核心系统应用平台设计

# 概述

强大的应用平台（这里，应用平台不是指基础架构层面的操作系统、数据库、编程工具等设施，而是指通常由开发者编写的、用以简化相关环节设计的各种基础组件、框架、工具：通信框架，应用框架，持久化框架，...）可以极大地降低核心系统之类复杂应用系统的开发成本，同时提升应用系统在可维护性、可靠性、处理性能等各方面的品质。

#### 目标

下移的核心系统高度重视应用平台的建设，所有业务处理应用都建立在应用平台上，依托应用平台开发和运行。通过为业务处理应用提供强有力的支持和约束，应用平台将能够确保整个核心系统达到以下目标：

1）具有较高的处理性能，联机交易处理耗时综合平均小于100ms；

2）处理能力具有极强的可伸缩性。支持集群部署，处理能力可以横向扩展，与集群节点数基本成正比；

3）具有极高的可靠性/可用性。部署方案不存在单点故障；单节点宕机率控制在1次/1万小时以下；支持单节点自动恢复，恢复时间控制在30分钟以内；支持热插拔业务应用组件；

4）具有强大的通信处理能力。单节点支持30000以上的外部节点基于TCP协议同时接入和访问；按报文综合平均长度2K字节计，单节点报文收发量峰值达到5万个/秒；支持外部节点采用异步方式访问（在获得应答前提交新的请求）；支持外部节点按WebService标准访问；

5）可以灵活更换基础架构层面运用的产品和设施：服务器，数据库，...；

6）能够实施高效的管控；

7）应用开发成果具有极高的可维护性。支持采用图形化架构规划和业务建模的方法开发应用；支持结合开发项目管理流程的团队协作开发；

#### 设计理念

为实现以上目标，我们在核心系统应用平台设计过程中运用了以下理念：

1. 封装基础接口并规范运用
2. 构建通用基础模型并规范运用
3. 通信和业务处理分离
4. 业务处理流程与数据访问分离
5. 针对不同类型应用开发的特性建立相应的应用框架
6. 低耦合

#### 模块划分

基于以上思路，核心系统应用平台总共规划了九大模块，包括：工具包、基础服务、组件对象、通信应用框架、数据访问框架、服务应用框架、批量应用框架、调度器和交易主控。



工具包模块对java基础库和常用开源软件接口进行了必要的封装和替换，并按照企业规范封装了日志输出、分布式缓存访问、分布式文件访问、SQL访问，加密等基础API，为所有java应用程序的编写提供了最基本的支持和约束。

组件对象模块致力于构建在所有业务应用领域都具有普遍需要的引擎、模型、框架和工具——序列化、OXM、组件、组件对象设计器、组件工厂等等，这些设计为所有面向对象的、组件化的应用程序的编写模式和规范奠定了基础。

调度器模块致力于为基于图形化的设计器和脚本快速编排、组合应用处理逻辑和控制应用处理任务提供设计和运行时支持。

通信应用框架模块致力于构建具有强大处理性能、适配能力和可扩展性，运行高度稳定可靠的通信基础设施，以及可以对任意拓扑结构的分布式应用提供通信支持的应用框架。

基础服务模块致力于构建应用协同普遍需要的时钟服务、信号服务、同步服务、消息队列服务等基础服务。这些基础服务都是分布式的，需要部署独立运行的服务系统，相关组件都以通信应用框架模块为基础构建，并给客户端系统提供访问代理。

数据访问框架模块致力于通过ORM等引擎、框架帮助应用中的数据访问对象类以极其简洁的方式为上层应用提供统一的、无需编写SQL的数据访问接口。

服务应用框架模块以通信应用框架模块为基础构建，致力于为面向所有外部系统开放接口的业务处理服务应用的开发、运行提供应用框架形式的完备的支持、约束，统一解决与业务无关的技术层面的问题。

批量应用框架模块以通信应用框架模块和调度器模块为基础构建，致力于为批处理应用的开发、运行提供应用框架形式的完备的支持、约束，统一解决与业务无关的技术层面的问题。

交易主控模块致力于实现核心系统联机和批量交易应用公共的控制和处理设计。

需要注意的是——

1、这个架构仅仅是核心系统应用平台的软件架构，它只需面对核心系统应用开发和部署的需求。它不是整个企业应用架构，因此，像应用网关、ESB之类被核心系统视作外部系统的应用系统，以及对它们的构建提供支持的基础组件，在这个架构中原本是不需要加以考虑的。

2、尽管这个架构只是核心系统应用平台的软件架构，但其中绝大多数模块都具有普遍适用性，不是仅仅只能用于构建核心系统。例如，通信应用框架完全可以用于构建ESB；服务应用框架完全可以用于构建前置类应用系统；基础服务完全可以部署成企业级统一的基础服务系统；组件对象完全可以在展示层应用系统中（例如网银，甚至Android版本的手机银行APP）加以运用；等等。在9大模块中，交易主控是唯一和核心系统应用场景紧密相关的模块。

# 工具包

工具包（Utility）模块从编程更加可靠、更加简洁、更加规范、性能更好等角度对java基础库和常用开源软件接口进行了必要的封装和替换（例如，字符串的比较、连接方法，操作DOM的API，等等），并按照企业规范封装了日志输出、分布式缓存访问、分布式文件访问、加密等基础API，为所有java应用程序的编写提供了最基本的支持和约束。

## 基本辅助类和接口

基本辅助类和接口子模块是java基础库之上的很薄的一层封装，使用其中的类、方法、接口，而不是直接使用java基础包中类似的类、方法（例如字符串的比较方法），能够让应用的编写更简便、更灵活，运行更可靠。

OutObject和RefInteger, RefString, RefBoolean

为APM定义的接口：IAsynchronousResult/ICallBack

UUID

Helper：字符串，日期，反射，字符串/数值与二进制编码数据转换，...

EncryptHelper

LogHelper

## DOM

开源文档对象模型（Document Object Modal，简称DOM）包并发处理性能较差，我们自行开发了DOM包，为方便上层应用的迁移，其接口基本与DOM4J中较常用的接口完全相同。

## 日志输出

## 分布式缓存访问

## 分布式文件访问

## SQL访问

SQL访问子模块在JDBC基础之上封装了一组新的SQL操作接口。相比JDBC，这些接口可以让SQL应用的编写更简便、更符合业务应用开发人员的习惯，运行更稳定，更可靠——

* 支持使用SELECT ... INTO [数据变量]这样简洁的、一行代码的方式获取、写入数据，开发人员无须面对Statement、ResultSet等复杂的接口
* 支持SqlDataTable（一次性获取全部记录到本地，再基于本地数据进行遍历）和SqlCursor（使用在服务器上创建的游标进行遍历，每次获取一条记录）两种结果集访问接口

SQL访问子模块还使用C3P0对数据库链接资源自动进行池化管理，提高了数据库访问操作的性能。

# 组件对象

组件对象（Component Object，简称CO）模块致力于构建在所有业务应用领域都具有普遍需要的引擎、模型、框架和工具——序列化、OXM、组件、组件对象设计器、组件工厂等等，这些设计为所有面向对象的、组件化的应用程序的编写模式和规范奠定了基础。

组件对象模块由：1、OXM引擎；2、序列化引擎；3、组件对象模型；4、组件对象模型扩展；共4个子模块构成——



## OXM引擎

OXM引擎子模块提供了一个帮助应用实现对象-XML映射（Object Xml Mapping，简称OXM，支持从对象转换成XML，以及从XML转换成对象）方法的引擎。

#### 需求背景和目标

#### OXM接口和OXM实现类

#### 转换图和转换器

#### 动态转换支持

#### OXM引擎特性分析

基于专用结构和接口运作，而非基于反射，执行高效、安全，机制灵活

应用仅有接口编码工作，接口实现简洁，工作量很低。

发布、部署方便，无需使用其他工具，发布时也无需其他配套文件。

灵活（支持未登记名字的节点，支持具有动态结构的对象，例如集合型对象；可以结合数据检查）

较高的处理速度，10K长度的XML，时间消耗控制在....毫秒内。

但接口开发对于很多应用场景而言，还是觉得较为麻烦。

## 序列化引擎

序列化引擎子模块提供了一个帮助应用实现对象序列化（支持从对象转换成字节流，以及从字节流转换成对象）方法的引擎。不同于一般的序列化方法，这是基于可设定格式（编码字符集，域长度，分隔符，等等）的字符串字节流的序列化方法。

#### 需求背景和目标

交易系统运行时刻，服务系统需要把客户端系统传来的字节流数据转换成业务数据对象，交由应用处理，还需要把应用处理后产生的业务数据对象转换成字节流数据发送回客户端系统，这个转换过程就称为序列化/反序列化。如果业务数据对象自身具备序列化/反序列化能力，则应用和应用框架的开发就会更加简洁优美。

但让所有业务数据对象自身都具备序列化/反序列化能力，不是一件简单的事情。如果要求所有业务数据对象类的开发者自行编写代码实现序列化/反序列化接口，开发工作量巨大，质量也难以保证。为此，我们提出序列化设计，这是用于支持业务数据对象类以极其简易的方式获得序列化/反序列化能力的一种框架设计。

序列化所产生或需处理的字节流可以有多种形式，在这个序列化设计中，我们假定的字节流形式是字符串数据、数值数据等基本类型数据按约定编码方式产生的二进制编码，按约定位置和分隔符紧密连接形成的字节流。

例如，有如下数据对象类和应用代码：

**public** **class** MyData

{

public Integer f1;

public String f2;

public String f3;

public Integer f4;

...

}

MyData data = new MyData();

data.f1 = 3;

data.f2 = “OK”;

data.f3 = “hello”;

data.f4 = 100;

byte[] by = new byte[1000];

data.Serialize(by);

...

执行后，我们希望by将是如下状态（‘\_’表示空格对应的ASCII码，即0x20，其他数字、字母字符均表示字符对应的ASCII码）：

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| \_ | 3 | , | O | K | , | h | e | l | l | o | \_ | \_ | , | \_ | 1 | 0 | 0 |  |  |  |  |  |

#### 序列化接口和可序列化类

一个方法，只要能够实现业务数据对象与字节流数据之间的转换，似乎就可以被称为序列化接口，不必在意其表现形式——但是，我们不这么看，统一一下序列化接口的形态具有很大的意义，我们提出如下的标准化的序列化接口：

**public** **interface** ISerialize

{

public void Serialize(boolean bIsLoading, byte[] byValue, int nStartPosition, int nEndPosition, RefInteger nLength, SerializeControl serializecontrol);

}

其中，nStartPosition、nEndPosition表示byValue中的有效数据的起始、终止位置；

nLength表示对于当前接口对象而言，所占用数据的长度。外部应用在调用Serialize方法时，如果填写的bIsLoading为false（表示要序列化），则nLength的Value值只会是0，如果填写的bIsLoading为true（表示要反序列化），则nLength的Value值可能不会是0——这表示调用者知道对于当前接口对象而言，所需占用数据的长度。不管怎样，Serialize方法执行完后，都应当在返回前设置nLength的Value值为经过处理后的准确的占用数据的长度。

只有实现了该接口的类，我们才称之为可序列化类。借助于这个统一的序列化接口，再借助于序列化图和序列化器的支持，我们就能够让聚合了大量可序列化类子对象的、结构复杂的数据对象类很轻松地实现这个序列化接口，也成为可序列化类。

#### 序列化图和序列化器

假定我们已经有了A、B、C3个可序列化类，现在，我们构建了一个新的、聚合了A、B、C3类子对象的数据类D：

**public** **class** D

{

public A m\_A;

public B m\_B;

public C m\_C;

}

我们只需在D类中实现一个接口来填写序列化图，标明其有哪些子对象（必须都是可序列化的）会参与到序列化过程中，随后，即可将此图交给序列化器，由序列化器完成D类对象的序列化操作。

序列化图其实就是一个序列化接口表：

**public** **class** SerializeMap

{

**private** ArrayList m\_arraylistSerialize;

...

**public** **int** getSerializeCount()

{

**return** m\_arraylistSerialize.size();

}

**public** ISerialize GetSerialize(**int** nIndex)

{

**return** (ISerialize)m\_arraylistSerialize.get(nIndex);

}

**public** **void** SetSerialize(ISerialize iserialize)

{

m\_arraylistSerialize.add(iserialize);

}

...

}

对于D类而言，实现一个填写序列化图的接口非常容易：

**public** **void** InitializeSerializeMap(SerializeMap serializemap)

{

serializemap.SetSerialize(m\_A);

serializemap.SetSerialize(m\_B);

serializemap.SetSerialize(m\_C);

}

再借助序列化器，D类就很容易成为一个可序列化类：

public void Serialize(boolean bIsLoading, byte[] byValue, int nStartPosition, int nEndPosition, RefInteger nLength, SerializeControl serializecontrol)

{

SerializeMap sm = new SerializeMap();

InitialzieSerializeMap(sm);

new Serializer(sm).Serialize(bIsLoading, byValue, nStartPosition, nEndPosition, nLength, serializecontrol);

}

D类对象序列化后的字节流数据，实际上就是其包含的A、B、C类子对象，序列化后的字节流数据的紧密连接（连接处是否有分隔符，取决于SerializeControl）。

#### 序列化控制

上文中有一个多次提到但未深入阐述的概念：序列化控制（SerializeControl），这实际上就是一个包含了一组可以控制、影响序列化过程的参数的对象类。参数包括：分隔符是哪个字符，字符串编码集，是否所有域之间都必须插入分隔符，是否直接用二进制布局存放数值，等等。要注意的是：对于一个具体的可序列化类而言，序列化控制中的参数是否起作用，是由可序列化类的序列化接口的实现决定的——这意味着参数不一定都能起作用，比如，序列化控制指出所有域之间都必须插入分隔符，但某个实现得不是很好的复合型的可序列化对象仍然产生了不含分隔符的序列化数据。

序列化控制的存在意味着同一个可序列化对象产生的多份序列化数据，可能是不同的。

#### 动态序列化支持

某些复合型数据类，其成员数量可能不定，例如集合型的数据类，为了支持这些类对象的序列化，我们对序列化器和序列化图都进行了增强，首先增加了一个接口IGetDynamicSerialize：

**public** **interface** IGetDynamicSerialize

{

**public** ISerialize GetDynamicSerialize(**boolean** bIsLoading, **int** nIndex, **byte**[] byValue, RefInteger nStartPosition, **int** nEndPosition, SerializeControl serializecontrol) **throws** Exception;

}

然后，在序列化图的接口中增加了：

**public** **void** SetGetDynamicSerialize(IGetDynamicSerialize igetdynamicserialize) **throws** Exception;

运行时，序列化器会调用序列化图的GetGetDynamicSerialize()方法来获得图中的IGetDynamicSerialize接口，进而反复调用该接口的GetDynamicSerialize方法，获得对动态序列化数据区域进行读、写的可序列化元素对象，并调用这些元素对象的Serialize方法，完成序列化过程。

nIndex表示是在序列化器中连续第几次调用GetDynamicSerialize方法。对于集合型数据对象而言，如果是要做序列化，则通常应该看看nIndex是否已经大于自己包含的元素的个数，如果已经大于，则返回null，序列化器将不会再调用GetDynamicSerialize方法，如果不大于，则返回第nIndex个元素对象（这个对象也必须是可序列化的）；如果是要做反序列化，则通常会依据byValue中的数据（从nStartPosition开始），创建新的、类型合适的元素对象加入集合，并返回新对象。由于在反序列化时，byValue中的部分信息可能是用于断定应该产生什么样的元素对象，而不是用于新的元素对象的序列化，因此，nStartPosition是一个引用型的整数，在该方法返回时，可以调整其Value值，指明用于新的元素对象的序列化数据的最新起始位置。

反序列化时，序列化器如何知道何时该停止调用GetDynamicSerialize方法？序列化器有3种方式来确定动态序列化数据区域中对象个数：1、用确知的总长度-静态序列化数据区域的长度，如果在动态反序列化进行过程中，已处理的动态序列化数据长度达到这个值，就表明所有动态序列化数据区域中对象都已完成反序列化；2、用确知的动态序列化数据区域长度，如果在动态反序列化进行过程中，已处理的动态序列化数据长度达到这个长度，就表明所有动态序列化数据区域中对象都已完成反序列化；3、用确知的动态序列化元素对象的个数——所谓“确知”的信息，通常都是写在序列化数据中的。究竟采用哪种方式，取决于序列化图的一个方法：

**public** **int** getLengthRecordMode();

它可以有以下值（1、2、3分别对应上述3种方法）：

**public** **final** **static** **int** *LengthRecordModeNone* = 0;

**public** **final** **static** **int** *LengthRecordModeTotalLength* = 1;

**public** **final** **static** **int** *LengthRecordModeDynamicLength* = 2;

**public** **final** **static** **int** *LengthRecordModeDynamicCount* = 3;

#### 分隔符的运用

不定长string对象，又不在字节流中表述其长度，就必须依赖分隔符终止。实际上，不只是string，所有不定长的、又不在字节流中表述其长度的可序列化信息，都必须依赖分隔符终止——这意味着即便序列化控制不要求所有域之间都必须插入分隔符，某些域后面也一定会跟着分隔符。

在序列化控制指定所有域之间都必须插入分隔符时，I5对象的序列化数据为：

Cao | 12534|DocumentHelper|Y|Cao | 33|N|DocumentHelper |Y| 444|Cao|222222| 1112222|N|DocumentHelper|Y| 2|@I21|Cao|12334| 454|N|DocumentHelper|Y| 2122|@I22|Cao|12334| 454|N|DocumentHelper|Y|N

#### 占位支持

某些可序列化对象中的可序列化子对象，在序列化过程中采用占位模式——即先在某个位置放置一个占位标志，然后，才在后面有一段距离的某个位置放置序列化数据。

为支持占位模式，我们在ISerialize接口中增加了：

**public** **int** getOccupyingMode();

它可以有以下值：

**public** **final** **static** **int** *OccupyingModeNone* = 0;

**public** **final** **static** **int** *OccupyingModeBit* = 1;

**public** **final** **static** **int** *OccupyingModeByte* = 2;

**public** **final** **static** **int** *OccupyingModeLength* = 3;

所有Component类型的子对象都可以使用占位模式，例如，稍加修改I4的OnConstruct：

**private** **void** OnConstruct() **throws** Exception

{

SetElement("FirstName", m\_stringFirstName = **new** co.String("", 8));

