**Nesoft.ServiceBus的设计介绍**

目 录

[**1．** **整体介绍** 3](#_Toc359171521)

[**(1)** **设计目的** 3](#_Toc359171522)

[**(2)** **整体逻辑结构** 4](#_Toc359171523)

[**(3)** **物理部署结构** 6](#_Toc359171524)

[**2．** **关于的Queue的实现** 7](#_Toc359171525)

[**(1)** **Nesoft.ServiceBus.Consumer** 8](#_Toc359171526)

[(1.1) 抽象基类EventPublisher、实现子类和相关工厂 8](#_Toc359171527)

[(1.2) 相关的队列配置 9](#_Toc359171528)

[(1.3) 发布方的事件消息的定义 10](#_Toc359171529)

[**(2)** **Nesoft.ServiceBus.Queue** 10](#_Toc359171530)

[(2.1) 抽象基类EventQueue和工厂 10](#_Toc359171531)

[(2.2) 派生的抽象类ReliableQueue 11](#_Toc359171532)

[(2.3) 基于SQL Server的具体实现 12](#_Toc359171533)

[(2.4) 基于MongoDB的具体实现 12](#_Toc359171534)

[**(3)** **Nesoft.ServiceBus.QueueManagement** 12](#_Toc359171535)

[**(4)** **消息存储结构的设计** 13](#_Toc359171536)

[**3．** **关于消息分发的实现** 16](#_Toc359171537)

[**(1)** **Nesoft.ServiceBus.Distributor** 16](#_Toc359171538)

[**(2)** **Nesoft.ServiceBus.DispatchHost** 17](#_Toc359171539)

[**4．** **扩展点介绍** 17](#_Toc359171540)

[**(1)** **对于Queue实现的扩展** 17](#_Toc359171541)

[**(2)** **对于订阅关系具体配置的读写的扩展** 18](#_Toc359171542)

[**(3)** **对于订阅者的Service调用方式的扩展** 18](#_Toc359171543)

[**(4)** **对异常处理、失败的消息分发任务的处理的扩展** 18](#_Toc359171544)

[**附1. ServiceProxy的设计** 18](#_Toc359171545)

1. **整体介绍**
2. **设计目的**

本应用组件主要希望达到的目的是：

* 解除系统与系统之间的设计耦合，解除设计层面的依赖关系，从而降低业务关联复杂度；
* 解除系统与系统之间的稳定性依赖，一个系统的处理响应速度过慢甚至down机不会对另一个系统的处理响应速度带来太大的影响；
* 方便处理系统根据任务压力情况灵活自由的通过横向scale out的方式来提升响应速度和处理能力；
* 避免分布式系统间发生分布式事务，通过保证数据最终一致性的方式来达到事务的目标；

