OMG DDS 规范
v1.4 中文版

南京磐优信息科技有限公司 译

目　录

1. 概论

1.1 引言

DDS 规范为分布式应用的通信和集成提供了一套***以数据为中心的发布订阅（DCPS）***通信模型，规范同时定义了应用接口和通信语义（行为和服务质量），在信息产生者与匹配的消费者之间提供了一条高效的信息传输通道。

DDS 规范的目的概括来说就是“为实现在正确的时间将正确的信息传递到正确的目的地提供高效且高鲁棒性的传输”

可预料的应用领域要求以数据为中心的发布订阅模型在使用资源的过程中保持高性能、可预测性和高效性。为了满足以上需求，在进行接口设计时必须保证按照以下方式设计：

允许中间件提前分配资源以便动态分配的资源减少到最低；

避免提供需要使用无限或者不可预测的资源实现的功能；

尽可能的减少数据拷贝。

DDS 最大程度地采用基于类型的接口（即考虑实际的数据类型进行设计的接口）。基于类型的接口主要有以下优点：

简单易用：程序开发人员可以直接处理自然象征数据的类型设计；

安全可用：在编译阶段即可进行验证；

高效好用：依赖于提前了解的精确数据类型来设计执行代码可以预分配资源使用。

需要注意的一点是，决定使用基于类型的接口意味着需要一款可以将类型定义转换成合适的接口和实现的生成工具，进而弥补基于类型的接口和通用中间件的不足。

服务质量（Quality of Service，QoS）是用来指定服务行为的一般概念。通过设置 QoS 的方式进行的编程服务行为具有以下优势，即应用开发者只用表明需要“什么”而不是“如何”获取该项服务质量。一般来说，QoS 由数个 QoS 策略组成。每一个 QoS 策略都是由一个名字和一个数值组成的独立的描述。通过采用一系列独立的 QoS 策略来表示 QoS 带来了更大的灵活性。

本规范设计时允许将发布端和订阅端进行明确的划分，仅作为发布者参与的应用进程可以嵌入与发布严格相关的内容。同理，仅作为订阅者参与的应用进程可以嵌入与订阅严格相关的内容。

1.2 宗旨

许多实时应用需要将他们的一些通信模式建模为纯粹的以数据为中心的交换模式，在这种模式下应用发布（提供或者流出）数据，对这些数据感兴趣的远端应用可以得到这些数据。相关实时应用可以在 C4I、工业自动化、分布式控制与仿真、通信设备控制、传感器网络和网络管理系统中找到。更普遍地，任何需要（选择性）信息传播的应用程序都是数据驱动网络架构的候选者。

这些实时应用主要关注以最小的开销实现可预测的数据分发。由于无限扩展需要的资源是不可行的，所以能够指定可用资源并提供允许中间件将资源分配给最关键需求的策略非常重要。这种必要性转化为控制服务质量（QoS）属性的能力，影响到可预测性，开销和资源

利用率。

以强大的方式扩展到数百或数千个发布者和订阅者的需求也是一个重要的要求。实际上这不仅仅是扩展性的需求，而且也是灵活性的需求：在许多这样的系统中，添加应用程序不需要或者说不可能重构整个系统。以数据为中心的通信模型将发送者与接收者分离；发布者和订阅者耦合程度越低，这些扩展就越容易。

分布式共享内存是提供以数据为中心的数据交换方式的经典模型。但是，这种模式很难在网络上高效实现，并且不提供所需的可扩展性和灵活性。因此，另一个以数据为中心的发布订阅通信模型在许多实时应用领域越来越流行。该模型基于所有感兴趣的应用程序都可访问的“全局数据空间”的概念，希望向此数据空间贡献信息的应用程序声明其意图成为“发布者”；类似地，希望获取此数据空间部分数据的应用程序声明其意图成为“订阅者”。每当发布者将新数据发布到这个“全局数据空间”时，中间件就将信息传播给所有感兴趣的订阅者。

任何以数据为中心的发布订阅模型的底层都是数据模型。该数据模型定义了“全局数据空间”并指定发布者和订阅者如何访问此数据空间。该数据模型可以像一组不相关的数据结构一样简单，每个数据结构都由一个主题和一个数据类型来标识。主题提供了一个唯一标识全局数据空间内某些数据项的标识符。数据类型提供了所需的结构信息，该结构信息告知中间件如何操作数据并允许中间件提供数据类型安全级别。但是，目标应用程序通常需要更高级别的数据模型，以允许表达数据元素之间的聚合和一致性关系。

2 以数据为中心的发布订阅模型

2.1 概要

本节介绍了以数据为中心的发布订阅（DCPS）模型。DCPS 模型定义了应用程序用于发布或订阅数据对象数值的功能。该模型允许：

·发布程序识别其计划发布的数据对象，并为这些对象赋值；

·订阅程序识别其感兴趣的数据对象，并访问这些对象的值；

·应用程序定义主题，将数据类型信息添加到该主题上，允许应用程序创建发布者或者订阅者实体，并将 QoS 策略添加到这些实体上，最终可以操作这些实体。

本节内容主要分为以下两个子条款：

·平台无关模型（PIM）；

·基于 PIM 的 OMG IDL 平台特点设计的模型，即平台特定模型（PSM）。

2.2 平台无关模型（PIM）

2.2.1 概述及设计原理

2.2.1.1 格式及约定

本子条款的目的是提供以数据为中心的发布订阅模型的平台无关模型的操作概述。为此，本子条款引入了很多术语。其中一些是常见术语，其在发布/订阅的上下文环境中的含义与常见用法不同。在合适的情况下，这些条款将用*斜体* 表示。其他术语在发布/订阅模型和本规范中的含义是唯一的，并作为类模型的关键元素。第一次使用这些术语时，它们将以***粗斜体***格式展示。后续再出现不再以任何方式突出显示。

除了 UML 图之外，构成服务的所有类都使用表格来记录。用于记录这些类的文档格式如下所示。

<class name>		
attributes		
<attribute name>	<attribute type>	
...	...	
operations		
<operation name>		<return type>
	<parameter>	<parameter type>
	...	...
...		...

类方法部分的参数可以在参数名前面包含修饰符“in”、“out”或“inout”。 如果修饰符省略，则暗示参数是“in”参数。

在某些情况下，方法的参数或返回值是由包含部分元素的给定<类型>的集合。这表明了存在符号“<类型> []。”这种表示法并不意味着它将以数组形式实现。实际执行情况由平台特定模型（PSM）定义：它最终可能被映射到序列，列表或其他类型的集合。

例如，下面名为“MyClass”的类有一个属性，名为“my_attribute”，类型为“long”，一个简单的返回值类型为“long”的方法“my_operation”。该方法有四个参数。第一个参数“param1”是“long”型的输出参数；第二个参数“param2”是“long”型的输入输出参数。第三个参数“param3”是“long”型的输入参数（in 修饰符缺省）。第四个参数“param4”是“long”型的输入参数。

Myclass		
attributes		
my_attribute	long	
operations		
my_operation		long
	out: param1	long
	inout: param2	long
	param3	long
	in: param4	long

在平台无关模型（PIM）这一级别，我们将错误建模为类型为 ReturnCode_t 的返回码。每一种平台特定模型（PSM）可以将这些映射到任意返回码或异常。完整的返回码列表如下所示。

返回码	
OK	成功返回
ERROR	一般的、未指明的错误
BAD_PARAMETER	非法的参数值
UNSUPPORTED	不支持的方法，只能通过可选的方法返回
ALREADY_DELETED	此方法的对象目标已经被删除
OUT_OF_RESOURCES	服务耗完了完成方法所需的资源
NOT_ENABLED	某个***实体***的方法还未使能就被调用了
IMMUTABLE_POLICY	应用程序尝试修改一个不可变的 ***QoS 策略***
INCONSISTENT_POLICY	应用程序指定一组互相之间不一致的策略
PRECONDITION_NOT_MET	方法的前置条件未满足
TIMEOUT	方法超时
ILLEGAL_OPERATION	在不适当的对象上或在不适当的时间（由规范或服务实现设置的策略确定）调用方法，没有可以改变的前提条件能够使方法调用成功
NO_DATA	表示方法在没有出现内部错误时未返回任何数据的短暂状况

任意返回类型为 ReturnCode_t 的方法都可能返回 OK，ERROR 或 ILLEGAL_OPERATION。使用输入参数的任何方法都可以额外返回 BAD_PARAMETER。任意工厂创建的对象的任何方法都可能额外返回 ALREADY_DELETED。任何声明为可选的操作可以额外返回 UNSUPPORTED。返回码 OK，ERROR，ILLEGAL_OPERATION，ALREADY_DELETED，UNSUPPORTED 和 BAD_PARAMETER 是标准返回码，本规范不会为每个方法明确提及它们。可能返回上述任何其他（非标准）错误码的方法本规范将明确说明。

应用程序使用已通过工厂上相应的删除方法删除的实体是错误的。如果应用程序执行此操作，则结果未指定，并且将取决于实现和平台特定模型（PSM）。在可以检测到使用已删除实体的情况下，方法调用应该失败并返回 ALREADY_DELETED。

2.2.1.2 概念大纲

2.2.1.2.1 概述

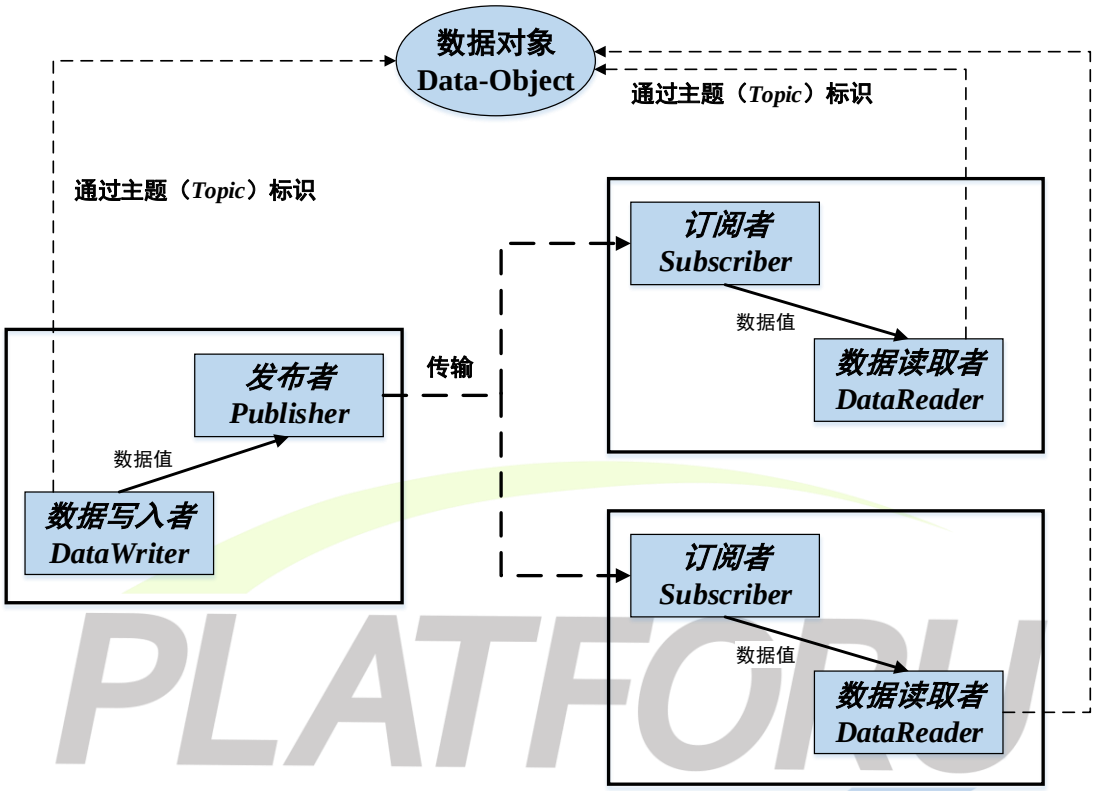

图 2-1 概述

信息流借助以下实体：发送方的***发布者（Publisher）***和***数据写入者（DataWriter）***；接收方的***订阅者（Subscriber）***和***数据读取者（DataReader）***。

- ***发布者***是负责数据分发的对象。它可以发布不同数据类型的数据。***数据写入者***充当发布者的类型访问者。***数据写入者***是应用程序用来向发布者传递给定类型的数据对象的存在和数值的必需对象。当数据对象值通过适当的数据写者传递给发布者时，发布者有责任执行分发（发布者将根据自己的 QoS 或相应数据写入者的 QoS 执行此操作）。一次发布是通过数据写入者与发布者的关联定义的。此关联表明应用程序有发布数据的意图，该数据由发布者提供的上下文中的数据写入者进行描述。

- ***订阅者***是负责接收已发布的数据并使其（根据订阅者的 QoS）可用于接收应用程序的对象。它可以接收和分发不同特定类型的数据。应用程序必须使用附加到订阅者的某一类型的***数据读取者***访问接收的数据。因此，一次*订阅*是通过数据读取者与订阅者的关联定义的。该关联表明应用程序有订阅数据的意图，该数据由订阅者提供的上下文中的数据读取者进行描述。

主题（Topic）对象在概念上适合发布和订阅。订阅必须明确地引用发布，发布必须以这样的方式为订阅所知。*主题*旨在实现该目的：它关联名称（数据域中唯一），数据类型和与数据本身相关的 QoS。除了主题 QoS 之外，与该主题相关联的*数据写入者（DataWriter）*的 QoS 和与*数据写入者*关联的*发布者*的 QoS 控制发布者这一方的行为，与此对应的*主题*，*数据读取者（DataReader）*和*订阅者 QoS* 控制订阅者这一方的行为。

当应用程序希望发布给定类型的数据时，它必须创建一个*发布者*（或使用已经创建的一个发布者）和一个具有所需发布的所有特征的*数据写入者*。类似地，当应用程序希望接收数据时，它必须创建*订阅者*（或重用已经创建的订阅者）和*数据读取者*来定义订阅。

2.2.1.2.2 整体概念模型

整体概念模型如图 2.2 所示。请注意，所有主要通信对象（*实体*的专业化）都遵循以下统一模式：

- 支持 QoS（由一些 QoS 策略组成）；QoS 为应用程序提供了一种通用机制，用于控制服务的行为并根据需要定制服务。每个*实体* 都支持其自己的专用 QoS 策略。完整的 QoS 策略列表及其含义在 2.2.3 小节中有详细描述。

- 接受*监听器（Listener）*；监听器为中间件提供通用机制，以通知应用程序相关的异步事件，例如订阅的数据到达，违反 QoS 设置等事件。每个 DCPS 实体支持其自己的专用类型监听器。监听器与状态条件的变化有关。完整的状态条件在 2.2.4 小节，Listeners，Conditions 和 Wait-sets 中有描述。

- 接受*状态条件（StatusCondition）*（以及数据读取者的一组*读取条件* 对象）；条件（与 *WaitSet* 对象一起）为中间件和应用程序之间的备用通信方式提供支持（即，基于等待而不是基于通知）。完整的状态条件集在 2.2.4 小节，监听器，条件和等待集中描述。

所有这些 DCPS 实体都附属于***域参与者（DomainParticipant）***。 域参与者表示域中应用程序的本地成员身份。*数据域*是一个分布式的概念，它链接所有的应用程序，使得这些应用程序能够相互通信。它代表一个通信平面：只有附属于同一数据域的发布者和订阅者才可以相互通信。

*数据域实体（DomainEntity）*是一个中间对象，其唯一目的是声明***域参与者*** 不能包含其他域参与成员。

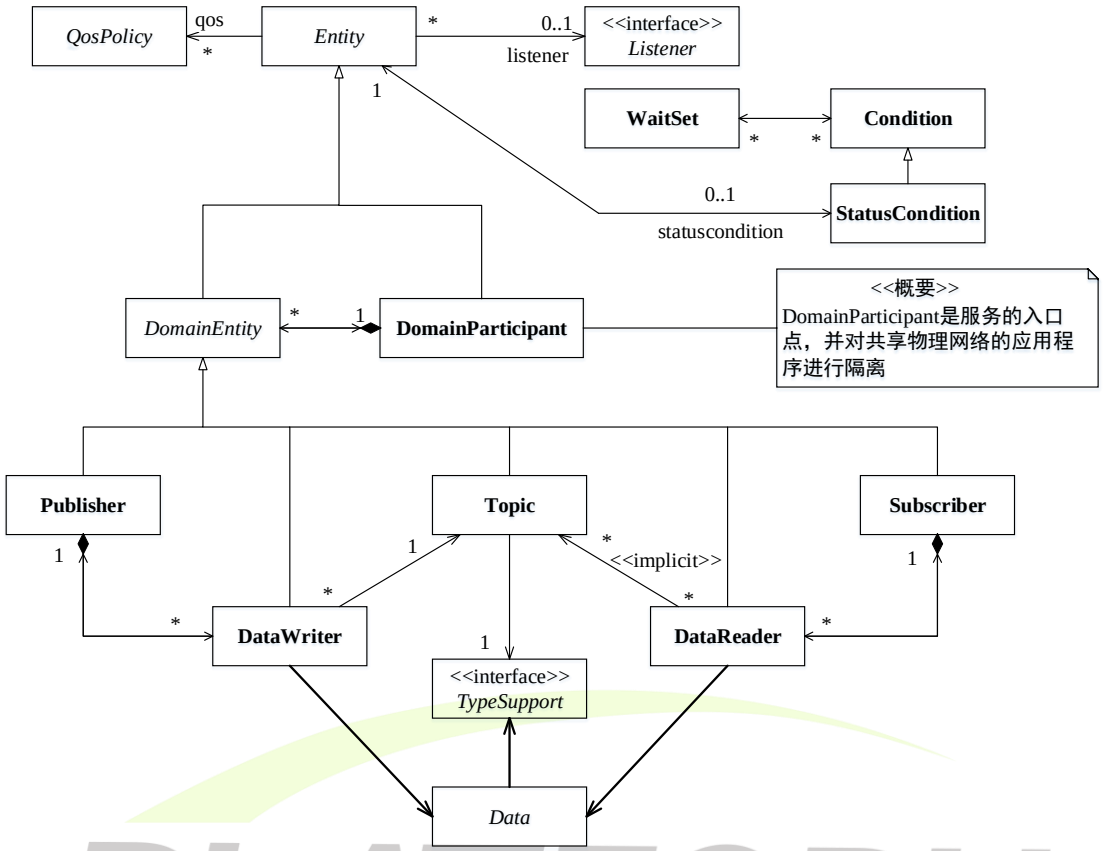

图 2-2 DCPS 概念模型

在 DCPS 这一级，数据类型意味着信息将会以原子形式发送出去。

默认情况下，每一次数据更改都是单独且独立地传播，与其他更改不相关。但是，应用程序可能会要求发送方将多次修改当做一次完整的数据发送出去，并在接收方进行对应的解析。此功能将以***发布者/订阅者（Publisher/Subscriber）***为基础而实现。也就是说，这些关系只能在隶属于同一***发布者（Publisher）***的***数据写入者（DataWriter）***对象中指定，并在隶属于同一***订阅者（Subscriber）***的***数据读取者（DataReader）***对象中接收恢复。

根据定义，***主题*** 对应单个数据类型。但是，多个主题可能指的是相同的数据类型。因此，***主题*** 标识单个类型的数据，范围从单个实例到该给定类型的整个实例集合。通过假设的数据类型“Foo”进行展示，如图 2.3 所示。

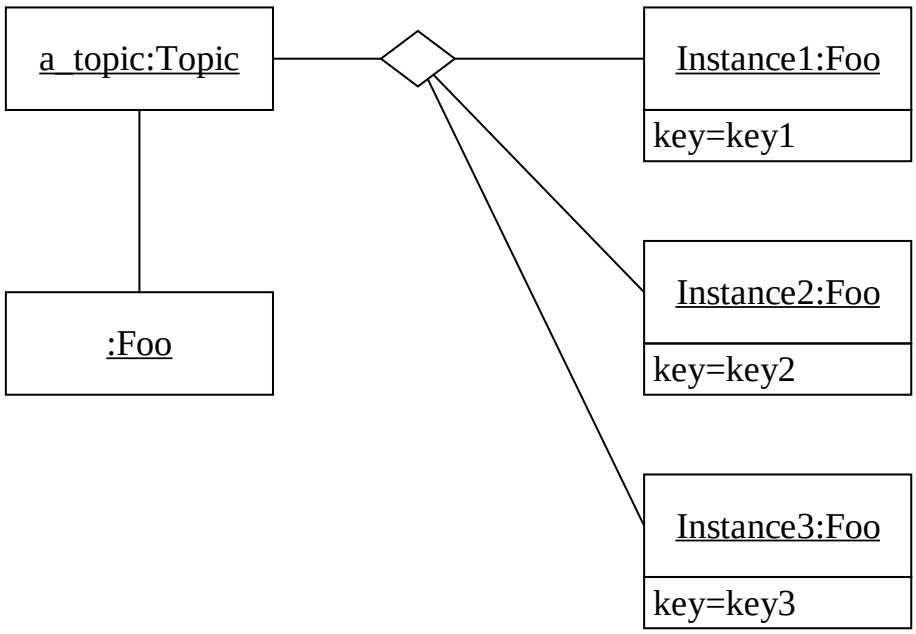

图 2-3 主题可以标识数据对象实例的集合

如果在同一主题下收集了一组实例，则必须区分不同的实例。通过指定某些数据字段为该数据集的***关键字*** 来实现这种区分。必须向中间件告知***关键字*** 的描述（即，组成关键字的数据字段列表）。规则很简单：具有相同键值的不同数值表示同一数据实例的连续值，而具有不同键值的不同数值表示不同的数据实例。如果未提供***关键字***，则与***主题*** 相关的数据集仅存在一个数据实例。

中间件需要知道主题并且该主题可能会进行数据传输。***主题*** 对象是通过***域参与者*** 提供的创建方法创建的。

发布者方面的交互方式很简单：当应用程序决定要使数据可供发布时，它会调用关联的***数据写入者*** 的相应操作（这将触发其所属的***发布者***）。然而，在订阅者方面，还有更多选择：当应用程序忙于执行其他操作或应用程序只是等待该信息时，相关信息可能会到达。因此，根据应用程序的设计方式，异步通知或同步访问可能更合适。中间件允许两种交互模式，***监听器（Listener）***用于为同步数据访问提供回调，与一个或多个 ***Condition*** 对象关联的 ***WaitSet*** 提供异步数据访问。

相同的同步和异步交互模式也可用于访问某些改变，这些改变会影响中间件通信状态。例如，当中间件异步检测到不一致时，可能通过这两种方式进行通知。此外，使用内置主题这种方式，应用程序可以作为普通应用数据，通过内置的数据读取者提供可能与应用程序相关的其他中间件信息（例如现有主题的列表）。

2.2.2 平台无关模型（PIM）描述

DCPS 由五大模块组成。

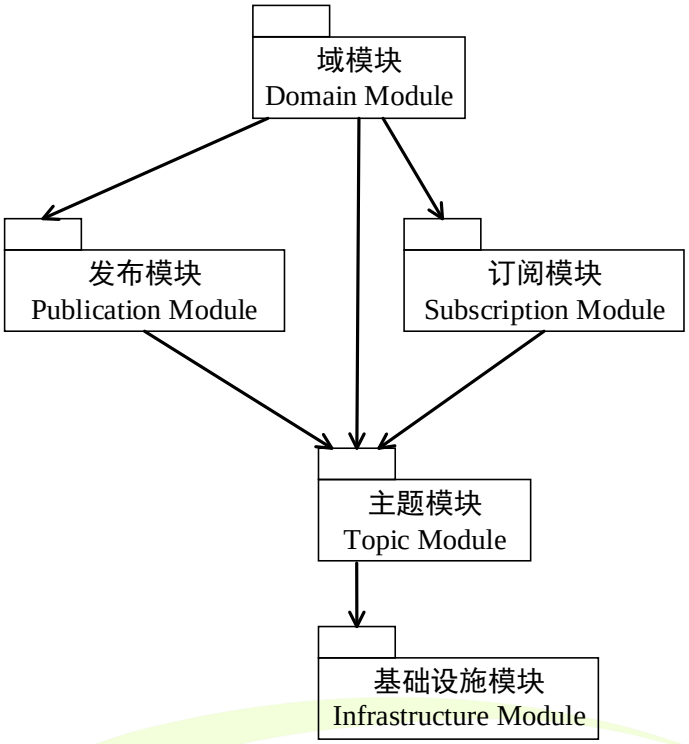

图 2-4 DCPS 模块分类

1. 基础设施模块定义了抽象类和由其他模块细化的接口。它还为中间件提供了两种交互方式（基于通知和等待）的支持。
2. 域模块包含 ***DomainParticipant*** 类，它充当中间件服务的入口点，并充当许多类的工厂。***DomainParticipant*** 类还充当组成服务的其他对象的容器。
3. 主题定义模块包含 ***Topic***，***ContentFilteredTopic***，***MultiTopic*** 类和 ***TopicListener*** 接口，也就是主题定义模块包含应用程序所需用来定义 ***Topic*** 对象并将 QoS 策略附加到对象上的全部内容。
4. 发布模块包含 ***Publisher*** 和 ***DataWriter*** 类以及 ***PublisherListener*** 和 ***DataWriterListener*** 接口，也就是包含发布端所需的全部内容。
5. 订阅模块包含 ***Subscriber***，***DataReader***，***ReadCondition*** 和 ***QueryCondition*** 类，以及 ***SubscriberListener*** 和 ***DataReaderListener*** 接口，也就是包含订阅方所需的全部内容。