SetElement("Phone", m\_integerPhone = **new** co.Integer(2, 13));

SetElement("LastName", m\_stringLastName = **new** co.String(""));

SetElement("EmailAddress", m\_booleanEmailAddress = **new** co.Boolean());

SetComponent("I1", m\_i1 = **new** I1());

SetComponent("I2", m\_i2 = **new** I2());

m\_i1.setOccupyingMode(Serializer.*OccupyingModeLength*);

}

则I4的序列化数据格式就被改变为（注意那个长度为4，内容为73的灰色域，这既是一个长度描述，也是一个占位信息，I1的序列化数据实际位置在87到160之间共73个字节）：

Cao 12434DocumentHelper|Y 73Cao|12334| 454NDocumentHelper|YCao 12334NDocumentHelper Y 454

#### 序列化引擎特性分析

该引擎具有以下特性——

* 基于专用结构和接口运作，而非基于反射，执行高效、安全，机制灵活
* 支持动态子对象的序列化，支持占位模式的序列化

## 组件对象模型

组件对象模型子模块确立了组件类的标准构建方式（以Component类为最终基类，并运用常用元素类和其他组件类组装新的组件类）。组件类自动具备OXM方法和序列化方法（基于OXM引擎、序列化引擎），以及可通过组件对象设计器查看、设计组件对象运行时状态的特性。

#### 需求背景和目标

我们知道，无论多么庞大的企业应用程序，运行时刻都可以被看作一组对象，这些对象的状态是至关重要的信息：一方面，我们需要了解对象的状态，用以判断系统当前的运行状况，诊断问题所在；另一方面，我们可以设置、调整对象的状态，用以控制系统的运行。

当企业的应用程序是由几千个组件共同构成的时候，每一个组件都提供其独特的查看和设置对象状态的手段，是不可想象的，我们必须采用某种规则来约束组件的开发，并在此基础上获得统一的查看和设置对象状态的手段。这个规则就是组件对象模型（Component Object Modal，简称COM），即要求所有组件开发时都必须遵照COM，按照特定的要求编写代码。请注意这里所讲的“组件”，它不是泛指一切应用逻辑，

#### COM基本规范

COM要求：1）组件开发时继承自组件对象模块提供的Component基类；2）组件对象的成员基本都应是各种Element派生类（Boolean/Integer/Float/Decimal/String/DateTime/…）和Component派生类的对象；3）根据Component基类的设计，每个组件对象都会内含一个构造图，组件对象构造时刻必须调用SetElement方法和SetComponent方法，在其构造图中登记成员。

#### 常用元素类和赋值校验接口

常用元素类有：Binary、Boolean、Integer、String、Float、Long、Decimal、DateTime，都是以Element为基础派生的类型。

常用元素类对象实例化时可以指定一个赋值校验接口，这些赋值校验接口在组件对象反序列化/OXM时刻都会自动发挥作用。组件对象模型提供了基于枚举值的校验器等最常用的校验器。

#### 组件对象的构造图



#### 组件对象的浏览、定位和查找方法

#### 组件对象的初始化方法和差分方法

#### 组件对象的OXM和序列化方法

按COM要求开发的组件将自动具备依据组件对象的构造图实现OXM的方法——转换（Translate）方法族（由Component基类提供）。



我们可以调用转换方法，把组件对象的状态输出到指定的XML文件中，这个XML文件也被称为组件对象的状态文件。同一类型的组件对象，状态文件具有相同的结构。例如，上面示例中的A类型的组件对象，其状态文件中根节点名称固定地为”A”（组件对象的类型名），并固定地具有名为”A1”，”A2”，”A3”的3个属性，它们与组件对象中通过SetElement方法登记了的Element派生类成员相对应；根节点下固定地含有名为”A4”，”A5”的2个子节点（这些子节点固定地具有名为”B1”，”B2”的属性），它们与组件对象中通过SetComponent方法登记了的Component派生类成员相对应。



当然，我们还可以调用转换方法，根据状态文件中的数据反过来设置对象的状态。需要特别说明的是，结构不完整的状态文件，只要确保其结构是完整结构的子集（例如，仅仅含有一个不具有属性的根节点），也被认为是合法的状态文件。

除了用于查看和设置对象状态外，转换方法和状态数据还被用于在分布式应用系统间进行数据传递，以便目标系统上能够产生与源系统状态完全相同的数据对象。例如，在OLTP类应用系统中，我们可以使用OXM技术来实现交易请求和应答的传递，所谓的交易报文就是交易请求和应答对象的状态数据。

为提高运行效率，组件对象（衍生自Component的类的对象）通常并非在转换前填写一个转换图，而是其自身就是一个转换图——我们定义了一个转换图接口ISerializeMap，所有实现了这个接口的类对象，都可被称转换图。转换器基于这个接口对象即可工作，而不一定非得是一个SerializeMap对象。Component也实现了ISerializeMap接口，这意味着Component的OXM实现很简洁，运行也很高效：

**public** **boolean** Serialize(**boolean** bIsLoading, **byte**[] byValue, **int** nStartPosition, **int** nEndPosition, RefInteger nLength, SerializeControl serializecontrol) **throws** Exception

{

**return** **new** Serializer(**this**).Serialize(bIsLoading, byValue, nStartPosition, nEndPosition, nLength, serializecontrol);

}

Component对象的序列化图信息是怎么形成的呢？是在我们调用SetElement、SetComponent方法构造Component对象时即填写完成的——

**class** I4 **extends** Component

{

//这个类用于演示两层全静态元素序列化

**private** co.String m\_stringFirstName;

**private** co.String m\_stringLastName;

**private** co.Integer m\_integerPhone;

**private** co.Boolean m\_booleanEmailAddress;

I1 m\_i1;

I2 m\_i2;

**public** I4() **throws** Exception

{

OnConstruct();

}

...

**private** **void** OnConstruct() **throws** Exception

{

SetElement("FirstName", m\_stringFirstName = **new** co.String("", 8));

SetElement("Phone", m\_integerPhone = **new** co.Integer(2, 13));

SetElement("LastName", m\_stringLastName = **new** co.String(""));

SetElement("EmailAddress", m\_booleanEmailAddress = **new** co.Boolean());

SetComponent("I1", m\_i1 = **new** I1());

SetComponent("I2", m\_i2 = **new** I2());

}

**protected** **void** OnInitialize() **throws** Exception

{

**super**.OnInitialize();

OnConstruct();

}

这和我们现在迁移工具生成的代码基本一样——因此，我们原先的应用、数据类，只要是衍生自Component 的，几乎不用修改，自动就都具备了OXM的能力。

I1 i1 = **new** I1();

i1.setAddress(**false**);

i1.setEmailAddress(**true**);

i1.setFirstName("Cao");

i1.setLastName("DocumentHelper");

i1.setPhone(33);

i1.setPostCode(444);

I2 i2 = **new** I2();

i2.setAddress(**false**);

i2.setEmailAddress(**true**);

i2.setFirstName("Cao");

i2.setLastName("DocumentHelper");

i2.setPhone(222222);

i2.setPostCode(1112222);

I4 i4 = **new** I4();

i4.setEmailAddress(**true**);

i4.setFirstName("Cao");

i4.setLastName("DocumentHelper");

i4.setPhone(12434);

i4.m\_i1.Initialize(i1);

i4.m\_i2.Initialize(i2);

基于以上操作，I4对象的序列化数据为：

Cao 12534DocumentHelper|YCao 33NDocumentHelper Y 444Cao|222222| 1112222NDocumentHelper|Y

## 组件对象模型扩展

组件对象模型扩展子模块提供常用的基础组件类、集合组件类和相关的辅助类，以及组件工厂，以方便设计具有复杂组件对象结构（容器，数组链表，哈希表，等等），以及需要频繁扩展、调整设计并热更新的应用。

#### 组件对象集合

在上面的示例中，A类型的组件对象固定地聚合了2个B类型的子组件对象，但在很多应用场景中，我们希望组件对象能够动态地聚合子组件对象，例如，某种类型的组件对象是另一种类型的子组件对象的集合，集合中的成员数量可以是任意数目。为此，组件框架提供了集合型组件——ComponentArrayList，在ComponentArrayList对象下可以聚合任意数目的符合类型要求的子组件对象。ComponentArrayList也继承自Component基类，因此，我们只需把ComponentArrayList对象用作所需开发的组件对象的一个成员，并在构造时刻调用SetComponent方法登记，即可满足动态聚合子组件对象的需要。



在上面的示例中，C类型组件对象的序列化文件根节点下会固定地包含一个名为”C1”的子节点，而”C1”子节点下可以有任意数目的子节点，这些子节点对应在C类型组件对象的ComponentArrayList成员对象下聚合的子组件对象，它们的名字是子组件对象的类型名，其结构与子组件对象的序列化文件根节点的结构一致。



我们同步改进了ComponentArrayList/Decorator和ComponentContainer/Decorator，让这些集合型组件类，或用于构建集合型组件类的装饰器类，基于动态序列化支持，实现可序列化，或装饰目标的可序列化。

**class** I5 **extends** I4

{

**private** ComponentArrayListDecorator m\_componentarraylistdecoratorI2;

**public** I5() **throws** Exception

{

OnConstruct();

}

**public** **int** getI2Count()

{

**return** m\_componentarraylistdecoratorI2.getComponentCount();

}

**private** **void** OnConstruct() **throws** Exception

{

m\_componentarraylistdecoratorI2 = **new** ComponentArrayListDecorator(**this**, ComponentFactory.*getDefaultComponentFactory*(), I2.**class**, **new** java.lang.Object[] { });

}

**protected** **void** OnInitialize() **throws** Exception

{

**super**.OnInitialize();

m\_componentarraylistdecoratorI2.Initialize();

OnConstruct();

}

I5对象的序列化数据为（注意那个长度为4，内容为2的灰色部分，它是动态序列化数据区开头的信息，表示动态序列化元素对象的个数——默认的，其序列化图的getLengthRecordMode方法是返回*LengthRecordModeDynamicCount*）：

Cao 12534DocumentHelper|YCao 33NDocumentHelper Y 444Cao|222222| 1112222NDocumentHelper|Y 2@I21|Cao|12334| 454NDocumentHelper|Y 2122@I22|Cao|12334| 454NDocumentHelper|YN

#### 组件工厂

当我们调用序列化方法，试图反过来设置一个C类型组件对象的状态时，如果序列化文件中”C1”子节点下含有一些子节点，则意味着序列化方法需要自动创建相应的子组件对象，并把它们添加到C类型组件对象的ComponentArrayList成员对象下。通常，这些子节点对应的子组件对象的类型都不是组件框架，甚至也不是整个CBF所提供的，那么，Component基类的序列化方法如何才能寻找到提供了这些子组件对象类型的程序集呢？为此，组件框架提供了组件工厂，只有在组件工厂（运行目录下的Eap.ComponentFactory.config文件）中登记了的组件类型，才是有效的——也就是说，我们必须在C类型组件对象运行目录下的Eap.ComponentFactory.config文件中登记B和C的组件类型，才能有效地运用上面示例中的序列化文件来设置对象状态。

Eap.ComponentFactory.config文件是一个XML文件，除去根节点和名为”Catalogue”的节点以外，其他每个节点都用以登记一个组件类型，所提供的信息包括：名称，以及该组件所在程序集和空间（即Java版本中的包）的名称。Eap.ComponentFactory.config文件通常由CBF的安装程序自动生成和修改，用户不可手工编辑。



在上面的示例中，C类型组件对象实现了动态聚合B类型子组件对象，但这个实现方式略有瑕疵。本质上，被动态聚合的子组件对象归属C类型组件对象的ComponentArrayList成员对象，而非C类型组件对象自身，与此对应，我们在其序列化文件中看到一个不想看到的”C1”子节点。为了达到更加理想的效果，组件框架提供了ComponentArrayListDecorator，它是一个装饰器类，用于把所需开发的组件自身装饰成集合型组件。



在上面的示例中，D类型组件对象的序列化文件根节点下可以有任意数目的子节点，这些子节点对应在D类型组件对象下聚合的子组件对象（当然，这些子组件对象本质上还需使用ComponentArrayListDecorator成员对象进行管理）。



除了ComponentArrayList之外，组件框架还提供了另一个常用的、同样也继承自Component基类的集合型组件——ComponentContainer。基于ComponentContainer，我们可以把组件对象开发成子组件对象的容器型集合——集合中只能有1个符合类型要求的子组件对象，或者没有。同样，为了达到更加理想的效果，组件框架还提供了ComponentContainerDecorator，用于把所需开发的组件自身装饰成容器型集合型组件。

可使用一个统一的、可视化的设计器程序（即组件对象设计器）产生与组件对象运行时状态完全对应的XML文件或其他格式的文本文件——这些文件往往被用作应用系统的配置文件，应用系统运行时可以从这些文件转换出相应的组件对象。组件对象运行时也可以将其状态输出到XML文件或其他格式的文本文件中，通过组件对象设计器查看这些文件，即可直观地了解组件对象运行时状态。

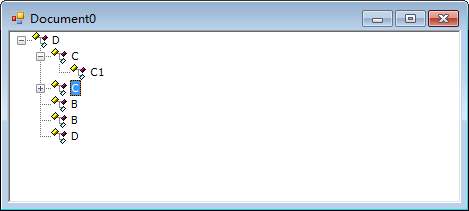
#### 组件对象设计器

按CCM要求开发的组件都将自动具备“可设计”功能，即可以通过CBF的组件设计器查看和设置组件对象的状态。

通常情况下，组件设计器不会直接访问已部署的CBF应用系统中的组件对象。最常见的处理方式是在工作台系统上创建同类型的组件对象，并运用组件设计器查看和设置其状态，然后将状态输出到序列化文件，最后，再操作已部署的CBF应用系统提供的管理工具，使其运行时刻系统中的组件对象能够加载这个序列化文件。这个由组件设计器生成和修改的组件对象的序列化文件也称为组件设计文档。

组件设计器在资源管理工作台系统上即已提供，因此，组件设计文档是资源管理工作台系统可查看和编辑的一种文件，其后缀为component。

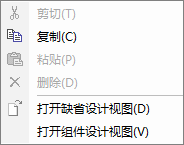
点击”打开”/”文件”菜单项，在Windows标准的文件对话框中选中所需编辑（或查看）的组件设计文档，并点击”确定”按钮（或者点击”新建/文件”菜单项，在”新建文件”对话框中选择一个组件设计文档模板，并点击”打开”按钮），组件设计器即会依据组件设计文档根节点的名称和组件工厂中登记的组件类型，创建一个组件对象，并根据组件设计文档中的数据设置其状态，同时，在工作区中打开一个内嵌了缺省设计视图的组件设计子窗口，用于查看和设置该组件对象的状态。对应同一个组件对象，组件设计器可能会提供多种类型的设计视图，所有类型的设计视图通常都会依据组件对象的构造图，为用户提供一个比较高效、智能的编辑界面，用于查看和设置对应的组件对象和其下的子组件对象的状态，而不是直接编辑XML文件。



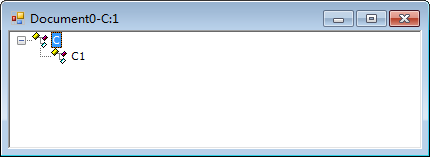
组件设计器新增了若干命令，并且对基本工作台系统提供的部分命令的功能进行了扩展，用以对工作区中当前活动的内嵌了设计视图的组件设计子窗口中唯一选定的组件对象（以下简称焦点组件对象）进行设计操作，包括：

|  |  |  |  |
| --- | --- | --- | --- |
| 名称 | 图标 | 快捷键 | 功能 |
| 剪切 |  | Ctrl+X | 删除焦点组件对象，并将其保存到剪贴板。注意只有集合型组件对象下的子组件对象才可以被删除 |
| 复制 |  | Ctrl+C | 复制焦点组件对象，并将其保存到剪贴板 |
| 粘贴 |  | Ctrl+V | 复制剪贴板中保存的组件对象，添加到焦点组件对象下。注意只能对集合型组件对象执行粘贴操作，并且剪贴板中保存的组件对象类型必须符合要求 |
| 删除 |  | Del | 删除焦点组件对象，和剪切不同，被删除的组件对象不会被保存到剪贴板 |
| 打开缺省设计视图 |  |  | 在工作区中打开一个内嵌了对应焦点组件对象的缺省管理视图的组件设计子窗口 |
| 打开组件设计视图 |  |  | 在工作区中打开一个内嵌了对应焦点组件对象的组件管理视图的组件设计子窗口 |

在焦点组件对象上点击鼠标右键，通常都会出现一个弹出式菜单，它的菜单项绑定了用户对该对象可以发出的所有组件设计器命令。对于适用于所有类型组件对象的命令，组件对象右键菜单几乎都有与之绑定的菜单项，例如：”剪切”、”复制”、”粘贴”、”删除”、”打开缺省设计视图”、”打开组件设计视图”等菜单项，它们分别绑定了”剪切”、”复制”、”粘贴”、”删除”、”打开缺省设计视图”、”打开组件设计视图”等命令。



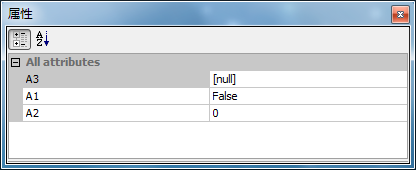
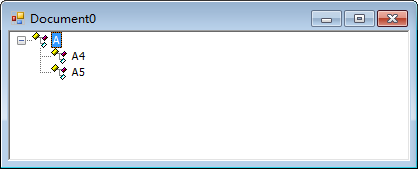
点击组件对象右键菜单中的”打开组件设计视图”菜单项，会在工作区中打开一个内嵌了对应此前焦点组件对象的组件设计视图的组件设计子窗口。