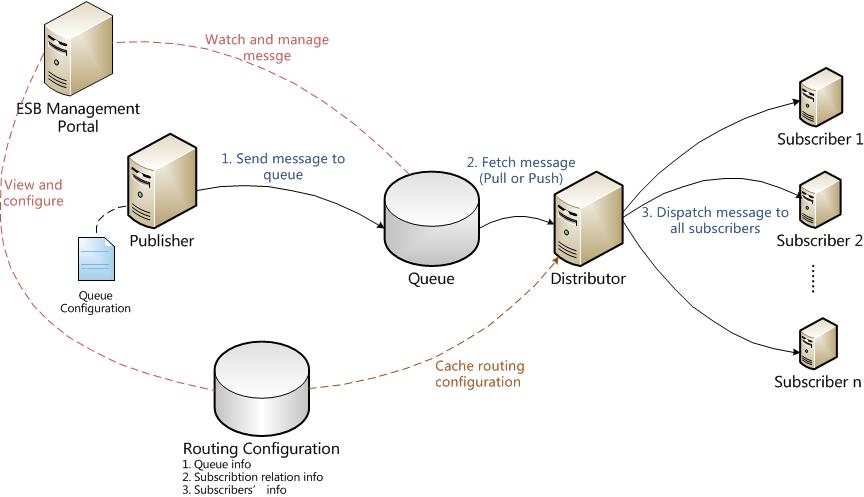
当前我们也有一些常用做法和方式可以达到上面的几点目的：

* 采用SQL Server Service Broker，比如前台网站下单后，任务通过SSB分发到后端订单处理系统来进一步处理；就目前的使用情况来看，该方式带来的问题有：
* 对底层数据库依赖很严重，如果底层数据库不是SQL Server则不行；
* 每次增加新的消息通道，都必须要DBA配合在数据库层面做相应的调整；
* 整套结构的部署实施需要DBA专业人士来处理，普通开发人员很难接手；
* 处理任务方必须是SQL Server的存储过程，SSB无法直接调用上层应用的Service，这导致了大量业务处理都在存储过程中，就现在情况来看还导致了一个逻辑会在上层应用的Service实现一遍，又会在存储过程中实现一遍，使得后期维护成本很高；
* 自行设计任务表（或在现有业务表里增加相关任务处理的标识字段），然后写Job来定时轮询处理，比如SO的欺诈检查等；就目前的使用情况来看，该方式带来的问题有：
* 使得单纯的业务设计中，糅杂了非业务需求的任务轮询处理逻辑设计，人为增加了系统复杂度；
* 增加了开发工作量，需要设计更多字段甚至表，需要自行设计job，控制调度的schedule；
* 对于job的稳定性、异常的合理处理、事务性（最终一致性）的保证都需要自行控制，不同水平的开发人员可能产出的结果质量差异很大，增加了项目的风险；
* 可复用性低，由于每次job的设计都是和具体业务点相关，在设计开发时往往抽象度不够，导致这部分设计很难复用，可能每个需要异步处理的业务点的job都要重复去实现这些执行调度、合理处理异常、保证数据一致性的代码；
* 处理任务的job只能采用定时轮询的调度方式，这会导致即时性不够好（和轮询的频率相关，但轮询频率很高也会带来性能上的较大损耗），另外也会导致空闲时段（比如整个晚上）无用的轮询执行；

