偏爱的另一处理工具，对流处理非常有用。你可以在Aerosolve查看我们在机器学习上的努力。 S3是一个独立的存储系统，我们可以从HDFS数据得到便宜的长期存储。Hive管理的表可以对自己存储改变，并指向S3的文件，容易访问和管理元数据。