请注意，工作区中的这个组件设计子窗口具有一个特别的标题，标题的格式是”文件名[-(子组件成员对象构造图登记名|集合型组件对象下的子组件对象的类型名:序号){/(子组件成员对象构造图登记名|集合型组件对象下的子组件对象的类型名:序号)}]”，用以根据组件设计文档中的组件对象的构造图关系准确定位一个子组件对象，其机制类似于我们常用的文件路径名，因此这个格式也称为构造图路径格式。

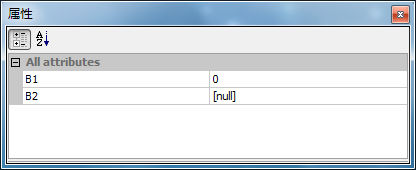
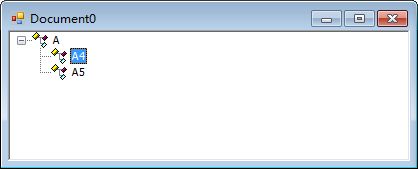
”组件设计视图”是常用的、最基本的、适用于所有类型组件对象的一种设计视图，也是许多CBF组件对象的缺省设计视图。组件设计视图是一个标准的TreeView，组件设计器在组件设计视图中使用根节点（TreeNode）来表示该设计视图对应的组件对象。

当用户选中根节点时，会在”属性”窗格中看到对应的组件对象具有的可查看和设置值的属性，它们与组件对象中通过SetElement方法登记了的Element派生类成员相对应。例如，对应上面的示例中A类型的组件对象，可看到其分别名为”A1”，”A2”，”A3”的3个属性。



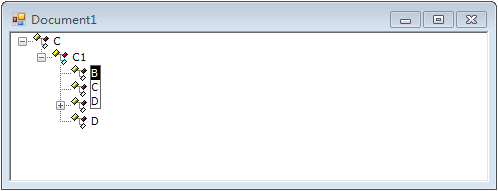
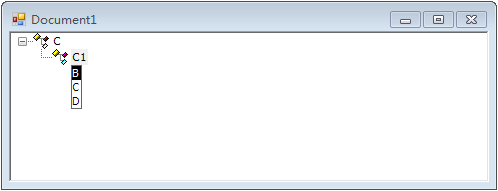
用户在组件设计视图中还会看到组件对象聚合的子组件对象（根节点下的子节点），它们与组件对象中通过SetComponent方法登记了的Component派生类成员相对应。例如，对应上面示例中的A类型的组件对象，可看到其分别名为”A4”，”A5”的2个B类型的子组件对象。这些由各层节点共同构成的树状结构向用户直观地展现了组件对象的构造。

选中与子组件对象对应的子节点，同样会在”属性”窗格中看到子组件对象具有的可查看和设置值的属性。

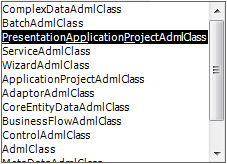


对于可动态聚合子组件对象的组件对象——集合型的组件对象，我们可以通过组件设计视图来删除或添加子组件对象。

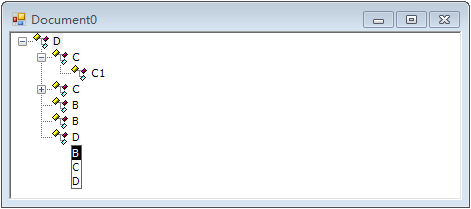
例如，对应上面示例中的C类型组件对象，在组件设计视图中选中”C1”子节点，按”Shift+Insert”键，即会在”C1”子节点之下的最后一个子节点的下方（如果”C1”子节点还没有子节点，则会在”C1”子节点的下方）出现一个（根据组件工厂登记的组件类型）列举了所有可添加的子组件对象类型的选择器窗口。在选择器窗口中选中一个类型，并按”Enter”键，即可完成子组件对象的创建和添加（这个新的子组件对象会被添加到子组件对象集合的最后一个位置）。如果”C1”子节点已经有了一些子节点，那么，选中其中一个子节点，按”Delete”键，即可完成对应的子组件对象的删除；按”Insert”键，则会在选中的子节点处出现一个选择器窗口，同样，选中一个类型，并按”Enter”键，也可完成子组件对象的创建和添加（这个新的子组件对象会被插入到子组件对象集合当前选中的子组件对象所在的位置）。



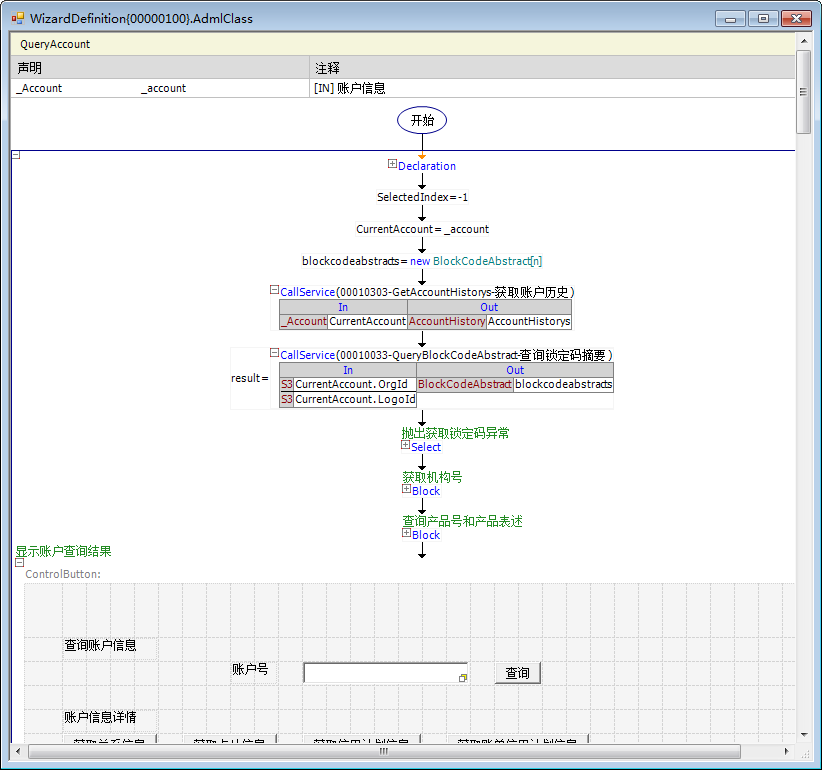
*选择器窗口是组件设计器提供的一个特殊的列表窗口，列表中当前选中的项目用反色（黑底白字）表示。用户即可以用上、下方向键来遍历所有项目，也可以输入字符，选择器窗口会自动检索所有项目中与输入字符最匹配的，将其设置为当前选中的项目，并在该项目文字下方用下划线标出匹配的输入字符（按”Backspace”键可清除最后一个匹配的输入字符）。按”Esc”键可关闭选择器窗口并取消当前操作。*



对应上面示例中的D类型组件对象，组件设计视图展现的构造更为简洁、理想——根节点下的子节点直接对应子组件对象。在组件设计视图中选中根节点，按”Shift+Insert”键即可进行添加子组件对象的操作。其他操作同C类型组件对象。



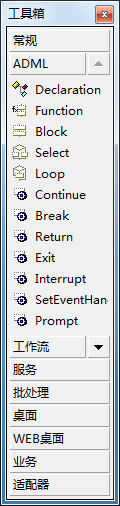
点击”打开缺省设计视图”菜单项，也会在工作区中打开一个内嵌了对应此前焦点组件对象的设计视图的组件设计子窗口，标题同样是构造图路径格式，但这里的设计视图有可能不是组件设计视图，而是对于此前焦点组件对象而言更适用的设计视图（例如，AdmlClass类型对象的图形设计视图）。采用哪种类型的设计视图作为缺省设计视图，取决于焦点组件对象的类型。图形化的设计视图通常都会使用一个深蓝色的方框标示出该视图中唯一选定的组件对象，并可通过鼠标点击来选择组件对象。



组件设计器支持在同一设计视图内部，以及不同设计视图之间的拖放操作。需要注意的是，拖放操作的目标组件对象必须是集合型组件对象（如果是容器型集合组件对象，还必须保证其下没有子组件对象），而源组件对象的类型必须符合要求。默认的拖放操作是复制源组件对象+粘贴到目标组件对象的操作组合，如果按住”Shift”键，则拖放操作是剪切源组件对象+粘贴到目标组件对象的操作组合。

相对在组件设计视图中按”Insert”键或”Shift+Insert”键，拖放操作是一种更加普遍适用的添加、删除子组件对象的操作方法，尤其是在图形化的设计视图中添加需要设定几何位置的子组件对象，拖放操作几乎是唯一可行的操作方法。

我们还可以基于工具箱进行拖放操作。工具箱是指组件设计器在资源管理工作台系统上创建的一个名为”工具箱”的窗格，这个窗格可管理并展示多个栏目，在栏目下设置一组项目，每个项目都对应一个常用的组件对象，当我们试图拖动一个项目时，其对应的组件对象会立即被复制到剪贴板。在”工具箱”窗格中，点击栏目的标题可展开和收起栏目，如果展开时由于项目数量过多而不能全部显示，还可点击栏目标题右侧的具有三角形图标的按钮，向上或向下翻阅项目。



组件对象右键菜单还会提供绑定了仅适用于当前焦点组件对象类型的设计命令的菜单项。例如，如果当前焦点组件对象是AdmlClass类型的，那么，其右键菜单中就会看到”添加函数”等菜单项。

#### 组件对象寄存器

#### 元素对象集合和元素工厂

#### 字符串枚举和可设定规则的字符串校验器

#### 可作为组件对象运用的XML文档对象

# 调度器

调度器（Schduler）模块以UCP平台...组件为基础构建，用于为基于图形化的设计器和脚本快速编排、组合应用处理逻辑和控制应用处理任务提供设计和运行时支持。

# 通信应用框架

通信应用框架（Communication Application Framework，简称CAF）模块致力于构建具有强大处理性能、适配能力和可扩展性，运行高度稳定可靠的通信基础设施，以及可以对任意拓扑结构的分布式应用提供通信支持的应用框架。

通信应用框架模块由：1、通信管理框架；2、常用适配器；3、应用框架；4、外围通信框架；共4个子模块构成——



## 通信管理框架

通信管理框架子模块是通信应用框架模块的核心子模块，它确立了以消息、外部服务器和路由器等对象为核心的整个服务器端通信应用对象体系的基本结构，建立了具体通信目标系统、数据收发方式、报文编组方式、数据加工应用的描述、登记和发现机制。

该框架具有以下特性——

* 统一管理的通信缓冲区、应用（路由器）线程池和流量控制
* 向应用提供基于外部服务器和消息对象的通信接口，而具体数据收发方式、报文编组方式等技术细节均对应用完全透明且可灵活扩展

## 常用适配器

常用适配器子模块提供了一组可装配到通信管理框架上的、用以应对常见的数据收发方式（基于TCP/IP的Socket长连接，短连接，等等）的通信适配器，以及常见的报文编组方式（固定报文头，以总长度开始，等等）的编组器。

基于TCP/IP的通信适配器均具有以下特性——

* 支持主动、被动两种通道建立模式，并自动、定时维护通道
* 专用线程（不占用应用线程）完全基于无阻塞机制执行IO操作（NIO）

## 应用框架

应用框架子模块扩展了通信管理框架子模块中的应用（路由器）模型，以方便几种常见模式通信应用（消息处理应用，服务应用，等等）的设计。

## 外围通信框架

外围通信框架子模块提供了在分布式应用架构中相对处于外围的（往往只需和1个相对处于中心的服务器采用指定的、专用的协议通信）服务器上的通信管理和应用框架。

# 基础服务

基础服务（Foundation Service，简称FS）

基础服务访问代理也能提供给本地应用使用，这是应用平台中的一个常见的设计模式。

## 共享资源服务

## 同步服务

## 消息队列服务

## 时钟服务

## 信号服务

# 数据访问框架

数据访问框架（Data Access Framework，简称DAF）模块致力于通过ORM等引擎、框架帮助应用中的数据访问对象类（Data Access Object，简称DAO）以极其简洁的方式为上层应用提供统一的、无需编写SQL的数据访问接口。

数据访问框架模块由：1、ORM引擎；2、综合数据访问框架；共2个子模块构成——



## ORM引擎

ORM引擎子模块提供了一个帮助DAO类实现对象-关系数据库映射（Object Relational Database Mapping，简称ORM，即为上层应用提供统一的数据访问接口，Select，Update，Delete，Insert，等等，使得数据库结构对上层应用透明，应用无需编写SQL语句来获取、写入数据）方法的引擎。

相对于常见的开源的ORM引擎（例如Hibernate），该引擎具有以下特性——

* 代码精干（主要逻辑<2000行），便于维护
* 基于专用结构和接口运作，而非基于反射，执行高效、安全，机制灵活
* DAO类接口的实现非常简洁，工作量很低，通常由开发工具自动生成，发布的DAO类包使用方便，无需配置文件，随处可用
* 为结合了OO设计理念的数据模型提供更有效的支持
* 完全适应中信银行分布式数据库和核心系统应用层面的特性要求，不仅支持基于主键和过滤条件的访问，还支持基于索引的访问，支持基于指定域组的访问，支持链接级缓存，支持分片键转换，支持与DAO绑定的SqlDataTable和SqlCursor，支持临时表映射，支持分页检索

## 综合数据访问框架

# 服务应用框架

服务应用框架（Service Application Framework，简称SAF）模块以通信应用框架模块为基础构建，致力于为服务应用——即按请求/应答模式设计的、面向所有外部系统开放接口的服务器端业务处理应用（例如联机交易应用）——的开发、运行提供应用框架形式的完备的支持、约束，统一解决与业务无关的技术层面的问题。

#### 服务系统集群

基于服务应用框架搭建的服务系统可以部署成服务系统集群，这个集群通常由N个服务处理系统、M个异步任务处理系统和1个服务控制系统等应用系统（逻辑节点）共同构成：



其中，服务控制系统面向运行管理平台，处理其对整个服务系统集群发出的运行管理命令，并向其转发整个服务系统集群运行时产生的各种监控消息，同时，面向服务处理系统提供多阶段事务处理过程中产生的一致性问题的处理（补偿处理）等服务；服务处理系统面向客户端系统，处理其提交的服务请求；异步任务处理系统面向服务处理系统，处理其在服务请求处理过程中产生并提交的异步任务。

#### 模块内部结构

服务应用框架模块由：1、接口数据对象模型和接口管理；2、环境资源信息访问对象；3、应用框架；4、处理器和代理；共4个子模块构成：

接口数据对象模型和接口管理子模块确立了服务接口数据对象体系的基本结构，建立了服务接口的描述、登记和发现机制。

环境资源信息访问对象子模块提供了一组用以获取、设置服务系统集群运行时所需环境资源、配置等信息的DAO。

应用框架子模块确立了服务应用对象体系的基本结构，建立了应用功能实现的描述、登记和发现机制，在系统层面和具体服务请求处理层面提供了支持，使应用设计可以得到最大程度的简化。其中，由于在某些环节上，应用框架子模块的内部设计相对自成一体，因此，应用框架子模块又细分出：1、中断/重入支持；2、异步任务支持；3、多阶段事务处理应用模型；等3个子模块。

处理器和代理子模块提供了通信层组件。处理器用于解析外部系统提交的请求报文，调用应用框架接口进行处理，并组装应答报文回发给外部系统。代理是为外部系统提供的一组支持其基于接口数据对象进行访问的API。



#### 运行时子模块之间的协作

服务系统集群维护管理时刻相关系统、子模块之间主要的协作关系为：

1、运行管理平台上的应用调用环境资源信息访问对象子模块提供的接口获知、设置服务系统集群运行时环境资源、配置等信息；

2、运行管理平台上的应用通过处理器和代理子模块提供的代理访问服务控制系统，向其发出运行管理命令；

3、服务控制系统上部署的处理器和代理子模块中的处理器调用应用框架子模块提供的接口，让其执行运行管理平台发来的运行管理命令；

4、服务控制系统上应用框架子模块调用环境资源信息访问对象子模块提供的接口获知服务系统集群运行时环境资源信息，并通过处理器和代理子模块提供的代理访问服务处理系统等集群中所有节点，向它们转发运行管理平台要求执行的运行管理命令；

5、服务处理系统上部署的处理器和代理子模块中的处理器调用应用框架子模块提供的接口，让其执行服务控制系统发来的运行管理命令；

6、服务处理系统上应用框架子模块调用环境资源信息访问对象子模块提供的接口获知服务处理系统运行时环境资源、配置等信息，执行运行管理命令相应的处理过程。



具体服务请求处理时刻相关系统、模块之间主要的协作关系为：

1、客户端系统上的应用调用接口数据对象模型和接口管理子模块提供的接口获知所需访问的服务系统的接口描述，并创建请求数据；

2、客户端系统上的应用通过处理器和代理子模块提供的代理访问服务处理系统，向其发出服务请求；

3、服务处理系统上部署的处理器和代理子模块中的处理器调用接口数据对象模型和接口管理子模块提供的接口，解析、校验客户端系统发来的服务请求报文，将其转换为请求数据，并调用应用框架子模块提供的接口，让其处理客户端系统发来的服务请求；

4、服务处理系统上应用框架子模块调用应用提供的接口，让其执行服务请求在应用层面的处理过程。应用接口处理过程中会调用接口数据对象模型和接口管理子模块提供的接口，创建应答数据；

5、服务处理系统上部署的处理器和代理子模块中的处理器将应用框架子模块接口返回的应答数据转换成服务应答报文，回发给客户端系统。客户端系统上处理器和代理子模块中的代理调用接口数据对象模型和接口管理子模块提供的接口，解析、校验服务处理系统发来的服务应答报文，将其转换为应答数据，返回给客户端系统上的应用。



## 接口数据对象模型和接口管理

接口数据对象模型和接口管理子模块确立了服务接口数据对象体系的基本结构，建立了服务接口的描述、登记和发现机制。

接口数据对象模型主要由应用标识（ApplicationIdentifier）、请求标识（RequestIdentifier）、请求文档（RequestDocument）、处理错误码（ProcessError）、应答标识（ResponsesIdentifier）和应答文档（ResponseDocument）等数据构成。