1. **整体逻辑结构**

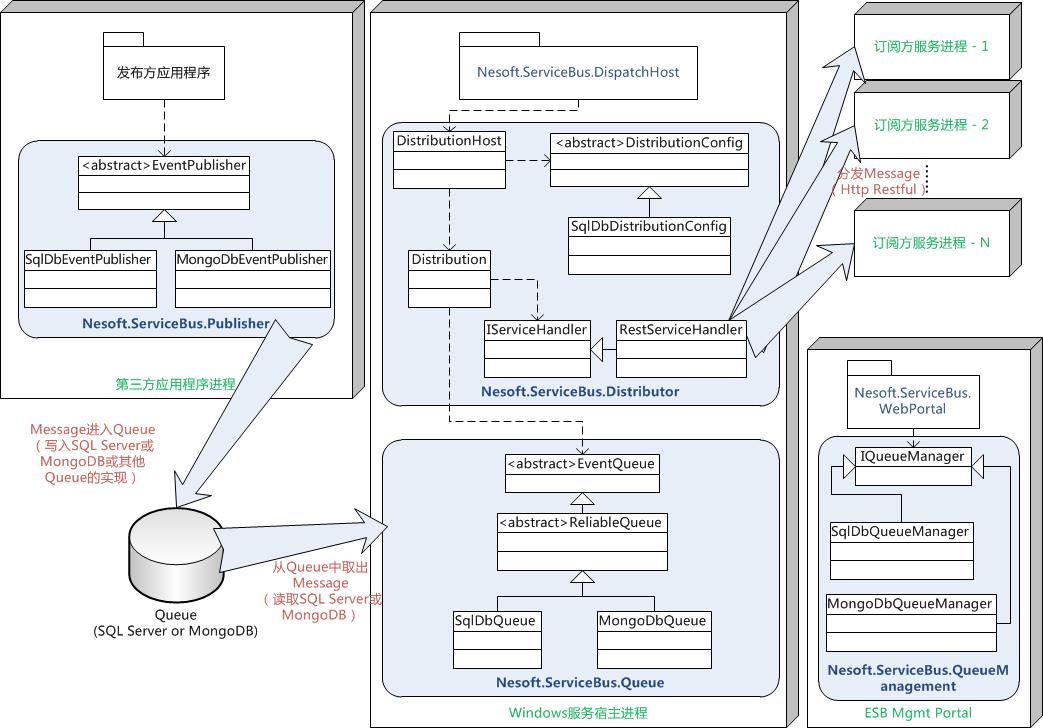
可以把整个应用可分为6大块：

* Routing Configuration：负责存储发布订阅关系和订阅者服务信息的配置（数据库或xml文件或其他存储方式）
* Publisher：负责发出消息
* Queue：负责存储消息
* Distributor：负责进行消息分发
* Subscriber：负责处理和消费消息
* ESB Management Portal：管理工具，用来进行订阅者服务配置、消息路由配置以及进行消息的实时处理监控



如图所示，Publisher会根据本地的Queue配置文件，将发布方的事件消息放入指定的消息队列中。消息分发器（Distributor）首先会根据所配置的消息订阅关系，给Publisher放入的消息来为每个订阅者生成一个消息分发任务，从而形成一个消息分发任务队列，然后开始执行消息分发任务队列中的每一条任务，根据消息任务自身所包含的订阅者（Subscriber）的标识到Routing Configuration的配置信息中找到该订阅者（Subscriber）的服务的物理地址信息，然后进行调用，从而完成了消息的分发投递。

下面是整体设计的逻辑结构图：

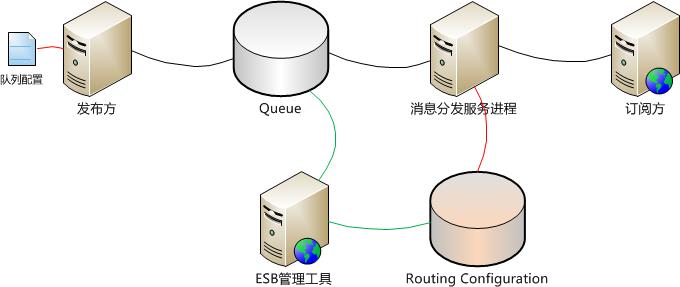


从上图可以把整个环境分为左、中、右上、右下四部分，其中：

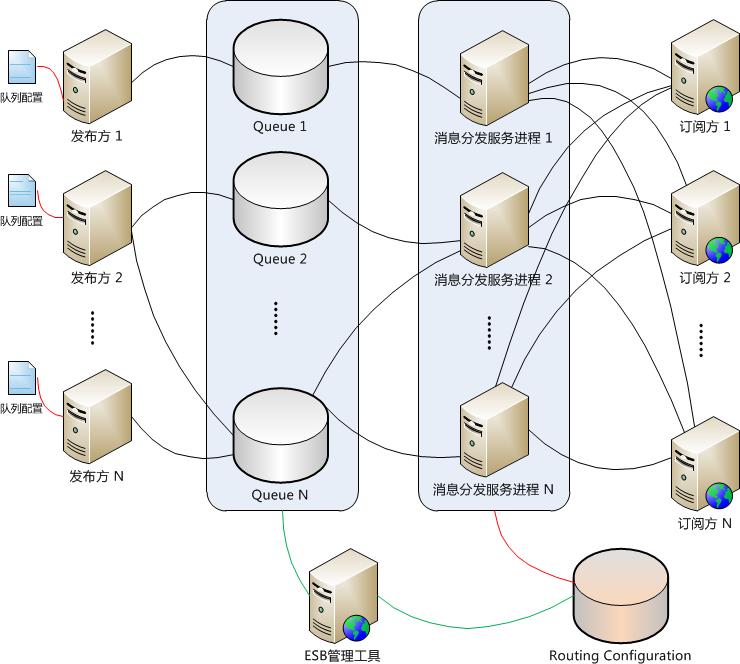
* 最左边为扮演消息发布方的第三方系统；可以看到，在第三方系统里，如果要发布消息，需要：
* 需要引用Nesoft.ServiceBus.Publisher这个程序集，使用EventPublisher这个类（一个带有静态工厂方法的抽象类）来制造一个其具体实现类的实例，然后进行消息发布；
* 因为需要Queue的配置信息，所以需要在第三方应用系统本地放置一个配置文件；
* 第三方系统的运行进程需要能够直接访问到Queue所在的SQL Server或MongoDB；
* 中间为扮演消息分发器的Windows服务；在该服务进程里会使用到Nesoft.ServiceBus.Distributor和Nesoft.ServiceBus.Queue这2个程序集，来实现从Queue里获取消息，并且分发出去；
* 最右边上面部分为扮演消息订阅者（消费者）的服务，要求就是以Http Restful的方式暴露出来，接受Http Post方式，数据格式为JSON；
* 最右边下面部分则是负责管理事件订阅关系和管理消息数据的Web Portal，其为一个宿主在IIS中的ASP.Net网站，该网站程序会使用到Nesoft.ServiceBus.QueueManagement程序集，该程序集主要提供了对消息队列中消息数据的管理能力；而对于订阅关系的管理的相关逻辑代码，则是直接在Nesoft.ServiceBus.WebPortal网站这一层中直接实现的。