在 PIM 级别，我们选择将任何实体建模为类或接口。然而值得注意的是这并不意味着它们中的任何一个都将被转换为 IDL 接口。通常我们选择将应用程序必须扩展的实体建模为接口，以便与中间件服务进行交互。其余实体已建模为类。

2.2.2.1 基础设施模块

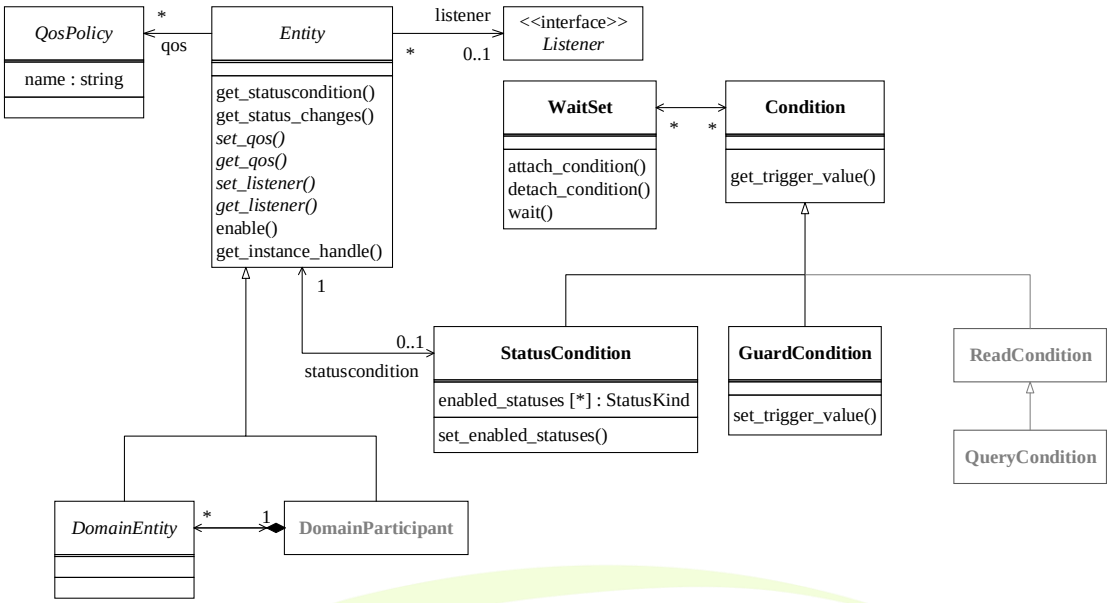

图 2-5 DCPS 基础设施模块的类模型

DCPS 基础设施模块由以下类组成：

- Entity
- DomainEntity
- QosPolicy
- Listener
- Status
- WaitSet
- Condition
- GuardCondition
- StatusCondition

2.2.2.1.1 Entity 类

此类是支持 QoS 策略，监听器和状态条件的所有 DCPS 对象的抽象基类。

Entity		
no attributes		
operations		
abstract set_qos		ReturnCode_t
	qos_list	QosPolicy []
abstract get_qos		ReturnCode_t
	out: qos_list	QosPolicy []
abstract set_listener		ReturnCode_t
	a_listener	Listener
	mask	StatusKind []
abstract get_listener		Listener
get_statuscondition		StatusCondition
get_status_changes		StatusKind []
enable		ReturnCode_t
get_instance_handle		InstanceHandle_t

StatusKind 是一个枚举类型，用于标识每个具体的 ***Status*** 类型。

以下子条款详细说明了 Entity 类的所有方法。

2.2.2.1.1.1 set_qos(抽象方法)

此方法用于设置实体的 QoS 策略，必须由每个派生的实体类（***DomainParticipant，Topic，Publisher，DataWriter，Subscriber，DataReader***）提供，以便可以设置对每个实体有意义的策略。

通过 *qos_list* 参数指定的策略集应用于现有 QoS 之上，替换先前设置的任何策略值。

如 2.2.3 支持的 QoS 所述，中某些策略是“不可变的”，它们只能在创建实体时设置，或者在实体启用之前设置。如果在实体启用后调用 *set_qos* 尝试更改 “不可变”策略的值，此操作将失败并返回 IMMUTABLE_POLICY。

2.2.3 支持的 QoS 小节还描述了 QoS 策略的某些值可能与其他策略的设置不兼容。如果通过 *set_qos* 方法指定一组 QoS 值，与现有 QoS 值组合将导致不一致的策略，则 set_qos 方法也将失败。在这种情况下，返回值为 INCONSISTENT_POLICY。

如果应用程序为服务不支持的 QoS 策略提供非默认值，则 *set_qos* 方法将失败并返回 UNSUPPORTED。

仅当 *set_qos* 方法成功才会更改现有策略集，此时返回 OK。在所有其他情况下，没有任何策略会被修改。

每个派生的实体类（***DomainParticipant，Topic，Publisher，DataWriter，Subscriber，DataReader***）都具有相应的 QoS 特殊值（PARTICIPANT_QOS_DEFAULT，PUBLISHER_QOS_DEFAULT，SUBSCRIBER_QOS_DEFAULT，TOPIC_QOS_DEFAULT，DATAWRITER_QOS_DEFAULT，DATAREADER_QOS_DEFAULT）。此特殊值可用作 **set_qos** 方法的参数，意味着应更改实体的 QoS 以匹配实体工厂中的当前默认 QoS 集。set_qos 无法修改不可变的 QoS，因此成功返回值表明已修改实体的可变 QoS 以匹配实体工厂的当前默认值。

除标准返回错误代码外，还可能返回错误代码：INCONSISTENT_POLICY，IMMUTABLE_POLICY。

2.2.2.1.1.2 get_qos（抽象方法）

此方法允许访问实体的现有 QoS 策略集，必须由每个派生的实体类（***DomainParticipant，Topic，Publisher，DataWriter，Subscriber，DataReader***）提供，以便获取对特定实体有意义的策略。

2.2.2.1.1.3 set_listener（抽象方法）

此方法在实体上安装监听器。只有在特定掩码代表的通信状态更改时才会调用监听器。

允许使用'nil'作为监听器的值。“nil”监听器不执行任何操作。

每个实体只能附加一个监听器。如果已经设置了监听器，*set_listener* 方法将使用新的替换它。因此，如果将值“nil”当做 *set_listener* 方法的监听器参数，则将删除任何现有监听器。

此方法必须由每个派生的实体类（***DomainParticipant，Topic，Publisher，DataWriter，Subscriber，DataReader***）提供，以便监听器具有适合特定实体的具体类型。

2.2.2.1.1.4 get_listener（抽象方法）

此方法允许访问附加到实体的现有监听器。

此方法必须由每个派生的实体类（***DomainParticipant，Topic，Publisher，DataWriter，Subscriber，DataReader***）提供，以便监听器具有适合特定实体的具体类型。

2.2.2.1.1.5 get_statuscondition

此方法允许访问与实体关联的***状态条件***（2.2.2.1.9，StatusCondition 类），可以将返回的条件添加到***等待集***（2.2.2.1.6，WaitSet 类），以便应用程序可以等待影响实体的特定状态的更改。

2.2.2.1.1.6 get_status_changes

此方法将检索实体中“已触发”的通信状态列表。即自上次应用程序读取状态以来其值更改的状态列表。 2.2.4.2 状态变化给出了通信状态“触发”的准确定义。

首次创建实体或未启用实体时，所有通信状态都处于“未触发”状态，因此 get_status_changes 方法返回的列表将为空。

get_status_changes 方法返回的状态列表是指在实体自身触发的状态，不包括适用于包含实体的状态。

2.2.2.1.1.7 enable

此方法启用实体。可以启用或禁用实体对象。这由实体相应的工厂上 ENTITY_FACTORY Qos 策略（2.2.3.20，ENTITY_FACTORY）的值控制。

默认情况下，ENTITY_FACTORY 的设置是不必在新创建的实体上显式调用 *enable* 方法（请参阅 2.2.3.20，ENTITY_FACTORY）。

enable 方法是幂等的。在已启用的 Entity 上调用 enable 将返回 OK 并且无效。

如果尚未启用实体，则可以在其上调用以下类型的操作：

- 设置或获取实体的 QoS 策略（包括默认 QoS 策略）和监听器的方法
- get_statuscondition
- “工厂”方法

•get_status_changes 和其他获取状态方法（尽管已禁用的实体状态永远不会更改）

•“查找”操作

其他方法可以明确声明可以在禁用的实体上调用它们，这种情况不会返回错误返回值 NOT_ENABLED。

通过工厂调用正确的方法来删除尚未启用的实体是合法的。

无论 ENTITY_FACTORY Qos 策略的设置如何，通过已禁用的工厂创建的*实体*都会被禁用。

在未启用工厂的*实体*上调用 enable 将失败并返回 PRECONDITION_NOT_MET。

如果 ENTITY_FACTORY QoS 策略将 autoenable_created_entities 设置为 TRUE，则工厂上的 *enable* 方法将自动启用从工厂创建的所有实体。

在启用实体之前，不会调用与实体关联的监听器。与未启用的实体关联的条件是“非活动”，即 ***trigger_value*** == FALSE（请参阅 2.2.4.4，条件和等待集）。

2.2.2.1.1.8 get_instance_handle

此方法返回表示*实体*的 InstanceHandle_t。

2.2.2.1.2 DomainEntity 类

DomainEntity 是所有 DCPS 实体的抽象基类，***DomainParticipant*** 除外。其唯一目的是表示 ***DomainParticipant*** 是一种特殊的实体，它充当所有其他*实体*的容器，但本身不能包含其他 ***DomainParticipant***。

DomainEntity
no attributes
no operations

2.2.2.1.3 QosPolicy 类

QosPolicy	
attributes	
name	string
no operations	

QosPolicy 类为应用程序提供了指定服务质量参数的基本机制，它具有一个属性 *name*，用于唯一标识每个 QoS 策略。所有具体的 ***QosPolicy*** 类都派生自此基类，并包含一个类型取决于具体 QoS 策略的值。

QosPolicy 值的类型可以是原子的，例如整数或浮点数，或复合类型（结构）。需要连续地设置多个参数以定义 ***QosPolicy*** 的一致值时，就会使用复合类型。

可以使用 ***QosPolicy*** 列表配置每个实体。但是，任一实体无法支持任意 ***QosPolicy***。例如，***DomainParticipant*** 支持与 ***Topic*** 或 ***Publisher*** 不同的 QosPolicy。

可以在创建实体时设置 ***QosPolicy***，也可以使用 ***set_qos*** 方法修改 ***QosPolicy***。列表中的每个 ***QosPolicy*** 都独立于其他 ***QosPolicy*** 进行处理，这种方法具有可扩展的优点。但是也可能存在多个策略发生冲突的情况。每次通过 ***set_qos*** 方法修改策略时，都会执行一致性检查。

在某个策略设置为给定值后需要更改时，不强制要求立即应用新值，允许服务在过渡阶

段后应用它。此外，一些 ***QosPolicy*** 具有“不可变”语义，这意味着它们只能在实体创建时指定，或者在实体上调用启用操作之前指定。

小节 2.2.3 支持的 QoS 提供所有 ***QosPolicy*** 的列表，包括它们的含义，特征和可能的值，以及它们适用的具体实体。

2.2.2.1.4 Listener 接口

Listener 是所有***监听器***接口的抽象根类。所有受支持的具体监听器接口（每个实体一个：***DomainParticipant***，***Topic***，***Publisher***，***DataWriter***，***Subscriber*** 和 ***DataReader***）都派生自此根类，并添加原型依赖于具体监听器的方法。

Listener
no attributes
no operations

有关已定义监听器接口的列表，请参见 2.2.4.3 通过监听器访问。监听器接口为服务提供了一种机制，此机制可以异步地通知应用程序通信状态的相关变化。

2.2.2.1.5 Status 类

Status 类是所有通信状态对象的抽象根类。所有具体的 ***Status*** 类都是此类的特殊化。

Status
no attributes
no operations

每个具体实体与一组 Status 对象相关联，其值表示该实体的“通信状态”。可以通过实体上的相应方法访问这些状态值。这些状态值的更改既会激活相应的 ***StatusCondition*** 对象，又会触发合适的 ***Listener*** 对象的调用，从而异步通知应用程序。

Status 对象及其与 ***Listener*** 和 ***Condition*** 对象的关系详见 2.2.4.1 通信状态。

2.2.2.1.6 WaitSet 类

WaitSet 对象允许应用程序等待，直到一个或多个附加的 ***Condition*** 对象的 ***trigger_value*** 为 ***TRUE***，否则直到超时到期。

WaitSet		
no attributes		
operations		
attach_condition		ReturnCode_t
	a_condition	Condition
detach_condition		ReturnCode_t
	a_condition	Condition
wait		ReturnCode_t

	inout: active_conditions	Condition []
	timeout	Duration_t
get_conditions		ReturnCode_t
	inout: attached_conditions	Condition []

WaitSet 类没有工厂。它通过编程语言中自然的方式直接创建对象（例如使用 C ++或 Java 中的“new”）。这是因为它不一定与单个 ***DomainParticipant*** 相关联，并且可以用于等待与不同 ***DomainParticipant*** 对象关联的 ***Condition*** 对象。

以下子项详细说明了所有方法。

2.2.2.1.6.1 attach_condition

将条件 ***Condition*** 附加到 ***WaitSet*** 中。

可以在当前（通过 ***wait*** 方法）正在等待的 ***WaitSet*** 上附加 ***Condition***。在这种情况下，如果 ***Condition*** 的 ***trigger_value*** 为 ***TRUE***，则附加条件将取消阻止 ***WaitSet***。

添加已附加到 ***WaitSet*** 的 ***Condition*** 无效。

除标准错误代码外，还可能返回错误代码：OUT_OF_RESOURCES。

2.2.2.1.6.2 detach_condition

从 ***WaitSet*** 中分离条件 ***Condition***。

如果条件未附加到 ***WaitSet***，则方法将返回 PRECONDITION_NOT_MET。

除标准错误代码外，还可能返回错误代码：PRECONDITION_NOT_MET。

2.2.2.1.6.3 wait

此方法允许应用程序线程等待某些条件的发生。如果附加到 ***WaitSet*** 的所有条件的 ***trigger_value*** 都不为 ***TRUE***，则 ***wait*** 方法将阻塞并挂起调用线程。

wait 操作的结果是所有 ***trigger_value*** 为 ***TRUE*** 的附加条件的列表（即取消等待阻塞的条件）。

wait 操作采用超时参数指定等待的最长持续时间。超过此持续时间并且没有附加的 ***Condition*** 对象为 true，wait 将返回 TIMEOUT。

不允许多个应用程序线程在同一个 ***WaitSet*** 上等待。如果在已经有线程阻塞的 ***WaitSet*** 上调用 ***wait*** 操作，则操作将立即返回 PRECONDITION_NOT_MET。

2.2.2.1.6.4 get_conditions

此方法获取附加条件列表。

2.2.2.1.7 Condition 类

Condition 类是所有可以附加到 ***WaitSet*** 的条件的根类。这个基类专门用于中间件已知的三个类：***GuardCondition***（2.2.2.1.8），***StatusCondition***（2.2.2.1.9）和 ***ReadCondition***（2.2.2.5.8）。

Condition
no attributes
operations

get_trigger_value		boolean

Condition 的 ***trigger_value*** 可以为 TRUE 或 FALSE，由服务自动设置。

2.2.2.1.7.1 get_trigger_value

此操作获取 ***Condition*** 的 ***trigger_value***。

2.2.2.1.8 GuardCondition 类

GuardCondition 对象是一个特殊的 ***Condition***，其 ***trigger_value*** 完全在应用程序的控制之下。

GuardCondition		
no attributes		
operations		
set_trigger_value		ReturnCode_t
	value	boolean

GuardCondition 类没有工厂。它通过编程语言中自然的方式直接创建对象（例如，在 C++或 Java 中使用“new”）。首次创建时，***trigger_value*** 设置为 FALSE。

GuardCondition 类的目的是为应用程序提供手动唤醒 ***WaitSet*** 的方法。通过将 ***GuardCondition*** 附加到 ***WaitSet***，然后调用 ***set_trigger_value*** 方法设置 ***trigger_value*** 来实现。

2.2.2.1.8.1 set_trigger_value

此方法设置 ***GuardCondition*** 的 ***trigger_value***。

WaitSet 对象的行为取决于其附加条件的 ***trigger_value*** 的更改。因此，***GuardCondition*** 附加到的任何 ***WaitSet*** 都可能受此方法影响。

2.2.2.1.9 StatusCondition 类

StatusCondition 对象是与每个实体关联的特定条件。

StatusCondition		
no attributes		
operations		
set_enabled_statuses		ReturnCode_t
	mask	StatusKind []
get_enabled_statuses		StatusKind []
get_entity		Entity

StatusCondition 的 ***trigger_value*** 取决于该实体的通信状态（例如，数据到达，信息丢失等），由 ***StatusCondition*** 上的 ***enabled_statuses*** 集“过滤”。

enabled_statuses 及其与 ***Listener*** 和 ***WaitSet*** 的关系详见 2.2.4.4.1 StatusCondition 的触发状态。

2.2.2.1.9.1 set_enabled_statuses

此方法定义了确定 ***StatusCondition*** 的 ***trigger_value*** 时需要考虑的通信状态列表，可能会更改 ***StatusCondition*** 的 ***trigger_value***。

WaitSet 对象的行为取决于其附加条件的 ***trigger_value*** 的更改。因此，***StatusCondition*** 附加到的任何 ***WaitSet*** 都可能受此方法的影响。

如果未调用此函数，则启用状态的默认列表将包括所有状态。

2.2.2.1.9.2 get_enabled_statuses

此方法获取需要考虑的通信状态列表，以确定 ***StatusCondition*** 的 ***trigger_value***，方法返回在上次调用 ***set_enabled_statuses*** 时显式设置的状态；如果从未调用 ***set_enabled_statuses***，则返回默认列表（请参阅 2.2.2.1.9.1）。

2.2.2.1.9.3 get_entity

此方法返回与 ***StatusCondition*** 关联的实体。请注意，每个 ***StatusCondition*** 只关联一个实体。

2.2.2.2 域模块

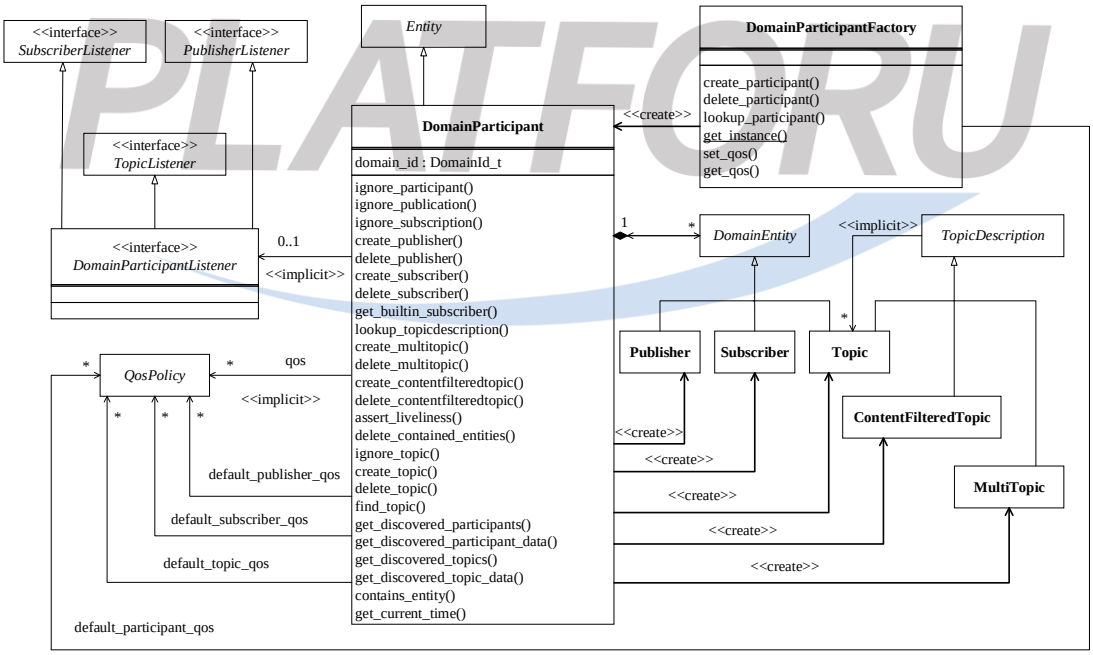

图 2-6 DCPS 域模块的类模型

DCPS 域模块由以下类组成：

- DomainParticipant
- DomainParticipantFactory
- DomainParticipantListener

2.2.2.2.1 DomainParticipant 类

DomainParticipant 对象扮演多个角色：

•它充当所有其他 ***Entity*** 对象的容器。

•它充当 ***Publisher***，***Subscriber***，***Topic*** 和 ***MultiTopic Entity*** 对象的工厂。

•它表示应用程序在通信平面上的参与，该通信平面将在同一物理计算机上运行的应用程序彼此隔离。域建立“虚拟网络”，链接共享相同 ***domainId*** 的所有应用程序，并将它们与在不同域上运行的应用程序隔离。通过这种方式，独立的分布式应用程序可以在同一物理网络中共存，而不会干扰甚至了解彼此。

•它在域中提供管理服务，允许应用程序在本地“忽略”与给定参与者（***ignore_participant***），发布（***ignore_publication***），订阅（***ignore_subscription***）或主题（***ignore_topic***）有关的任何信息。

DomainParticipant		
no attributes		
operations		
(inherited) get_qos		ReturnCode_t
	out: qos_list	QosPolicy []
(inherited) set_qos		ReturnCode_t
	qos_list	QosPolicy []
(inherited) get_listener		Listener
(inherited) set_listener		ReturnCode_t
	a_listener	Listener
	mask	StatusKind []
create_publisher		Publisher
	qos_list	QosPolicy []
	a_listener	PublisherListener
	mask	StatusKind []
delete_publisher		ReturnCode_t
	a_publisher	Publisher
create_subscriber		Subscriber
	qos_list	QosPolicy []
	a_listener	SubscriberListener
	mask	StatusKind []
delete_subscriber		ReturnCode_t
	a_subscriber	Subscriber
create_topic		Topic
	topic_name	string
	type_name	string
	qos_list	QosPolicy []
	a_listener	TopicListener
	mask	StatusKind []
delete_topic		ReturnCode_t
	a_topic	Topic
create_contentfilteredtopic		ContentFilteredTopic
	name	string

	related_topic	Topic
	filter_expression	string
	expression_parameters	string []
delete_contentfilteredtopic		ReturnCode_t
	a_contentfilteredtopic	ContentFilteredTopic
create_multitopic		MultiTopic
	name	string
	type_name	string
	subscription_expression	string
	expression_parameters	string []
delete_multitopic		ReturnCode_t
	a_multitopic	MultiTopic
find_topic		Topic
	topic_name	string
	timeout	Duration_t
lookup_topicdescription		TopicDescription
	name	string
get_builtin_subscriber		Subscriber
ignore_participant		ReturnCode_t
	handle	InstanceHandle_t
ignore_topic		ReturnCode_t
	handle	InstanceHandle_t
ignore_publication		ReturnCode_t
	handle	InstanceHandle_t
ignore_subscription		ReturnCode_t
	handle	InstanceHandle_t
get_domain_id		DomainId_t
delete_contained_entities		ReturnCode_t
assert_liveliness		ReturnCode_t
set_default_publisher_qos		ReturnCode_t
	qos_list	QosPolicy []
get_default_publisher_qos		ReturnCode_t
	out: qos_list	QosPolicy []
set_default_subscriber_qos		ReturnCode_t
	qos_list	QosPolicy []
get_default_subscriber_qos		ReturnCode_t
	out: qos_list	QosPolicy []
set_default_topic_qos		ReturnCode_t
	qos_list	QosPolicy []
get_default_topic_qos		ReturnCode_t
	out: qos_list	QosPolicy []
get_discovered_participants		ReturnCode_t
	inout: participant_handles	InstanceHandle_t []

get_discovered_participant_data		ReturnCode_t
	inout: participant_data	ParticipantBuiltin-TopicData
	participant_handle	InstanceHandle_t
get_discovered_topics		ReturnCode_t
	inout: topic_handles	InstanceHandle_t []
get_discovered_topic_data		ReturnCode_t
	inout: topic_data	TopicBuiltinTopicData
	topic_handle	InstanceHandle_t
contains_entity		boolean
	a_handle	InstanceHandle_t
get_current_time		ReturnCode_t
	inout: current_time	Time_t