#### 请求数据

客户端系统调用服务接口时必须发送以下请求数据：

1. 应用标识，是所需执行的某个应用功能的唯一标识（交易系统开发人员通常也称之为交易码）。应用标识必须是8位长度由数字构成的字符串。应用标识的唯一性是相对于所访问的服务系统上的全部应用功能而言的——服务系统A上应用标识“00009001”所对应的应用功能，和服务系统B上应用标识“00009001”所对应的应用功能，并不一定是同一个。
2. 请求标识，是客户端系统产生的某一次服务接口调用的唯一标识（交易系统开发人员通常也称之为前端流水号）。请求标识由客户端标识（ClientIdentifier）和客户端调用标识（ClientInvokingIdentifier）两部分内容构成。客户端标识是某一个客户端系统的唯一标识。客户端标识必须是8位长度由数字构成的字符串。客户端标识的唯一性是相对于所访问的服务系统能够接入的所有客户端系统而言的——服务系统A能够接入的客户端标识“80000001”所对应的客户端系统，和服务系统B能够接入的客户端标识“80000001”所对应的客户端系统，并不一定是同一个。客户端调用标识则是相对于某一个客户端系统发出的全部服务接口调用，其中某一次调用的唯一标识。客户端调用标识必须是18位长度由数字构成的字符串。
3. 请求文档，是一个RequestDocument派生类（最终基类是Component）对象，包含了由应用标识所标定的应用功能执行时所需输入的全部数据。请求文档包含的内容都是仅用来应对具体应用层面处理需要的，服务系统框架层面处理需要的应用标识、请求标识等等，都不是请求文档的内容。具体应用功能所需输入的请求文档类型是限定的。例如，QueryAccount应用功能限定输入的请求文档必须是QueryAccountReqestDocument类型，因此，客户端系统调用服务接口时，如果其应用标识标定的是QueryAccount应用功能，而请求文档并非QueryAccountReqestDocument类型，则服务请求将不能得到正常的处理——需要注意的是，输入QueryAccountReqestDocument派生类型的请求文档是可以的，此外，不同应用功能限定的请求文档类型有可能是同一个。

#### 应答数据

服务系统完成处理后，会向客户端系统回送以下应答数据：

1. 处理错误码，表示服务请求是否得到正常处理。注意，这里的“正常”并不意味着“交易成功”，只要是获得了指定应用功能执行后产生的处理结果，均被视作正常，包括处理结果是“交易失败”的情况。处理错误码是一个整数，0表示正常，非0表示“系统暂停服务”、“系统异常”、“请求标识重复”、...等异常。
2. 应答标识，是服务系统产生的某一次服务接口调用的唯一标识（交易系统开发人员通常也称之为交易日志号）。应答标识由应答日期（ResponseDate）和应答序号（ResponseSequence）两部分内容构成。应答日期是所访问的服务系统认定的该次服务接口调用的处理日期，是符合“YYYY-MM-DD”格式的10位长度的日期字符串。应答序号则是相对于所访问的服务系统某一个应答日期内处理的所有服务接口调用，其中某一次调用的唯一标识。应答序号是14位长度由数字构成的字符串。
3. 应答文档，是一个ResponseDocument派生类（最终基类是Component）对象，包含了由应用标识所标定的应用功能执行后会输出的全部数据。应答文档包含的内容都是仅用来表达具体应用层面处理结果的，服务系统框架层面处理产生的处理错误码、应答标识等等，都不是应答文档的内容。具体应用功能在处理成功的情况下能够输出的应答文档类型是限定的。例如，QueryAccount应用功能限定输出的应答文档是QueryAccountResponseDocument类型，因此，客户端系统调用服务接口时，如果其应用标识标定的是QueryAccount应用功能，则处理成功情况下其能获得的应答文档只会是QueryAccountResponseDocument类型——需要注意的是，不同应用功能限定的应答文档类型有可能是同一个。

应用处理失败时，都必须输出RejectResponseDocument类型（也是ResponseDocument派生类）的应答文档，应用可以设置其OperationHint属性值（字符串类型），表达失败的原因（通常是一个失败描述，或者是一个错误码），并向客户端操作人员提示处理的方法。

#### 服务声明注册器和服务接口声明

客户端系统可以通过服务声明注册器（ServiceDeclarationRegister）对象获知所访问的服务系统当前能够提供哪些应用功能，并且基于应用标识，调用ServiceDeclarationRegister对象的GetServiceInterfaceDeclaration方法，就可以查找到描述应用功能接口的服务接口声明（ServiceInterfaceDeclaration）对象。

ServiceInterfaceDeclaration对象的RequestDocumentType和ResponseDocumentType属性值（均为字符串类型）就是相应应用功能所限定的请求文档类型名和应答文档类型名。

#### 服务描述

每一个业务处理功能相对完整的应用模块开发完成时通常都应包含一个服务描述类（ServiceDescription派生类）。服务描述类负责把描述该应用模块所提供的全部应用功能的接口的ServiceInterfaceDeclaration对象按既定的应用标识登记到ServiceDeclarationRegister对象中（实现InitializeServiceDeclarationRegister方法接口）。客户端系统只需基于应用模块提供的服务描述类，实例化服务描述对象，并调用其InitializeServiceDeclarationRegister方法，即可完成ServiceDeclarationRegister对象的初始化。

服务描述类InitializeServiceDeclarationRegister方法的一般实现方式为：



## 环境资源信息访问对象

环境资源信息访问对象子模块提供了一组用以获取、设置服务系统集群运行时所需环境资源、配置等信息（例如，有哪些服务处理系统节点，所在物理节点的端口号，地址，等等）的DAO——与用配置文件来存放相关信息这种传统的做法不同，服务应用框架模块把几乎所有相关信息都存放在数据库中。

#### 服务中心对象

服务中心（ServiceCenter）对象是这里最重要的一种环境资源信息访问对象，它对应一个已经部署好的服务系统集群，其相关属性内容描述了对应服务系统集群的基本设置，包括要加载哪些应用模块（ServiceProjectCollectionConfigName属性），应用运行时的参数设置（ServiceRuntimeSettingConfigName属性），由哪些服务描述组件来登记服务接口声明（ServiceDescriptionCollectionConfigName属性），对外提供服务所遵循的标准（ServiceType属性），日志数据库的链接字符串（SqlServiceConnectionString属性），等等。

#### 配置对象

配置（Config）对象是被广泛运用的一种环境资源信息访问对象，它通常对应一个被持久化了的组件对象，可以被看作是保存了该组件对象组件状态的文件。通常，我们也把存放在Config对象里的组件对象，称为“配置组件对象”。除了Name属性外，Config对象最主要的两个属性是Type和Content。Type属性值就是配置组件对象的关键基类名称，Content属性值则是通过调用配置组件对象的OXM方法形成的XML文本——在数据库中，Content属性对应的域类型是ntext，可以存放长度不受限制的XML文本数据。

对应服务系统集群的ServiceCenter对象的ServiceProjectCollectionConfigName、ServiceRuntimeSettingConfigName、ServiceDescriptionCollectionConfigName、...等属性引出的内容所含数据量可能较大，并且可能会被多个服务系统集群所共用，因此，我们把这类属性引出的内容封装成数据组件，并用Config对象来保存这些数据组件对象，属性值仅仅保存Config对象的名称。需要注意的是，这是一种具有普遍意义的设计思路，采用这种思路设计的环境资源信息访问对象类不仅仅是ServiceCenter。

#### 服务中心组件对象和应用服务器对象

ServiceCenter对象将对应一组服务中心组件（ServiceCenterComponent）对象，后者是对应服务系统集群中具体逻辑节点，描述具体逻辑节点基本设置（例如，在物理节点上的安装位置，逻辑节点序号，等等）的环境资源信息访问对象。有三种ServiceCenterComponent派生类对象：服务处理组件（ServiceProcessComponent）对象、服务控制组件（ServiceControlComponent）对象和异步任务处理组件（AsynchronousTaskProcessComponent）对象，分别与服务处理系统、服务控制系统和异步任务处理系统三类逻辑节点对应。

应用服务器（ApplicationServer）对象是用来对应物理节点，描述物理节点基本设置（IP地址，管控端口，等等）的环境资源信息访问对象。不同逻辑节点可以被部署到同一个物理节点上，因此，一个ApplicationServer对象与一组ServiceCenterComponent对象对应。

#### 环境资源服务和访问代理

为方便外部系统（指那些不能直接访问环境资源信息数据库的系统，例如运行管理平台）访问这些环境资源信息，环境资源信息访问对象子模块提供了一个基础服务——环境资源服务（EnvironmentResource），这是通信应用框架的应用插件，它解析外部系统提交的环境资源信息访问请求报文，调用其内部基于环境资源信息访问对象方法进一步封装的API（GetResource方法、GetConfig方法、GetConfigComponentArrayList方法、...等）进行处理，并组装环境资源信息访问应答报文回发给外部系统。

外部系统可以基于基础服务访问协议对其进行访问——在基础服务访问请求报文应用区中填写环境资源服务能够处理的指令和相关输入参数，并在基础服务访问应答报文应用区中读取环境资源服务的指令处理结果和其他输出参数。这些指令和输入、输出参数通常都是中/英文词或词组，一些具有复杂结构的输入、输出参数则会用XML格式字符串表示，都用ASCII编码填写并用空格分隔。

环境资源服务支持GetConfig、GetDevice、GetApplicationComponent等命令，这些命令的作用，以及相关参数的含义、格式如下：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 命令 | 作用 | 输入参数表 | 处理结果 | 其他输出参数表 |
| GetDevice |  |  |  |  |

为进一步简化外部系统访问环境资源服务时的编程，环境资源信息访问对象子模块还提供了环境资源服务访问代理（EnvironmentResourceService），它包含了一组API，这些API自动建立、维护通信通道，并自动完成符合环境资源信息访问报文格式要求的请求报文的组装和应答报文的解析。环境资源服务访问代理也是按本地优先的原则设计的，因此，即便是和环境资源服务在同一线程中执行的上层应用，也大多会使用环境资源服务访问代理来访问环境资源信息。

## 应用框架

应用框架子模块是服务应用框架模块的核心子模块，它确立了服务应用对象体系的基本结构，建立了应用功能实现的描述、登记和发现机制，在系统层面，为应用和配置的加载，运行时管控（流量限制，运行级别，日志级别，等等）和热更新提供统一支持，在具体服务请求处理层面，统一完成资源（数据库链接，事务，日志号计数器，等等）准备和释放、应用对象实例化和协作、日志记录（支持堵重）、异常处理、处理过程监控，并统一对外提供查询处理结果、重做、冲正（也称补偿处理、撤销）、自动冲正（异步模式的冲正，也称异步补偿处理）等服务接口，使应用设计可以得到最大程度的简化。

框架（Framework）类是应用框架子模块的核心设计。Framework类的设计可以分为静态和非静态两大部分，静态设计用以应对系统层面统一支持的需要，非静态设计则用以应对具体服务请求处理层面统一支持的需要。

* + 1. Environment对象

Framework类管理的最重要的全局静态对象是一个Environment内部类对象，该对象聚合了在服务系统生命周期内可能会被运行管理操作变更的所有环境资源对象，包括：一个ComponentFactory对象，一个服务描述集合（ServiceDescriptionCollection）对象，一个ServiceDeclarationRegister对象，一个配置组件对象集合（ConfiguredComponentCollection）对象，一个项目集合（ProjectCollection）对象和一个运行时设置（RuntimeSetting）对象。Framework类提供了相应的用以获取这些环境资源对象的静态属性（属性名称与访问属性获得对象的类名相同）。

具体服务请求处理时，会首先通过一个非静态的变量引用这个全局静态Environment对象，其后，其处理过程都会使用那个非静态的变量来获取相关环境资源对象。由于在服务系统生命周期内可能会被运行管理操作变更的所有环境资源对象都被放入了Environment对象，因此，如果我们需要对服务系统进行热更新，只需创建一个新的Environment对象，并在基于最新版本的包、配置完成其初始化后，将Framework类中的静态变量指向新的Environment对象即可——在此期间开始的具体服务请求的整个处理过程都会使用更新前的Environment对象，所获得的相关环境资源对象版本是一致的。

Environment对象是依据配置文件和与服务系统集群对应的ServiceCenter对象的相关属性内容来完成主要的初始化过程的。例如，基于组件工厂配置文件完成Enviroment对象中ComponentFactory对象初始化；依据ServiceCenter对象的ServiceRuntimeSettingConfigName属性值、ServiceDescriptionCollectionConfigName属性值和ServiceProjectCollectionConfigName属性值获得相应的Config对象，并利用其中保存的组件对象——RuntimeSetting对象、ServiceDescriptionCollection对象和ProjectCollection对象——来完成Enviroment对象中相应对象的初始化。

#### 组件工厂对象

Environment对象中的ComponentFactory对象被用来代替new方法，完成服务系统运行时刻几乎所有组件对象的动态创建。接口数据被封装成数据组件类后，从请求报文转换成接口数据对象的过程，就会利用这个ComponentFactory对象，实现限定基类的数据对象的多态接收。此外，把应用内部的业务处理逻辑封装成应用组件类，就可利用这个ComponentFactory对象，完成应用组件对象的实例化，这种实例化方式能够让热更新机制工作起来更加干净彻底，同时也更有利于应用包的合理组织。

#### 服务描述集合对象和服务声明注册器对象

ServiceDescriptionCollection对象就是ServiceDescription对象的集合，服务系统启动时会依次调用Environment对象中的ServiceDescriptionCollection对象包含的所有ServiceDescription对象的InitializeServiceDeclarationRegister方法，完成Environment对象中的ServiceDeclarationRegister对象的初始化——服务请求报文合法性的检查过程会使用到Environment对象中的ServiceDeclarationRegister对象。

#### 配置组件对象集合对象

服务系统启动时会读取所有Type属性值为"component"的Config对象，将其中保存的配置组件对象加入Environment对象中的ConfiguredComponentCollection对象。ConfiguredComponentCollection对象内部建立了一个以Config对象的Name属性值为键值的配置组件对象哈希表，ConfiguredComponentCollection对象提供了GetConfiguredComponent方法，可以根据Config对象的Name属性值在该哈希表中查找到配置组件对象。配置组件对象通常被用作服务系统中所有应用模块在具体服务请求处理时共用的参数——这意味着我们在通过组件对象设计器设计配置组件对象时即可决定运行时服务系统的相关参数。

#### 项目、功能实现和应用

应用框架子模块提供了Project类和Application类，共同构成了服务应用模型的基础。任何一个基于服务应用框架进行的业务处理功能相对完整的应用模块的开发工作，都需要定义一个项目类（Project派生类）和一组应用类（Application派生类），并实现其接口。

项目类最重要的接口是模块功能实现（ModuleImplementation）属性，该接口的实现要求返回一个模块功能实现（ModuleImplementation）对象，ModuleImplementation类是功能实现（Implementation）类的派生类，其内部又定义了一个功能实现对象集合成员，因此，可以把ModuleImplementation对象看作一棵Implementation对象树的中间节点。

Implementation类是纯虚类，真正可以实例化的Implementation类，除了ModuleImplementation类，还有应用功能实现（ApplicationImplementation）类，实际上，后者的实例才是Implementation对象树上更关键的、能够提供具体应用功能实现信息的对象——它不只具有所有Implementation对象都具有的Identifier（即应用标识）属性和Description属性，还有Application属性，访问Application属性即可获得执行相应应用功能的原型应用对象。

项目类ModuleImplementation（属性）接口的一般实现方式为：



#### 项目集合对象

ProjectCollection对象就是Project对象的集合。完成开发的应用模块，如果要加载到某个服务系统集群中运行，只需通过组件对象设计器把相关项目对象添加到相应ServiceCenter对象ServiceProjectCollectionConfigName属性值所指向的Config对象内含的ProjectCollection对象中即可。

项目类是典型的组件类（最终基类是Component），Environment对象中的ProjectCollection对象包含的所有Project对象在服务系统生命周期内又是一直存在的，因此，我们可以在项目类中定义一些可配置数据成员，作为应用模块内部具体服务请求处理时共用的参数——这意味着我们在通过组件对象设计器把Project对象添加到ProjectCollection对象时即可决定运行时服务系统中的由Project对象描述的参数（除了这些可配置数据成员外，项目类中还可以定义一些由相应应用模块自行管理的状态数据和环境资源对象成员，项目类可以提供相应的用以获取这些成员数据或对象的属性）。

为便于访问Environment对象中的ProjectCollection对象包含的Project对象，服务系统启动时会在Enviroment对象中创建一个以这些Project对象的Name属性值为键值的Project对象哈希表，Framework类提供了GetProject静态方法，可以根据这些Project对象的Name属性值在该哈希表中查找到Project对象。

#### 应用注册器对象

Enviroment对象管理了一个以应用标识为键值的应用注册器（ApplicationRegister）对象哈希表（与交易系统开发人员通常具有的交易注册表概念相近），其中，一个ApplicationRegister对象对应一个应用功能。ApplicationRegister对象内部管理了相应应用功能的实时流量统计值、流量控制阈值、运行级别等数据，并关联了一个用以执行相应应用功能的原型应用对象。服务系统启动时会依据从Environment对象中的ProjectCollection对象包含的所有Project对象获得的ApplicationImplementation对象，完成Environment对象中的ApplicationRegister对象哈希表的初始化。Framework类提供了GetApplicationRegister静态内部方法，可以根据应用标识在该哈希表中查找到ApplicationRegister对象。

#### 项目关联配置组件对象

在具体应用模块开发工作中，往往需要把项目的参数分成两个层面，一是服务系统完成初始化后即不再变化的参数，二是服务系统生命周期内会被运行管理操作变更的参数。仅仅依靠项目类组件对象模型层面的初始化和更新方法难以适应分层管控参数的需要，因此，我们通常用项目类的可配置数据成员满足第一个层面的需要，而为满足第二个层面的需要，我们要求每个Project对象都关联一个含有所需可配置数据成员的组件对象——我们称之为项目关联配置组件对象，服务系统运行时，可以调用这个对象的初始化和更新方法，更新其中的参数值，而不会对Project对象造成任何影响。