1. **物理部署结构**

简单的单实例的方式部署：（黑色线条表示消息数据，红色线条表示配置数据，绿色线条为管理监控数据）



当消息的数据量很大，或者消息的并发写入压力很大，或者希望消息分发能力更强时，可以采用横向扩展的分布式部署：（黑色线条表示消息数据，红色线条表示配置数据，绿色线条为管理监控数据）



如上图所示：

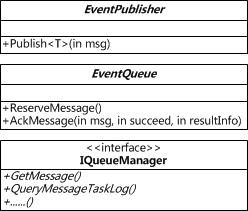
* 可以通过增加Queue的数量（通过增加DB甚至增加服务器的方式），设计一定的消息数据分割策略（比如不同Event主题的消息放到不同的物理Queue或者对消息按顺序依次循环放到不同的物理Queue等等），来缓解消息数据量过大以及缓解对Queue的高并发写入的压力。
* 另外可以通过增加消息分发服务的进程（开多个进程甚至用更多的应用服务器），来增强对消息的分发能力。一个消息分发服务进程可以配置为监控一个或多个Queue，而同一个Queue也支持被多个不同的消息分发服务进程进行监控和处理。
* 而一个逻辑上的订阅方，也可以提供多个Restful Service的地址来实现该订阅方处理消息的负载均衡和防止单点失败。

1. **关于的Queue的实现**

整个应用实现的核心，在于中心的Message Queue的实现部分。对于Message Queue的实现，核心又在于3个部分：

1. 把消息放入队列
2. 从队列中取消息
3. 查看队列中的消息情况

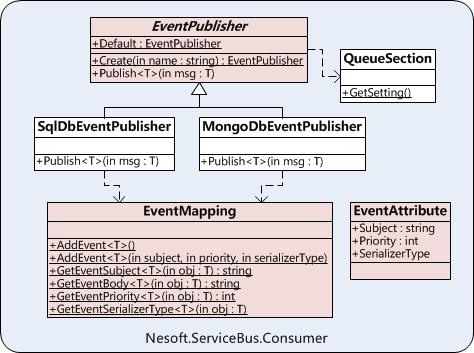
在这里的实现里，我们将3个部分拆开来，分别做到了Nesoft.ServiceBus.Consumer和Nesoft.ServiceBus.Queue以及Nesoft.ServiceBus.QueueManagement三个工程里。在这三个工程中分别定义了三个很重要的核心抽象（2个抽象类和1个接口）：



1. **Nesoft.ServiceBus.Consumer**
   1. 抽象基类EventPublisher、实现子类和相关工厂

该工程里定义了抽象基类EventPublisher，该类定义了抽象方法Publish<T>用来将一个泛型T类型的对象实例发布到Queue里去；当前工程中对于抽象基类EventPublisher实现了基于SQLServer技术和MongoDB技术的2个实现类。

该抽象基类EventPublisher类同时还兼任了静态工厂类的作用，其静态属性Default会根据配置文件的设置返回所配置的默认的EventPublisher的实现类的对象实例；而静态方法Create则根据入参字符串，到配置文件中找到对应的EventPublisher的实现类，创建出其对象实例，并调用其Init方法进行初始化。