以下详细说明了所有方法。

即使未启用 ***DomainParticipant***，也可以调用以下方法。除此以外如果在禁用的 ***DomainParticipant*** 上调用其他方法，则返回 NOT_ENABLED：

- 在基类级别定义的方法，即 ***set_qos***，***get_qos***，***set_listener***，***get_listener*** 和 ***enable***。
- 工厂方法：**create_topic**，**create_publisher**，**create_subscriber**，**delete_topic**，**delete_publisher**，**delete_subscriber**
- 访问状态的方法：**get_statuscondition**

2.2.2.2.1.1 create_publisher

此方法将创建具有所需 QoS 策略的 ***Publisher***，并将指定的 ***PublisherListener*** 附加到该 ***Publisher***。

如果指定的 QoS 策略不一致，方法调用将失败，且不会创建 ***Publisher***。

特殊值 PUBLISHER_QOS_DEFAULT 表明应使用工厂中设置的默认 ***Publisher*** QoS 创建 ***Publisher***。使用这个特殊值等同于应用程序通过 get_default_publisher_qos（2.2.2.2.1.21）方法获取默认 ***Publisher*** QoS 并使用返回的 QoS 创建 ***Publisher***。

创建的 ***Publisher*** 隶属于作为其工厂的 ***DomainParticipant***。

如果方法调用失败，将返回“nil”值（由平台指定）。

2.2.2.2.1.2 delete_publisher

此方法将删除现有的 ***Publisher***。

如果 ***Publisher*** 还有需要管理的 ***DataWriter*** 对象，则无法将该 ***Publisher*** 删除。在这种情况下调用 ***delete_publisher*** 方法，将返回 PRECONDITION_NOT_MET。

必须在创建 ***Publisher*** 的同一 ***DomainParticipant*** 对象上调用 ***delete_publisher*** 方法。如果在另一个 ***DomainParticipant*** 上调用 ***delete_publisher***，该方法将不起作用，返回 PRECONDITION_NOT_MET。

除标准错误代码外，还可能返回错误代码：PRECONDITION_NOT_MET。

2.2.2.2.1.3 create_subscriber

此方法将创建具有所需 QoS 策略的 ***Subscriber***，并将指定的 ***SubscriberListener*** 附加到

该 ***Subscriber***。

如果指定的 QoS 策略不一致，方法调用将失败，且不会创建 ***Subscriber***。

特殊值 SUBSCRIBER_QOS_DEFAULT 表明应使用工厂中设置的默认 ***Subscriber*** QoS 创建 ***Subscriber***。使用这个特殊值等同于应用程序通过 get_default_subscriber_qos（2.2.2.2.1.23）方法获取默认 ***Subscriber*** QoS 并使用返回的 QoS 创建 ***Subscriber***。

创建的 ***Subscriber*** 隶属于作为其工厂的 ***DomainParticipant***。

如果方法调用失败，将返回“nil”值（由平台指定）。

2.2.2.2.1.4 delete_subscriber

此方法将删除现有的 ***Subscriber***。

如果 ***Subscriber*** 还有需要管理的 ***DataReader*** 对象，则无法将该 ***Subscriber*** 删除。在这种情况下调用 ***delete_subscriber*** 方法，将返回 PRECONDITION_NOT_MET。

必须在创建 ***Subscriber*** 的同一 ***DomainParticipant*** 对象上调用 ***delete_subscriber*** 方法。如果在另一个 ***DomainParticipant*** 上调用 ***delete_subscriber***，该方法将不起作用，返回 PRECONDITION_NOT_MET。

除标准错误代码外，还可能返回错误代码：PRECONDITION_NOT_MET。

2.2.2.2.1.5 create_topic

此方法创建具有所需 QoS 策略的 ***Topic***，并将指定的 ***TopicListener*** 附加到该 ***Topic***。

如果指定的 QoS 策略不一致，方法调用将失败，并且不会创建 ***Topic***。

特殊值 TOPIC_QOS_DEFAULT 表明应使用工厂中设置的默认 ***Topic*** QoS 创建 ***Topic***。使用这个特殊值等同于应用程序通过 get_default_topic_qos（2.2.2.2.1.25）方法获取默认 ***Topic*** QoS 并使用返回的 QoS 创建 ***Topic***。

创建的 ***Topic*** 隶属于作为其工厂的 ***DomainParticipant***。

Topic 需要绑定 ***type_name*** 参数描述的类型。在创建 ***Topic*** 之前，必须在服务中完成该类型的注册。这是在 ***TypeSupport*** 接口的派生类上使用 ***register_type*** 方法完成的，如 2.2.2.3.6 TypeSupport 接口中所述。

如果方法调用失败，将返回“nil”值（由平台指定）。

2.2.2.2.1.6 delete_topic

此方法将删除一个 ***Topic***。

如果还有存在的 ***DataReader***，***DataWriter***，***ContentFilteredTopic*** 或 ***MultiTopic*** 需要使用该 ***Topic***，则无法将该 ***Topic*** 删除。在这种情况下调用 ***delete_topic*** 方法，将返回 PRECONDITION_NOT_MET。

必须在创建 ***Topic*** 的同一 ***DomainParticipant*** 对象上调用 ***delete_topic*** 方法。如果在另一个 ***DomainParticipant*** 上调用 ***delete_topic***，该方法将不起作用，返回 PRECONDITION_NOT_MET。

除标准错误代码外，还可能返回错误代码：PRECONDITION_NOT_MET。

2.2.2.2.1.7 create_contentfilteredtopic

此方法将创建一个 ***ContentFilteredTopic***。如 2.2.2.3 主题定义模块中所述，***ContentFilteredTopic*** 可用于实现基于内容的订阅。

订阅的相关主题是通过 ***related_topic*** 参数指定的。***ContentFilteredTopic*** 仅与在该主题下

发布的数据样本有关系，并根据其内容进行过滤。通过评估样本中某些数据字段值的逻辑表达式来完成过滤。逻辑表达式来源于 ***filter_expression*** 和 ***expression_parameters*** 参数。

过滤表达式和参数的语法在附录 B 中描述。

如果调用失败，将返回“nil”值（由平台指定）。

2.2.2.2.1.8 delete_ contentfilteredtopic

此方法将删除一个 ***ContentFilteredTopic***。

如果还有存在的 ***DataReader***，***DataWriter***，***ContentFilteredTopic*** 或 ***MultiTopic*** 需要使用该 ***Topic***，则无法将该 ***Topic*** 删除。在这种情况下调用 ***delete_topic*** 方法，将返回 PRECONDITION_NOT_MET。

必须在创建 ***Topic*** 的同一 ***DomainParticipant*** 对象上调用 ***delete_topic*** 方法。如果在另一个 ***DomainParticipant*** 上调用 ***delete_topic***，该方法将不起作用，返回 PRECONDITION_NOT_MET。

除标准错误代码外，还可能返回错误代码：PRECONDITION_NOT_MET。

2.2.2.2.1.9 create_multitopic

此方法创建 ***MultiTopic***。如 2.2.2.3 主题定义模块中所述，***MultiTopic*** 可用于订阅多个主题，并将接收的数据组合/过滤为作为结果的类型。特别地，***MultiTopic*** 提供了基于内容的订阅机制。

作为结果的类型由 type_name 参数指定。在创建 ***MultiTopic*** 之前，必须在服务中完成该类型的注册。这是在 ***TypeSupport*** 接口的派生类上使用 ***register_type*** 方法完成的，如 2.2.2.3.6 中所述。

subscription_expression 和 ***expression_parameters*** 参数指定主题列表，并组合过滤与重新排列每个主题信息的逻辑。

表达式和参数的语法在附录 B 中描述。

如果调用失败，将返回“nil”值（由平台指定）。

2.2.2.2.1.10 delete_multitopic

此方法将删除一个 ***MultiTopic***。

如果还有存在的 ***DataReader*** 对象需要使用该 ***MultiTopic***，则无法将该 ***MultiTopic*** 删除。在这种情况下调用 ***delete_multitopic*** 方法，将返回 PRECONDITION_NOT_MET。

必须在创建 ***MultiTopic*** 的同一 ***DomainParticipant*** 对象上调用 ***delete_multitopic*** 方法。如果在其他 ***DomainParticipant*** 上调用 ***delete_multitopic***，该方法将不起作用，返回 PRECONDITION_NOT_MET。

除标准错误代码外，还可能返回错误代码：PRECONDITION_NOT_MET。

2.2.2.2.1.11 find_topic

find_topic 方法使用 ***Topic*** 的名称和超时时间作为参数，可根据参数传入的名称访问现有（或准备存在）的主题。

如果已经存在同名 ***Topic***，本方法便可以访问该主题，否则等待（阻塞调用者），直到其他机制创建该主题（或设定的时间到期超时）。这个其他机制可以是另一个线程，配置工具或一些其他中间件服务。需要注意的是，***Topic*** 是一个本地对象，充当指定主题全局概念的“代理”。中间件可以选择传播主题并在本地获取远程创建主题的信息。

通过 ***find_topic*** 获取的主题必须通过 ***delete_topic*** 删除，确保释放本地资源。如果通过 ***find_topic*** 或 ***create_topic*** 多次获取主题，那必须使用 ***delete_topic*** 删除相同次数的主题。

无论中间件是否选择传播主题，***delete_topic*** 操作都只删除本地代理。

如果方法超时，则返回“nil”值（由平台指定）。

2.2.2.2.1.12 lookup_topicdescription

lookup_topicdescription 方法将 ***TopicDescription*** 的 ***name*** 作为参数，根据参数传入的名称访问现有的本地创建的 ***TopicDescription***。

如果已经存在同名的 ***TopicDescription***，本方法便可以访问该对象，否则返回“nil”值。本方法永远不会阻塞。

lookup_topicdescription 方法可用于定位任何本地创建的 ***Topic***，***ContentFilteredTopic*** 和 ***MultiTopic*** 对象。

与 ***find_topic*** 方法不同，***lookup_topicdescription*** 方法仅在本地创建的主题中搜索。因此，它永远不应该创建新的 ***TopicDescription***。***lookup_topicdescription*** 返回的 ***TopicDescription*** 不需要额外删除。在一种特殊情况下可以删除 ***lookup_topicdescription*** 方法返回的 ***TopicDescription***，即该 ***TopicDescription*** 没有读者或写者，会被删除，后续对其查找将失败。

如果本方法无法找到 ***TopicDescription***，则返回“nil”值（由平台指定）。

2.2.2.2.1.13 get_builtin_subscriber

此方法允许访问内置 ***Subscriber***。每个 ***DomainParticipant*** 都包含几个内置的 ***Topic*** 对象，以及相应的 ***DataReader*** 对象来访问这些 ***Topic*** 对象。所有这些 ***DataReader*** 对象都属于单个内置 ***Subscriber***。

内置主题用于传递有关其他 ***DomainParticipant***，***Topic***，***DataReader*** 和 ***DataWriter*** 对象的信息。这些内置对象在 2.2.5 内置主题中描述。

2.2.2.2.1.14 ignore_participant

此方法允许应用程序指导中间件服务在本地忽略远程域参与者。从那时起，服务将在本地运行，就好像远程参与者不存在一样。这意味着它将忽略源自该域参与者的任何主题，发布或订阅。

该方法可以与通过“DCPSParticipant”内置主题实现远程参与者的发现结合使用，以提供不同功能（例如访问控制）。应用程序数据可以通过 USER_DATA QoS 策略与 ***DomainParticipant*** 相关联。此应用程序数据作为内置主题中的字段传播，辅助应用程序实现其自己的访问控制策略。有关内置主题的更多详细信息，请参见 2.2.5 内置主题。

要忽略的域参与者由 ***handle*** 参数标识。此句柄是 ***SampleInfo*** 中出现的句柄，该 ***SampleInfo*** 通过主题为“DCPSParticipant”的内置 ***DataReader*** 收取。使用与任何 DataReader 相同的读取/获取操作读取内置 DataReader。这些数据访问方法将在 2.2.2.5 订阅模块中描述。

ignore_participant 方法不需要是可逆的。服务无法撤销。

除标准错误代码外，还可能返回错误代码：OUT_OF_RESOURCES。

2.2.2.2.1.15 ignore_topic

此方法允许应用程序指导中间件服务在本地忽略主题，意味着它将在本地忽略某一主题的任何发布或订阅操作。

当应用程序知道它永远不会在某些主题下发布或订阅数据时，此方法可用于节约本地资

源。

要忽略的主题由 ***handle*** 参数标识。此句柄是从主题为“DCPSTopic”的内置 ***DataReader*** 接收到的数据样本 SampleInfo 中获得的句柄。

ignore_topic 方法不需要是可逆的。服务无法撤销。

除标准错误代码外，还可能返回错误代码：OUT_OF_RESOURCES。

2.2.2.2.1.16 ignore_publication

此方法允许应用程序指导中间件服务在本地忽略远程发布; 发布由主题名称、用户数据与 ***Publisher*** 设置的分块的关联定义（请参阅 2.2.5 中的“DCPSPublication”内置主题）。在此调用之后，服务将忽略与该发布有关的任何数据。

要忽略的 ***DataWriter*** 由 ***handle*** 参数标识。此句柄是从主题为“DCPSPublication”的内置 ***DataReader*** 接收到的数据样本 SampleInfo 中获得的句柄。

ignore_publication 操作不需要是可逆的。服务无法撤销。

除标准错误代码外，还可能返回错误代码：OUT_OF_RESOURCES。

2.2.2.2.1.17 ignore_subscription

此方法允许应用程序指导中间件服务在本地忽略远程订阅; 订阅由主题名称、用户数据与 ***Subscriber*** 设置的分块的关联定义（请参阅 2.2.5 中的“DCPSSubscription”内置主题）。在此调用之后，服务将忽略与该订阅有关的任何数据。

要忽略的 ***DataReader*** 由 ***handle*** 参数标识。此句柄是在从主题为“DCPSSubscription”的内置 ***DataReader*** 接收到的数据样本 SampleInfo 中获得的句柄。

ignore_subscription 操作不需要是可逆的。服务无法撤销。

除标准错误代码外，还可能返回错误代码：OUT_OF_RESOURCES。

2.2.2.2.1.18 delete_contained_entities

此方法将删除通过 ***DomainParticipant*** 的“create”方法创建的所有实体，即删除所有包含的 ***Publisher***，***Subscriber***，***Topic***，***ContentFilteredTopic*** 和 ***MultiTopic*** 实体对象。

在删除每个包含的实体对象之前，此方法将递归地调用每个包含的实体对象上的相应 ***delete_contained_entities*** 方法（如果适用），递归地使用该模式。通过这种方式，***DomainParticipant*** 上的 ***delete_contained_entities*** 方法将最终删除 ***DomainParticipant*** 中递归包含的所有实体对象，例如 ***DataWriter***，***DataReader*** 对象，以及隶属于所包含的 ***DataReader*** 对象的 ***QueryCondition*** 和 ***ReadCondition*** 对象。

如果任何包含的实体处于无法删除的状态，方法将返回 PRECONDITION_NOT_MET。

一旦 ***delete_contained_entities*** 成功返回，应用程序可能会删除 ***DomainParticipant***，因为应用程序知道它没有包含的实体。

2.2.2.2.1.19 assert_liveliness

此方法手动断言 ***DomainParticipant*** 的活跃性，与 LIVELINESS QoS 策略（参见 2.2.3，支持的 QoS）结合使用，以向服务提示实体处于活动状态。

仅当 ***DomainParticipant*** 包含将 LIVELINESS QoS 设置为 MANUAL_BY_PARTICIPANT 的 ***DataWriter*** 实体并且 ***DomainParticipant*** 仅影响这些 ***DataWriter*** 实体的活跃性时，才需要调用此方法。否则，此方法没有任何效果。

注意：通过 ***DataWriter*** 对象上的 ***write*** 方法写入数据会断言 ***DataWriter*** 对象本身及其所

属 ***DomainParticipant*** 对象的活跃性。因此，只有在应用程序不定期写入数据时才需要使用 ***assert_liveliness***。

完整的细节在 2.2.3.11 LIVELINESS 中提供。

2.2.2.2.1.20 set_default_publisher_qos

此方法设置发布者 QoS 策略的默认值，在调用 ***create_publisher*** 方法时作为默认 QoS 策略用于新创建的发布者（***Publisher***）实体。

此方法将检查生成的策略是否是自相容的；如果不是，则方法无效并返回 INCONSISTENT_POLICY。

可以将特殊值 PUBLISHER_QOS_DEFAULT 当做本方法的输入参数，以指示应将默认 QoS 重置为工厂使用的初始值，即从未调用 ***set_default_publisher_qos*** 方法时使用的值。

2.2.2.2.1.21 get_default_publisher_qos

此方法获取发布者（***Publisher***）QoS 的默认值，即在 ***create_publisher*** 方法使用默认 QoS 策略的情况下，用于新创建的发布者（***Publisher***）实体的 QoS 策略。

通过此方法获取到的值将与上次成功调用 ***set_default_publisher_qos*** 时指定的值集保持一致；如果 ***set_default_publisher_qos*** 从未被调用，默认值在 2.2.3 支持的 QoS 中的 QoS 表中列出。

2.2.2.2.1.22 set_default_subscriber_qos

此方法设置订阅者（***Subscriber***）QoS 策略的默认值，在调用 ***create_subscriber*** 方法时作为默认 QoS 策略用于新创建的订阅者（***Subscriber***）实体。

此方法将检查生成的策略是否是自相容的；如果不是，则方法无效并返回 INCONSISTENT_POLICY。

可以将特殊值 SUBSCRIBER_QOS_DEFAULT 当做本方法的输入参数，以指示应将默认 QoS 重置为工厂使用的初始值，即从未调用 ***set_default_subscriber_qos*** 方法时使用的值。

2.2.2.2.1.23 get_default_subscriber_qos

此方法获取订阅者（***Subscriber***）QoS 的默认值，即在 ***create_subscriber*** 方法使用默认 QoS 策略的情况下，用于新创建的订阅者（***Subscriber***）实体的 QoS 策略。

通过此方法获取到的值将与上次成功调用 ***set_default_subscriber_qos*** 时指定的值集保持一致；如果 ***set_default_subscriber_qos*** 从未被调用，默认值在 2.2.3 支持的 QoS 中的 QoS 表中列出。

2.2.2.2.1.24 set_default_topic_qos

此方法设置主题（***Topic***）QoS 策略的默认值，在调用 ***create_topic*** 方法时作为默认 QoS 策略用于新创建的主题（***Topic***）实体。

此方法将检查生成的策略是否是自相容的；如果不是，则方法无效并返回 INCONSISTENT_POLICY。

可以将特殊值 TOPIC_QOS_DEFAULT 当做本方法的输入参数，以指示应将默认 QoS 重置为工厂使用的初始值，即从未调用 ***set_default_topic_qos*** 方法时使用的值。

2.2.2.2.1.25 get_default_topic_qos

此方法获取主题（***Topic***）QoS 的默认值，即在 ***create_topic*** 方法使用默认 QoS 策略的情况下，用于新创建的主题（***Topic***）实体的 QoS 策略。

通过此方法获取到的值将与上次成功调用 ***set_default_topic_qos*** 时指定的值集保持一致；如果 ***set_default_topic_qos*** 从未被调用，默认值在 2.2.3 支持的 QoS 中的 QoS 表中列出。

2.2.2.2.1.26 get_domain_id

此方法获取用于创建 ***DomainParticipant*** 对象的 *domain_id*。*domain_i* 标识 ***DomainParticipant*** 所属的 DDS 域。如 2.2.2.2.1 的介绍中所述，每个 DDS 域代表与其他域隔离的单独数据“通信平面”。

2.2.2.2.1.27 get_discovered_participants

此方法获取已在域中发现的 DomainParticipants 列表，并且应通过 DomainParticipant ignore_participant 方法“忽略”应用程序未加入的域。

如果服务的底层不在本地维护连接信息，则方法可能会失败。在这种情况下，方法将返回 UNSUPPORTED。

2.2.2.2.1.28 get_discovered_participant_data

此方法获取在网络上发现的 ***DomainParticipant*** 的有关信息。参与者必须与调用此方法的参与者位于同一域中，并且不能被 ***DomainParticipant ignore_participant*** 方法“忽略”过。*participant_handle* 必须对应于上面描述的这种 ***DomainParticipant***。否则，此方法调用将失败并返回 PRECONDITION_NOT_MET。

使用 ***get_discovered_participants*** 方法获取当前发现的 DomainParticipants。

如果服务的底层不包含填写 *participant_data* 所需的信息，则方法调用也可能失败。这种情况下，将返回 UNSUPPORTED。

2.2.2.2.1.29 get_discovered_topics

此方法获取已在域中发现的主题列表，并且应通过 DomainParticipant ignore_topic 方法“忽略”应用程序未创建的主题列表。

2.2.2.2.1.30 get_discovered_topic_data

此方法获取在网络上发现的主题的有关信息。该主题必须由与调用此方法的参与者位于同一域中的参与者创建，并且不能被 ***DomainParticipant ignore_topic*** 方法“忽略”过。*topic_handle* 必须对应于上面描述的这类主题。否则，此方法调用将失败并返回 PRECONDITION_NOT_MET。

使用 ***get_discovered_topics*** 方法获取当前发现的主题。

如果服务的底层不包含填写 *topic_data* 所需的信息，则方法调用也可能失败。这种情况下，将返回 UNSUPPORTED。

如果服务的底层不在本地维护连接信息，则方法可能会失败。在这种情况下，方法将返回 UNSUPPORTED。

2.2.2.2.1.31 contains_entity

此方法检查给定的 a_handle 是否为该 ***DomainParticipant*** 创建，此从属关系在该 ***DomainParticipant*** 包含的实体中递归适用。也就是说，它既适用于使用 ***DomainParticipant*** 直接创建的实体（***TopicDescription***，***Publisher*** 或 ***Subscriber***），也适用于使用包含的 ***Publisher*** 或 ***Subscriber*** 作为工厂创建的实体等。

实体（***Entity***）的实例句柄可以从内置主题数据，各种状态或实体方法 ***get_instance_handle*** 获得。

2.2.2.2.1.32 get_current_time

此方法返回当前时间值，辅助服务对需要发布的数据加时间戳，并为其接收的数据设置接收时间戳。

2.2.2.2.2 DomainParticipantFactory 类

此类的唯一目的是创建和销毁 ***DomainParticipant*** 对象。

DomainParticipantFactory 本身没有工厂。它是一个预先存在的单例对象，可以通过 ***DomainParticipantFactory*** 上的 ***get_instance*** 类方法进行访问。

DomainParticipantFactory		
no attributes		
operations		
create_participant		DomainParticipant
	domain_id	DomainId_t
	qos_list	QosPolicy []
	a_listener	DomainParticipantListener
	mask	StatusKind []
delete_participant		ReturnCode_t
	a_participant	DomainParticipant
(static) get_instance		DomainParticipantFactory
lookup_participant		DomainParticipant
	domain_id	DomainId_t
set_default_participant_qos		ReturnCode_t
	qos_list	QosPolicy []
get_default_participant_qos		ReturnCode_t
	out: qos_list	QosPolicy []
get_qos		ReturnCode_t
	out: qos_list	QosPolicy []
set_qos		ReturnCode_t
	qos_list	QosPolicy []

以下子小节提供了有关方法的详细信息。

2.2.2.2.2.1 create_participant

此方法将创建一个新的 ***DomainParticipant*** 对象。该 ***DomainParticipant*** 对象的创建意味着调用此方法的应用程序打算加入 *domain_id* 参数标识的域。

如果指定的 QoS 策略不一致，则方法调用将失败，并且不会创建 ***DomainParticipant*** 对象。

特殊值 PARTICIPANT_QOS_DEFAULT 表明应使用工厂中设置的默认 ***DomainParticipant*** QoS 创建 ***DomainParticipant***。使用此值等同于应用程序通过 ***get_default_participant_qos***（2.2.2.2.2.6）方法获取默认 ***DomainParticipant*** QoS 并使用返回的 QoS 创建 ***DomainParticipant***。

如果调用失败，将返回“nil”值（由平台指定）。

2.2.2.2.2.2 delete_participant

此方法将删除现有的 ***DomainParticipant***。只有在属于该参与者的所有域实体都被删除的情况下，才能调用此方法。否则返回错误 PRECONDITION_NOT_MET。

除标准错误代码外，还可能返回错误代码：PRECONDITION_NOT_MET。

2.2.2.2.2.3 get_instance

此方法返回 ***DomainParticipantFactory*** 单例。该方法是幂等的，也就是说它可以被多次调用而没有副作用，返回相同的 ***DomainParticipantFactory*** 实例。

get_instance 方法是使用本机语言的语法实现的静态方法，因此不能在 IDL PSM 中表示。

预定义值 ***TheParticipantFactory*** 也可以用作 ***get_instance*** 方法返回的单例工厂的别名。

2.2.2.2.2.4 lookup_participant

此方法获取之前创建的属于特定 *domain_id* 的 ***DomainParticipant*** 对象。如果不存在此类 ***DomainParticipant*** 对象，将返回“nil”值。