访问Project对象的ConfigedComponent属性可以获得相应的项目关联配置组件对象，ConfigedComponent属性是要求项目类实现的一个接口，因此，项目关联配置组件对象是什么类型，完全由项目类的实现决定——在应用模块开发工作中，对应每个项目类，几乎都会定义一个项目关联配置组件对象类。服务系统启动时也会在Enviroment对象中创建一个以Project对象的Name属性值为键值的项目关联配置组件对象哈希表，Framework类提供了GetConfigedComponent静态方法，可以根据Project对象的Name属性值在此哈希表中查找到相应的项目关联配置组件对象。

#### 运行时设置对象

项目关联配置组件对象可通过RuntimeSetting对象进行初始化和更新。RuntimeSetting对象内部聚合了一个项目运行时设置（ProjectRuntimeSetting）对象集合对象，其中包含的每个ProjectRuntimeSetting对象内部又聚合了一个项目关联配置组件对象。服务系统启动和热更新时刻，会获取Environment对象中的RuntimeSetting对象聚合的全部项目关联配置组件对象，用其作为原型，初始化Environment对象中的Project对象哈希表里的Project对象所关联的项目关联配置组件对象——这意味着我们在通过组件对象设计器设计Config对象中的RuntimeSetting对象时即可决定运行时服务系统中的由项目关联配置组件对象描述的参数。

应用类也是典型的组件类（最终基类是Component），因此，我们可以在应用类中定义一些可配置数据成员，作为应用内部处理时的参数。RuntimeSetting对象内部聚合了一个应用运行时设置（ApplicationRuntimeSetting）对象集合对象，其中包含的每个ApplicationRuntimeSetting对象内部又聚合了一个原型应用对象。服务系统启动和热更新时刻，会获取Environment对象中的RuntimeSetting对象聚合的全部原型应用对象，用其作为原型，初始化Environment对象中的ApplicationRegister对象所引用的原型应用对象——这意味着我们在通过组件对象设计器设计Config对象中的RuntimeSetting对象时即可决定运行时服务系统中的由应用对象描述的参数。

除了用以初始化和更新运行时Envionment对象中的项目关联配置组件对象和原型应用对象以外，RuntimeSetting对象还表达整个服务系统的日志级别、运行级别（运行时设置对象的JournalLevel属性值和RunLevel属性值），初始化和更新Environment对象中的ApplicationRegister对象的FluxLimit（流量控制阈值）、RunLevel（运行级别）等属性值（基于ApplicationRuntimeSetting对象的FluxLimit属性值和RunLevel属性值）。如果系统的日志级别高于应用代码中记录日志操作所申报的级别，则应用代码中记录日志的操作将被忽略。如果具体应用功能（对应的ApplicationRegister对象）的运行级别低于系统的运行级别，则该应用功能将不能被执行。

* + 1. 其他静态设计

除了Environment对象，Framework类还管理了一组服务系统在从启动到关停整个生命周期内都需要使用的静态的状态数据和环境资源对象，并提供了相应的用以获取这些对象的静态属性（访问属性获得对象的类名基本都与属性名称相同），包括：系统状态（OperationState属性，共有尚未启动/已经关停、正在启动、正在提供服务、正在关停4种状态），逻辑节点对应的环境资源信息（ServiceControlComponent属性、AsynchronousTaskProcessComponent属性、ServiceProcessComponent属性和ServiceCenter属性），用以与服务控制系统通信的外围通信框架（AroundServerFramework属性，如果当前节点就是服务控制系统节点，则该属性返回null）和所需调用的各种基础服务访问代理：服务控制服务（由服务控制系统提供的补偿处理服务、运行控制服务）访问代理（ServiceControlService），异步任务处理服务（由异步任务处理系统提供的异步任务处理服务）访问代理（AsynchronousTaskProcessService），等等。

Framework类还封装了一组静态方法，用以完成服务系统启动时的处理过程（StartService方法，由于Framework类对服务系统集群中所有类型逻辑节点都提供支持，因此，StartService方法要依据当前节点的类型执行不同的初始化操作），系统关闭时的处理过程（StopService方法），以及系统运行过程中的更新、查询（SetRuntimeSetting方法、LoadComponentClass方法、QueryState方法、...等）等运行管理操作的处理过程。

* + 1. 框架对象

服务系统在处理一个具体服务请求时，需实例化一个Framework对象，并调用Framework对象提供的服务请求处理方法——这个非静态的Framework对象其生命周期与具体服务请求在应用框架中的整个处理过程对应。

Framework对象的内部，管理了一组与具体服务请求处理对应的状态数据和环境资源对象，包括：服务请求处理周期内所引用的Environment对象，所有服务接口数据，日志序号和章节位置，数据库链接对象，入口标志，等等，访问Framework对象的属性（ApplicationIdentifier属性、RequestIdentifier属性、ResponseIdentifier属性、EnterpriseResourceSqlService属性、Entrance属性、...等）即可获得这些数据或对象。

#### InstanceFramework属性与框架类的自适应静态设计

为了简化应用的编写，Framework对象并不作为参数传递给应用对象，而是由Framework类提供了一个静态属性（InstanceFramework），可以让应用随时获得Framework对象——显然，不同线程访问InstanceFramework属性时获得的Framework对象不会是同一个，这就决定了这个属性决不可能基于一个普通的静态变量来实现，事实上，这是基于ThreadLocal对象实现的，服务系统会在某个时刻把实例化好的Framework对象放入一个静态ThreadLocal对象，其后，在同一线程中任何时刻，就可以通过这个静态ThreadLocal对象来获得先前实例化好的Framework对象。

有了InstanceFramework属性，某些原来必须以非静态方式声明的Framework类的属性和方法，为了方便运用，现在也都可以静态方式声明，这导致某些静态属性（主要是用以获得Environment对象，以及其下环境资源对象的属性）既可以用来获得全局静态的环境资源对象，也可以用来获得Framework对象中的环境资源对象，其内部执行逻辑取决于调用时的线程状态，如果线程已经实例化了Framework对象，则使用Framework对象中的环境资源对象，如果还没有创建，则使用全局静态的环境资源对象——这种设计模式我们也称为自适应静态设计。

#### BeginProcess/EndProcess方法

Framework对象提供的最主要的服务请求处理方法是BeginProcess/EndProcess，一般的服务请求都是通过调用Framework对象的BeginProcess/EndProcess方法完成处理。

BeginProcess/EndProcess方法是遵照微软提出的异步应用编程模型（Asynchronous Programming Modal，简称APM）设计的，这样的设计是为了让处理过程中有通信IO等长时间阻塞操作（例如调用外部系统服务接口）的应用**理论上有可能**获得使用有限线程资源完成超大流量的任务处理的能力——当应用处理过程需要较长时间等候某个条件得到满足（例如接收到外部系统的应答）才能继续时，可以先返回以释放当前线程，其后，当条件满足时再使用新的线程执行应用处理过程的后续操作。

APM通常要求应用提供一个Begin...方法和一个End...方法。Begin...方法需要返回一个实现了IAsynchronousResult接口的对象（即异步处理状态对象），此对象将通过其IsCompletedSynchronously属性值和IsCompleted属性值告知调用者应用处理过程是否已经结束，如果在Begin...方法返回时处理过程就已结束，那么，此对象的IsCompletedSynchronously属性值将为true，否则，调用者就需要在后续操作中通过检查其IsCompleted属性值是否为true来判定应用处理过程是否已结束。为了让调用者及时知道应用处理过程的进展，Begin...方法要求传入一个实现了ICallback接口的对象，应用处理过程有任何进展，包括已经完成，都有可能调用此对象的Callback方法通知调用者——因此，Callback方法的实现往往都是立即检查先前获得的异步处理状态对象的IsCompleted属性值，如果为true，则立即调用End...方法获得具体的应用处理结果——与传统的回调模式的异步应用编程模型相比，这个APM的优势在于调用者能够获知应用处理过程的中间进展，而不只是最终完成的结果。

Framework对象提供的BeginProcess方法的原型为：



BeginProcess方法的主要处理过程如下：

1、把当前Framework对象放入静态ThreadLocal对象，以便后续设计访问Framework类的InstanceFramework静态属性获得当前Framework对象。

2、创建一个被用作BeginProcess方法返回值的异步处理状态对象，并初始化Framework对象中的相应变量，以便后续设计能够不断更新这个异步处理状态对象。

3、依据BeginProcess方法参数中的请求数据初始化Framework对象中的相应变量。

4、创建应答标识并初始化Framework对象中的相应变量。

5、记录与“进入”事件对应的服务日志（日志数据库中的ServiceJournal表），并发送对应的服务处理监控消息。

6、依据应用标识检索到相应的ApplicationRegister对象，并以ApplicationRegister对象关联的应用对象为原型，克隆一个用来执行具体处理过程的应用对象（我们也称之为工作应用对象）。

7、创建业务数据库链接对象，初始化Framework对象中的相应变量，以便后续设计访问Framework对象的EnterpriseResourceSqlService属性获得业务数据库链接对象，并把该数据库链接对象设置为数据访问框架模块中的默认数据库链接对象。

8、以请求文档为参数调用工作应用对象的Process方法，把其返回的应答文档放入待返回的异步处理状态对象。

9、关闭业务数据库链接对象，调用Complete方法并返回异步处理状态对象——调用者获得该异步处理状态对象后即可知道服务请求处理过程已完成，调用者随后即可调用Framework对象的EndProcess方法来获得具体服务请求的处理结果。

BeginProcess方法中调用的Complete方法的原型为：



这个Complete方法执行时，静态ThreadLocal对象中的Framework对象引用将被清除，继而把处理错误码（参数nProcessError）放入待返回的异步处理状态对象，并设置异步处理状态对象的IsCompletedSynchronously属性值为true。

Framework对象提供的EndProcess方法的原型为：



EndProcess方法的主要处理过程如下：

1、记录与“返回”事件对应的服务日志（日志数据库中的ServiceJournal表），并发送对应的服务处理监控消息。

2、使用Framework对象和先前返回的异步处理状态对象中的相应变量，设置EndProcess方法参数中的应答数据。

#### 应用对象的Process方法和OnProcess接口

与Framework对象协作完成整个服务请求处理过程的，是工作应用对象。Framework对象会调用工作应用对象的Process方法（这是一个内部方法），Process方法则会调用该应用对象的OnProcess接口来完成具体服务请求在应用层面的处理过程。OnProcess接口调用前，Process方法会检测相应应用功能的运行级别和流量是否符合要求，如果不符合要求则立即返回一个内含运行级别或流量不符合要求等拒绝原因的RejectResponseDocument对象，如果符合要求则加计流量统计值，并在OnProcess接口调用后，减计流量统计值。

应用对象的OnProcess接口的原型为：



应用在实现OnProcess接口时，首先应当创建一个在完成所有预期处理过程后才会返回的应答文档，其类型应当和相应应用功能的ServiceInterfaceDeclaration对象的ResponseDocumentType属性值保持一致。其后，可以访问Framework对象的EnterpriseResourceSqlService属性获得业务数据库链接对象，执行所需的数据库操作，并在处理过程中依据数据库操作结果设置应答文档中的数据。如果由于某些业务规则不能满足，而无法完成所有预期的处理过程，则需立即创建一个RejectResponseDocument对象，设置其OperationHint属性值并返回。

#### 异常处理

OnProcess接口实现的处理过程通常无需自行捕捉异常。当异常发生时，应用对象的Process方法会统一捕捉异常，完成异常信息的记录（日志数据库中的ServiceException表）和异常监控消息的发送，并返回一个OperationHint属性值被设置为"系统异常，请联系技术支持中心"的RejectResponseDocument对象。但这并不是绝对的。某些异常是业务设计时即预期可能发生的（例如，向数据库插入业务设计层面就有可能键值重复的记录），这样的异常就需要自行捕捉，捕捉到这样的异常后，应当依据业务设计将其理解为某个业务规则不能满足，或者进入某个处理过程分支的先决条件。还有一些异常虽然并非业务设计时即预期可能发生的，但为了提高其发生时的诊断效率，需要进行异常描述信息的转换，更加清晰地指出是哪一个处理环节发生了错误，这样的异常也需要自行捕捉，捕捉到这样的异常后，通常都是创建一个以原异常作为内部异常的新异常，设置新异常的描述信息并立即抛出。

#### 日志记录

应用框架子模块在具体服务请求的处理过程中会自行记录主要环节的日志（日志数据库中的ServiceJournal表），例如，进入应用层处理时的应用标识、时间、请求文档，从应用层处理返回时的时间、应答文档等等，因此，应用在OnProcess接口实现的处理过程中，通常无需再自行记录日志。但在开发和试运行阶段，为了方便查找问题，应用仍然有在OnProcess接口实现的处理过程中记录日志的需要。Framework对象提供了SaveJournal方法用以支持应用记录日志（日志数据库中的ServiceJournal表）——实际上，应用框架子模块自己在记录日志时也是调用SaveJournal方法。SaveJournal方法的原型为：



SaveJournal方法调用时都需要申报日志级别（nJournalLevel参数）。如果系统的日志级别高于调用SaveJournal方法时所申报的级别，则SaveJournal方法就不会执行日志记录操作。

**日志记录的筛选控制**

SaveJournal方法支持进行有章节结构的日志记录——对应一个具体服务请求的处理过程的所有日志记录，每一条都会有一个章节位置。这样，我们通过工具查看日志记录，就像在查看一本书的目录，能够更直观地了解具体服务请求的处理过程。

Framework对象提供了BeginChapter方法和EndChapter方法（这两个方法的原型都没有参数和返回值）用以控制当前日志记录所处的章节位置（我们也称之为“当前章节”）。调用BeginChapter方法，即会在当前章节下创建一个新的章节，并把这个新章节设置为当前章节，此后，调用SaveJournal方法产生的日志记录都会位于这个新章节。调用EndChapter方法，则会终止当前章节，并把其上级章节设置为当前章节。

#### 数据加工应用和服务登记簿

某些应用OnProcess接口实现的处理过程中需要加工（包括插入、修改和删除）数据库中的数据，这种应用我们称之为数据加工应用（Data Process Application），应用框架子模块为数据加工应用专门定义了DataProcessApplication类（也派生自Application类）作为其基类。

如果一个具体服务请求需要由数据加工应用来完成处理（这样的服务我们也称之为“数据加工服务”），那么这个具体服务请求的处理过程的相关信息将会被记入服务登记簿（日志数据库中的ServiceRegister表），包括：应用标识，请求标识，应答标识，处理状态，请求文档，应答文档和处理完成时的数据加工应用对象的组件状态等域。这一动作由Framework对象的BeginProcess方法在创建工作应用对象后执行。

服务登记簿基于请求标识域定义了唯一索引——这意味着如果客户端系统调用数据加工服务接口时提交的请求标识重复，则会引发服务登记簿插入异常，则立即调用Complete方法并返回异步处理状态对象，处理过程结束，客户端系统将会得到“请求标识重复”处理错误码——这就是交易系统开发人员通常所讲的“堵重”机制。注意，客户端系统调用非数据加工服务接口时，相关信息不会记入服务登记簿，提交重复的请求标识不会引发服务登记簿插入异常，仍然会被正常处理。

客户端系统调用数据加工服务接口时可能会由于通信故障，导致其未能在指定时间内收到应答，此时，客户端系统通常都有查明处理结果的需要。基于服务登记簿，Framework对象的BeginProcess方法就可以满足这个需要：调用BeginProcess方法时，把第一个参数bAllowRepeat设置为true，如果发生服务登记簿插入重复的异常，BeginProcess方法就会从服务登记簿中找出请求标识相同的那条记录，并返回其中的应答标识和应答文档，如果没有发生插入重复异常，则按照BeginProcess的正常逻辑进行后续处理——这就是交易系统开发人员通常所讲的“查证-重做”机制。

在执行服务登记簿插入操作时，应答文档域和处理完成时的数据加工应用对象的组件状态域都只能填充null，处理状态域将被填充为"P"——正在处理中。Framework对象的BeginProcess方法完成数据加工应用对象的Process方法调用后，会依据其返回的应答文档更新先前插入的服务登记簿记录的应答文档、处理完成时的数据加工应用对象的组件状态、处理状态等域。如果返回的应答文档是一个RejectResponseDocument对象，则处理状态域将被填充为"R"——服务请求被拒绝，否则，处理状态域将被填充为"A"——服务请求被接受。

在处理状态域尚为"P"时，如果客户端系统提交查明处理结果的请求，只能得到“原服务请求还在处理中”处理错误码。

基于服务登记簿，Framework类还提供了QueryService静态方法，可以依据部分请求标识信息进行模糊查询，返回所有符合条件的服务接口调用的应答标识，以便调用者利用这些应答标识进行精准的、进一步的处理。

某些应用场景需要扩展、调整服务登记簿的结构，为此，应用框架子模块把对服务登记簿的操作都定义到了IRegisterService接口中，并在Framework类中定义了一个静态变量，该变量引用的服务登记簿操作对象（实现IRegisterService接口的对象）将决定对服务登记簿操作的实现方式——这意味着在具体应用环境中，我们可以自定义服务登记簿操作类，实例化相应对象并用以设置Framework类中的静态变量，从而达到扩展、调整服务登记簿结构的目的。

#### 补偿处理

数据加工应用大多需要支持冲正功能，即完成所有预期处理过程后又应客户端系统的要求进行撤销，因此，数据加工应用类不仅要实现OnProcess接口，还要实现在DataProcessApplication类中定义的OnProcessCompensatively接口，完成对先前通过OnProcess接口进行的数据变更的补偿处理：



OnProcessCompensatively接口要求返回RejectResponseDocument对象，如果补偿处理过程全部完成，那么，返回的RejectResponseDocument对象的OperationHint属性值必须被设置为null，否则，该属性值应该被设置为补偿处理失败的原因和提供给客户端操作人员的处理方法提示。

Framework对象提供了BeginProcessCompensatively/EndProcessCompensatively方法，用以支持客户端系统提交的撤消请求的处理。撤消接口请求数据主要由原服务请求处理的应用标识和应答标识共同构成，撤消接口应答数据主要由表示撤消处理结果的RejectResponseDocument对象构成。