* 1. 相关的队列配置

需要在应用的app.config或web.config文件中加入类似于下面的配置：



在<queue>节点上的default属性用来为EventPublisher.Default配置所使用的默认的Queue，而其下的<queue>/<item>/<parameters>节点下的键值对配置，则是每个EventPublisher实现子类各自所需要的配置数据，EventPublisher的实现子类可以通过override基类的方法：

protected abstract void InitPublisher(Dictionary<string, string> queueArgs);

来得到这些配置参数，并做相关的初始化工作。

* 1. 发布方的事件消息的定义

静态类EventMapping和特性类型EventAttribute是用来定义消息的，有2种方法将一个类标记为特定的消息：

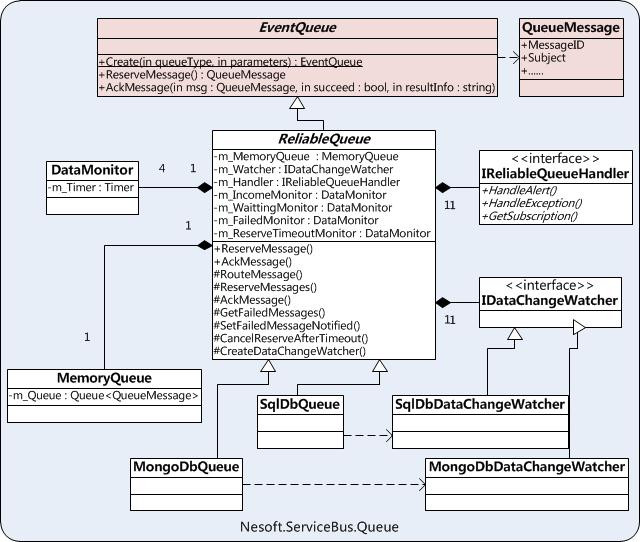
1. 在一个class上加上EventAttribute的特性，通过该特性的构造参数可以为该class指定其所代表的消息的Subject、被分发处理的优先级以及被序列化为字符串的方式；这种方式是代码声名式的方式，方式优雅、简单，但要求该class的工程（一般为Entity工程）必须要依赖Nesoft.ServiceBus.Publisher这个dll，这有一定的侵入性，也增加了不必要的耦合。
2. 在发布方的应用程序里，使用EventMapping.AddEvent<T>静态方法来注册添加一个class和消息的映射关系，可以在应用启动的地方，如asp.net web的Application\_Start事件里，一次性把本应用所有需要用的消息和对应的class都注册添加好，当然这里也可以自行扩展使用读取xml配置文件然后根据配置来动态注册添加映射关系；因为发布方应用程序本身就需要引用Nesoft.ServiceBus.Publisher这个dll，所以不会额外增加耦合，但需要发布方多写些代码；

对于上面的2种方式，使用者根据自己的实际情况来选择。

1. **Nesoft.ServiceBus.Queue**
2. 抽象基类EventQueue和工厂

该工程里定义了抽象基类EventQueue，该类定义了抽象方法ReserveMessage用来从队列中获取一个消息，而抽象方法AckMessage则是用来反馈一个消息是否被外部程序处理成功（以便将该消息从队列中移除）。

该抽象基类EventQueue类同时还兼任了静态工厂类的作用，其静态方法Create则根据入参指定的Queue类型名称和键值对参来创建EventQueue的实现类型，创建出其对象实例，并调用Init方法进行初始化。



1. 派生的抽象类ReliableQueue

ReliableQueue中将具体的消息的持久化和获取的动作抽象为抽象方法，交由具体实现子类在完成，而该类自身中组合了一个内存队列，一个数据变化侦听器，四个数据Monitor和一个统一处理异常、失败消息以及获取订阅关系数据的handler：

* IDataChangeWatcher：是一个队列的侦听器，有个Start()和Stop()两个接口方法来启动侦听和停止侦听，另外该接口有1个事件：

event DataChangedEventHandler DataChanged;