如果存在属于该 *domain_id* 的多个 ***DomainParticipant*** 对象，则返回其中一个对象，并不具体指定为哪一个。

2.2.2.2.2.5 set_default_participant_qos

此方法设置 ***DomainParticipant*** QoS 策略的默认值，在调用 ***create_participant*** 方法时作为默认 QoS 策略用于新创建的 ***DomainParticipant*** 实体。

此方法将检查生成的策略是否是自相容的；如果不是，则方法无效并返回 INCONSISTENT_POLICY。

2.2.2.2.2.6 get_default_participant_qos

此方法获取 ***DomainParticipant*** QoS 的默认值，即在 ***create_participant*** 方法使用默认 QoS 策略的情况下，用于新创建的 ***DomainParticipant*** 的 QoS 策略。

通过此方法获取到的值将与上次成功调用 ***get_default_participant_qos*** 时指定的值集保持一致；如果 ***get_default_participant_qos*** 从未被调用，默认值在 2.2.3 支持的 QoS 中的 QoS 表中列出。

2.2.2.2.2.7 set_qos

此方法设置 ***DomainParticipantFactory*** QoS 策略的值。这些策略控制实体的工厂对象的行为。

请注意，尽管具有 QoS，但 ***DomainParticipantFactory*** 不是实体。

此方法将检查生成的策略是否是自相容的；如果不是，则方法无效并返回 INCONSISTENT_POLICY。

2.2.2.2.2.8 get_qos

此方法返回 ***DomainParticipantFactory*** QoS 策略的值。

2.2.2.2.3 DomainParticipantListener 接口

这是一个可以由应用程序提供的类实现的接口，该接口需要向 ***DomainParticipant*** 注册，以便 DCPS 服务可以向应用程序通知相关状态的更改。

DomainParticipantListener 接口扩展了所有其他监听器（***Listener***）接口，除了监听器定义的普通方法之外没有其他方法。

DomainParticipantListener		
no attributes		
operations		
on_inconsistent_topic		void
	the_topic	Topic
	status	InconsistentTopicStatus
on_liveliness_lost		void
	the_writer	DataWriter
	status	LivelinessLostStatus
on_offered_deadline_missed	the_writer	DataWriter
	status	OfferedDeadlineMissedStatus
on_offered_incompatible_qos	the_writer	DataWriter
	status	OfferedIncompatibleQosStatus
on_data_on_readers		void
	the_subscriber	Subscriber
on_sample_lost		void
	the_reader	DataReader
	status	SampleLostStatus
on_data_available		void
	the_reader	DataReader
on_sample_rejected		void
	the_reader	DataReader
	status	SampleRejectedStatus
on_liveliness_changed		void
	the_reader	DataReader
	status	LivelinessChangedStatus
on_requested_deadline_missed		void

	the_reader	DataReader
	status	RequestedDeadlineMissedStatus
on_requested_incompatible_qos		void
	the_reader	DataReader
	status	RequestedIncompatibleQosStatus
on_publication_matched		void
	the_writer	DataWriter
	status	PublicationMatchedStatus
on_subscription_matched		void
	the_reader	DataReader
	status	SubscriptionMatchedStatus

DomainParticipantListener 对象的目的是作为最后的监听器，通知附加到 ***DomainEntity*** 对象的特定监听器（***Listener***）未捕获的状态的变化。当相关状态发生变化时，DCPS 服务将首先尝试通知附加到相关 ***DomainEntity*** 的监听器（如果已安装）。否则，DCPS 服务将通知 ***DomainParticipant*** 附带的监听器。2.2.4 Listeners，Conditions 和 Wait-sets 中描述了监听器之间的关系。

2.2.2.3 主题定义（Topic-Definition）模块

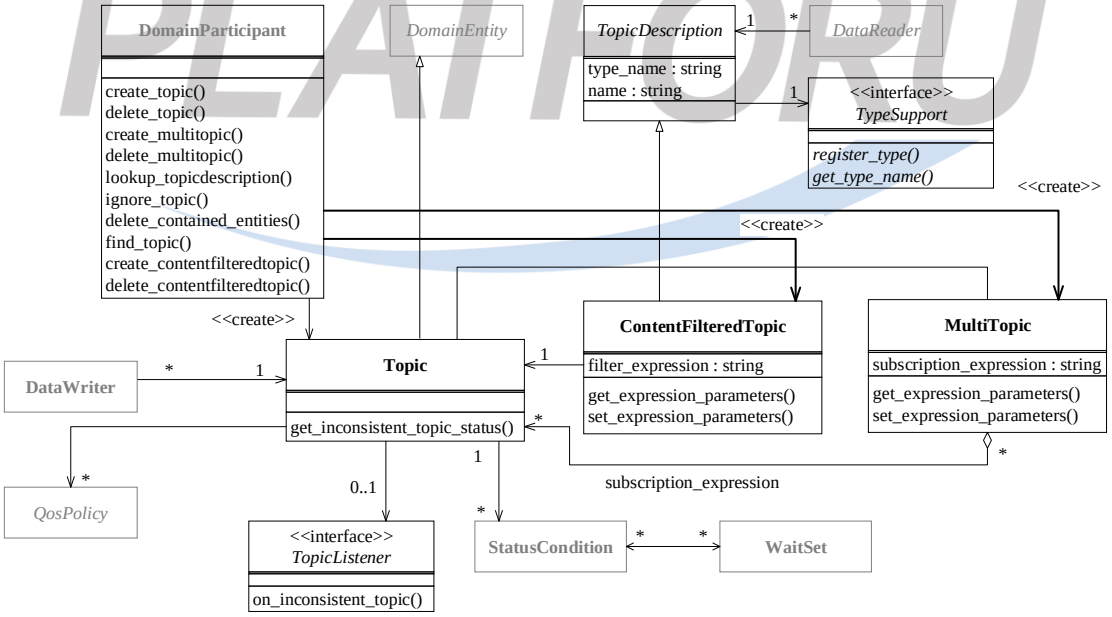

图 2-7 DCPS 主题定义模块的类模型

主题定义模块由以下类组成：

- TopicDescription
- Topic
- ContentFilteredTopic
- MultiTopic
- TopicListener

•TypeSupport

2.2.2.3.1 TopicDescription 类

此类是一个抽象类。它是 ***Topic***，***ContentFilteredTopic*** 和 ***MultiTopic*** 的基类。

TopicDescription		
attributes		
readonly: name	string	
readonly: type_name	string	
operations		
get_participant		DomainParticipant
get_type_name		string
get_name		string

TopicDescription 类代表了发布和订阅都绑定到单个数据类型的事实。其属性 type_name 为发布或订阅定义唯一的结果类型，因此创建了与 ***TypeSupport*** 类的隐式关联。***TopicDescription*** 还有一个 name 属性，允许在本地检索获取它。

2.2.2.3.1.1 get_participant

此方法返回 ***TopicDescription*** 所属的 ***DomainParticipant***。

2.2.2.3.1.2 type_name

属性值 *type_name* 用来创建 ***TopicDescription*** 对象。

2.2.2.3.1.3 name

属性值 *name* 用来创建 ***TopicDescription*** 对象。

2.2.2.3.2 主题（Topic）类

主题（Topic）类是要发布和订阅数据的最基本描述方式。

Topic		
no attributes		
operations		
(inherited) get_qos		ReturnCode_t
	out: qos_list	QosPolicy []
(inherited) set_qos		ReturnCode_t
	qos_list	QosPolicy []
(inherited) get_listener		Listener
(inherited) set_listener		ReturnCode_t
	a_listener	Listener
	mask	StatusKind []

get_inconsistent_topic_status		ReturnCode_t
	out: status	InconsistentTopicStatus

主题(Topic) 由其 ***名称(name)*** 标识，在整个域中必须是唯一的。此外（通过扩展 ***TopicDescription***）它完全指定了在发布或订阅主题***(Topic)***时可以传达的数据类型。

主题（Topic） 是唯一可用于发布的 ***TopicDescription***，因此与 ***DataWriter*** 相关联。

除基类方法 ***set_qos***，***get_qos***，***set_listener***，***get_listener***，***enable*** 和 ***get_status_condition*** 之外的所有方法都可能返回 NOT_ENABLED。

以下子条目描述了其部分方法。

2.2.2.3.2.1 get_inconsistent_topic_status

此方法允许应用程序检索***主题（Topic）*** 的 INCONSISTENT_TOPIC 状态。

每个 ***DomainEntity*** 都有一组相关的通信状态。状态改变会导致调用相应的监听器，也可以通过关联的 ***StatusCondition*** 进行监视。

2.2.4.1 通信状态中提供了完整的通信状态列表、它们的值以及它们适用的 ***DomainEntities***。

2.2.2.3.3 ContentFilteredTopic 类

ContentFilteredTopic 类是 ***TopicDescription*** 类的一种特殊化，允许基于内容的订阅。

ContentFilteredTopic		
attributes		
readonly: filter_expression	string	
operations		
get_related_topic		Topic
get_expression_parameters		ReturnCode_t
	out: expression_parameters	string []
set_expression_parameters		ReturnCode_t
	expression_parameters	string []

ContentFilteredTopic 描述了一种更复杂的订阅场景，这种情况下订阅者不一定要收到此***主题（Tpic）*** 下发布的每个实例的内容。相反，订阅者只想接收内容满足特定条件的值。因此，***ContentFilteredTopic*** 类可用于发起基于内容的订阅。

通过使用参数为 ***expression_parameters*** 的 ***filter_expression*** 属性完成基于内容的选择。

- ***filter_expression*** 属性是一个字符串，用于指定选择感兴趣的数据样本的条件。它类似于 SQL 语句中的 WHERE 部分。
- ***expression_parameters*** 属性是一系列字符串，它们为 ***filter_expression*** 中的“parameters”（即“%n”标记）赋值。提供的参数的数量必须与 ***filter_expression*** 中的请求值一致（即%n 符号的数量）。

附件 B 描述了 ***filter_expression*** 和 ***expression_parameters*** 的语法。

2.2.2.3.3.1 get_related_topic

此方法返回与 ***ContentFilteredTopic*** 关联的***主题（Topic）***，即在创建 ***ContentFilteredTopic*** 时指定的主题。

2.2.2.3.3.2 filter_expression

与 ***ContentFilteredTopic*** 关联的 ***filter_expression***，即创建 ***ContentFilteredTopic*** 时指定的表达式。

2.2.2.3.3.3 get_expression_parameters

此方法返回与 ***ContentFilteredTopic*** 关联的 ***expression_parameters***，即上次成功调用 ***set_expression_parameters*** 时指定的参数，或者从未调用 ***set_expression_parameters***，在创建 ***ContentFilteredTopic*** 时指定的参数。

2.2.2.3.3.4 set_expression_parameters

此方法设置与 ***ContentFilteredTopic*** 关联的 ***expression_parameters***。

2.2.2.3.4 MultiTopic 类[可选]

MultiTopic 类是 ***TopicDescription*** 类的一种特殊化，允许订阅方组合/过滤/重新排列来自多个主题的数据。

MultiTopic		
attributes		
readonly: subscription_expression	string	
operations		
get_expression_parameters		ReturnCode_t
	out: expression_parameters	string []
set_expression_parameters		ReturnCode_t
	expression_parameters	string []

MultiTopic 允许一种更复杂的订阅场景，这种场景下订阅方可以选择同时订阅多个主题的数据，并将从多个主题接收的数据组合成单个结果类型（由继承的 ***type_name*** 指定）。然后根据参数为 ***expression_parameters*** 的 ***subscription_expression*** 属性，对这些数据进行过滤（选择）甚至重新排列（聚合/投影）。

- ***subscription_expression*** 属性是一个字符串，用于标识从关联主题中选择和重新排列数据。它类似于一个 SQL 语句，其中 SELECT 部分提供要保留的字段，FROM 部分提供字段来源的主题名称，WHERE 部分提供内容过滤器。组合的主题可能具有不同的类型，但它们受到限制，因为用于 NATURAL JOIN 操作的字段类型必须相同。
- ***expression_parameters*** 属性是一个字符串序列，它为 ***subscription_expression*** 中的“parameters”（即“%n”标记）赋值。提供的参数数量必须与 ***subscription_expression*** 中的请求值一致（即%n 符号的数量）。
- 只要与 ***MultiTopic*** 相关的任何主题的数据发生修改，就会通过常规的监听器或条件机制（参见 2.2.4，监听器，条件和等待集）提醒与 ***MultiTopic*** 关联的 ***DataReader*** 实

体。

•与 ***MultiTopic*** 关联的 DataReader 实体会访问以下实例，这些实例在 DataReader 端通过多个 ***DataWriter*** 实体写入的实例“构造”。只要所有构成主题实例都存在，***MultiTopic*** 访问实例就会存在。***view_state*** 和 ***instance_state*** 是根据构成实例的相应状态计算的：

•如果至少有一个构成实例具有 ***view_state*** = NEW，则 ***MultiTopic*** 实例的 ***view_state***（参见 2.2.2.5.1）为 NEW，否则为 NOT_NEW。

•如果所有构成主题实例的 ***instance_state*** 为 ALIVE，则 ***MultiTopic*** 实例的 ***instance_state***（参见 2.2.2.5.1）为“ALIVE”。如果至少有一个构成主题实例是 NOT_ALIVE_DISPOSED，则为“NOT_ALIVE_DISPOSED”，否则为 NOT_ALIVE_NO_WRITERS。

附件 B 描述了 ***subscription_expression*** 和 ***expression_parameters*** 的语法。

2.2.2.3.4.1 subscription_expression

与 ***MultiTopic*** 相关联的 ***subscription_expression***jie，即创建 ***MultiTopic*** 时指定的表达式。

2.2.2.3.4.2 get_expression_parameters

此方法返回与 ***MultiTopic*** 关联的 ***expression_parameters***，即上次成功调用 ***set_expression_parameters*** 时指定的参数，或者从未调用 ***set_expression_parameters***，创建 ***MultiTopic*** 时指定的参数。

2.2.2.3.4.3 set_expression_parameters

此方法改变与 ***MultiTopic*** 关联的 ***expression_parameters***。

2.2.2.3.5 TopicListener 接口

由于 ***Topic*** 是一种实体（***Entity***），因此它具有关联监听器的能力。在这种情况下，关联的监听器应该具有具体类型 ***TopicListener***。

TopicListener		
no attributes		
operations		
on_inconsistent_topic		void
	the_topic	Topic
	status	InconsistentTopicStatus

2.2.2.3.6 TypeSupport 接口

TypeSupport 接口是一个抽象接口，必须根据应用程序使用的具体类型进行定制。

此接口要求 DDS 服务的实现方提供自动的方法，该方法通过类型的描述生成此类型特定的类（例如在 OMG IDL 映射中使用 IDL）。必须先在此类型特定的类上使用 ***register_type*** 方法注册 ***TypeSupport***，然后才能使用它创建 ***Topic*** 对象。

TypeSupport		
no attributes		
operations		
register_type		ReturnCode_t
	participant	DomainParticipant
	type_name	string
get_type_name		string

2.2.2.3.6.1 register_type

此方法允许应用程序告知 DDS 服务其存在的数据类型。该方法的实现包含了需要传递给中间件的所有知识，使得中间件能够管理该数据类型的数据内容。实现还特别地包含了允许 DDS 服务区分相同类型的不同实例的***关键字（Key）***定义。

在同一个 ***DomainParticipant*** 中使用相同的 ***type_name*** 注册两个不同的 ***TypeSupport*** 是一种“前提条件错误”。如果应用程序尝试此操作，此方法将失败并返回 PRECONDITION_NOT_MET。但是，允许在同一个 ***DomainParticipant*** 中使用相同或不同的 ***type_name*** 值多次注册相同的 ***TypeSupport***。如果在一个 ***DomainParticipant*** 中使用相同的 ***type_name*** 在同一个 ***TypeSupport*** 上多次调用 ***register_type***，则忽略第二个（和后续）***register_type***，方法返回正常值 OK。

应用程序可以将 nil 作为 *type_name* 的值。在这种情况下，将使用由 ***TypeSupport*** 定义的默认类型名称（即 ***get_type_name*** 方法的返回值）。

除标准错误代码外，还可能返回错误代码：PRECONDITION_NOT_MET 和 OUT_OF_RESOURCES。

2.2.2.3.6.2 get_type_name

此方法返回 TypeSupport 代表的数据类型的默认名称。

2.2.2.3.7 应用程序类的派生类

对于应用程序定义的数据类，需要定制的类来促进应用程序与 DDS 服务之间的类型安全交互。

DDS 服务的每个实现方都需要提供自动生成这些特定于类型的类的方法。***TypeSupport*** 是这些自动生成的类必须实现的接口之一。假设应用程序数据类型为“Foo”，为该类型创建的完整自动类集如图 2.8 所示。

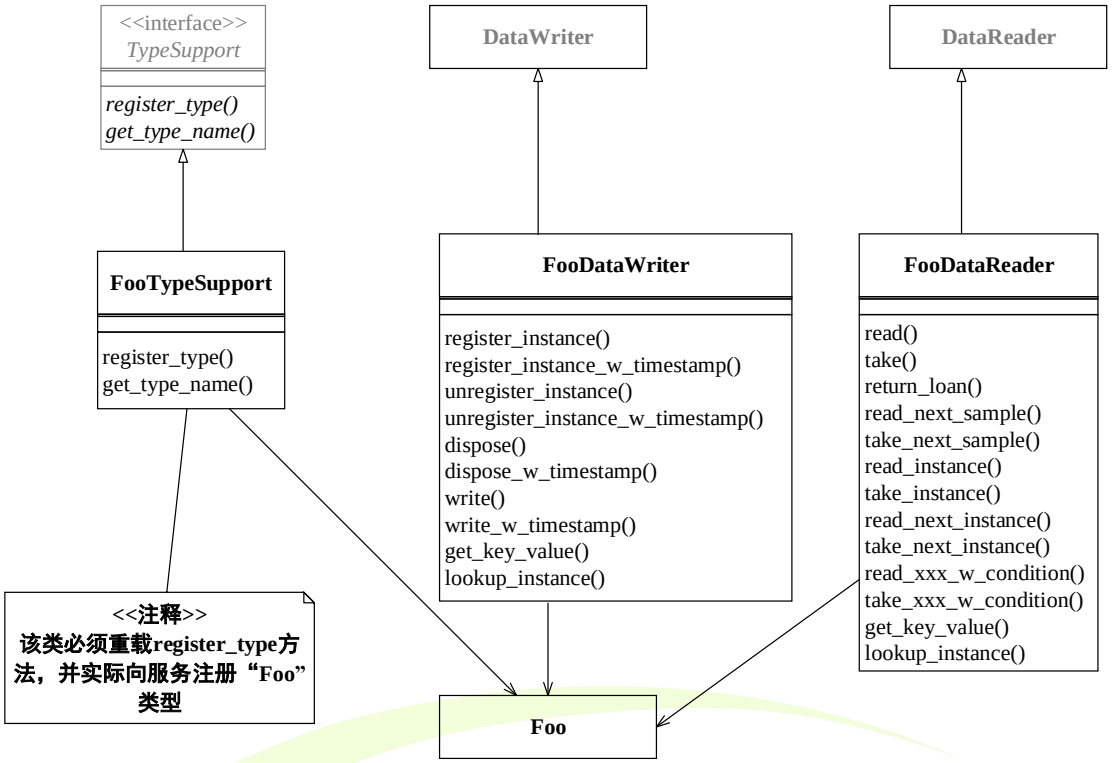

图 2-8　为名为 Foo 的应用程序数据类型自动创建的类

2.2.2.4 发布模块

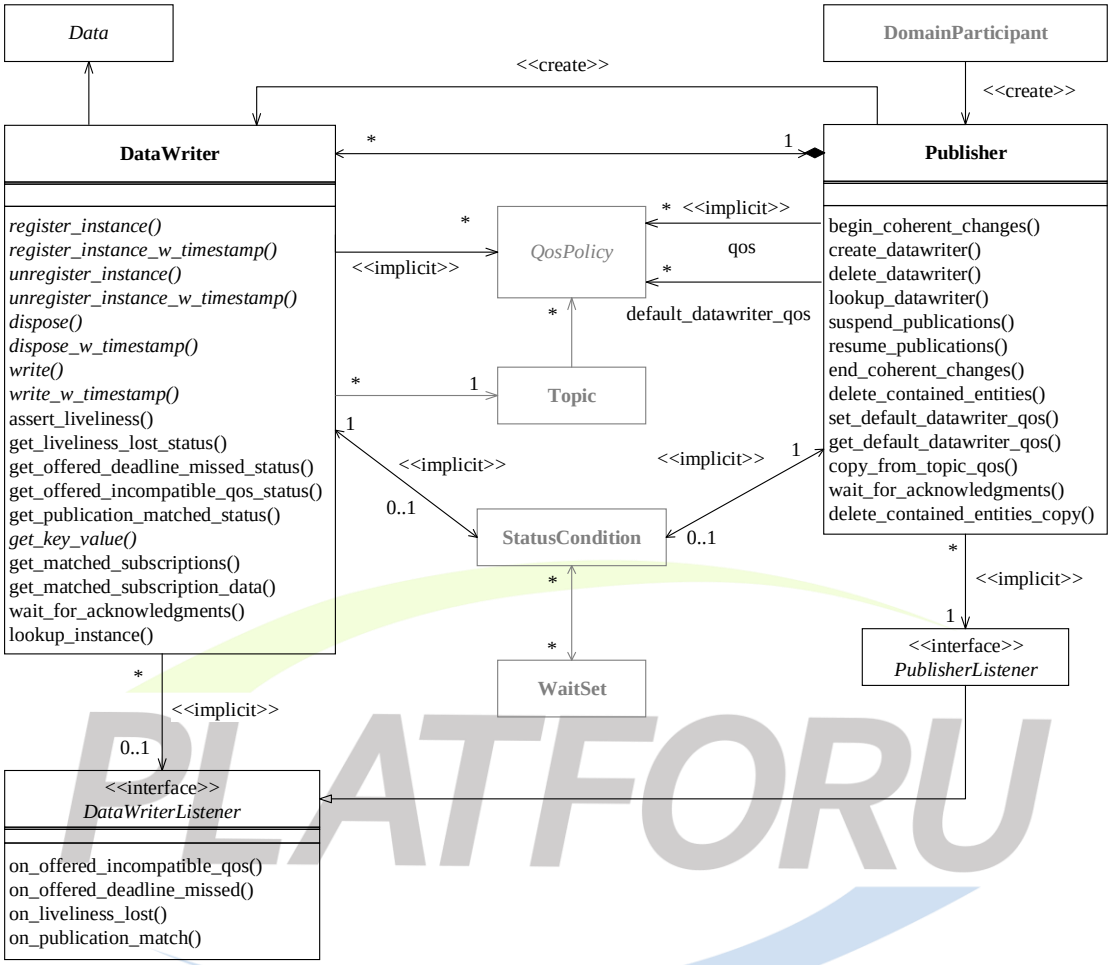

图 2-9 DCPS 发布模块的类模型

DCPS 发布模块由以下类组成：

- Publisher
- DataWriter
- PublisherListener
- DataWriterListener

2.2.2.4.1 Publisher 类

发布者（Publisher）类是负责实际数据发布的对象。

Publisher		
no attributes		
operations		
(inherited) get_qos		ReturnCode_t
	out: qos_list	QosPolicy []
(inherited) set_qos		ReturnCode_t

	qos_list	QosPolicy []
(inherited) get_listener		Listener
(inherited) set_listener		ReturnCode_t
	a_listener	Listener
	mask	StatusKind []
create_datawriter		DataWriter
	a_topic	Topic
	qos	QosPolicy []
	a_listener	DataWriterListener
	mask	StatusKind []
delete_datawriter		ReturnCode_t
	a_datawriter	DataWriter
lookup_datawriter		DataWriter
	topic_name	string
suspend_publications		ReturnCode_t
resume_publications		ReturnCode_t
begin_coherent_changes		ReturnCode_t
end_coherent_changes		ReturnCode_t
wait_for_acknowledgments		ReturnCode_t
	max_wait	Duration_t
get_participant		DomainParticipant
delete_contained_entities		ReturnCode_t
set_default_datawriter_qos		ReturnCode_t
	qos_list	QosPolicy []
get_default_datawriter_qos		ReturnCode_t
	out: qos_list	QosPolicy []
copy_from_topic_qos		ReturnCode_t
	inout: a_datawriter_qos	QosPolicy []
	a_topic_qos	QosPolicy []

发布者（Publisher）代表属于它的一个或多个***数据写入者（DataWriter）***对象。当其中一个 ***DataWriter*** 对象关联的数据发生改变时，发布者决定何时真正发送数据更新消息。在做出此决定时，发布者会考虑与数据有关的（时间戳，写入者等）以及与***发布者（Publisher）***和***数据写入者（DataWriter）***的 QoS 有关的额外信息，根据这些信息判断何时执行数据发送或者取消数据发送。

除基类方法 ***set_qos，get_qos，set_listener，get_listener，enable，get_statuscondition，create_datawriter 和 delete_datawriter*** 之外的所有方法都可能返回 NOT_ENABLED。

2.2.2.4.1.1 set_listener (来自实体类 Entity)