BeginProcessCompensatively方法的原型为：



BeginProcessCompensatively方法的主要处理过程如下：

1、把当前Framework对象放入静态ThreadLocal对象，以便后续设计访问Framework类的InstanceFramework静态属性获得当前Framework对象。

2、创建一个被用作BeginProcessCompensatively方法返回值的异步处理状态对象，并初始化Framework对象中的相应变量，以便后续设计能够不断更新这个异步处理状态对象。

3、依据BeginProcessCompensatively方法参数中的请求数据初始化Framework对象中的相应变量。

4、记录与“进入撤消”事件对应的服务日志（日志数据库中的ServiceJournal表），并发送对应的服务处理监控消息。

5、依据应用标识、应答标识在服务登记簿中检索到先前服务请求处理过程对应记录，从中获取请求文档、处理完成时的数据加工应用对象的组件状态等信息，并初始化Framework对象中的相应变量。若未能检索到，则立即调用Complete方法并返回异步处理状态对象，结束处理过程，客户端系统将会得到“原服务不存在”处理错误码。

6、更新服务登记簿中先前服务请求处理过程对应记录，把处理状态域重设为"P"，应答文档域重设为null，条件是原处理状态域必须为"A"，如果失败，则立即调用Complete方法并返回异步处理状态对象，结束处理过程，客户端系统将会得到“原服务请求处理状态冲突”处理错误码——这可以确保客户端系统提交的重复的撤消请求只能有一个得到正常处理。

7、创建业务数据库链接对象，初始化Framework对象中的相应变量，以便后续设计访问Framework对象的EnterpriseResourceSqlService属性获得业务数据库链接对象，并把该数据库链接对象设置为数据访问框架模块中的默认数据库链接对象。

8、以请求文档为参数调用数据加工应用对象的ProcessCompensatively方法，把其返回的RejectResponseDocument对象放入待返回的异步处理状态对象。ProcessCompensatively方法是数据加工应用对象的内部方法，它会继续调用本对象的OnProcessCompensatively接口来完成撤消请求在应用层面的处理过程，注意，数据加工应用对象在开始进行撤消处理时，其组件状态与先前服务请求处理完成时的组件状态是完全一致的。

9、关闭业务数据库链接对象，更新服务登记簿中先前服务请求处理过程对应记录，把处理状态域重设为"R"，应答文档域重设为待返回的异步处理状态对象中的应答文档，数据加工应用对象的组件状态域设置为当前工作应用对象的组件状态，调用Complete方法并返回异步处理状态对象——调用者获得该异步处理状态对象后即可知道撤消处理过程已完成，调用者随后即可调用Framework对象的EndProcessCompensatively方法来获得撤消处理结果。

EndProcessCompensatively方法的原型为：



EndProcessCompensatively方法的主要处理过程如下：

1、记录与“返回”事件对应的服务日志（日志数据库中的ServiceJournal表），并发送对应的服务处理监控消息。

2、使用Framework对象和先前返回的异步处理状态对象中的相应变量，设置EndProcessCompensatively方法参数中的应答数据。

某些应用系统需要支持基于原服务请求标识进行撤消处理，为此，Framework对象提供BeginProcessCompensativelyWithRequestIdentifier/EndProcessCompensativelyWithRequestIdentifier方法，在这种形态的撤消接口中，请求数据主要由原服务请求处理的应用标识和请求标识共同构成。

BeginProcessCompensativelyWithRequestIdentifier方法的原型为：



BeginProcessCompensativelyWithRequestIdentifier方法的主要处理过程如下：

1、依据原服务请求处理的应用标识、请求标识在服务登记簿中检索到先前服务请求处理过程对应记录，从中获取原服务请求处理的应答标识。若未能检索到，则立即调用Complete方法并返回异步处理状态对象，结束处理过程，客户端系统将会得到“原服务请求不存在”处理错误码。

2、调用BeginProcessCompensatively方法完成撤消处理过程。

EndProcessCompensativelyWithRequestIdentifier方法的原型和EndProcessCompensatively方法相同，其处理过程就是调用EndProcessCompensatively方法。

#### 异步补偿处理

客户端系统调用数据加工服务接口时，如果由于通信故障，导致其未能在指定时间内收到应答，还可能会用到一种被称为“自动冲正”的方案来解决问题——客户端系统无需查明原服务请求的处理结果，而是直接提交包含了原服务请求标识的自动冲正请求，要求服务系统据此撤消原服务请求处理；服务系统收到自动冲正请求后，需要立即向客户端系统回发应答，表示其已经收到请求，客户端系统只要收到这个应答，就可以认为原服务请求处理一定会被撤消。而在服务系统一端，实际的撤消处理过程往往是在应答回发后的某个时刻，由另一个进程/线程来完成的——这是典型的异步处理模式，因此，我们也把这种自动冲正请求的处理称作“异步补偿处理”。

为支持自动冲正，Framework对象提供了ProcessCompensativelyInAsynchronousWay方法，在这个自动冲正接口中，请求数据就是原服务请求标识。

ProcessCompensativelyInAsynchronousWay方法的原型为：



ProcessCompensativelyInAsynchronousWay方法的主要处理过程如下：

1、往服务登记簿插入一条记录，处理状态域填充为"R"，请求标识域填充为原服务请求标识，其他域均填充null，成功则返回“原服务请求不存在”处理错误码（注意，虽然客户端系统收到的处理错误码不为0，但仍然可以认为原服务请求处理一定会被撤消，符合自动冲正机制的要求）。

2、如果插入时发生重复键值异常，则依据原服务请求标识在服务登记簿中检索到先前服务请求处理过程对应记录，从中获取原服务请求处理的处理状态、应用标识、请求文档、应答标识和处理完成时的数据加工应用对象的组件状态等信息。

3、如果原服务请求处理状态为"R"，则返回0。

4、如果原服务请求处理状态为"P"，则返回“原服务请求还在处理中”处理错误码（注意，这是一个不符合自动冲正机制要求的应答，客户端系统在收到此应答时，需要在其后一个时刻再次提交自动冲正请求）。

5、如果原服务请求处理状态为"A"，则首先更新服务登记簿中先前服务请求处理过程对应记录的处理状态为"C"——原服务请求正在撤消，然后创建一个补偿处理请求（Compensation）对象，把先前服务请求处理过程的相关信息放入Compensation对象，并以之为参数调用ServiceControlService对象的ActivateCompensation方法向服务控制系统提交补偿处理请求，最后返回0。

6、如果原服务请求处理状态为"C"，则首先创建一个补偿处理请求（Compensation）对象，把先前服务请求处理过程的相关信息放入Compensation对象，再以之为参数调调用ServiceControlService对象的ActivateCompensation方法向服务控制系统提交补偿处理请求，最后返回0。

服务控制系统上部署的补偿处理器（CompensationProcess）收到服务处理系统提交的补偿处理请求后，会把其中包含的Compensation对象放入一个哈希表，补偿处理器会定时遍历此哈希表，基于其中等候时间间隔+上次执行时间>=当前时间的Compensation对象执行实际的撤消处理过程。

为支持基于Compensation对象执行的撤消处理，Framework对象提供了BeginProcessCompensativelyInAsynchronousWay/EndProcessCompensativelyInAsynchronousWay方法。

BeginProcessCompensativelyInAsynchronousWay方法的原型为：



BeginProcessCompensativelyInAsynchronousWay方法的主要处理过程如下：

1、把当前Framework对象放入静态ThreadLocal对象，以便后续设计访问Framework类的InstanceFramework静态属性获得当前Framework对象。

2、创建一个被用作BeginProcessCompensativelyInAsynchronousWay方法返回值的异步处理状态对象，并初始化Framework对象中的相应变量，以便后续设计能够不断更新这个异步处理状态对象。

3、依据参数compensation中的先前服务请求处理过程的相关信息初始化Framework对象中的相应变量。

4、依据应答标识在服务登记簿中检索到先前服务请求处理过程对应记录，如果该记录的处理状态域不为"C"，则立即调用Complete方法并返回异步处理状态对象，结束处理过程，客户端系统将会得到“原服务请求处理状态冲突”处理错误码。

5、创建业务数据库链接对象，初始化Framework对象中的相应变量，以便后续设计访问Framework对象的EnterpriseResourceSqlService属性获得业务数据库链接对象，并把该数据库链接对象设置为数据访问框架模块中的默认数据库链接对象。

6、以请求文档为参数调用数据加工应用对象的ProcessCompensatively方法。注意，数据加工应用对象在开始进行撤消处理时，其组件状态（来自参数compensation）与先前服务请求处理完成时的组件状态是完全一致的。

7、关闭业务数据库链接对象，更新服务登记簿中先前服务请求处理过程对应记录，把处理状态域重设为"R"，应答文档域重设为待返回的异步处理状态对象中的应答文档，数据加工应用对象的组件状态域设置为当前工作应用对象的组件状态，调用Complete方法并返回异步处理状态对象——调用者获得该异步处理状态对象后即可知道撤消处理过程已完成，调用者随后即可调用Framework对象的EndProcessCompensativelyInAsynchronousWay方法来获得撤消处理结果。

EndProcessCompensativelyInAsynchronousWay方法的原型为：



EndProcessCompensativelyInAsynchronousWay方法的主要处理过程就是返回处理错误码。

显然，实际的撤消处理过程是有可能未完成的，如果失败，则补偿处理器通常会把对应Compensation对象从哈希表中清除，但是如果失败时获得的处理错误码是“需再次尝试补偿处理”（某些数据加工应用的OnProcessCompensatively接口实现会调用外部系统的补偿处理接口，如果在预定时间内未收到应答，即不能确定外部系统是否完成了补偿处理，就会产生“需再次尝试补偿处理”的处理错误码，如果补偿处理器再次发起撤销处理过程，则该应用的OnProcessCompensatively接口实现会先去查询外部系统补偿处理结果，如果没有成功，则会再次调用其补偿处理接口，如果已经成功，就会返回OperationHint属性值为null的RejectResponseDocument对象），则在哈希表中保留这个Compensation对象，但会将其等候时间间隔加倍。

## 中断/重入支持

中断/重入支持子模块用以支持内含通信IO等长时间阻塞操作（例如调用外部系统服务接口）的应用具备中断/重入能力，使其能够使用有限的线程资源完成超大流量的任务处理。

通过一个OnProcess接口完成应用的服务请求处理全过程，意味着我们只能按同步编程模型开发那些需要通过通信IO操作来访问、集成外部系统服务接口的应用——这种按同步编程模型开发出来的应用运行起来似乎必定会由于阻塞操作而需要消耗大量线程资源，整个系统根本不可能具备极高的吞吐能力。

为解决这个关键问题，我们提出了一个独特的、创新的应用设计理念——重入。所谓可重入的应用，是指该应用在运行过程中可以中断（在Java中，可以用抛出异常来实现中断），释放出线程资源，在此后某个时刻，用一个新的线程重新调用该应用先前的入口方法，该应用能够让线程恢复中断时的上下文，并从中断处继续向下运行。

#### 框架对象提供的重入支持

中断/重入支持子模块为开发可重入的应用提供了支撑，应用的OnProcess接口和OnProcessCompensatively接口都可以被实现成可重入的。中断/重入支持子模块提供了Interruption类，在应用的OnProcess/OnProcessCompensatively接口实现内部，运行时刻只要创建Interruption对象并调用其Execute方法，就会抛出一个InterruptionException类型的异常，Framework对象的BeginProcess/BeginProcessCompensatively/Begin...等方法都会在应用对象的OnProcess/OnProcessCompensatively接口调用前后捕获这种类型的异常，一旦捕获到，就会关闭业务数据库链接对象，并调用Await方法，返回异步处理状态对象——注意，此后该线程即可被用于处理其他具体服务请求，即实现了中断。

BeginProcess/BeginProcessCompensatively/Begin...等方法中调用的Await方法的原型为：



这个Await方法执行时，首先会调用Interruption对象（捕获到的InterruptionException类型的异常中会关联应用创建的Interruption对象）的ExecuteAfterInterruption接口，继而清除静态ThreadLocal对象中的Framework对象引用，记录与“中断”事件对应的服务日志（日志数据库中的ServiceJournal表），并发送对应的服务处理监控消息。注意，Await方法执行时没有改变异步处理状态对象的IsCompletedSynchronously属性值和IsCompleted属性值，它们仍然是false。

在其后某个时刻，在同一JVM的某个线程中，只要调用先前捕获InterruptionException类型异常的Framework对象的Resume方法即可实现重入。

Resume方法的原型为：



Resume方法的主要处理过程如下：

1、把当前Framework对象放入静态ThreadLocal对象，以便后续设计访问Framework类的InstanceFramework静态属性获得当前Framework对象。

2、记录与“重入”事件对应的服务日志（日志数据库中的ServiceJournal表），并发送对应的服务处理监控消息。

3、如果入口是BeginProcess方法，则执行与BeginProcess方法主要处理过程中第7、8、9步基本相同的处理过程。

4、如果入口是BeginProcessCompensatively方法，则执行与BeginProcessCompensatively方法主要处理过程中第7、8、9步基本相同的处理过程。

5、如果入口是BeginProcessSequentially方法（见“框架对象提供的多阶段事务处理一致性问题处理支持”一节），则执行与BeginProcessSequentially方法主要处理过程中第7、8、9步基本相同的处理过程。

6、如果入口是BeginProcessCompensativelyInAsynchronousWay方法，则执行与BeginProcessCompensativelyInAsynchronousWay方法主要处理过程中第5、6、7步基本相同的处理过程。

之所以是基本相同，而非完全相同，主要就在于Resume方法无需返回异步处理状态对象，处理错误码放入异步处理状态对象的动作是在Complete方法调用之前进行，并且所调用的Complete方法原型不同。这里调用的Complete方法的原型为：



这个Complete方法执行时，静态ThreadLocal对象中的Framework对象引用将被清除，继而设置异步处理状态对象的IsCompleted属性值为true，并立即回调Begin...方法调用时传入的实现了ICallback接口的对象的Callback方法，以通知调用者整个处理过程完成，促使其调用End...方法来获取处理结果。

中断发生后，先前捕获InterruptionException类型异常的Framework对象保存在哪里，以及Framework对象的Resume方法是如何调用起来的，可能有些令人困惑。如果是直接利用Interruption对象的Execute方法产生的中断，那么，在Interruption对象的ExecuteAfterInterruption接口的基类默认实现中，我们可以看到，其中设置了一个计时器，计时器到达预定时间（预定时间是在创建Interruption对象时设置的）就会在一个新的线程中执行指定的回调接口，Framework对象被保存在Interruption对象中，而Interruption对象被保存在回调接口对象中，回调接口实现会调用Framework对象的Resume方法。

当然，直接创建Interruption对象并调用其Execute 方法意义是不大的，重入机制最常见的应用是访问某些提供了异步服务接口的外部系统，在这些应用中，我们应该基于Interruption类创建一个派生类，例如AsynchronousCall，并重载ExecuteAfterInterruption接口实现以指定中断时刻需要执行的动作——向外部系统发送请求数据，并注册一个回调接口对象以处理回送的应答数据，注意，Framework对象被保存在AsynchronousCall对象中，而AsynchronousCall对象被保存在回调接口对象中，回调接口实现会调用Framework对象的Resume方法。

#### 应用对象可重入方法的编程规则

我们必须认识到，重入机制运作的关键除了Framework对象外，还有应用自身的可重入方法。在同一个具体服务请求/撤销请求的处理过程中，这些可重入方法可能会被反复调用，而不是仅仅执行一次。因此，需要严格按照一定的规则和样板来编写。

以OnProcess接口实现为例，在编写可重入的OnProcess接口实现的各个步骤时，辨别哪些是需要记忆的，哪些是不需要记忆的，这一点非常关键，如果处理不好，在重入时线程就不能恢复原先中断时的上下文和执行路线，或者大量浪费计算资源。

所谓需要记忆的步骤，就是指在重入时不能再次执行具体处理过程的步骤，包括但不限于：不同时刻执行其处理过程无法保证都能获得同样的处理结果的步骤，或者再次执行其处理过程会让相关数据状态发生不能接受的改变的步骤，或者执行其处理过程会消耗大量资源并且较为漫长的步骤，等等。

这些需要记忆的步骤的处理结果也经常被用作后续步骤的处理参数，如何能够在避免重复处理的同时，又能获得先前的处理结果？中断/重入支持子模块提供的Memory类和MemoryRegister类可以帮助应用解决这个问题，Memory对象被用来对应一个已执行的需要记忆的步骤，记录该步骤的处理结果，而MemoryRegister对象则被用来保存、恢复所有与一个具体服务请求/撤消请求处理过程中已执行的需要记忆的步骤对应的Memory对象。

每个应用对象内部都聚合了一个MemoryRegister对象。在需要记忆的步骤的处理过程开始时先创建一个Memory对象，并以之为参数调用应用内部MemoryRegister对象的InstaurateMemory方法，然后判断InstaurateMemory方法返回的Memory对象的IsInitial属性值是否为true，如果是true，才执行该步骤的处理过程，并用处理结果来填充返回的Memory对象。如果是第一次执行该步骤，则InstaurateMemory方法会返回刚刚创建的那个Memory对象，此时该对象的IsInitial属性值为true，因此，接下来应用就会执行该步骤的处理过程，并用该步骤的处理结果来填充该对象。如果是因为重入而再次执行到该步骤，则InstaurateMemory方法将返回先前执行该步骤前创建并已经被填充了该步骤处理结果的Memory对象，此时，该对象的IsInitial属性值为false，因此，应用能够避免再次执行该步骤。无论是第一次执行，还是重入时再次执行，应用都应当从返回的Memory对象中获取相应步骤的处理结果，并据此执行后续处理过程。

实际上，可重入的OnProcess接口实现的处理过程可以看作一个由不需要记忆的步骤和需要记忆的步骤组成的一个步骤集（需要注意的是步骤之间不能嵌套），在这个步骤集的尾部，最后一个动作是返回一个应答文档。

其中，不需要记忆的步骤的编写样板如下：



需要记忆的步骤的编写样板如下：



可能会中断的步骤一定是一个需要记忆的步骤——Interruption类其实就是Memory类的派生类。如果我们想在OnProcess接口实现中添加一个一执行就中断并在60秒后自动重入的步骤，只需编写如下代码：