当有新消息进入队列后，就会即时触发该事件，从而达到底层队列即时通知上层代码有新消息到达的目的；

* DataMonitor：该类里通过一个Timer，定时地去执行一个动作。在ReliableQueue中组合了4个DataMonitor的对象实例，他们分别用来：
* 定时的检查消息队列中是否有新消息，如果有则根据订阅关系生成消息分发任务到任务队列；
* 定时的检查消息分发任务队列中还未执行的任务，并获取出来进行分发；
* 定时的检查还在处理中的消息分发任务，看是否有执行超时的任务分发任务，若有则将该消息处理失败次数加1，并检查失败次数是否达到上限，如没有到上限则还原任务为待处理以便后续继续处理，如果失败次数已达上限，则将任务放进失败队列中；
* 定时的检查警告提醒队列中是否有处理失败的任务，如果有则调用IReliableQueueHandler的HandleAlert方法来处理失败队列中的任务；
* MemoryQueue：一个基于Queue<T>的内存队列，当有新的消息进入队列后，DataMonitor会首先将消息分发任务加载入内存队列中，当外部应用调用ReliableQueue的ReserveMessage方法时则是从内存队列中来获取数据；
* IReliableQueueHandler：该接口有3个方法：
* 处理警告提醒任务
* 获取订阅关系的数据
* 处理异常

该接口的具体实现类是通过配置中key为handler的键值对的value来指定的。

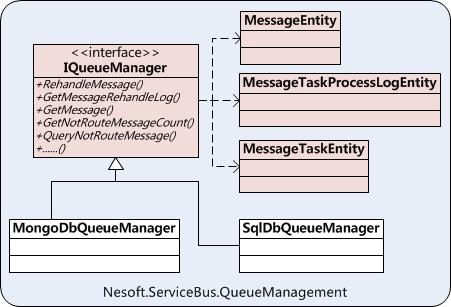
1. 基于SQL Server的具体实现

* SqlDbDataChangeWatcher：内部通过ADO.net 2.0之后提供的SqlDependency类，依赖于SQL Server的Service Broker的Services来实现数据库主动推送信息给.net应用程序；
* SqlDbQueue：通过ADO.net来实现对SQL Server相关库表的操作来实现队列效果；

1. 基于MongoDB的具体实现

* MongoDbDataChangeWatcher：利用基于Capped Collection 上的Tailable Cursors，实现当然MongoDB中有新数据进入后通知到.net应用；
* MongoDbQueue：通过Mongodb官方的C#驱动来完成相关数据的操作来实现队列效果；

1. **Nesoft.ServiceBus.QueueManagement**

****

该程序集主要通过接口IQueueManager暴露出来一系列方法，用于查看和管理队列中的消息、分发任务以及处理日志，包括对处理失败队列中的消息分发任务进行重新处理功能。

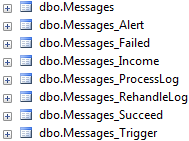
SqlDbQueueManager类通过ADO.net实现了对SQL Server库表中消息和相关日志数据的查看、管理；