通过扩展***实体（Entity）***类，***发布者（Publisher）***可以在创建时或创建后使用 ***set_listener*** 方法将其与***监听器（Listener）***进行绑定。附加的 ***Listener*** 必须继承自 ***PublisherListener***。监听器在 2.2.4 监听器，条件和等待集中描述。

2.2.2.4.1.2 get_listener (来自实体类 Entity)

获取 ***Publisher*** 附加绑定的 ***PublisherListener***。

2.2.2.4.1.3 set_qos (来自实体类 Entity)

通过扩展***实体（Entity）***类，***发布者（Publisher）***可以在创建时或创建后使用 ***set_qos*** 方法设置 QoS 策略。关于可以在 ***Publisher*** 上设置的 QoS 策略，请参阅 2.2.3 支持的 QoS。

除标准错误代码外，还可能返回错误代码：IMMUTABLE_POLICY，INCONSISTENT_POLICY。

2.2.2.4.1.4 get_qos (来自实体类 Entity)

此方法允许访问 QoS 策略的取值。

2.2.2.4.1.5 create_ datawriter

此方法将创建一个 ***DataWriter***。返回的 DataWriter 将附加并属于此 ***Publisher***。

create_datawriter 方法返回的 ***DataWriter*** 实际上是一个派生类，特定于与主题关联的数据类型。如 2.2.2.3.7 所述，对于每个应用程序定义的数据类型“Foo”，都有一个隐含的，自动生成的类 ***FooDataWriter***，它扩展了 ***DataWriter*** 并包含写入“Foo”类型数据的方法。

如果方法调用失败，将返回“nil”值（由平台指定）。

请注意，为 ***DataWriter*** 构建 QoS 的常见模式如下：

- 通过***主题（Topic）***对象的 ***get_qos*** 方法获取关联***主题（Topic）***的 QoS 策略。
- 通过 ***Publisher*** 对象的 ***get_default_datawriter_qos*** 方法获取默认的 ***DataWriter*** QoS。
- 结合上述两种 QoS 策略，根据需要有选择地修改。
- 使用生成的 QoS 策略创建 ***DataWriter***。

特殊值 DATAWRITER_QOS_DEFAULT 表明应使用工厂中设置的默认 ***DataWriter*** QoS 创建 ***DataWriter***。使用这个特殊值等同于应用程序通过 ***get_default_datawriter_qos***（2.2.2.4.1.15）方法获取默认 ***DataWriter*** QoS 并使用获得的 QoS 创建 ***DataWriter***。

特殊值DATAWRITER_QOS_USE_TOPIC_QOS表明应使用默认***DataWriter*** QoS 和 ***Topic*** QoS 组合创建 ***DataWriter***。使用此值等同于应用程序获取默认 ***DataWriter*** QoS 和 ***Topic*** QoS（通过方法 ***Topic :: get_qos***），然后使用 ***copy_from_topic_qos*** 方法组合这两个 QoS，从而使用 ***Topic*** QoS 中设置的 QoS 策略“覆盖”对应的默认 QoS 策略。最终将生成的 QoS 用于创建 ***DataWriter***。

传递给方法的***主题（Topic）***所在的 ***DomainParticipant*** 必须与创建此 ***Publisher*** 的 ***DomainParticipant*** 相同。如果主题是在其他 ***DomainParticipant*** 中创建，则此方法将失败并返回 nil。

2.2.2.4.1.6 delete_datawriter

此方法删除属于 ***Publisher*** 的 ***DataWriter***。

必须在创建 ***DataWriter*** 的同一 ***Publisher*** 对象上调用 ***delete_datawriter*** 方法。如果在另一个 ***Publisher*** 上调用 ***delete_datawriter***，该方法将不起作用，并返回 PRECONDITION_NOT_MET。

删除 ***DataWriter*** 将自动取消注册所有实例。根据 WRITER_DATA_LIFECYCLE QosPolicy，删除 ***DataWriter*** 也可以处理所有数据实例。详细信息请参阅 2.2.3.21。

除标准错误代码外，还可能返回错误代码：PRECONDITION_NOT_MET。

2.2.2.4.1.7 lookup_datawriter

此方法查找获取之前创建成功的属于***发布者（Publisher）***的 ***DataWriter***，需要查找的 ***DataWriter*** 主题为 *topic_name*。如果不存在此类 ***DataWriter***，则方法将返回“nil”。

如果 ***Publisher*** 中存在多个 ***DataWriter*** 满足此条件，则方法将返回其中一个，不指定具体是哪一个。

2.2.2.4.1.8 suspend_publications

此方法告知 DDS 服务，应用程序将利用属于本 ***Publisher*** 的 ***DataWriter*** 对象进行多次修改。

这是对服务的提示，服务可以通过例如控制修改的扩散然后对这些修改进行批处理来优化其性能。

不要求服务以任何方式使用此提示。

使用此方法必须有匹配的 ***resume_publications*** 方法调用，表明修改集合已完成。如果在调用 ***resume_publications*** 之前删除了发布者，则将丢弃尚未发布的所有暂停更新。

2.2.2.4.1.9 resume_publications

此方法告知服务，应用程序已完成之前 ***suspend_publications*** 方法启动的多个修改。服务实现者可以使用这种提示批量处理自 ***suspend_publications*** 以来所做的所有修改。

对 ***resume_publications*** 的调用必须与之前对 ***suspend_publications*** 的调用相匹配，否则此方法将返回错误值 PRECONDITION_NOT_MET。

除标准错误代码外，还可能返回错误代码：PRECONDITION_NOT_MET。

2.2.2.4.1.10 begin_coherent_changes

此方法要求应用程序使用属于 ***Publisher*** 的 ***DataWriter*** 对象开始“一致集”的修改。“一致集”将使用匹配的 ***end_coherent_changes*** 方法调用来完成。

“一致集”是修改的集合，必须以如下方式传播：它们在接收方被解析为一致的修改集合;也就是说，接收方只能在集合中的所有修改都可用后才能访问数据。

连接变化可能发生在一组一致变化的中间步骤，例如，发布者或其订阅者之一使用的分区集可能会发生变化，后期加入的 ***DataReader*** 可能会出现在网络上，或者可能发生通信故障。如果此类更改阻止实体接收完整的一致修改集合，则该实体必须表现好像它没有收到任何修改集合。

这些调用可以嵌套。在这种情况下，一致集仅在最后一次调用 ***end_coherent_ changes*** 时终止。

对“一致性更改”的支持使发布端的应用程序能够更改可能属于相同或不同主题的多个数据实例的值，并使这些更改被订阅方“原子地”获取。这在数据取值是相互关联的情况下很有用（例如，如果有两个数据实例表示同一架飞机的“高度”和“速度矢量”并且两者都被更改，以“订阅者可以同时观测到两者的变化”这种方式传输这些值就会很有用;否则，它可能被错误地解释成飞机处于碰撞过程中）。

2.2.2.4.1.11 end_coherent_changes

此方法终结由 ***begin_coherent_changes*** 方法的启动的“一致集”。如果之前没有调用匹

配的 ***begin_coherent_ changes***，则返回错误 PRECONDITION_NOT_MET。

除标准错误代码外，还可能返回错误代码：PRECONDITION_NOT_MET

2.2.2.4.1.12 wait_for_acknowledgments

此方法会阻塞调用线程，直到所有匹配的可靠的 ***DataReader*** 实体都确认收到可靠的 ***DataWriter*** 实体写入的所有数据，或者 *max_wait* 参数设定的持续时间已经到期，以先发生者为准。返回值为 OK 表示所有匹配的可靠的数据读取者已确认收到所有写入的数据样本；返回值 TIMEOUT 表示在确认收到所有数据之前已经超过 *max_wait* 设定的时间周期。

2.2.2.4.1.13 get_participant

此方法返回发布者 ***Publisher*** 所属的 ***DomainParticipant***。

2.2.2.4.1.14 delete_contained_entities

此方法将删除通过 ***Publisher*** 对象的“创建”方法创建的所有实体。也就是说，它会删除所有包含的 ***DataWriter*** 对象。

如果任一包含的实体处于无法删除的状态，则方法将返回 PRECONDITION_NOT_MET。

一旦 ***delete_contained_entities*** 成功返回，应用程序可能会删除***发布者 Publisher***，因为应用程序知道它没有包含 ***DataWriter*** 对象。

2.2.2.4.1.15 set_default_datawriter_qos

此方法设置 ***DataWriter*** QoS 策略的默认值。在 ***create_datawriter*** 方法使用默认 QoS 策略的情况下，该策略值将用于新创建的 ***DataWriter*** 实体。

此方法将检查生成的策略是否是自相容的；如果不是，则方法调用无效并返回 INCONSISTENT_POLICY。

可以将特殊值 DATAWRITER_QOS_DEFAULT 传递给此方法，表明将默认 QoS 重置为工厂使用的初始值，即从未调用 ***set_default_datawriter_qos*** 方法时使用的值。

2.2.2.4.1.16 get_default_datawriter_qos

此方法获取 ***DataWriter*** QoS 的默认值，即在 ***create_datawriter*** 方法使用默认 QoS 策略的情况下用于新创建的 ***DataWriter*** 实体的 QoS 策略。

get_default_datawriter_qos 获取的值与上次成功调用 ***set_default_datawriter_qos*** 时指定的值集保持一致；如果从未调用过 ***set_default_datawriter_qos***，则在 2.2.3 支持的 QoS 中的 QoS 表中列出了 ***DataWriter*** QoS 默认值。

2.2.2.4.1.17 copy_from_topic_qos

此方法将 *a_topic_qos* 中的策略复制到 *a_datawriter_qos* 中的相应策略（即替换 *a_datawriter_qos* 中的值，如果该位置存在值）。

这是一个“方便”的方法，与 ***get_default_datawriter_qos*** 和 ***Topic :: get_qos*** 方法结合使用最为有用。***copy_from_topic_qos*** 方法可用于将 ***DataWriter*** 默认 QoS 策略与 ***Topic*** 的相应 QoS 策略合并。可以使用生成的 QoS 创建新的 ***DataWriter***，或设置其 QoS。

此方法不会检查生成的 ***a_datawriter_qos*** 的一致性。这是因为“合并的”*a_datawriter_qos* 可能不是最后一个，因为应用程序仍然可以在将策略应用于 ***DataWriter*** 之前修改某些策略。

2.2.2.4.2 DataWriter 类

DataWriter 类允许应用程序设置给定主题下发布的数据的值。

DataWriter		
no attributes		
operations		
(inherited) get_qos		ReturnCode_t
	out: qos_list	QosPolicy []
(inherited) set_qos		ReturnCode_t
	qos_list	QosPolicy []
(inherited) get_listener		Listener
(inherited) set_listener		ReturnCode_t
	a_listener	Listener
	mask	StatusKind []
register_instance		InstanceHandle_t
	instance	Data
register_instance_w_timestamp		InstanceHandle_t
	instance	Data
	timestamp	Time_t
unregister_instance		ReturnCode_t
	instance	Data
	handle	InstanceHandle_t
	timestamp	Time_t
get_key_value		ReturnCode_t
	inout: key_holder	Data
	handle	InstanceHandle_t
lookup_instance		InstanceHandle_t
	instance	Data
write		ReturnCode_t
	data	Data
	handle	InstanceHandle_t
write_w_timestamp		ReturnCode_t
	data	Data
	handle	InstanceHandle_t
	timestamp	Time_t
dispose		ReturnCode_t
	data	Data
	handle	InstanceHandle_t
dispose_w_timestamp		ReturnCode_t
	data	Data
	handle	InstanceHandle_t

	timestamp	Time_t
wait_for_acknowledgments		ReturnCode_t
	max_wait	Duration_t
get_liveliness_lost_status		ReturnCode_t
	out: status	LivelinessLostStatus
get_offered_deadline_missed_status		ReturnCode_t
	out: status	OfferedDeadlineMissedStatus
get_offered_incompatible_qos_status		ReturnCode_t
	out: status	OfferedIncompatibleQosStatus
get_publication_matched_status		ReturnCode_t
	out: status	PublicationMatchedStatus
get_topic		Topic
get_publisher		Publisher
assert_liveliness		ReturnCode_t
get_matched_subscription_data		ReturnCode_t
	inout: subscription_data	SubscriptionBuiltinTopicData
	subscription_handle	InstanceHandle_t
get_matched_subscriptions		ReturnCode_t
	inout: subscription_handles	InstanceHandle_t []

DataWriter 只附加到一个充当它的工厂的 ***Publisher***。

DataWriter 只与一个***主题（Topic）***绑定，因此只与一种数据类型绑定。***主题（Topic）***必须在 ***DataWriter*** 创建之前存在。

DataWriter 是一个抽象类。它必须专门用于特定的应用数据类型，如图 2.8 所示。在自动生成的类中必须为假设的应用类型“Foo”定义的其他方法如下表所示：

FooDataWriter		
no attributes		
operations		
register_instance		InstanceHandle_t
	instance	Foo
register_instance_w_timestamp		InstanceHandle_t
	instance	Foo
	timestamp	Time_t
unregister_instance		ReturnCode_t
	instance	Foo
	handle	InstanceHandle_t
unregister_instance_w_timestamp		ReturnCode_t

	instance	Foo
	handle	InstanceHandle_t
	timestamp	Time_t
get_key_value		ReturnCode_t
	inout: key_holder	Foo
	handle	InstanceHandle_t
lookup_instance		InstanceHandle_t
	instance	Foo
write		ReturnCode_t
	instance_data	Foo
	handle	InstanceHandle_t
write_w_timestamp		ReturnCode_t
	instance_data	Foo
	handle	InstanceHandle_t
	timestamp	Time_t
dispose		ReturnCode_t
	instance	Foo
	handle	InstanceHandle_t
dispose_w_timestamp		ReturnCode_t
	instance	Foo
	handle	InstanceHandle_t
	timestamp	Time_t

除基类方法 ***set_qos，get_qos，set_listener，get_listener，enable 和 get_statuscondition*** 之外的所有方法都可能返回值 NOT_ENABLED。

以下子条目提供了有关方法的详细信息。

2.2.2.4.2.1 set_listener (来自实体类 Entity)

通过扩展***实体（Entity）***类，***数据写入者（DataWriter）***可以在创建时或创建后使用 ***set_listener*** 方法将其与***监听器（Listener）***进行绑定。附加的 ***Listener*** 必须继承自 ***DataWriterListener***。监听器在 2.2.4 监听器，条件和等待集中有详细描述。

2.2.2.4.2.2 get_listener (来自实体类 Entity)

获取 ***DataWriter*** 附加绑定的 ***DataWriterListener***。

2.2.2.4.2.3 set_qos (来自实体类 Entity)

通过扩展***实体（Entity）***类，***数据写入者（DataWriter）***可以在创建时或创建后使用 ***set_qos*** 方法设置 QoS 策略。关于可以在 ***DataWriter*** 上设置的 QoS 策略，请参阅 2.2.3 支持的 QoS。

除标准错误代码外，还可能返回错误代码：IMMUTABLE_POLICY，INCONSISTENT_POLICY。

2.2.2.4.2.4 get_qos (来自实体类 Entity)

此方法允许访问 QoS 策略的取值。

2.2.2.4.2.5 register_instance

此方法通知服务应用程序将修改特定数据实例。它为服务提供了预先配置自身的机会，以提高服务性能。

它将一个实例（用于获取关键字值）作为参数，并返回一个可用于连续写入或处理数据的方法的句柄。

应在调用修改实例的方法之前调用此方法，例如 ***write，write_w_timestamp，dispose 和 dispose_w_timestamp***。

如果服务不想为该实例分配任何句柄，则可以返回特殊值 HANDLE_NIL。

此方法可能会在 ***write*** 方法（2.2.2.4.2.11）所述的相同情况下阻塞并返回 TIMEOUT。

在与 ***write*** 方法(2.2.2.4.2.11)相同的情况下，此操作可能会返回 OUT_OF_RESOURCES。

register_instance 方法是幂等的。如果传入参数为已注册的实例，此方法只返回已分配的句柄。这可用于查找和获取分配给给定实例的句柄。显式使用此方法是可选的，因为应用程序可以直接调用 ***write*** 方法并指定 HANDLE_NIL 以表明应检查“关键字”以确认实例。

2.2.2.4.2.6 register_instance_w_timestamp

此方法与 ***register_instance*** 功能相同，在应用程序希望指定 ***source_timestamp*** 值的情况下，可以代替 ***register_instance***。***source_timestamp*** 可能会影响读取者观察来自多个写入者的事件的相对顺序。有关 DESTINATION_ORDER QoS 策略的详细信息，请参见 2.2.3.17。

此方法可能会在 ***write*** 方法（2.2.2.4.2.11）所述的相同情况下阻塞并返回 TIMEOUT。

在与 ***write*** 方法(2.2.2.4.2.11)相同的情况下，此操作可能会返回 OUT_OF_RESOURCES。

2.2.2.4.2.7 unregister_instance

此方法为 ***register_instance*** 的逆方法。它只在当前已经注册的实例上调用。

无论该实例调用多少次 ***register_instance*** 方法，每个实例只调用一次 ***unregister_instance*** 方法。

此方法通知 DDS 服务，数据写入者 ***DataWriter*** 不会再修改该数据实例。调用此方法意味着服务可以删除该实例的所有本地信息。在调用 ***unregister_instance*** 之后，应用程序不能再使用先前分配给该实例的句柄。

特殊值 HANDLE_NIL 可当做 ***handle*** 参数的取值。这表明应该（通过关键字）从***实例（instance）***自动推导出实例的身份标识。

如果 ***handle*** 是除 HANDLE_NIL 之外的任何值，则它必须对应在注册实例（由其关键字标识）时调用 ***register_instance*** 方法返回的值。否则行为如下：

- 如果句柄 ***handle*** 对应于某一存在的实例但与***“instance”***参数引用的实例不一样，则行为一般为未指定。如果 Service 实现者可检测到这种不一致，则方法将失败并返回错误代码 PRECONDITION_NOT_MET “。
- 如果句柄与所有已经存在的实例都不对应，则行为一般为未指定。如果 Service 实现者可检测出这种情况，则方法将失败并返回错误代码“BAD_PARAMETER”。

如果在实例取消注册之后，应用程序想要再次修改（写入或处理）实例，必须再次注册该实例，否则使用特殊句柄 ***handle*** 值 HANDLE_NIL。

此方法不代表实例已删除（这是 ***dispose*** 方法的目的）。操作 ***unregister_instance*** 只是表明 ***DataWriter*** 不再有关于该实例的“任何说法”。如果没有 ***DataWriter*** 实体正在对该实例进行数据写入，则正在读取实例数据的 ***DataReader*** 实体最终将接收具有 NOT_ALIVE_NO_WRITERS 实例状态的数据样本。

此方法会影响数据实例的所有权（如 2.2.3.9 OWNERSHIP 和 2.2.3.23.1 冗余系统上的所有权解析所述）。如果 ***DataWriter*** 是实例的唯一所有者，则调用 ***unregister_instance*** 方法将放弃该所有权。

此方法可能会在 ***write*** 方法（2.2.2.4.2.11 write）所述的相同情况下阻塞并返回 TIMEOUT。

除标准错误代码外，还可能返回错误代码：TIMEOUT，PRECONDITION_NOT_MET。

2.2.2.4.2.8 unregister_instance_w_timestamp

此方法与 ***unregister_instance*** 功能相同，并在应用程序希望指定 ***source_timestamp*** 值的情况下，可以代替 ***unregister_instance***。***source_timestamp*** 可能会影响读取者观察来自多个写入者的事件的相对顺序。有关 DESTINATION_ORDER QoS 策略的详细信息，请参见 2.2.3.17 DESTINATION_ORDER）。

handle 参数值的约束和相应的错误行为与 ***unregister_instance*** 方法（2.2.2.4.2.7）相同。

此方法可能会在 ***write*** 方法（2.2.2.4.2.11）所述的相同情况下阻塞并返回 TIMEOUT。

2.2.2.4.2.9 get_key_value

此方法可用于获取与 ***instance_handle*** 对应的实例关键字值。该方法仅填充 ***key_holder*** 实例内形成关键字的字段。

如果 ***InstanceHandle_t*** *a_handle* 与 ***DataWriter*** 已知的现有数据对象不对应，则此方法可以返回 BAD_PARAMETER。如果 DDS 服务的实现无法检查出无效句柄，则此情况下的结果未指定。

2.2.2.4.2.10 lookup_instance

此方法将实例 *instance* 作为参数，并返回一个句柄，该句柄可当做后续方法的参数。*instance* 参数仅用于检查定义关键字（key）的字段。

此方法不会注册新的实例。如果实例先前未注册过，或者由于其他原因服务无法提供实例句柄，则服务将返回特殊值 HANDLE_NIL。

2.2.2.4.2.11 write

此方法修改数据实例的值。使用此方法时，DDS 服务将自动根据 ***SampleInfo*** 中 ***source_timestamp*** 属性提供可供 ***DataReader*** 对象使用的 ***source_timestamp*** 的值。有关读取端数据时间戳的更多详细信息，请参见 2.2.2.5 订阅模块;有关 DESTINATION_ORDER QoS 策略的详细信息，请参见 2.2.3.17。

特定应用程序的将要写入数据的数据类型生成的专用类上必须提供此方法实现。这样，承载数据的 ***data*** 参数具有适当的应用程序定义类型（例如“Foo”）。

此方法维持了 ***DataWriter*** 本身，***Publisher*** 和 ***DomainParticipant*** 的存活性。

特殊值 HANDLE_NIL 可当做 ***handle*** 参数的取值。这表明应该（通过关键字）从***实例数据（instance_data）***自动推导出实例的身份标识。

如果 ***handle*** 是除 HANDLE_NIL 之外的任何值，则它必须对应在注册实例（由其关键字标识）时调用 ***register_instance*** 方法返回的值。否则行为如下：

•如果句柄 ***handle*** 对应于某一存在的实例但与“***data***”参数引用的实例不一样，则行为一般为未指定。如果 DDS 服务实现者可检测到这种不一致，则方法将失败并返回错误代码 PRECONDITION_NOT_MET “。

•如果句柄与所有已经存在的实例都不对应，则行为一般为未指定。如果 DDS 服务实现者可检测出这种情况，则方法将失败并返回错误代码“BAD_PARAMETER”。

如果 RELIABILITY QoS 的 ***kind*** 设置为 RELIABLE，当数据修改导致数据出现丢失或者导致 RESOURCE_LIMITS QoS 指定的资源超限，***write*** 方法即会阻塞。在这些情况下，可以通过 RELIABILITY QoS 的 ***max_blocking_time*** 参数配置 ***write*** 方法的最长阻塞时间，用以等待数据存储空间变得可用。如果在 ***DataWriter*** 能够存储修改而不超出限制之前经过了 ***max_blocking_time***，则 ***write*** 方法将失败并返回 TIMEOUT。

具体来说，即使 HISTORY 类型为 KEEP_LAST，***DataWriter write*** 方法仍可能会在以下情况阻塞（注意该列表可能并非详尽无遗的）。

•如果（RESOURCE_LIMITS ***max_samples*** <RESOURCE_LIMITS ***max_instances*** * HISTORY ***depth***），则在 ***max_samples*** 资源限制用尽的情况下，只要此类实例下至少还有一个数据样本，则允许 DDS 服务丢弃某些其他实例的数据样本。如果仍然无法使空间可用于存储修改，则允许数据写入者阻塞。

•如果（RESOURCE_LIMITS ***max_samples*** <RESOURCE_LIMITS ***max_instances***），则无论 HISTORY ***depth*** 为多少，***DataWriter*** 都可能会阻塞。

如果满足以下两个条件，则允许 ***write*** 方法立即返回错误代码 OUT_OF_RESOURCES，而不是阻塞：

1.阻塞的原因是超出了 RESOURCE_LIMITS 限制。

2.服务确定等待 ***max_waiting_time*** 时间后仍然没有机会释放必需的资源。例如，如果获得必要资源的唯一方法是用户取消注册实例。

如果句柄 ***handle*** 对应于某一存在的实例但与“***data***”参数引用的实例不一样，则行为一般为未指定。如果 DDS 服务实现者可检测到这种不一致，则方法将失败并返回错误代码 PRECONDITION_NOT_MET“。如果句柄与所有已经存在的实例都不对应，即 *handle* 无效，则行为一般为未指定。如果 DDS 服务实现者可检测出这种情况，则方法将失败并返回错误代码“BAD_PARAMETER”。