更多时候，可能会中断的步骤是一些调用异步服务接口访问外部系统的步骤，在这些步骤中，OnProcess接口实现通常会使用进一步封装的异步服务接口调用对象——即应用自定义的一种Interruption对象，例如前面提到的AsynchronousCall对象，其编写样板如下：



#### 可重入应用开发示范

可重入应用的编写看起来似乎很繁琐，处处都是陷阱，但是，通过下面的例子我们就可以看出来，只要事先封装好一些基础的可重入方法，编写可重入的应用就会显得很轻松。

我们首先封装一个应用PurchaseFundFromBank，这是一个在银行端服务系统中供其它应用调用的基础应用，它的作用是调用基金公司交易系统的接口，提交银行网点客户购买基金的交易请求。由于基金公司提供的访问其交易系统的相关方法是支持回调函数的异步模式的API，因此，我们可以把PurchaseFundFromBank设计成可重入的，即在其OnProcess接口实现调用基金公司交易系统访问代理（RemoteServiceProxy）对象提供的相关方法（Process）后中断，在回调函数执行时再调用Framework对象的Resume方法，重新进入OnProcess接口实现，并在先前中断的位置执行后续处理。



但是，在上面展示的PurchaseFundFromBank类的OnProcess接口实现代码中，我们却看到，其处理过程非常简洁，丝毫看不出可重入方法编程规则的痕迹——奥妙就在于其中BankService对象的CallFundServiceProcess方法被设计成了一个基础的可重入方法，使得PurchaseFundFromBank类的OnProcess接口实现的整个处理过程都成为不需要记忆的步骤：





PurchaseFundFromBank的数据处理显然是不完整的——既然购买基金，那么必然要操作客户在银行的帐户、支出资金才行。因此，我们接下来要开发一个银行服务系统中完整的、对客户端系统提交的客户购买基金的服务请求直接进行处理的应用PurchaseFund，其OnProcess接口实现将首先调用借计储蓄帐户应用DebitSavingAccount，然后再调用PurchaseFundFromBank。



注意，帮助完成应用调用的DataProcessRegister对象的InvokeDataProcessApplication方法是一个应用框架层面提供的可重入的方法——这个方法执行时会自动判断之前是否已经执行过应用调用，如果已经执行过，则返回之前调用时获得的应答文档，这使得PurchaseFund类的OnProcess接口实现的整个处理过程都成为不需要记忆的步骤，因此，其处理过程也显得非常简洁。

类似的方法还有Framework类的InvokeApplication静态方法。这些应用框架提供的可重入的应用调用方法对于高度组件化设计（业务逻辑被封装在组件内部，组件可以调用更基础的组件，应用大都是通过调用更基础的组件和应用的方法来完成其处理过程）的应用系统而言意义重大，这意味着只要基础组件和应用提供的方法是可重入的，那么，基于应用框架提供的可重入的应用调用方法构建的上层应用都自动是可重入的。

## 多阶段事务处理应用模型

多阶段事务处理应用模型子模块确立了多阶段事务处理应用对象体系的基本结构，建立了相应的一致性问题处理机制。

在普通的数据加工应用对象的OnProcess接口和OnProcessCompensatively接口调用前后，Framework对象会自动地开启和关闭业务数据库事务，应用的所有数据变更操作都被放在这个业务数据库事务中管理，因此，我们也称之为单事务处理应用。在编写单事务处理应用类的OnProcess接口和OnProcessCompensatively接口的实现时通常不可以再自行创建、管理业务数据库事务。

如果具体服务请求的处理过程要求应用创建、管理多个数据库事务，应用该怎么办呢？例如，OnProcess接口的实现要访问、变更多个数据库中的数据。这种情况下，应用就只能自行创建、管理多个数据库事务了，这就是我们所讲的多阶段事务处理应用。注意，由于这些数据库事务可能属于不同的数据库，甚至可能属于多个不同类型的数据库（例如Oracle、DB2和MySQL），因此，多阶段事务处理应用必然面临多阶段事务处理一致性问题——即在其处理过程中，会发生其中某些数据库事务处理成功（数据变更操作全部成功并Commit），而其后的数据库事务却处理失败（数据变更操作失败并Rollback）的问题，导致整个数据体系的完整性、一致性被破坏。

为更好地应对这类需求，多阶段事务处理应用模型子模块提供了专门的多阶段事务处理应用基类MultistageDataProcessApplication（也是DataProcessApplication的派生类），并提供了2种处理机制用来解决多阶段事务处理一致性问题：1是根据先前有哪些处理成功的数据库事务，执行相应的补偿处理，直到数据全部恢复原状（服务请求被拒绝或撤销），简称“补偿-拒绝”机制；2是在先前处理失败的数据库事务上继续尝试，直到所有数据库事务全部处理成功（服务请求被接受），简称“续做-接受”机制。

#### 框架对象提供的多阶段事务处理一致性问题处理支持

多阶段事务处理应用（即MultistageDataProcessApplication派生类）对象的OnProcess接口和OnProcessCompensatively接口被调用前后，Framework对象不会自动开启、关闭业务数据库事务。

多阶段事务处理应用模型子模块提供了DataProcess类和DataProcessRegister类，用来支持应用记录哪些数据变更步骤已经执行以及处理成功与否，其中，DataProcess对象被用来对应一个已发生的数据变更步骤（通常一个数据变更步骤会对应一个数据库事务，但这并不绝对），记录该步骤的处理情况（包括处理结果），而DataProcessRegister对象则被用来保存、恢复所有与一个具体服务请求处理过程中已发生并且未被补偿处理的数据变更步骤对应的DataProcess对象。

每一个多阶段事务处理应用对象内部都聚合了一个DataProcessRegister对象。当一个数据变更步骤处理失败，多阶段事务处理应用对象的OnProcess接口返回内含失败原因的RejectResponseDocument对象时，Framework对象的BeginProcess/Resume方法会立即检查应用内部的DataProcessRegister对象，如果其中含有DataProcess对象，就会在给调用者准备的应答数据中设置存在一致性问题的标志（调用EndProcess方法获得的bIsConsistent参数值将为false），并把服务登记簿中原先服务请求处理过程对应记录的处理状态域重设为"I"——服务请求处理存在一致性问题。

调用者获得存在一致性问题的应答数据后，可以选择提交一个撤消请求的方案来处理一致性问题。在服务系统上，（一个新的）Framework对象的BeginProcessCompensatively方法被用来处理这个撤消请求（注意，BeginProcessCompensatively方法主要处理过程的第6步中的“条件是原处理状态域必须为"A"”，应修正为“条件是原处理状态域必须为"A"或"I"”），在BeginProcessCompensatively方法内部，将创建一个多阶段事务处理应用对象（注意，这个对象的组件状态与原先产生一致性问题的多阶段事务处理应用对象的最后时刻的组件状态完全相同——它们内部的DataProcessRegister对象所含信息是一样的，因此，在撤消请求处理时刻，应用能够知道先前产生一致性问题的处理过程已经完成了哪些数据变更步骤），并调用这个对象的OnProcessCompensatively接口进行补偿处理。随着补偿处理过程的推进，应用需要自行、及时地把相应的DataProcess对象从DataProcessRegister对象中清理出去。如果补偿处理过程也出现了数据变更失败，OnProcessCompensatively接口返回内含失败原因的RejectResponseDocument对象，则Framework对象的BeginProcessCompensatively/Resume方法会在应答数据中继续设置存在一致性问题的标志（调用EndProcessCompensatively方法获得的bIsConsistent参数值将为false），并把服务登记簿中原先服务请求处理过程对应记录的处理状态域重设为"I"。

“补偿-拒绝”机制除了由调用者发起，Framework对象也可能会自动发起。多阶段事务处理应用对象的OnProcess接口返回内含失败原因的RejectResponseDocument对象时，Framework对象的BeginProcess/Resume方法实际上会首先检查应用内部的DataProcessRegister对象包含的全部DataProcess对象是否都允许自动补偿，如果存在不允许自动补偿的DataProcess对象，才会在应答数据中设置存在一致性问题的标志，并把服务登记簿中原先服务请求处理过程对应记录的处理状态域重设为"I"，如果都允许自动补偿，则会把服务登记簿中原先服务请求处理过程对应记录的处理状态域重设为"C"，并创建一个Compensation对象，把先前服务请求处理过程的相关信息放入Compensation对象，向服务控制系统提交补偿处理请求。此时，调用Framework对象的EndProcess方法会获得一个内含失败原因的RejectResponseDocument对象，但应答数据不会有一致性问题。

调用者获得存在一致性问题的应答数据后，还可以选择提交一个续做请求的方案来处理一致性问题。续做接口请求数据主要由原服务请求处理的应用标识和应答标识共同构成，续做接口应答数据就是在服务接口应答数据的基础上去除了应答标识。在服务系统上，（一个新的）Framework对象的BeginProcessSequentially方法被用来处理续做请求。

BeginProcessSequentially方法的原型为：

 BeginProcessSequentially方法的主要处理过程如下：

1、把当前Framework对象放入静态ThreadLocal对象，以便后续设计访问Framework类的InstanceFramework静态属性获得当前Framework对象。

2、创建一个被用作BeginProcessSequentially方法返回值的异步处理状态对象，并初始化Framework对象中的相应变量，以便后续设计能够不断更新这个异步处理状态对象。

3、依据BeginProcessSequentially方法参数中的请求数据初始化Framework对象中的相应变量。

4、记录与“进入续做”事件对应的服务日志（日志数据库中的ServiceJournal表），并发送对应的服务处理监控消息。

5、依据应用标识、应答标识在服务登记簿中检索到先前服务请求处理过程对应记录，从中获取请求文档、处理完成时的数据加工应用对象的组件状态等信息，并初始化Framework对象中的相应变量。若未能检索到，则立即调用Complete方法并返回异步处理状态对象，结束处理过程，客户端系统将会得到“原服务不存在”处理错误码。

6、更新服务登记簿中先前服务请求处理过程对应记录，把处理状态域重设为"P"，应答文档域重设为null，条件是原处理状态域必须为"I"，如果失败，则立即调用Complete方法并返回异步处理状态对象，结束处理过程，客户端系统将会得到“原服务请求处理状态冲突”处理错误码——这可以确保客户端系统提交的重复的续做请求只能有一个得到正常处理。

7、创建业务数据库链接对象，初始化Framework对象中的相应变量，以便后续设计访问Framework对象的EnterpriseResourceSqlService属性获得业务数据库链接对象，并把该数据库链接对象设置为数据访问框架模块中的默认数据库链接对象。

8、以请求文档为参数调用数据加工应用对象的Process方法，把其返回的应答文档放入待返回的异步处理状态对象。注意，数据加工应用对象在开始进行续做处理时，其组件状态与先前服务请求处理完成时的组件状态完全相同——它们内部的DataProcessRegister对象所含信息是一样的，因此，在续做请求处理时刻，应用能够知道先前产生一致性问题的处理过程已经完成了哪些数据变更步骤。

9、关闭业务数据库链接对象。更新服务登记簿中先前服务请求处理过程对应记录，把应答文档域重设为待返回的异步处理状态对象中的应答文档，数据加工应用对象的组件状态域设置为当前工作应用对象的组件状态。如果续做请求处理过程也出现了数据变更失败（OnProcess接口返回内含失败原因的RejectResponseDocument对象），则在应答数据中继续设置存在一致性问题的标志（调用EndProcessSequentially方法获得的bIsConsistent参数值将为false），并把服务登记簿中原先服务请求处理过程对应记录的处理状态域重设为"I"；如果续做请求处理过程全部完成，则把服务登记簿中先前服务请求处理过程对应记录的处理状态域重设为"A"。调用Complete方法并返回异步处理状态对象——调用者获得该异步处理状态对象后即可知道续做处理过程已完成，调用者随后即可调用Framework对象的EndProcessSequentially方法来获得续做处理结果。

EndProcessSequentially方法的原型为：



EndProcessSequentially方法的主要处理过程如下：

1、记录与“续做返回”事件对应的服务日志（日志数据库中的ServiceJournal表），并发送对应的服务处理监控消息。

2、使用Framework对象和先前返回的异步处理状态对象中的相应变量，设置EndProcessSequentially方法参数中的应答数据。

#### 多阶段事务处理应用OnProcess接口编程规则

我们必须认识到，“续做-接受”机制运作的关键除了Framework对象外，还有多阶段事务处理应用自身的OnProcess接口的实现。在同一个具体服务请求处理引发的多阶段事务处理一致性问题的处理过程中，该接口可能会被反复调用，而不是仅仅执行一次。因此，需要严格按照一定的规则和编写样板来实现。

在编写多阶段事务处理应用OnProcess接口实现的各个步骤时，辨别哪些是可重复的，哪些是不可重复的，这一点非常关键。

所谓可重复的步骤，严格意义上是指在不同时刻执行其处理过程都能产生同样的结果、并且不会对数据进行变更操作的步骤，例如，根据请求文档中的信息填写某个处理动作所需要的参数，再例如，查询某个账户的余额（如果在这个时间段内该账户信息没有变化的话），等等。

数据变更步骤通常都是不可重复的。但数据变更步骤的处理结果也经常需要被作为后续步骤处理动作的参数，如何能够在避免重复执行的同时，又能获得先前的处理结果？DataProcessRegister对象和DataProcess对象可以解决这个问题。

在数据变更步骤的处理过程开始时创建一个DataProcess对象，设置其对应步骤的Name属性值，向其填充该步骤处理过程所需的参数，并以之为参数调用多阶段事务处理应用内部DataProcessRegister对象的EnterDataProcess方法，然后判断EnterDataProcess方法返回的DataProcess对象的IsCommitted属性值是否为false，如果是false，才执行该步骤的处理过程，并用处理结果来填充返回的DataProcess对象，失败就立即返回一个内含失败原因的RejectResponseDocument对象，成功就调用DataProcess对象的Commit方法（这将导致其IsCommitted属性值被改为true）。如果是第一次执行该步骤，则EnterDataProcess方法会返回刚刚创建的那个DataProcess对象，此时该对象的IsCommitted属性值为false，因此，接下来应用就会执行该步骤的处理过程，并用该步骤的处理结果来填充该对象。如果是因为一致性问题处理而再次执行到该步骤，则EnterDataProcess方法将返回先前执行该步骤前创建并已经被填充了该步骤处理结果的DataProcess对象，此时，该对象的IsCommitted属性值为true，因此，应用能够避免再次执行该步骤。无论是第一次执行，还是一致性问题处理时再次执行，应用都应当从返回的DataProcess对象中获取相应的数据变更步骤的处理结果，并据此执行后续处理过程。

实际上，多阶段事务处理应用OnProcess接口实现的处理过程可以看作一个由可重复的步骤和数据变更步骤组成的一个步骤集（需要注意的是步骤之间不能嵌套），在这个步骤集的尾部，最后一个动作是返回应答文档。

其中，可重复的步骤的编写样板如下：



数据变更步骤的编写样板如下：



在应用开发调试和运行管理工作中，我们会有查看DataProcessRegister对象内部信息的需要，如果其中包含的DataProcess对象数量庞大的话，我们会希望DataProcessRegister对象使用目录结构来保存这些DataProcess对象，以便我们能够更高效地了解具体服务请求处理过程中数据变更步骤的执行情况。MultistageDataProcess对象可以用来解决这个问题，这是一个DataProcess派生类对象，其内部又包含了一组DataProcess对象，因此，MultistageDataProcess对象实际对应了一个数据变更步骤集，但由于本身没有任何处理过程，因此并没有破坏步骤之间不可嵌套的原则。需要注意的是，MultistageDataProcess对象对应的数据变更步骤集中每个步骤都要根据该步骤的处理参数和结果填写MultistageDataProcess对象，而不是应答文档。



如果一个数据变更步骤未能执行到DataProcess对象的Commit方法或者CommitPartially 方法，那么，在应用OnProcess接口调用结束后，Framework对象通常会把该对象从DataProcessRegister对象中清除，但如果该对象是MultistageDataProcess对象则会例外。

有些数据变更步骤有可能产生不确定的处理结果（例如，调用外部系统提供的服务接口），这种步骤的编写样板如下：



有些数据变更步骤不但有可能产生不确定的处理结果，而且其获知处理结果的时间也是不确定的（例如，调用外部系统提供的服务接口，其应答数据会在请求数据发出之后的某个时段内送回），这种步骤的编写样板如下：



如果在上述应用场景中运用重入机制，而非阻塞等候应答数据，则可以确保这类应用不会在限定时间内一直占用线程资源，编写样板如下：



#### 多阶段事务处理应用OnProcessCompensatively接口编程规则

我们同样必须认识到，“补偿-拒绝”机制运作的关键除了Framework对象外，还有多阶段事务处理应用自身的OnProcessCompensatively接口实现。在同一个具体服务请求处理引发的一致性问题的处理过程中，该接口也可能会被反复调用，而不是仅仅执行一次。因此，也需要严格按照一定的规则和编写样板来实现。

DataProcessRegister对象记录的信息可以帮助应用判定该执行哪些补偿处理步骤，并提供原先的数据变更步骤执行时的处理参数和结果。注意，并非所有在DataProcessRegister对象中有相应的DataProcess对象的数据变更操作步骤都必须要有和其对应的补偿处理步骤。

补偿处理步骤的编写样板如下：



如果原步骤其内部处理过程被编写为多步骤，那么对应的补偿处理步骤的编写样板如下：



如果原步骤有可能产生不确定的处理结果，那么对应的补偿处理步骤的编写样板如下（实际上，如果单事务处理应用的补偿处理过程有可能产生不确定的处理结果的话，也会应用类似的样板）：



如果原步骤获知处理结果的时间也是不确定的，例如，调用外部系统提供的服务接口，那么对应的补偿处理步骤的编写样板如下（实际上，如果单事务处理应用的补偿处理过程获知补偿处理结果的时间也是不确定的话，也会应用类似的样板）：