MongoDbQueueManager类通过Mongodb官方的C#驱动实现了对MongoDB库表中消息和相关日志数据的查看、管理；

1. **消息存储结构的设计**

以SQL Server的实现为例：

在SQL Server中，一个逻辑上的Queue是由8张表来构成的，如下面的例子：



这里的7张表是以Messages这个名字来做的例子，而这个名字是可以换的，比如可以换为下面6张表：

dbo.TestMsg

dbo.TestMsg\_Alert

dbo.TestMsg\_Failed

dbo.TestMsg\_Income

dbo.TestMsg\_ProcessLog

dbo.TestMsg\_RehandleLog

dbo.TestMsg\_Succeed

dbo.TestMsg\_Trigger

那么只需要在配置中心为这个Queue配置“dbo.TestMsg”这个名字即可，其他几张表的名字会按照这个规律自动计算出来。

我们仍然以Messages这个名字作为例子来讲：

* dbo.Messages里存储为待处理的消息分发任务；
* dbo.Messages\_Alert里为某条失败一定次数后就会生成一个Alert警告提醒任务，前面提到的一个DataMonitor类就会监控这张表，发现新的还未发送通知的警告提醒任务，就会取出来发送通知，通知发送成功后会更新这个表的该记录，标识其已经发送了通知；
* dbo.Messages\_Failed里为超过重试次数后仍然失败的消息分发任务；
* dbo.Messages\_Income里为发布者发布的原始消息数据；
* dbo.Messages\_Succeed里为已经成功分发处理的消息；
* dbo.Messages\_ProcessLog里则是每一次的消息分发任务的处理（无论成功与否）都会记录一个日志；
* dbo.Messages\_Trigger是个辅助表，这个表只有一个bigInt字段，且默认就一条记录，前面提到的SqlDbDataChangeWatcher类里的SqlDependency就是监控的这张表，每次有新消息进来时就会对这条记录这个字段的值进行自增长加一的操作，从而触发SqlDependency，通知SqlDbQueue来从SQL Server中获取新消息；
* dbo. Messages\_RehandleLog表也是辅助表，是在ESB管理工具中把一个处理失败的消息重新扔回待处理队列（即dbo.Messages表）里时会记录一条日志，以便将来追踪数据处理情况；

我们详细看dbo.Messages表的设计结构，如下：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 字段名 | 字段类型 | 字段长度 | 允许为空 | 描述 |
| TransID | Int | 4 | 否 | 自增主键 |
| MessageID | Uniqueidentifier |  | 否 | 消息的标识，由Publisher来负责生成 |
| Subject | Varchar | 50 | 否 | 消息的主题 |
| Body | XML |  | 否 | 消息的内容 |
| Subscriber | Uniqueidentifier |  | 否 | 订阅者的标识 |
| FailedTimes | Int | 4 | 否 | 该消息处理失败的次数 |
| InTime | Datetime | 8 | 否 | 消息进入数据库表的时间 |
| Priority | Int | 4 | 否 | 消息的优先级，默认为1024，数值越小优先级越高，若数值相同则先进入队列的优先级高 |
| TimeoutSeconds | Int | 4 | 是 | 设置消息处理超时的时间，如果为null，则使用Distributor程序里的默认配置超时时间 |
| ReserveTime | Datetime | 8 | 是 | 消息被Distributor程序读取到内存中的时间，当Distributor读取该消息后会更新该字段为当前时间，通过扫描该字段来查看哪些消息处理超时，如果一旦发现有超时的消息，那么就会将该消息的该字段更新为NULL，并让FailedTimes字段自增1 |
| PlanReserveTime | Datetime | 8 | 是 | 计划下次被读取的时间，如果该字段为null，则Distributor认为可以直接读取出来进行处理；如果当前时间小于该字段时间，则Distributor不会读取出该消息，直到当前时间超过了改字段时间；这个字段的主要用处在于可以控制消息延时被Distributor读取出处理（比如消息处理失败后会延时5分钟再重试，而不是马上再被重试）。 |

我们可以模拟一条消息的处理过程，看看在Queue的这些表里会做些什么操作：

1. Publisher发布一条新的消息到Queue里：

* 向dbo.MessagesIncome里insert一条消息数据
* 更新dbo.Messages\_Trigger表字段，用以触发SqlDependency

1. Distributor根据订阅关系和新消息的Subject，为每个订阅者生成一条消息分发的任务

* 根据订阅关系，将dbo.MessagesIncome里的数据复制一条或多条到dbo.Messages表，每条记录上都会有相应的订阅者；

1. Distributor到dbo.Messages表里获取待处理的消息分发任务，获取条件为：

WHERE [ReserveTime] IS NULL AND ([PlanReserveTime] IS NULL OR [PlanReserveTime] <= GETDATE())

1. Distributor取出消息到内存队列中：

* 更新dbo.Messages表里该消息分发任务记录的ReserveTime为GETDATE()表示该任务已经在处理中及开始处理的时间；

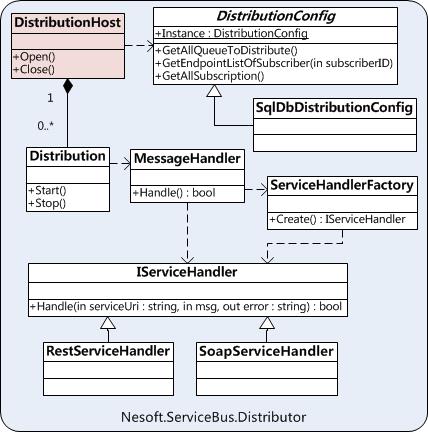
1. Distributor将消息投递给订阅者的Service处理
   1. 处理成功的话：