2.2.2.4.2.12 write_w_timestamp

此方法与 ***write*** 功能相同，除此之外还可以通过 ***SampleInfo*** 中 ***source_timestamp*** 属性提供可供 ***DataReader*** 对象使用的 ***source_timestamp*** 的值。有关读取端数据时间戳的更多详细信息，请参见 2.2.2.5 订阅模块;有关 DESTINATION_ORDER QoS 策略的详细信息，请参见 2.2.3.17。

handle 参数值的约束和相应的错误行为与 ***write*** 方法（2.2.2.4.2.11）相同。

此方法可能会在 ***write*** 方法（2.2.2.4.2.11）所述的相同情况下阻塞并返回 TIMEOUT。

在与 ***write*** 方法(2.2.2.4.2.11)相同的情况下，此方法可能会返回 OUT_OF_RESOURCES。

在与 ***write*** 方法(2.2.2.4.2.11)相同的情况下，可能会返回 PRECONDITION_NOT_MET。

此方法可能在 ***write*** 方法（2.2.2.4.2.11）所述的相同情况下返回 BAD_PARAMETER。

与 ***write*** 类似，特定应用程序的将要写入数据的数据类型生成的专用类上必须提供此方法实现。

2.2.2.4.2.13 dispose

此方法请求中间件删除数据（实际执行删除会被推迟，直到整个系统中不再使用该数据）。通常应用程序通过 ***DataReader*** 对象的方法（该对象已知该实例将被删除）了解数据删除（更多详细信息请参阅 2.2.2.5 订阅模块）。

此方法不会修改实例的值。传递 ***instance*** 参数只是为了识别实例。

使用此方法时，DDS 服务将通过 ***SampleInfo*** 中 ***source_timestamp*** 属性提供可供 ***DataReader*** 对象使用的 ***source_timestamp*** 的值。

handle 参数值的约束和相应的错误行为与 ***unregister_instance*** 方法（2.2.2.4.2.7）相同。

此方法可能会在 ***write*** 方法（2.2.2.4.2.11）所述的相同情况下阻塞并返回 TIMEOUT。

在与 ***write*** 方法（2.2.2.4.2.11）相同的情况下，此方法可能会返回 OUT_OF_RESOURCES。

2.2.2.4.2.14 dispose_w_timestamp

此方法与 ***dispose*** 功能相同，除此之外应用程序还可以通过 ***SampleInfo*** 中 ***source_timestamp*** 属性提供可供 ***DataReader*** 对象使用的 ***source_timestamp*** 的值。（请参阅 2.2.2.5 订阅模块）。

handle 参数值的约束和相应的错误行为与 ***dispose*** 操作（2.2.2.4.2.13）相同。

此方法可能会在 ***dispose*** 方法（2.2.2.4.2.13）所述的相同情况下返回 PRECONDITION_NOT_MET。

在与 ***dispose*** 方法（2.2.2.4.2.13）相同的情况下，此方法可能返回 BAD_PARAMETER。

此方法可能会在 ***write*** 方法（2.2.2.4.2.11）所述的相同情况下阻塞并返回 TIMEOUT。

在与 ***write*** 方法（2.2.2.4.2.11）相同的情况下，此方法可能会返回 OUT_OF_RESOURCES。

除标准错误代码外，还可能返回错误代码：TIMEOUT，PRECONDITION_NOT_MET。

2.2.2.4.2.15 wait_for_acknowledgments

仅当 ***DataWriter*** 将 RELIABILITY QoS ***kind*** 设置为 RELIABLE 时，才会使用此方法。否则将立即返回 RETCODE_OK。

wait_for_acknowledgments 方法阻塞调用线程，直到 ***DataWriter*** 写入的所有数据都被 RELIABILITY QoS kind 取值为 RELIABLE 的所有匹配的 ***DataReader*** 实体确认，或者直到 ***max_wait*** 参数指定的持续时间到期，以先发生者为准。返回值为 OK 表示所有可靠的匹配数据读取者已确认所有写入的数据样本；返回值 TIMEOUT 表示在确认所有数据之前已经过了 ***max_wait***。

2.2.2.4.2.16 get_liveliness_lost_status

此方法允许调用者访问 LIVELINESS_LOST 通信状态。通信状态在 2.2.4.1 通信状态中描述。

2.2.2.4.2.17 get_offered_deadline_missed_status

此方法允许调用者访问 OFFERED_DEADLINE_MISSED 通信状态。通信状态在 2.2.4.1 通信状态中描述。

2.2.2.4.2.18 get_offered_incompatible_qos_status

此方法允许调用者访问 OFFERED_INCOMPATIBLE_QOS 通信状态。通信状态在 2.2.4.1

通信状态中描述。

2.2.2.4.2.19 get_publication_matched_status

此方法允许调用者访问 PUBLICATION_MATCHED 通信状态。通信状态在 2.2.4.1 通信状态中描述。

2.2.2.4.2.20 get_topic

此方法返回与 ***DataWriter*** 关联的***主题（Topic）***，与用于创建 ***DataWriter*** 的***主题（Topic）***相同。

2.2.2.4.2.21 get_publisher

此方法返回 ***DataWriter*** 所属的 ***Publisher***。

2.2.2.4.2.22 assert_liveliness

此方法手动断言 ***DataWriter*** 的活跃性，与 LIVELINESS QoS 策略（参见 2.2.3 支持的 QoS）结合使用，以向服务提示实体处于活动状态。

仅当 LIVELINESS QoS 设置为 MANUAL_BY_PARTICIPANT 或 MANUAL_BY_TOPIC 时，才需要使用此方法。否则，它没有任何效果。

注意：通过 ***DataWriter*** 对象上的 ***write*** 方法写入数据会断言 ***DataWriter*** 本身及其 ***DomainParticipant*** 的活跃性。因此，只有在应用程序不定期写入数据时才需要使用 ***assert_liveliness***。

完整的细节在 2.2.3.11 LIVELINESS 中提供。

2.2.2.4.2.23 get_matched_subscription_data

此方法获取当前与 ***DataWriter*** “关联”的订阅方信息，即应用程序尚未指示的具有匹配的 ***Topic*** 和兼容 QoS 的订阅应通过 ***DomainParticipant ignore_subscription*** 方法进行“忽略”。

subscription_handle 必须对应于当前与 ***DataWriter*** 关联的订阅方，否则方法调用将失败并返回 BAD_PARAMETER。 ***get_matched_subscriptions*** 方法可用于查找当前与 ***DataWriter*** 匹配的订阅方。

如果底层不包含填写 *subscription_data* 所需的信息，则方法也可能失败。在这种情况下，将返回 UNSUPPORTED。

2.2.2.4.2.24 get_matched_subscriptions

此方法获取当前与 ***DataWriter*** “关联”的订阅列表，即应用程序尚未指示的具有匹配的 ***Topic*** 和兼容 QoS 的订阅应通过 ***DomainParticipant ignore_subscription*** 方法进行“忽略”。

“subscription_handles” 列表中返回的句柄是 DDS 实现方用于标识本地相应匹配的 ***DataReader*** 实体的句柄。在读取“DCPSSubscriptions”内置主题时，这些句柄与 ***SampleInfo*** 的“instance_handle”字段的句柄相对应。

如果底层不在本地维护连接信息，则方法可能会失败。

2.2.2.4.3 PublisherListener 接口

PublisherListener
no attributes
no operations

由于 ***Publisher*** 是一种实体，因此它可以拥有与之关联的监听器。在这种情况下，关联的监听器应该是拥有具体类型的 ***PublisherListener***。 2.2.4 监听器，条件和等待集中描述了此监听器的使用及其与 ***Publisher*** 通信状态变化的关系。

2.2.2.4.4 DataWriterListener 接口

DataWriterListener		
no attributes		
operations		
on_liveliness_lost		void
	the_writer	DataWriter
	status	LivelinessLostStatus
on_offered_deadline_missed	the_writer	DataWriter
	status	OfferedDeadlineMissedStatus
on_offered_incompatible_qos	the_writer	DataWriter
	status	OfferedIncompatibleQosStatus
on_publication_matched		
	the_writer	DataWriter
	status	PublicationMatchedStatus

由于 ***DataWriter*** 是一种实体，因此它可以拥有与之关联的监听器。在这种情况下，关联的监听器应该是拥有具体类型的 ***DataWriterListener***。 2.2.4 监听器，条件和等待集中描述了此监听器的使用及其与 ***DataWriter*** 通信状态变化的关系。

2.2.2.4.5 并发行为

本规范没有假设发布端应用程序的设计方式。特别是多个 ***DataWriter*** 可以在不同的线程中运行。如果它们隶属于同一个 ***Publisher***，则中间件保证其操作是线程安全的。不要求每个请求线程与其他请求线程隔离处理（例如导致几个隔离的一致变化集合）。如果每个请求线程与其他请求线程相互隔离是期望的行为，那么正确的设计是为每个线程创建一个 ***Publisher***。

2.2.2.5 订阅模块

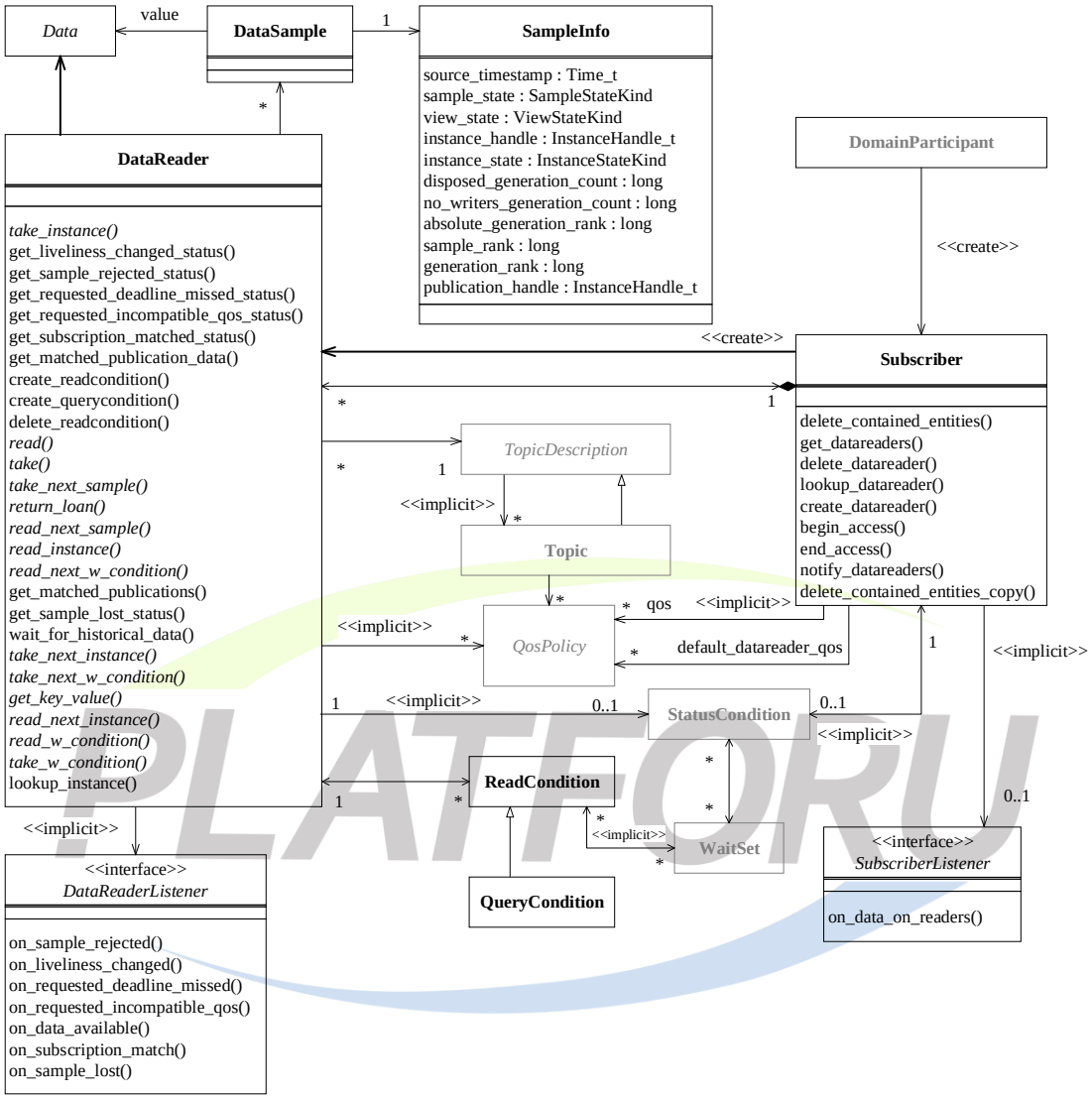

图 2-10 DCPS 订阅模块的类模型

DCPS 订阅模块由以下类组成

- Subscriber
- DataReader
- DataSample
- SampleInfo
- SubscriberListener
- DataReaderListener
- ReadCondition
- QueryCondition

以下小节介绍了如何访问获取数据并介绍了 ***sample_state，view_state 和 instance_state***。小节 2.2.2.5.2（“订阅者类”）到 2.2.2.5.9（“QueryCondition 类”）介绍了属于该模块的

每个类的详细信息。

2.2.2.5.1 访问获取数据

应用程序通过 ***DataReader*** 对象的以下方法实现数据的访问获取：***read***，***read_w_condition***，***take*** 和 ***take_w_condition***。“***read***”方法的语义是应用程序只能访问相应的数据，数据生命周期的管理仍然是中间件的责任，而且数据可以被再次“***read***”。“***take***”方法的语义是应用程序对数据生命周期负全部责任，***DataReader*** 将无法再访问该数据。***DataReader*** 可以多次访问同一数据样本，前提是之前所有的访问都是通过 ***read*** 方法完成。

以上每一个方法都会返回数据值和关联的 ***SampleInfo*** 对象的有序集合。每个数据值表示数据的原子信息（即一个实例的值）。此集合可能包含与相同或不同实例（由关键字标识）相关的数据样本。如果 HISTORY QoS（2.2.3.18）的设置允许，多个样本可以引用相同的实例。

2.2.2.5.1.1 SampleInfo 说明

SampleInfo 包含与关联数据（***Data***）值相关的信息：

- ***Data*** 值的 ***sample_state***（数据样本已被同一 ***DataReader*** READ 或 NOT_READ）。
- 相关实例的 ***view_state***（对 ***DataReader*** 而言该实例为 NEW 或 NOT_NEW）-请参阅下文。
- 相关实例的 ***instance_state***（实例为 ALIVE，NOT_ALIVE_DISPOSED 或 NOT_ALIVE_NO_WRITERS）-请参阅下文。
- ***valid_data*** 标志。该标志指明是否存在与样本相关联的数据。某些样本不包含数据，则表明相应实例的 ***instance_state*** 有更改-请参阅下文。
- 收到样本时相关实例的 ***dispos_generation_count*** 和 ***no_writers_generation_count*** 属性的值。这些计数器给出了实例在收到样本时已经变为 ALIVE 状态（通过 ***instance_state*** = ALIVE）的次数-见下文。
- 返回序列中数据样本的 ***sample_rank*** 和 ***generation_rank***。这些排名给出了 ***read*** 或 ***take*** 方法返回的序列中数据样本的预览。
- ***DataReader*** 中数据样本的 ***absolute_generation_rank***。此排名给出了 ***DataReader*** 中可用内容的预览。
- 样本的 ***source_timestamp***。这是 ***DataWriter*** 在生成样本时提供的时间戳。

2.2.2.5.1.2 SampleInfo 的 sample_state 说明

对于收到的每个数据样本，中间件在内部维护一个对应于每个 ***DataReader*** 的 ***sample_state***。***sample_state*** 可以是 READ 或 NOT_READ。

- READ 表示 ***DataReader*** 已通过 ***read*** 方法访问该数据样本。
- NOT_READ 表示 ***DataReader*** 之前未访问过该数据样本。

对于 ***read*** 或 ***take*** 方法返回的集合中的每个样本，***sample_state*** 通常都是不同的。

2.2.2.5.1.3 SampleInfo 的 instance_state 说明

对于每个实例，中间件在内部维护一个 ***instance_state***。***instance_state*** 可以是 ALIVE，NOT_ALIVE_DISPOSED 或 NOT_ALIVE_NO_WRITERS。

- ALIVE 表示（a）已收到实例的样本，（b）有活动的 DataWriter 实体正在写入实例，（c）实例还未被显示处理（或者在实例被处理删除后收到更多数据样本）。

•NOT_ALIVE_DISPOSED 表示 ***DataWriter*** 通过 ***dispose*** 方法显式地删除了实例。
•NOT_ALIVE_NO_WRITERS 表示 ***DataReader*** 已将实例声明为非活动状态，因为它检测到没有活动的 ***DataWriter*** 实体正在编写该实例数据。

引起 ***instance_state*** 变化的精确行为事件取决于 OWNERSHIP QoS 的设置：
•如果 OWNERSHIP 设置为 EXCLUSIVE，则只有当“拥有”该实例的 ***DataWriter*** 显式处理它时，***instance_state*** 才会变为 NOT_ALIVE_DISPOSED。仅当拥有该实例的 ***DataWriter*** 写入该实例数据时，***instance_state*** 才会再次变为 ALIVE。
•如果 OWNERSHIP 设置为 SHARED，任何 ***DataWriter*** 显式地处理该实例，***instance_state*** 将变为 NOT_ALIVE_DISPOSED。只要 ***DataWriter*** 再次写入实例数据，***instance_state*** 就会变为 ALIVE。

SampleInfo 中可用的 ***instance_state*** 是获取返回集合时实例的 ***instance_state*** 快照（即在调用 ***read*** 或 ***take*** 方法时）。因此，***instance_state*** 对于返回集合中引用同一实例的所有样本都是相同的。

2.2.2.5.1.4 SampleInfo 的 valid_data 说明

通常，每个 ***DataSample*** 都包含 ***SampleInfo*** 和 ***Data***。但是在某些情况下，***DataSample*** 仅包含 ***SampleInfo*** 并没有关联的数据。当 DDS 服务通知应用程序相关实例状态发生了更改，这种更改是由某些内部机制（例如超时）引起的，便会出现只有 ***SampleInfo*** 却没有关联数据这种情况。一个典型的例子是当 DDS 服务检测到实例没有写入者并将相应的 ***instance_state*** 更改为 NOT_ALIVE_NO_WRITERS 时。

中间件返回不包含 Data 的 DataSamples 的实际场景集是取决于 DDS 实现的。应用程序可以通过检查 ***valid_data*** 标志的值来区分特定 ***DataSample*** 是否具有数据。如果此标志设置为 TRUE，则 ***DataSample*** 包含有效数据，如果该标志设置为 FALSE，则 ***DataSample*** 不包含数据。

为了确保正确性和可移植性，在访问 ***DataSample*** 的 Data 之前，应用程序必须检查 ***valid_data*** 标志，如果标志设置为 FALSE，则应用程序不应访问 ***DataSample*** 的 ***Data*** 部分，即应用程序应该只访问 ***SampleInfo***。

2.2.2.5.1.5 SampleInfo 的 disposed_generation_count 和 no_writers_generation_count 属性说明

对于每个实例，中间件在内部维护两个计数器：***disposed_generation_count*** 和 ***no_writers_generation_count***，涉及到每个 ***DataReader***：
•当 ***DataReader*** 首次检测到从未见过的实例时，***disposed_generation_count*** 和 ***no_writers_generation_count*** 被初始化为零。
•每次实例的 ***instance_state*** 从 NOT_ALIVE_DISPOSED 更改为 ALIVE 时，***disposed_generation_count*** 都会增加。
•每次实例的 ***instance_state*** 从 NOT_ALIVE_NO_WRITERS 更改为 ALIVE 时，***no_writers_generation_count*** 都会增加。

SampleInfo 中的 ***disposed_generation_count*** 和 ***no_writers_generation_count*** 表示接收数据样本时相应计数器的快照。

2.2.2.5.1.6 SampleInfo 的 sample_rank, generation_rank 和 absolute_generation_rank 属性说明

SampleInfo 中可用的 ***sample_rank*** 和 ***generation_rank*** 仅基于 ***read*** 或 ***take*** 方法返回的有序集合中的实际样本计算。

- ***sample_rank*** 给出了返回集合中同一实例的样本数。
- ***SampleInfo*** 中可用的 ***generation_rank*** 给出了样本（S）与返回集合中同一实例的最新样本（MRSIC）之间“代”的差异，即 ***generation_rank*** 计算从接收 S 到接收 MRSIC 之间实例从非活动状态转变为活动状态的次数。

generation_rank 采用以下公式计算：

generation_rank =
(MRSIC.disposed_generation_count + MRSIC.no_writers_generation_count)
- (S.disposed_generation_count + S.no_writers_generation_count)

SampleInfo 中可用的 ***absolute_generation_rank*** 给出了样本（S）与中间件收到的同一实例的最新样本（MRS）之间“代”的差异，即 ***absolute_generation_rank*** 计算实例从接收 S 到调用 ***read*** 或 ***take*** 方法之间从非活动状态转换为活动状态的次数。

absolute_generation_rank =
(MRS.disposed_generation_count + MRS.no_writers_generation_count)
- (S.disposed_generation_count + S.no_writers_generation_count)

2.2.2.5.1.7 SampleInfo 的 counters 和 ranks 属性说明

计数器和排名使得应用程序可以区分实例不同“代”的样本。在应用程序调用 ***read*** 或 ***take*** 方法访问数据之前，实例可能会多次从非活动状态转换为活动状态（再由活动态转换为非活动态）。在这种情况下，返回的集合可能包含跨代的样本（即实例变为非活动状态之前接收的样本，实例重新出现之后接收的其他样本）。使用 ***SampleInfo*** 中的信息，应用程序可以预测返回的集合中以及底层结构中出现的同一实例的其他信息，从而做出适当的决策。例如应用程序仅考虑每个实例最新样本的话，则其只会查看 ***sample_rank*** == 0 的样本。类似地如果应用程序仅考虑集合中最新一代的样本的话，则其只会查看 ***generation_rank*** == 0 的样本。如果应用程序仅需要可用的最新一代的样本，则其将忽略 ***absolute_generation_rank*** ! = 0 的样本。也可以使用其他应用定义的标准。

2.2.2.5.1.8 SampleInfo 的 view_state 属性说明

对于每个实例（由关键字 ***key*** 标识），中间件在内部维护一个 ***view_state***，涉及到每个 ***DataReader***。***view_state*** 取值为 NEW 或 NOT_NEW。

- NEW 表示这是 ***DataReader*** 第一次访问该实例的样本，或者 ***DataReader*** 已访问该实例之前的样本，但该实例已重生（即实例先变为非活动状态，然后再次变为活动状态）。通过检查 ***disposed_generation_count*** 和 ***no_writers_generation_count*** 来区分这两种情况。
- NOT_NEW 表示 ***DataReader*** 已经访问了实例的样本，而且该实例还没有重生。

SampleInfo 中可用的 ***view_state*** 是获取返回集合时（在 ***read*** 或 ***take*** 方法被调用时）实例的 ***view_state*** 快照，该实例的 ***view_state*** 与访问样本的 ***DataReader*** 有关。因此对于返回集合中同一实例的所有样本，其 ***view_state*** 都是相同的。

一旦检测到实例没有任何“存活的”写入者并且 ***DataReader*** 已经“获取”与实例有关的所有样本，中间件就可以回收该实例的所有本地资源。将来的数据样本将被视为“从未见过”。

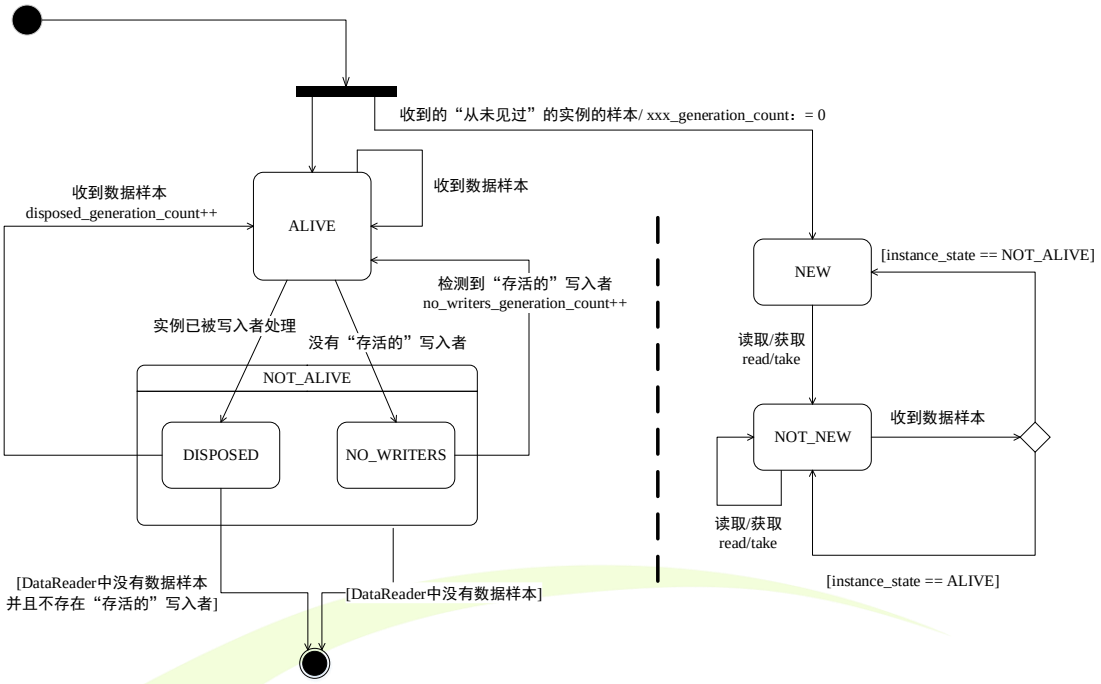

图 2-11 单个实例的 *instance_state* 和 *view_state* 的状态图

2.2.2.5.1.9 数据访问模式

应用程序通过 ***DataReader*** 的 ***read*** 或 ***take*** 方法来访问数据。这些方法返回由 ***SampleInfo*** 部分和 ***Data*** 部分组成的有序 ***DataSamples*** 集合。中间件构建此集合的方式取决于 ***DataReader*** 和 ***Subscriber*** 设置的 QoS 策略、数据样本的源时间戳以及传递给 ***read/take*** 方法的参数，即：