如果在上述应用场景中运用重入机制，而非阻塞等候应答数据，则可以确保这类应用不会在限定时间内一直占用线程资源，编写样板如下：



#### 应用调用控制

与可重入应用的开发类似，基于应用框架层面提供的应用调用方法，多阶段事务处理应用的开发也可以大大简化。但是，在大量运用应用框架层面提供的应用调用方法前，我们必须认识到，应用能否被正常调用，和调用者的类型，以及被调用者的类型都有关系——

非数据加工型的应用（Application派生类，通常都是查询应用）通过Framework类的InvokeApplication静态方法，可以在所有类型的应用中被调用；

普通数据加工应用（DataProcessApplication派生类，通常都是单阶段事务处理应用）通过Framework类的InvokeApplication静态方法，可以在普通数据加工应用中被调用，注意调用者和被调用者的数据变更操作都在同一个由应用框架管理的数据库事务中；

如果在多阶段事务处理应用中调用非数据加工型的应用，最好通过DataProcessRegister对象的InvokeApplication方法（而不是Framework类的InvokeApplication静态方法）进行；

普通数据加工应用和多阶段事务处理应用可以在多阶段事务处理应用中被调用，但只能通过DataProcessRegister对象的InvokeDataProcessApplication方法进行；

## 异步任务支持

异步任务支持子模块用以支持异步任务的登记和执行，以便应用能够把处理时间较长、并且无需在当前交易处理过程中获知处理结果的动作分离到专门的异步任务处理线程中执行，缩短“交易响应时间”。

#### 异步任务的基本开发模式

应用可以自行定义各种在具体服务请求/撤消请求处理过程中可以生成的异步任务，每种异步任务都要有一个异步任务类，例如，用来执行发送短消息任务的异步任务类SendShortMessageTask。异步任务支持子模块提供了两种异步任务基类：AsynchronousTask和SynchronizedAsynchronousTask。

如果异步任务类派生自SynchronizedAsynchronousTask，则实现相对简单，就是实现Process接口，注意，这个接口实现异步任务的具体处理逻辑，它会在执行异步任务的线程中被调用，而不是在执行服务请求/撤消请求处理过程的线程中被调用，这个Process接口的原型为：



如果在Process接口实现中调用一个耗时较长的方法，例如调用一个封装得不太完美的TransportFile方法，而这个方法会确保文件传送到位后才返回，那么，如此实现就会导致执行异步任务的线程池可能经常处于资源紧缺，执行请求排队很长，而CPU资源并未得到充分利用的状态。这种情况下，我们建议基于AsynchronousTask派生异步任务类（实际上，SynchronizedAsynchronousTask也派生自AsynchronousTask），并实现BeginProcess/EndProcess接口。

BeginProcess/EndProcess接口的原型为：



显然，派生自AsynchronousTask的异步任务类都遵循APM开发，能够更高效地利用线程资源。

#### 运行时异步任务的登记和处理

每个应用对象内部都聚合了一个异步任务登记器（AsynchronousTaskRegister）对象。在应用的OnProcess接口和OnProcessCompensatively接口实现中，随时都可以创建异步任务对象，并以之为参数调用AsynchronousTaskRegister对象的SetAsynchronousTask方法，把异步任务对象加入到AsynchronousTaskRegister对象中——即完成异步任务的登记。调用SetAsynchronousTask方法时，需要通过nTaskProcessMode参数注明该任务将在服务请求/撤消请求处理过程执行成功时执行，还是失败时执行，还是都执行。

Framework对象的BeginProcess/BeginProcessCompensatively/Resume/Begin...等方法调用应用对象的OnProcess接口和OnProcessCompensatively接口返回后，会首先创建一个异步任务组（AsynchronousTaskGroup）对象，并根据应用对象接口处理的成败把其AsynchronousTaskRegister对象中包含的相应情况下需要执行的异步任务对象放入AsynchronousTaskGroup对象（在这个AsynchronousTaskGroup对象中，除了那些登记时注明的无论成败都要执行的异步任务对象，要么都是登记时注明的成功时执行的异步任务对象，要么都是登记时注明的失败时执行的异步任务对象），如果这个AsynchronousTaskGroup对象中最终存在需要执行的异步任务对象，则先以之为参数调用异步任务处理服务访问代理的Process方法，向异步任务处理系统提交异步任务组执行请求，然后才会调用Complete方法结束处理过程。

异步任务处理系统上部署的异步任务处理器（AsynchronousTaskProcess）收到服务处理系统提交的异步任务组执行请求后，会恢复其中包含的AsynchronousTaskGroup对象中的所有异步任务对象，并基于既定的线程池并发执行这些异步任务对象的处理过程。注意，异步任务支持子模块不保证一个包含了AsynchronousTaskGroup对象的消息能够及时、可靠地送达异步任务处理器，但保证所有曾经产生并且需要执行的异步任务对象都会在数据库中留有相应的记录。

#### 按重入模式开发的异步任务

除了实现BeginProcess/EndProcess接口，派生自AsynchronousTask的异步任务类也可以按重入模式开发，实现OnProcess接口，OnProcess接口的原型为：



和应用的中断/重入机制类似，异步任务基类也为按重入模式开发的异步任务提供了支持：在异步任务类OnProcess接口实现内部，可以创建Interruption对象并调用其Execute方法，抛出InterruptionException类型的异常，异步任务基类的BeginProcess接口的默认实现会在OnProcess接口调用前后捕获这种类型的异常，一旦捕获到，就会调用Interruption对象的ExecuteAfterInterruption接口，然后返回异步处理状态对象——注意，此后该线程即可被用于处理其他异步任务，即实现了中断。在其后某个时刻，在同一JVM的某个线程中，只要调用先前捕获InterruptionException类型异常的异步任务对象的Resume方法（在基类中提供）即会重入。

异步任务类OnProcess接口实现内部，同样使用MemoryRegister对象和Memory对象，避免重复处理需要记忆的步骤，并获得先前的处理结果。因此，按重入模式开发的异步任务类OnProcess接口的编程规则和应用可重入方法的编程规则基本一致，这里不再赘述。

异步任务和应用在中断/重入运作机制上一个重要区别是中断/重入之间对象状态的保存方式——由于异步任务重入时间间隔一般不像应用那样有限制（由于客户端系统调用服务接口时通常能够接受的应答等候时间都在1分钟内，所以，应用的重入时间间隔通常都会设置为几十秒），所以，在异步任务中断时，对象状态会被存入数据库，而不是在内存中保留对象。在条件具备可以继续处理时，异步任务处理器会根据数据库中的记录恢复异步任务对象，并调用其Resume方法进行重入处理。

开发异步任务时，选择重入模式，还是APM模式，主要取决于异步任务的处理内容，以及底层接口形式。例如，如果异步任务是要进行文件传送，而文件传送底层接口本身就是APM模式的，那么，采用APM模式开发异步任务就会非常清晰、简洁；如果底层接口本身是普通的支持回调函数的异步模式的业务处理接口，则可以考虑采用重入模式开发。

#### 按重入模式开发异步任务时常用的基础方法

在具体应用场景中，很多异步任务实质上是一些任务控制器，例如，联机交易处理过程中提交了一个异步任务，其内容是要执行一个批处理任务，但该批处理任务的执行时机需要加以控制，而不是何时提交就何时执行——那样很容易让数据库产生大量并发锁，甚至死锁，导致系统崩溃。

上述任务控制器工作时需要时钟服务的配合，在它的OnProcess接口实现执行时，如果当前时刻不在限定时段内，就会立即中断并发送一个到时提醒的请求到时钟服务系统，待约定时间到达，时钟服务系统送来提醒消息，OnProcess接口实现重入执行时，才会向批处理系统提交批处理任务。

为简化上述控时逻辑的设计，异步任务基类提供了可重入的WaitClockAlarm方法，供任务控制器在其OnProcess接口实现中调用。WaitClockAlarm方法被调用时会自行判断是否重入执行，如果是首次执行，则判断当前时刻是否在限定时段内，如果在，就立即返回，如果不在，就会抛出Interruption异常，完成中断，同时向时钟服务系统发出到时提醒的请求；如果是由于时钟服务系统发来的提醒消息而导致的重入执行，则立即返回。

任务控制器常用的基础服务，除了时钟服务，还有信号服务、队列服务等。针对这些常用的基础服务，异步任务基类都提供了可重入的方法，包括：WaitSignal方法，ReadQueueData方法，等等，以简化相应控制逻辑的设计。

## 处理器和代理

处理器和代理子模块提供了通信层的组件，在服务系统端负责解析外部系统（客户端系统和运行管理平台）提交的请求报文，调用应用框架接口进行处理，并组装应答报文回发给外部系统——即所谓的处理器。

为进一步简化外部系统访问服务系统时的编程，处理器和代理子模块还为外部系统提供了一组API，这些API自动建立、维护通信通道，并自动完成符合处理器访问报文格式要求的请求报文的组装和应答报文的解析——即所谓的代理。

外部系统可以访问的处理器有两种。一是部署在服务控制系统上的、用于处理运行管理平台发来的运行控制命令的处理器，即运行控制器；二是部署在服务处理系统上的、用于处理客户端系统发来的具体服务请求的，即服务处理器。

#### 运行控制器和服务控制访问代理

运行控制器是基于通信应用框架模块的基础服务应用模型开发的（ServiceApplication派生类OperationControl），所登记的基础服务标识为"SCOC"，运行管理平台可以基于基础服务访问协议对其进行访问——在基础服务访问请求报文应用区中填写运行控制器能够处理的命令和相关输入参数，并在基础服务访问应答报文应用区中读取运行控制器的命令处理结果和其他输出参数。这些命令和输入、输出参数通常都是中/英文词或词组，一些具有复杂结构的输入、输出参数则会用XML格式字符串表示，都用ASCII编码填写并用空格分隔。

运行控制器支持StartService、StopService、GetRuntimeSetting、SetRuntimeSetting、LoadComponentClass等命令，这些命令的作用，以及相关参数的含义、格式如下：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 命令 | 作用 | 输入参数表 | 处理结果 | 其他输出参数表 |
| StartService |  |  |  |  |
| StopService |  |  |  |  |

为访问运行控制器封装的代理就是服务控制访问代理，它对应运行控制器支持的命令，提供以下方法：

#### 服务处理器和服务代理

服务处理器实际上是两个，一是基于通信应用框架模块的异步基础服务应用模型开发的（AsynchronousServiceApplication派生类AsynchronousServiceProcessor），所登记的基础服务标识为"ASCS"，调用Framework对象提供的BeginProcess.../EndProcess...等异步API，处理客户端系统提交的具体服务请求或相应的补偿处理请求、续做处理请求，这些请求处理时都有可能会产生中断；二是基于通信应用框架模块的基础服务应用模型开发的（ServiceApplication派生类ServiceProcessor和AsynchronousServiceProcessor），所登记的基础服务标识为"SSCS"，调用Framework类/对象提供的Query.../Process...等同步API，处理客户端系统提交的日志查询、异步补偿处理等执行时不会有中断的请求。

客户端系统可以基于基础服务访问协议对其进行访问——在基础服务访问请求报文应用区中填写服务处理器能够执行的处理方法名称和相关输入参数，并在基础服务访问应答报文应用区中读取服务处理器的处理结果和其他输出参数。在应用区中，这些处理方法名称和处理结果、输入、输出参数等要素都采用ASCII编码的[8位数字字符串表示的要素数据长度]+[单个字母表示的要素数据类型（I表示整数类型，B表示布尔类型，S表示字符串类型）]+[字符串表示的要素数据]的格式来填写，例如，如果请求报文应用区内填写的数据为"00000006I00012300000004BTrue0000030SHello!This is a 30Byte string!"，服务处理器就会认为其中包含了3个要素，第一个要素是一个整数，值为123，第二个要素是一个布尔数，值为True，第三个要素是一个字符串，值为"Hello!This is a 30Byte string!"。

无论哪个服务处理器，请求报文应用区中的第一个要素都是处理方法名称，请求报文中的处理方法名称决定了其他要素（包括应答报文应用区的要素）的个数和含义，AsynchronousServiceProcessor支持Process、ProcessSequentially 、ProcessCompensatively、ProcessCompensativelyWithRequestIdentifier等处理方法，ServiceProcessor支持QueryService、ProcessCompensativelyInAsynchronousWay等处理方法，这些处理方法的作用和相应的其他要素的含义、类型和填写格式如下：

|  |  |  |  |
| --- | --- | --- | --- |
| 处理方法名 | 作用 | 请求报文其他要素 | 应答报文要素 |
| Process |  |  |  |

报文应用区中的接口数据基本都XML格式字符串表示。服务处理器从请求报文应用区中获得XML格式字符串形态的接口数据后，会依据应用标识在ServiceDeclarationRegister对象中查找到对应的ServiceInterfaceDeclaration对象，然后依据该对象的RequestDocumentType属性值创建相应的请求文档对象，再调用请求文档对象的OXM方法处理XML格式字符串形态的接口数据，在请求文档对象的OXM方法执行时刻，会自动对XML格式字符串形态的接口数据进行合法性检查。

为访问服务处理器封装的代理就是服务代理（ServiceProxy），它对应服务处理器支持的处理方法，提供以下方法：

服务代理从应答报文应用区中获得XML格式字符串形态的接口数据后，也会依据应用标识在本地ServiceDeclarationRegister对象中查找到对应的ServiceInterfaceDeclaration对象，然后依据该对象的ResponseDocumentType属性值创建相应的应答文档对象，再调用应答文档对象的OXM方法处理XML格式字符串形态的接口数据，在应答文档对象的OXM方法执行时刻，会自动对XML格式字符串形态的接口数据进行合法性检查。

#### 定制服务处理器和服务代理

由于应用框架子模块被分离成为一个独立的设计，因此，理论上，我们可以开发出很多种服务处理器和服务代理，例如，我们可以基于Tomcat开发符合WebService规范的服务处理器和服务代理。此外，某些应用场景希望自行定义请求和应答报文（例如，核心系统交易请求和应答报文都是设定分隔符的字符串字节流），并在应用框架之外统一进行一些个性化的处理，因此，定制服务处理器和服务代理是很常见的需求。

我们通常在具体应用场景中定制、开发服务处理器和相应的服务代理。定制开发的服务处理器和服务代理，与服务应用框架模块提供的默认的服务处理器和服务代理，其主要处理过程是基本相同的。

## 附录

* + 1. 框架类管理的全局静态对象示意图



* + 1. 具体服务处理时相关对象的协作图





* + 1. 服务登记簿记录处理状态域的状态图



# 批量应用框架

批量应用框架（Batch Application Framework，简称BAF）模块以通信应用框架模块和INTIGER平台为基础构建，致力于为批处理应用的开发、运行提供应用框架形式的完备的支持、约束，统一解决与业务无关的技术层面的问题。

批量应用框架模块由：1、接口数据对象模型；2、应用框架；3、处理器和代理；4、调度器扩展；共4个子模块构成——



## 接口数据对象模型

接口数据对象模型子模块确立了批处理应用接口数据对象体系的基本结构。

## 应用框架

应用框架子模块确立了批处理应用对象体系的基本结构，建立了应用功能实现的描述、登记和发现机制。在系统级，为应用和配置的加载，运行时管控提供统一支持；在交易（具体批处理任务的处理）级，统一完成资源（数据库链接、事务、日志号计数器、...）准备和释放、应用对象实例化和调度、日志记录（支持堵重）、错误/异常处理，并统一对外提供查询处理结果、重做等接口，使应用设计可以得到最大程度的简化。

## 处理器和代理

处理器和代理子模块提供了一个通信应用框架上的应用插件，它解析客户端系统（例如批量调度平台）提交的请求报文（主体部分是一个XML），调用应用框架接口进行处理，并组装应答报文（主体部分是一个XML）回发给客户端系统——即所谓的处理器。报文解析和组装都是调用接口数据对象的OXM方法进行。

处理器和代理子模块还为客户端系统提供了一组支持其基于接口数据对象提交批处理任务的API，它自动建立、维护通信通道，并自动完成符合处理器格式要求的请求报文的组装和应答报文的解析——即所谓的代理。

## 调度器扩展

调度器扩展子模块提供了一组扩展的INTIGER调度指令，通过代理对接处理器提交批处理任务，并支持更加智能（例如动态任务分配）的批量调度流程设计。

# 交易主控

交易主控（Transaction Control，简称TC）模块致力于实现核心系统联机和批量交易应用公共的控制和处理设计，为确立整个核心系统交易应用开发的基本模式和规范奠定了基础。

目前，交易主控模块由：1、业务组件模型和公共组件；2、参数组件；3、联机交易应用模型；4、联机交易处理器；5、批处理交易应用模型；共5个子模块构成——



## 业务组件模型和公共组件

业务组件模型和公共组件子模块确立了业务组件对象体系的基本结构，并提供了一组公共的业务控制和处理组件（各种检查，交易环境初始化，交易统一的数据加工处理和登记，等等）。

## 参数组件

参数组件子模块负责参数加载及日切时的版本更新，为上层应用提供统一的参数访问接口。

## 联机交易应用模型

联机交易应用模型子模块以服务应用框架模块中的应用模型为基础构建，确立了联机交易应用对象体系的基本结构，包括联机交易应用对象、接口数据对象和上下文数据对象（TWA、TIA、TOA、...）。联机交易应用基类方法负责调用联机交易应用对象接口完成交易数据处理，并调用公共组件对象接口完成交易数据处理前后统一的控制处理。联机交易应用基类方法还统一提供了列表数据翻页查询功能。

## 联机交易处理器

联机交易处理器子模块提供了一个通信应用框架上的应用插件，它解析客户端系统提交的符合中信银行现有核心系统接口规范的定长/分隔符格式和XML格式的请求报文，调用公共组件对象接口完成交易系统外围检查，再调用服务应用框架接口（会逐层调用到联机交易应用基类方法和联机交易应用对象数据处理接口）获得交易系统处理结果，并组装应答报文回发给客户端系统。

## 批处理交易应用模型

批处理交易应用模型子模块以批量应用框架模块中的应用模型为基础构建，确立了批处理交易应用对象体系的基本结构，包括批处理交易应用对象、接口数据对象和上下文数据对象。批处理交易应用基类方法负责调用批处理交易应用对象接口完成交易数据处理，并调用公共组件对象接口完成交易数据处理前后统一的控制处理。