* 向dbo. Messages\_ProcessLog表里插入一条日志数据
* 向dbo.Messages\_Succeed表里插入该消息分发任务的数据（数据来源于dbo.Messages表）
* 从dbo.Messages表里delete该消息分发任务的记录
  1. 处理失败的话：
* 向dbo. Messages\_ProcessLog表里插入一条日志数据
* 检查该消息的失败次数是否达到配置的上限
* 如果还未达到失败次数的上限，那么就将dbo.Messages表里该消息分发任务记录的ReserveTime更新为NULL，PlanReserveTime更新为GETDATE()后的5分钟，FailedTimes自增1；
* 如果已经达到失败次数的上限，那么就将向dbo.Messages\_Failed表里插入该消息分发任务的数据（数据来源于dbo.Messages表），并且从dbo.Messages表里delete该消息分发任务的记录。

而监控处理中的消息是否有超时的DataMonitor类实例，则是定时检查dbo. Messages表里[ReserveTime]字段不为空，并且超过了timeout时间的记录，将这些记录的[ReserveTime]字段设为NULL，将其[FailedTimes]自增1，同时还会向dbo. Messages\_ProcessLog表里插入一条日志数据；

1. **关于消息分发的实现**
2. **Nesoft.ServiceBus.Distributor**

在该程序集里实现了DistributionHost作为消息分发器的数组，可以根据app.config/web.config文件中的配置来决定负责监控和分发哪些Queue的数据；该程序集中主要的类如下：



DistributionHost根据配置来组合0个或1个或多个Distribution类实例，而每一个Distribution类实例负责监控和分发一个队列的任务；Distribution内部会使用EventQueue来读取出消息分发任务，然使用MessageHandler来根据订阅关系中的订阅者地址进行分发，该MessageHandler类支持一个订阅者有多个Service地址，以实现load balance和防止单点失败；MessageHandler类里具体发送消息给service则是使用的IServiceHandler接口的Handle方法，当前接口IServiceHandler有2个默认实现：

RestServiceHandler -- 它实现了将消息通过Http JSON方式扔给Restful Service；

SoapServiceHandler --它实现了将消息通过标识Soap 1.1的方式扔给Webservice；

1. **Nesoft.ServiceBus.DispatchHost**

Nesoft.ServiceBus.DispatchHost为一个单纯的Windows服务，是默认的负责分发消息的宿主，里面主要调用了程序集Nesoft.ServiceBus.Distributor里的Distribution来实现真正的消息监控和分发工作；开发人员也可以通过调用 Nesoft.ServiceBus.Distributor里的Distribution来实现自己的消息分发宿主程序。

1. **扩展点介绍**
2. **对于Queue实现的扩展**

需要实现2个抽象类和一个接口：（当前实现了SQL Server和MongoDB的）

1. Nesoft.ServiceBus.Queue.EventQueue
2. Nesoft.ServiceBus.Consumer.Publisher
3. Nesoft.ServiceBus.QueueManagement.IQueueManager
4. **对于订阅关系具体配置的读写的扩展**

需要实现2个抽象类：（当前使用的SQL Server来存储）

1. Nesoft.ServiceBus.Distributor.Config.DistributionConfig
2. **对于订阅者的Service调用方式的扩展**

需要实现1个接口：（当前仅支持基于JSON格式接受Http Post的Restful Service的调用，以及标准SOAP 1.1的Webservice的调用）

1. Nesoft.ServiceBus.Distributor.ServiceHandler.IServiceHandler
2. **对异常处理、失败的消息分发任务的处理的扩展**

当前都只是记录个txt的日志文件并发送提醒邮件；

**附1. ServiceProxy的设计**

ServiceProxy既是抽象基类，同时也具体有工厂类的作用，其静态方法Find就为工厂方法，用来根据方法入参的service key到ServiceSetting配置中获取相关配置数据，然后构建和返回具体的ServiceProxy实现类；