- 所需的 sample 状态（即 READ，NOT_READ 或两者都需要）。
- 所需的 view 状态（即 NEW，NOT_NEW 或两者都需要）。
- 所需的 instance 状态（ALIVE，NOT_ALIVE_DISPOSED，NOT_ALIVE_NO_WRITERS 或这些的组合）。

read 和 ***take*** 方法是非阻塞的，只提供与指定状态匹配的当前可用的内容。

read_w_condition 和 ***take_w_condition*** 方法将 ***ReadCondition*** 对象作为参数，而不是采用 sample，view 和 instance 状态作为参数。表现为将条件为 TRUE 的样本返回。这两种方法与 ***ReadCondition*** 对象和 ***WaitSet*** 一起执行等待读取（见下文）。

一旦数据样本可供数据读取者使用，它们就可以被应用程序读取或获取。基本规则是应用程序可以按照自己的意愿以任意顺序执行此方法。这种方式非常灵活，可以实现应用程序的最终控制。但是应用程序必须使用特定的访问模式，以防需要以正确的顺序获取样本或者访问一组完整的连贯变化。

要连贯地或按顺序访问数据，必须正确设置 PRESENTATION QoS（在 2.2.3.6 中说明），应用程序必须使用符合下面描述的访问模式，否则应用程序虽然仍将访问数据，但不一定会获取到所有连贯的变化，也不会获取到顺序正确的变化。

有一种通用模式可以跨多个 ***DataReader*** 提供有序和连贯的访问。此模式适用于

PRESENTATION QoS 的任何设置。更简单的模式也可以用于 QoS 的特定设置，具体内容如下所述。

1.**将数据样本作为连贯集访问和（或）跨 *DataReader* 实体按顺序访问的一般模式**。当 PRESENTATION QoS 指定“***access_scope*** = GROUP”时，这种情况适用。

- 在通知 ***SubscriberListener*** 或启用类似的 ***StatusCondition*** 后，应用程序使用 ***Subscriber*** 的 ***begin_access*** 方法告知服务它将通过 ***Subscriber*** 访问数据。
- 然后调用 ***Subscriber*** 的 ***get_datareaders*** 方法以获取数据样本可用的 ***DataReader*** 对象列表。
- 在此之后，它会以返回对象列表的相同顺序调用每个 ***DataReader*** 的 ***read*** 或 ***take*** 方法，从而访问 ***DataReader*** 中的所有相关更改。
- 一旦它调用了所有读取者的 ***read*** 或 ***take*** 方法，就会调用 ***end_access***。

请注意，如果 PRESENTATION QoS 策略指定 ***ordered_access*** = TRUE，则 ***DataReader*** 列表可能会多次返回同一个 ***DataReader***。通过这种方式可以在不同 ***DataReader*** 对象的样本之间维护正确的样本顺序。

2.***DataReader* 实体之间不需要维持按序或连贯性的专用模式**。这种场景适用于 PRESENTATION QoS 策略的 ***access_scope*** 值不为 GROUP。

- 在这种情况下，应用程序不需要调用 ***begin_access*** 和 ***end_access***。调用了这两个方法不会出错，也不会产生任何影响。
- 应用程序通过每个 ***DataReader*** 的 ***read*** 或 ***take*** 方法按照自己的意愿以任意顺序来访问数据。
- 应用程序仍然可以调用 ***get_datareaders*** 来确定哪些读取者具有要读取的数据，但不需要取得所有的读取者，也不需要按特定顺序取得它们。此外，***get_datareaders*** 方法的返回值在逻辑上将是一个“集合”，即相同的读取者不会出现两次，而且未指定返回的读取者的顺序。

3.**应用程序访问 *SubscriberListener* 中数据的为专用模式**。当应用程序通过监听器的 ***on_data_on_readers*** 方法访问数据时，不需要考虑 PRESENTATION QoS 策略如何。

- 与前一种情况（上面的 2）类似，应用程序不需要调用 ***begin_access*** 和 ***end_access***，这样做无效。
- 应用程序可以按照其希望的任何顺序在每个 ***DataReader*** 上调用 ***read*** 或 ***take*** 方法来访问数据。
- 应用程序还可以通过调用 ***notify_datareaders*** 方法将数据访问委派给安装在每个 ***DataReader*** 上的 ***DataReaderListener*** 对象。
- 与前一种情况（上面的 2）类似，应用程序仍然可以调用 ***get_datareaders*** 来确定哪些读取器者有要读取的数据，但它不需要取得所有读取者，也不需要按特定顺序取得它们。此外，***get_datareaders*** 的返回值逻辑上是一个“集合”。

2.2.2.5.2 Subscriber 类

订阅者（***Subscriber***）是负责实际接收其订阅所得的数据的对象。

Subscriber

no attributes		
operations		
(inherited) get_qos		ReturnCode_t
	out: qos_list	QosPolicy []
(inherited) set_qos		ReturnCode_t
	qos_list	QosPolicy []
(inherited) get_listener		Listener
(inherited) set_listener		ReturnCode_t
	a_listener	Listener
	mask	StatusKind []
create_datareader		DataReader
	a_topic	TopicDescription
	qos	QosPolicy []
	a_listener	DataReaderListener
	mask	StatusKind []
delete_datareader		ReturnCode_t
	a_datareader	DataReader
lookup_datareader		DataReader
	topic_name	string
begin_access		ReturnCode_t
end_access		ReturnCode_t
get_datareaders		ReturnCode_t
	out: readers	DataReader []
	sample_states	SampleStateKind []
	view_states	ViewStateKind []
	instance_states	InstanceStateKind []
notify_datareaders		ReturnCode_t
get_participant		DomainParticipant
delete_contained_entities		ReturnCode_t
set_default_datareader_qos		ReturnCode_t
	qos_list	QosPolicy []
get_default_datareader_qos		ReturnCode_t
	out: qos_list	QosPolicy []
copy_from_topic_qos		ReturnCode_t
	inout: a_datareader_qos	QosPolicy []
	a_topic_qos	QosPolicy []

订阅者（Subscriber）代表与其相关的一个或多个***数据读取者（DataReader）***对象。 当它（从系统的其他部分）接收数据时，便会构建相关 ***DataReader*** 对象的列表，通过其监听器或启用相关条件向应用程序提示有可用数据。应用程序可以通过 ***get_datareaders*** 方法访问关联 ***DataReader*** 对象列表，然后通过 ***DataReader*** 提供的方法访问可用数据。

除基类操作 ***set_qos，get_qos，set_listener，get_listener，enable，get_statuscondition 和 create_datareader*** 之外的所有方法都可能返回 NOT_ENABLED。

2.2.2.5.2.1 set_listener (来自实体类 Entity)

通过扩展***实体（Entity）***类，***订阅者（Subscriber）***可以在创建时或创建后使用 ***set_listener*** 方法将其与***监听器（Listener）***进行绑定。附加的 ***Listener*** 必须继承自 ***SubscriberListener***.。监听器在 2.2.4 监听器，条件和等待集中描述。

2.2.2.5.2.2 get_listener (来自实体类 Entity)

访问绑定的 ***SubscriberListener***。

2.2.2.5.2.3 set_qos (来自实体类 Entity)

通过扩展***实体（Entity）***类，***订阅者（Subscriber）***可以在创建时或创建后使用 ***set_qos*** 方法设置 QoS 策略。关于可以在 ***Subscriber*** 上设置的 QoS 策略，请参阅 2.2.3 支持的 QoS。

除标准错误代码外，还可能返回错误代码：IMMUTABLE_POLICY，INCONSISTENT_POLICY。

2.2.2.5.2.4 get_qos (来自实体类 Entity)

此方法允许访问 QoS 策略的取值。

2.2.2.5.2.5 create_datareader

此方法将创建一个 ***DataReader***。返回的 ***DataReader*** 对象将附加到并属于此 ***Subscriber***。

create_datareader 方法返回的 ***DataReader*** 对象实际上是一个派生类，特定于与***主题（Topic）***关联的数据类型。如 2.2.2.3.7 所述，对于每个应用程序定义的数据类型“Foo”，都有一个隐含的，自动生成的类 ***FooDataReader***，它扩展了 ***DataReader*** 并包含写入“Foo”类型数据的方法。

如果方法调用失败，将返回“nil”值（由平台指定）。

请注意，为 ***DataReader*** 构建 QoS 的常见应用模式如下：

- 通过***主题（Topic）***对象的 ***get_qos*** 方法获取关联***主题（Topic）***的 QoS 策略。
- 通过 ***Subscriber*** 对象的 ***get_default_datareader_qos*** 方法获取默认的 ***DataReader*** QoS。
- 结合上述两种 QoS 策略，根据需要有选择地修改。
- 使用生成的 QoS 策略创建 ***DataReader***。

特殊值 DATAREADER_QOS_DEFAULT 表明应使用工厂中设置的默认 ***DataReader*** QoS 创建 ***DataReader***。使用这个特殊值等同于应用程序通过 ***get_default_datareader_qos***（2.2.2.5.2.16）方法获取默认 ***DataReader*** QoS 并使用获得的 QoS 创建 ***DataReader***。

如果 ***TopicDescription*** 参数传入的为 ***Topic*** 或者 ***ContentFilteredTopic*** 类型对象，特殊值 DATAREADER_QOS_USE_TOPIC_QOS 表明应使用默认 ***DataReader***QoS 和 ***Topic*** QoS 组合创建 ***DataReader***（在传入参数为 ***ContentFilteredTopic*** 类型的情况下，讨论的主题是 ***ContentFilteredTopic*** 的“相关主题”）。使用此值等同于应用程序获取默认 ***DataReader*** QoS 和 ***Topic*** QoS（通过方法 ***Topic :: get_qos***），然后使用 ***copy_from_topic_qos*** 方法组合这两个 QoS，从而使用 ***Topic*** QoS 中设置的 QoS 策略“覆盖”对应的默认 QoS 策略。最终将生成的 QoS 用于创建 ***DataReader***。在使用 ***MultiTopic*** 创建 ***DataReader*** 时使用 DATAREADER_QOS_USE_TOPIC_QOS 是错误的，方法将返回“nil”。

传递给方法的***主题描述（TopicDescription）***所在的 ***DomainParticipant*** 必须与创建此

Subscriber 的 ***DomainParticipant*** 相同。如果***主题描述（TopicDescription）***是在其他 ***DomainParticipant*** 中创建，则此方法将失败并返回 nil。

2.2.2.5.2.6 delete_datareader

此方法删除属于 ***Subscriber*** 的 ***DataReader***。如果 ***DataReader*** 不属于该 ***Subscriber***，此方法将返回错误 PRECONDITION_NOT_MET。

如果 ***DataReader*** 上还存在绑定的 ***ReadCondition*** 或 ***QueryCondition*** 对象，则不允许删除 ***DataReader***。在这种情况下调用 ***delete_datareader*** 将返回 PRECONDITION_NOT_MET。

如果应用程序调用 ***read***，***take*** 方法或其中任意一个方法的变体而未得到对应返回结果，则不允许删除 ***DataReader***。在这种情况下调用 ***delete_datareader*** 将返回 PRECONDITION_NOT_MET。

必须在创建 ***DataReader*** 的同一 ***Subscriber*** 对象上调用 ***delete_datareader*** 方法。如果在另一个 ***Subscriber*** 上调用 ***delete_datareader***，该方法将不起作用并返回 PRECONDITION_NOT_MET。

除标准错误代码外，还可能返回错误代码：PRECONDITION_NOT_MET。

2.2.2.5.2.7 lookup_datareader

此方法查找获取之前创建成功的属于***订阅者（Subscriber）***的 ***DataReader***，需要查找的 ***DataReader*** 主题为 ***topic_name***。如果不存在此类 ***DataReader***，则方法将返回“nil”。

如果 ***Subscriber*** 中存在多个 ***DataReader*** 满足此条件，则方法将返回其中一个，不指定具体是哪一个。

内置 ***Subscriber*** 对象使用此方法可以访问采用内置主题创建的内置 ***DataReader*** 实体。

2.2.2.5.2.8 begin_access

此方法表示应用程序即将访问附加到***订阅者（Subscriber）***上的任何 ***DataReader*** 对象中的数据样本。

仅当 ***DataReader*** 所属的***订阅者（Subscriber）***将 PRESENTATION QosPolicy ***access_scope*** 设置为“GROUP”时，才需要应用程序调用此方法。

在上述情况下，必须在调用任一数据样本访问方法之前调用 ***begin_access***，主要包括：**Subscriber** 的 ***get_datareaders*** 方法和任一 ***DataReader*** 的 ***read，take，read_w_condition，take_w_condition*** 方法。否则样本访问方法将返回错误 PRECONDITION_NOT_MET。应用程序完成访问数据样本后，必须调用 ***end_access***。

如果 PRESENTATION QosPolicy 将 ***access_scope*** 设置为“GROUP”以外的其他内容，则应用程序不需要调用 ***begin_access/end_access***。在这种情况下调用 ***begin_access/ end_access*** 不会被视为错误且无效。

对 ***begin_access/end_access*** 的调用可能是嵌套的。在这种情况下，应用程序调用 ***end_access*** 的次数必须与调用 ***begin_access*** 的次数保持一致。

除标准错误代码外，还可能返回错误代码：PRECONDITION_NOT_MET。

2.2.2.5.2.9 end_access

此方法表示应用程序已完成访问 ***Subscriber*** 管理的 ***DataReader*** 对象中的数据样本。

必须使用此方法来“关闭”对应的 ***begin_access***。

在调用 ***end_access*** 之后，应用程序应不再访问通过样本访问方法返回的任何 ***Data*** 或

SampleInfo 元素。此调用必须关闭之前对 ***begin_access*** 的调用，否则将返回错误 PRECONDITION_NOT_MET。

除标准错误代码外，还可能返回错误代码：PRECONDITION_NOT_MET。

2.2.2.5.2.10 get_datareaders

此方法允许应用程序访问包含具有指定 ***sample_states***，***view_states*** 和 ***instance_states*** 的样本的 ***DataReader*** 对象。

如果 ***DataReader*** 所属的 ***Subscriber*** 将 PRESENTATION QoS 策略 ***access_scope*** 设置为“GROUP”，此方法只应在 ***begin_access/end_access*** 块内调用，否则它将返回错误 PRECONDITION_NOT_MET。

根据 PRESENTATION QoS 策略的设置（参见 2.2.3.6），返回的 ***DataReader*** 对象集可能是一个“集合”，其中每个 ***DataReader*** 最多出现一次，而且没有指定的顺序；或者是一个“列表”，其中每个 ***DataReader*** 会出现一次或者多次，而且必须按指定顺序排列。

1.如果 PRESENTATION ***access_scope*** 是 INSTANCE 或 TOPIC，则返回一个“集合”。

2.如果 PRESENTATION ***access_scope*** 为 GROUP 且 ***ordered_access*** 设置为 TRUE，则返回一个“列表”。

在第二种情况下需要以特定顺序访问属于不同 ***DataReader*** 对象的样本，最终导致这种差异。在这种情况下，应用程序应按照每个 ***DataReader*** 在“列表”中显示的顺序来按序处理，并且每次只从 ***DataReader*** 中读取或获取一个样本。应用程序用于访问数据的模式在 2.2.2.5.1 访问数据中有完整描述。

2.2.2.5.2.11 notify_datareaders

此方法调用绑定到 ***DataReader*** 实体的 ***DataReaderListener*** 对象的 ***on_data_available*** 方法，该实体的 DATA_AVAILABLE 状态被视为已更改，如 2.2.4.2.2 读取通信状态的更改所述。

通常在 ***SubscriberListener*** 的 ***on_data_on_readers*** 方法中调用此方法。这样一来 ***SubscriberListener*** 可以委托 ***DataReaderListener*** 对象处理数据。

2.2.2.5.2.12 get_sample_lost_status

此方法允许访问 SAMPLE_LOST 通信状态。通信状态在 2.2.4.1 中描述。

2.2.2.5.2.13 get_participant

此方法返回***订阅者（Subscriber）***所属的 ***DomainParticipant***。

2.2.2.5.2.14 delete_contained_entities

此方法将删除通过 ***Subscriber*** 对象的“创建”方法创建的所有实体。也就是说，它会删除所有包含的 ***DataReader*** 对象。该模式会被递归地应用。这种模式下 ***Subscriber*** 的 ***delete_contained_entities*** 方法将最终递归地删除 ***Subscriber*** 中包含的所有实体，包括 ***DataReader*** 的 ***QueryCondition*** 和 ***ReadCondition*** 对象。

如果任一包含的实体处于无法删除的状态，则方法将返回 PRECONDITION_NOT_MET。例如以下这种情况，应用程序已调用 ***read*** 或 ***take*** 方法，并且尚未调用相应的 ***return_loan*** 方

法来返回已借出的样本，则无法删除所包含的 ***DataReader***，删除失败。

一旦 ***delete_contained_entities*** 成功返回，应用程序可能会删除 ***订阅者 Subscriber***，因为应用程序知道它没有包含 ***DataReader*** 对象。

2.2.2.5.2.15 set_default_datareader_qos

此方法设置 ***DataReader*** QoS 策略的默认值。在 ***create_datareader*** 方法使用默认 QoS 策略的情况下，该策略值将用于新创建的 ***DataReader*** 实体。

此方法将检查生成的策略是否是自相容的；如果不是，则方法调用无效并返回 INCONSISTENT_POLICY。

可以将特殊值 DATAREADER_QOS_DEFAULT 传递给此方法，表明将默认 QoS 重置为工厂使用的初始值，即从未调用 ***set_default_datareader_qos*** 方法时使用的值。

2.2.2.5.2.16 get_default_datareader_qos

此方法获取 ***DataReader*** QoS 的默认值，即在 ***create_datareader*** 方法使用默认 QoS 策略的情况下用于新创建的 ***DataReader*** 实体的 QoS 策略。

get_default_datareader_qos 获取的值与上次成功调用 ***set_default_datareader_qos*** 时指定的值集保持一致；如果从未调用过 ***set_default_datareader_qos***，则在 2.2.3 支持的 QoS 中的 QoS 表中列出了 ***DataReader*** QoS 默认值。

2.2.2.5.2.17 copy_from_topic_qos

此方法将 *a_topic_qos* 中的策略复制到 *a_datareader_qos* 中的相应策略（即替换 *a_datareader_qos* 中的值，如果该位置存在值）。

这是一个“方便”的方法，与 ***get_default_datareader_qos*** 和 ***Topic :: get_qos*** 方法结合使用最为有用。***copy_from_topic_qos*** 方法可用于将 ***DataReader*** 默认 QoS 策略与 ***Topic*** 的相应 QoS 策略合并。可以使用生成的 QoS 创建新的 ***DataReader***，或设置其 QoS。

此方法不会检查生成的 *a_datareader_qos* 的一致性。这是因为“合并的”*a_datareader_qos* 可能不是最后一个，因为应用程序仍然可以在将策略应用于 ***DataReader*** 之前修改某些策略。

2.2.2.5.3 DataReader 类

DataReader 允许应用程序（1）声明它希望接收的数据（即进行订阅），以及（2）访问由绑定的 ***订阅者 Subscriber*** 接收的数据。

DataReader		
no attributes		
operations		
(inherited) get_qos		ReturnCode_t
	out: qos_list	QosPolicy []
(inherited) set_qos		ReturnCode_t
	qos_list	QosPolicy []
(inherited) get_listener		Listener
(inherited) set_listener		ReturnCode_t
	a_listener	Listener

	mask	StatusKind []
read		ReturnCode_t
	inout: data_values	Data []
	inout: sample_infos	SampleInfo []
	max_samples	long
	sample_states	SampleStateKind []
	view_states	ViewStateKind []
	instance_states	InstanceStateKind []
take		ReturnCode_t
	inout: data_values	Data []
	inout: sample_infos	SampleInfo []
	max_samples	long
	sample_states	SampleStateKind []
	view_states	ViewStateKind []
	instance_states	InstanceStateKind []
read_w_condition		ReturnCode_t
	inout: data_values	Data []
	inout: sample_infos	SampleInfo []
	max_samples	long
	a_condition	ReadCondition
take_w_condition		ReturnCode_t
	inout: data_values	Data []
	inout: sample_infos	SampleInfo []
	max_samples	long
	a_condition	ReadCondition
read_next_sample		ReturnCode_t
	inout: data_value	Data
	inout: sample_info	SampleInfo
take_next_sample		ReturnCode_t
	inout: data_value	Data
	inout: sample_info	SampleInfo
read_instance		ReturnCode_t
	inout: data_values	Data []
	inout: sample_infos	SampleInfo []
	max_samples	long
	a_handle	InstanceHandle_t
	sample_states	SampleStateKind []
	view_states	ViewStateKind []
	instance_states	InstanceStateKind []
take_instance		ReturnCode_t
	inout: data_values	Data []
	inout: sample_infos	SampleInfo []
	max_samples	long

	a_handle	InstanceHandle_t
	sample_states	SampleStateKind []
	view_states	ViewStateKind []
	instance_states	InstanceStateKind []
read_next_instance		ReturnCode_t
	inout: data_values	Data []
	inout: sample_infos	SampleInfo []
	max_samples	long
	previous_handle	InstanceHandle_t
	sample_states	SampleStateKind []
	view_states	ViewStateKind []
	instance_states	InstanceStateKind []
take_next_instance		ReturnCode_t
	inout: data_values	Data []
	inout: sample_infos	SampleInfo []
	max_samples	long
	previous_handle	InstanceHandle_t
	sample_states	SampleStateKind []
	view_states	ViewStateKind []
	instance_states	InstanceStateKind []
read_next_instance_w_condition		ReturnCode_t
	inout: data_values	Data []
	inout: sample_infos	SampleInfo []
	max_samples	long
	previous_handle	InstanceHandle_t
	a_condition	ReadCondition
take_next_instance_w_condition		ReturnCode_t
	inout: data_values	Data []
	inout: sample_infos	SampleInfo []
	max_samples	long
	previous_handle	InstanceHandle_t
	a_condition	ReadCondition
return_loan		ReturnCode_t
	inout: data_values	Data []
	inout: sample_infos	SampleInfo []
get_key_value		ReturnCode_t
	inout: key_holder	Data
	handle	InstanceHandle_t
lookup_instance		InstanceHandle_t
	instance	Data
create_readcondition		ReadCondition
	sample_states	SampleStateKind []
	view_states	ViewStateKind []

	instance_states	InstanceStateKind []
create_querycondition		QueryCondition
	sample_states	SampleStateKind []
	view_states	ViewStateKind []
	instance_states	InstanceStateKind []
	query_expression	string
	query_parameters	string []
delete_readcondition		ReturnCode_t
	a_condition	ReadCondition
get_liveliness_changed_status		ReturnCode_t
	out: status	LivelinessChangedStatus
get_requested_deadline_missed_status		ReturnCode_t
	out: status	RequestedDeadlineMissedStatus
get_requested_incompatible_qos_status		ReturnCode_t
	out: status	RequestedIncompatibleQosStatus
get_sample_lost_status		ReturnCode_t
	out: status	SampleLostStatus
get_sample_rejected_status		ReturnCode_t
	out: status	SampleRejectedStatus
get_subscription_matched_status		ReturnCode_t
	out: status	SubscriptionMatchedStatus
get_topicdescription		TopicDescription
get_subscriber		Subscriber
delete_contained_entities		ReturnCode_t
wait_for_historical_data		ReturnCode_t
	max_wait	Duration_t
get_matched_publication_data		ReturnCode_t
	inout: publication_data	PublicationBuiltinTopicData
	publication_handle	InstanceHandle_t
get_matched_publications		ReturnCode_t
	inout: publication_handles	InstanceHandle_t []

DataReader 仅参考 ***TopicDescription***（一个 ***Topic***，***ContentFilteredTopic*** 或 ***MultiTopic***）来识别接收的数据类型。订阅具有唯一的结果类型。数据读取者可以访问结果类型的多个实例，这些实例可以通过他们的***关键字 key*** 彼此区分（如 2.2.1.2.2 整体概念模型中所述）。

DataReader 是一个抽象类。它必须专门用于特定的应用数据类型，如图 2.8 所示。在自动生成的类中必须为假设的应用类型“Foo”定义的其他方法如下表所示：

FooDataReader
no attributes
operations

read		ReturnCode_t
	inout: data_values	Foo []
	inout: sample_infos	SampleInfo []
	max_samples	long
	sample_states	SampleStateKind []
	view_states	ViewStateKind []
	instance_states	InstanceStateKind []
take		ReturnCode_t
	inout: data_values	Foo []
	inout: sample_infos	SampleInfo []
	max_samples	long
	sample_states	SampleStateKind []
	view_states	ViewStateKind []
	instance_states	InstanceStateKind []
read_w_condition		ReturnCode_t
	inout: data_values	Foo []
	inout: sample_infos	SampleInfo []
	max_samples	long
	a_condition	ReadCondition
take_w_condition		ReturnCode_t
	inout: data_values	Foo []
	inout: sample_infos	SampleInfo []
	max_samples	long
	a_condition	ReadCondition
read_next_sample		ReturnCode_t
	inout: data_value	Foo
	inout: sample_info	SampleInfo
take_next_sample		ReturnCode_t
	inout: data_value	Foo
	inout: sample_info	SampleInfo
read_instance		ReturnCode_t
	inout: data_values	Foo []
	inout: sample_infos	SampleInfo []
	max_samples	long
	a_handle	InstanceHandle_t
	sample_states	SampleStateKind []
	view_states	ViewStateKind []
	instance_states	InstanceStateKind []
take_instance		ReturnCode_t
	inout: data_values	Foo []
	inout: sample_infos	SampleInfo []
	max_samples	long
	a_handle	InstanceHandle_t

	sample_states	SampleStateKind []
	view_states	ViewStateKind []
	instance_states	InstanceStateKind []
read_next_instance		ReturnCode_t
	inout: data_values	Foo []
	inout: sample_infos	SampleInfo []
	max_samples	long
	previous_handle	InstanceHandle_t
	sample_states	SampleStateKind []
	view_states	ViewStateKind []
	instance_states	InstanceStateKind []
take_next_instance		ReturnCode_t
	inout: data_values	Foo []
	inout: sample_infos	SampleInfo []
	max_samples	long
	previous_handle	InstanceHandle_t
	sample_states	SampleStateKind []
	view_states	ViewStateKind []
	instance_states	InstanceStateKind []
read_next_instance_w_condition		ReturnCode_t
	inout: data_values	Foo []
	inout: sample_infos	SampleInfo []
	max_samples	long
	previous_handle	InstanceHandle_t
	a_condition	ReadCondition
take_next_instance_w_condition		ReturnCode_t
	inout: data_values	Foo []
	inout: sample_infos	SampleInfo []
	max_samples	long
	previous_handle	InstanceHandle_t
	a_condition	ReadCondition
return_loan		ReturnCode_t
	inout: data_values	Foo []
	inout: sample_infos	SampleInfo []
get_key_value		ReturnCode_t
	inout: key_holder	Foo
	handle	InstanceHandle_t
lookup_instance		InstanceHandle_t
	instance	Foo

除基类方法 ***set_qos***，***get_qos***，***set_listener***，***get_listener***，***enable*** 和 ***get_statuscondition*** 之外的所有方法都可能返回错误 NOT_ENABLED。

所有样本访问方法，即 ***read*** 或 ***take*** 方法的所有变体都可能会返回错误

PRECONDITION_NOT_MET。由此产生的情况在 2.2.2.5.2.8 中描述。

以下子条目提供了有关方法的详细信息。

2.2.2.5.3.1 set_listener (来自实体类 Entity)

通过扩展***实体（Entity）***类，***数据读取者（DataReader）***可以在创建时或创建后使用 ***set_listener*** 方法将其与***监听器（Listener）***进行绑定。附加的 ***Listener*** 必须继承自 ***DataReaderListener***。监听器在 2.2.4 监听器，条件和等待集中描述。

2.2.2.5.3.2 get_listener (来自实体类 Entity)

访问绑定的 ***DataReaderListener***。

2.2.2.5.3.3 set_qos (来自实体类 Entity)

通过扩展***实体(Entity)***类，***数据读取者(DataReader)***可以在创建时或创建后使用 ***set_qos*** 方法设置 QoS 策略。关于可以在 ***DataReader*** 上设置的 QoS 策略，请参阅 2.2.3 支持的 QoS。

除标准错误代码外，还可能返回错误代码：IMMUTABLE_POLICY，INCONSISTENT_POLICY。

2.2.2.5.3.4 get_qos (来自实体类 Entity)

此方法允许访问 QoS 策略的取值。

2.2.2.5.3.5 create_readcondition

此方法创建一个 ***ReadCondition***，返回的 ***ReadCondition*** 将被绑定到并属于 ***DataReader***。

如果方法调用失败，将返回“nil”值（由平台指定）。

2.2.2.5.3.6 create_querycondition

此方法创建一个 ***QueryCondition***，返回的 ***QueryCondition*** 将被绑定到并属于 ***DataReader***。

query_expression 和 ***query_parameters*** 参数的语法在附录 B 中描述。

如果方法调用失败，将返回“nil”值（由平台指定）。

2.2.2.5.3.7 delete_readcondition

此方法删除附加到 ***DataReader*** 的 ***ReadCondition***。由于 ***QueryCondition*** 是一种特殊的 ***ReadCondition***，因此此方法也可用于删除 ***QueryCondition***。如果被删除的 ***ReadCondition*** 未附加到 ***DataReader***，方法调用将返回错误 PRECONDITION_NOT_MET。

除标准错误代码外，还可能返回错误代码：PRECONDITION_NOT_MET。

2.2.2.5.3.8 read

此方法通过 ***DataReader*** 访问 ***Data*** 值的合集。返回合集的大小不超过指定的 *max_samples*。*data_values* 合集的属性和 PRESENTATION QoS 策略的设置（见 2.2.3.6）可能会对返回的“列表”的大小增加进一步的限制。

1.如果 PRESENTATION QoS 的 ***access_scope*** 是 INSTANCE，则返回的合集是“列表”，其中属于同一数据实例的样本是连续的。

2.如果 PRESENTATION QoS 的 ***access_scope*** 是 TOPIC 且 ***ordered_access*** 设置为 FALSE，则返回的合集是“列表”，其中属于同一数据实例的样本是连续的。

3.如果 PRESENTATION QoS 的 ***access_scope*** 是 TOPIC 且 ***ordered_access*** 设置为 TRUE，则返回的合集是“列表”，属于同一实例的样本可能连续也可能不连续。这是因为为了保持顺序，可能需要混合来自不同实例的样本。

4.如果 PRESENTATION QoS 的 ***access_scope*** 是 GROUP 且 ***ordered_access*** 设置为 FALSE，则返回的合集为“列表”，其中属于同一数据实例的样本是连续的。

5.如果 PRESENTATION QoS 的 ***access_scope*** 是 GROUP 且 ***ordered_access*** 设置为 TRUE，则返回的合集最多包含一个样本。这种情况不一样是因为需要应用程序以指定顺序读取属于不同 ***DataReader*** 对象的样本。

在任何情况下，一个实例的样本之间的相对顺序与 DESTINATION_ORDER QosPolicy 保持一致：

•如果 DESTINATION_ORDER 为 BY_RECEPTION_TIMESTAMP，则属于相同实例的样本将按接收的相对顺序出现（FIFO，较早样本在后面的样本之前）。

•如果 DESTINATION_ORDER 是 BY_SOURCE_TIMESTAMP，则属于相同实例的样本将以 ***source_timestamp*** 隐含的相对顺序出现（FIFO， source_timestamp 较小值排在较大值之前）。

除了样本合集之外，***read*** 方法还使用了 ***SampleInfo*** 结构的合集（*sample_infos*），参见 2.2.2.5.5 SampleInfo 类。

data_values 和 *sample_infos* 合集的初始（输入）属性将确定 ***read*** 方法的精确行为。为了更好的解释此方法描述，上述合集被建模为具有三个属性：当前长度（*len*），最大长度（*max_len*），以及合集容器是否拥有（*owns*）内部元素的内存操作权限。不提供这些功能的平台特定模型（PSM）映射可能需要更改 ***read*** 方法的签名以补偿它。

data_values 和 *sample_infos* 合集的 *len*，*max_len* 和 *owns* 属性的初始（输入）值控制 ***read*** 方法的行为，详见以下规则：

1.两个合集的 *len*，*max_len* 和 *owns* 的值必须相同，否则 ***read*** 方法将返回 PRECONDITION_NOT_MET。

2.成功输出时，*len*，*max_len* 和 *owns* 的值对于两个合集都是相同的。

3.如果输入的 *max_len* = = 0，则 *data_values* 和 *sample_infos* 合集将由 ***DataReader*** “借出”的元素填充。输出时，*owns* 将为 FALSE，*len* 将设置为返回值的数量，*max_len* 将设置为任一满足 *max_len*> = *len* 的值。使用此变体允许对数据进行零拷贝访问，并且应用程序需要使用 ***return_loan*** 方法将“贷款返还”给 ***DataReader***（参见 2.2.2.5.3.20）。

4.如果输入的 *max_len*> 0 且输入的 *owns*= = FALSE，则 ***read*** 方法将失败并返回 PRECONDITION_NOT_MET。这避免了由于应用程序忘记“退还贷款”而导致的潜在难以检测的内存泄漏。

5.如果输入的 *max_len*> 0 且输入的 *owns* = = TRUE，则 ***read*** 方法会将 ***Data*** 值和 ***SampleInfo*** 值复制到合集内部的元素中。在输出时，*owns* 将为 TRUE，*len* 将设置为复制的值的数量，*max_len* 将保持不变。使用此变体会强制进行复制，但应用程序可以控制副本的放置位置，并且应用程序不需要“返回贷款”。复制的样本数取决于 *max_len* 和 *max_samples* 的相对值：

•如果 *max_samples* = LENGTH_UNLIMITED，则最多将复制 *max_len* 条样本。

使用此变体可以使应用程序限制返回的样本数，确保样本数在序列可容纳的范围之内。

•如果 *max_samples* <= max_len，则最多将复制 max_samples 条样本。使用此变体可以使应用程序限制返回的样本数，确保样本数在序列可容纳的范围之内。

• 如果 *max_samples*> max_len，则 ***read*** 方法将失败并返回 PRECONDITION_NOT_MET。这避免了一种潜在的混淆场景，即应用程序期望能够访问 *max_samples* 条（即使它们在 ***DataReader*** 中是可用的）数据，但是本方法永远不可能正确返回，因为输出序列无法容纳这么多条数据。

如上所述，返回时 *data_values* 和 *sample_infos* 合集可能包含从 ***DataReader***“借出”的元素。如果是这种情况，应用程序在不再使用合集中的数据时需要调用 ***return_loan*** 方法（请参阅 2.2.2.5.3.20）返回“贷款”。***return_loan*** 返回后，该合集的 *max_len* 为 0 且 *owns* 为 FALSE。

应用程序可以根据调用 ***read*** 方法时的合集状态或访问“*owns*”属性来确定是否有必要“返还贷款”。在大多数情况下，总是调用 ***return_loan*** 可能更简单一些，因为如果合集没有贷款，此操作是无害的（即保持所有元素不变）。

为避免潜在的内存泄漏，***Data*** 和 ***SampleInfo*** 合集的实现应禁止更改 *owns*==FALSE 的合集的长度。此外，删除 *owns* == FALSE 的合集是一种错误调用。

在输出时，***Data*** 值的合集和 SampleInfo 结构的合集具有相同的长度并且一一对应。每个 ***SampleInfo*** 都提供对应样本的相关信息，例如 ***source_timestamp***，***sample_state***，***view_state*** 和 ***instance_state*** 等。

返回合集中的某些元素可能没有有效数据。如果 ***SampleInfo*** 中的 ***instance_state*** 为 NOT_ALIVE_DISPOSED 或 NOT_ALIVE_NO_WRITERS，则合集中该实例的最后一个样本，即 sample_rank == 0 的 ***SampleInfo*** 样本不包含有效数据。不包含数据的样本不受 RESOURCE_LIMITS QoS 策略的限制。

读取样本的行为将其 ***sample_state*** 设置为 READ。如果样本属于最新一代实例，还会将实例的 ***view_state*** 设置为 NOT_NEW。它不会影响实例的 ***instance_state***。

特定应用程序的将要写入数据的数据类型生成的专用类上必须提供此方法实现。

如果 ***DataReader*** 没有满足约束的数据样本，则返回值为 NO_DATA。

2.2.2.5.3.9 take

此方法访问 ***DataReader*** 中的数据样本合集以及相应的 ***SampleInfo*** 结构合集。该方法将返回样本的“列表”或单个样本，这由 PRESENTATION QoS 策略控制，使用与 ***read*** 方法相同的逻辑（见 2.2.2.5.3.8）。

获取样本的行为将样本从 DataReader 中删除，因此该样本无法再次被“读取”或“获取”。如果样本属于最新一代实例，还会将实例的 ***view_state*** 设置为 NOT_NEW，并不会影响实例的 ***instance_state***。

take 方法的行为遵循与 ***read*** 方法相同的规则，该规则是关于 *data_values* 和 *sample_infos* 合集的前置条件和后置条件的。与 ***read*** 方法类似，***take*** 方法可以“贷出”元素到输出合集，然后必须通过 ***return_loan*** 归还。与 ***read*** 方法的唯一区别在于，***take*** 方法归还的样本将不再用于后续 ***read*** 或 ***take*** 调用。

与 ***read*** 方法类似，特定应用程序的将要写入数据的数据类型生成的专用类上必须提供此方法实现。

如果 ***DataReader*** 没有满足约束的数据样本，则返回值为 NO_DATA。

2.2.2.5.3.10 read_w_condition

此方法通过“读取”访问与 ***ReadCondition*** 中指定的条件匹配的样本。此方法与 ***QueryCondition*** 结合使用，可根据内容过滤数据样本。

必须将指定的 ***ReadCondition*** 附加到 ***DataReader***，否则方法调用将失败并返回 PRECONDITION_NOT_MET。

如果 ***ReadCondition*** 是一个“普通”的 ***ReadCondition*** 而不是专门的 ***QueryCondition***，则该方法等同于调用 ***read*** 并将 ***read_condition*** 中相应属性的值当参数传递给 ***sample_states***，***view_states*** 和 ***instance_states***。应用程序使用此方法可以避免创建 ***ReadCondition*** 时指定重复的参数。

访问样本的语义与 ***read*** 方法相同。

与 ***read*** 方法类似，特定应用程序的将要写入数据的数据类型生成的专用类上必须提供此方法实现。

如果 ***DataReader*** 没有满足约束的数据样本，则返回值为 NO_DATA。

2.2.2.5.3.11 take_w_condition

此方法类似于 ***read_w_condition***，除了它通过 ***take*** 方法访问样本。

必须将指定的 ***ReadCondition*** 附加到 DataReader，否则方法调用将失败并返回 PRECONDITION_NOT_MET。

访问样本的语义与 ***take*** 方法相同。

此操作与 QueryCondition 结合使用，可根据内容过滤数据样本。

与 ***take*** 方法类似，特定应用程序的将要写入数据的数据类型生成的专用类上必须提供此方法实现。

如果 ***DataReader*** 没有满足约束的数据样本，则返回值为 NO_DATA。

2.2.2.5.3.12 read_next_sample

此方法从 ***DataReader*** 复制下一个之前未访问过的 ***Data*** 值，同时复制相应的 ***SampleInfo***。存储在 ***DataReader*** 中的样本之间的隐含顺序与 ***read*** 方法（2.2.2.5.3.8）相同。

read_next_sample 方法在语义上等效于 ***read*** 方法，其中输入的 ***Data*** 序列具有 *max_len* = 1，*sample_states* = NOT_READ，*view_states* = ANY_VIEW_STATE，以及 *instance_states* = ANY_INSTANCE_STATE 等特点。

read_next_sample 方法提供了一个简化的 API 来“读取”样本，从而避免了应用程序管理序列和指定状态的需要。

如果 ***DataReader*** 中没有未读数据，方法将返回 NO_DATA 并且不会复制任何内容。

2.2.2.5.3.13 take_next_sample

此方法从 ***DataReader*** 复制下一个之前未访问过的 ***Data*** 值，并从 ***DataReader*** 中“删除”它，使其不再可访问，同时复制相应的 ***SampleInfo***。此方法类似于 ***read_next_sample***，除了此方法会从 ***DataReader*** 中“删除”样本。

t***ake_next_sample*** 方法在语义上等同于 ***take*** 操作，其中输入序列具有 *max_len* = 1，*sample_states* = NOT_READ，*view_states* = ANY_VIEW_STATE，以及 *instance_states* = ANY_INSTANCE_STATE 等特点。

此方法提供了一个简化的 API 来“获取”样本，从而避免了应用程序管理序列和指定状态的需要。

如果 ***DataReader*** 中没有未读数据，方法将返回 NO_DATA 并且不会复制任何内容。

2.2.2.5.3.14 read_instance

此方法从 ***DataReader*** 访问 ***Data*** 值的合集。行为与 ***read*** 相同，只是返回的所有样本都属于句柄为 *a_handle* 的实例。

成功返回后，***Data*** 合集将包含属于同一实例的样本。相应的 ***SampleInfo*** 确认 ***instance_handle*** == *a_handle*。

此方法语义与 ***read*** 方法相同，除了在构建合集时，***DataReader*** 将检查样本是否属于指定的实例，如果不属于则不会将样本放在返回的合集中。

read_instance 方法遵循与 ***read*** 方法相同的规则，该规则是关于 *data_values* 和 *sample_infos* 合集的前置条件和后置条件的。与 ***read*** 方法类似，***read_instance*** 方法可以将“贷出”元素“贷款”到输出合集，然后通过 ***return_loan*** 方法归还。

与 ***read*** 方法类似，特定应用程序的将要写入数据的数据类型生成的专用类上必须提供此方法实现。

如果 ***DataReader*** 没有满足约束的样本，则返回值为 NO_DATA。

如果 ***InstanceHandle_t*** *a_handle* 与 ***DataReader*** 已知的现有数据对象不对应，则此方法可能返回 BAD_PARAMETER。如果实现无法检查句柄是否有效，则未指定此情况下的结果。

2.2.2.5.3.15 take_instance

此方法从 ***DataReader*** 访问 ***Data*** 值的合集，行为与 ***take*** 相同，只是返回的所有样本都属于句柄为 a_handle 的实例。

此方法语义与 ***take*** 方法的语义相同，除了在构建集合时，***DataReader*** 将检查样本是否属于指定的实例，如果不属于则不会将样本放在返回的合集中。

take_instance 方法遵循与 ***read*** 方法相同的规则，该规则是关于 *data_values* 和 *sample_infos* 合集的前置条件和后置条件的。与 ***read*** 方法类似，***take_instance*** 方法可以将“贷出”元素“贷款”到输出合集，然后通过 ***return_loan*** 归还。

与 ***read*** 方法类似，特定应用程序的将要写入数据的数据类型生成的专用类上必须提供此方法实现。

如果 ***DataReader*** 没有满足约束的样本，则返回值为 NO_DATA。

如果 ***InstanceHandle_t*** *a_handle* 与 ***DataReader*** 已知的现有数据对象不对应，则此方法可能返回 BAD_PARAMETER。如果实现无法检查句柄是否有效，则未指定此情况下的结果。

2.2.2.5.3.16 read_next_instance

此方法从 ***DataReader*** 访问一组 Data 值，其中所有样本都属于单个实例。行为类似于 ***read_instance***，但不直接指定实际实例。相反，样本将全部属于“下一个”实例，其 ***instance_handle*** 比指定的具有可用样本的 *previous_handle* 更大。

此方法意味着实例句柄之间存在“大于”的顺序关系。这种关系的细节并不都重要的，并且是特定于实现的。重要的是根据中间件的实现，所有实例都是相对于彼此排序的。这种排序是在实例句柄之间进行的，不应该依赖于实例的状态（例如它是否有数据），并且与 ***DataReader*** 当前管理的实例不对应的实例句柄也必须参与到排序中。出于排序的目的，每个实例句柄都应该“好像”表示为唯一整数。

read_next_instance 表现得“好像”***DataReader*** 使用最小的 ***instance_handle*** 调用 ***read_instance***，其中 ***instance_handle*** 满足（a）大于 ***previous_handle***，（b）有可用的样本（即

满足指定状态强加的约束的样本）。

特殊值 HANDLE_NIL 保证“小于”任何有效的 ***instance_handle***。因此，使用参数值 *previous_handle* == HANDLE_NIL 将返回包含可用样本的所有实例中具有最小 ***instance_handle*** 的实例的样本。

read_next_instance 方法旨在用于应用驱动的迭代，其中应用通过传递 *previous_handle* = = HANDLE_NIL 开始，检查返回的样本，然后使用 ***SampleInfo*** 中返回的 ***instance_handle*** 作为下一次 ***previous_handle*** 参数的值来继续调用 ***read_next_instance***。迭代继续，直到 ***read_next_instance*** 返回值 NO_DATA。

请注意，可以使用与 ***DataReader*** 当前管理的实例不对应的 *previous_handle* 调用 ***read_next_instance*** 方法。如前所述，即使是不由 ***DataReader*** 管理的句柄，也会定义“大于”关系。可能发生这种情况的一种场景是，当应用程序迭代所有实例时，获取到 NOT_ALIVE_NO_WRITERS 实例的所有样本，归还贷款（此时实例信息可能被删除，因此句柄变为无效）后尝试阅读下一个实例。

read_next_instance 方法遵循与 ***read*** 方法相同的规则，该规则是关于 *data_values* 和 *sample_infos* 合集的前置条件和后置条件的。与 ***read*** 方法类似，***read_next_instance*** 方法可以将“贷款”元素“贷款”到输出集合，然后通过 ***return_loan*** 归还。

与 ***read*** 方法类似，特定应用程序的将要写入数据的数据类型生成的专用类上必须提供此方法实现。

如果 ***DataReader*** 没有满足约束的样本，则返回值为 NO_DATA。

2.2.2.5.3.17 take_next_instance

此方法从 ***DataReader*** 访问 ***Data*** 值合集，并从 ***DataReader*** 中“删除”它们。

此方法与 ***read_next_instance*** 具有相同的行为，除了从 ***DataReader*** “获取”样本，使得它们不可再通过后续的 ***read*** 或 ***take*** 方法访问。

与 ***read_next_instance***（参见 2.2.2.5.3.16）方法类似，可以使用 *previous_handle* 调用 ***take_next_instance***，该 *previous_handle* 可以与 ***DataReader*** 当前管理的实例不对应。

take_next_instance 方法遵循与 ***read*** 方法相同的规则，该规则是关于 *data_values* 和 *sample_infos* 合集的前置条件和后置条件的。与 ***read*** 方法类似，***take_next_instance*** 方法可以将“贷出”元素“贷款”到输出集合，然后通过 ***return_loan*** 归还。

与 ***read*** 方法类似，特定应用程序的将要写入数据的数据类型生成的专用类上必须提供此方法实现。

如果 ***DataReader*** 没有满足约束的样本，则返回值为 NO_DATA。

2.2.2.5.3.18 read_next_instance_w_condition

此方法从 ***DataReader*** 访问 ***Data*** 值的集合，行为与 ***read_next_instance*** 相同，只是返回的所有样本都必须满足指定的条件。也就是调用成功时，所有返回的样本都属于同一个实例，并且是所有实例中 instance_handle 值“最小”的即（a）***instance_handle*** > = *previous_handle* 和（b）样本的 ***ReadCondition*** 评估为真。

与 ***read_next_instance*** 方法类似（参见 2.2.2.5.3.16），可以使用与 DataReader 当前管理的实例不对应的 *previous_handle* 参数调用 ***read_next_instance_w_condition***。

read_next_instance_w_condition 方法遵循与 ***read*** 方法相同的规则，该规则是关于 *data_values* 和 *sample_infos* 合集的前置条件和后置条件的。与 ***read*** 方法类似，***read_next_instance_w_condition*** 方法可以“贷出”元素到输出集合，然后通过 ***return_loan*** 归还。

与 ***read*** 方法类似，特定应用程序的将要写入数据的数据类型生成的专用类上必须提供此方法实现。

如果 ***DataReader*** 没有满足约束的样本，则返回值为 NO_DATA。

2.2.2.5.3.19 take_next_instance_w_condition

此方法从 ***DataReader*** 访问 ***Data*** 值合集，并从 ***DataReader*** 中“删除”它们。

此方法与 ***read_next_instance_w_condition*** 具有相同的行为，除了从 ***DataReader***“获取”样本，使得它们不可再通过后续的 ***read*** 或 ***take*** 方法访问。

与 ***read_next_instance*** 方法（参见 2.2.2.5.3.16）类似，可以使用参数 *previous_handle* 调用 ***take_next_instance_w_condition***，该 *previous_handle* 可以与 ***DataReader*** 当前管理的实例不对应。

take_next_instance_w_condition 方法遵循与 ***read*** 方法相同的规则，该规则是关于 *data_values* 和 *sample_infos* 合集的前置条件和后置条件的。与 ***read*** 方法类似，***take_next_instance_w_condition*** 方法可以“贷款”元素到输出集合，然后通过 ***return_loan*** 归还。

与 ***read*** 方法类似，特定应用程序的将要写入数据的数据类型生成的专用类上必须提供此方法实现。

如果 ***DataReader*** 没有满足约束的样本，则返回值为 NO_DATA。

2.2.2.5.3.20 return_loan

此方法向 ***DataReader*** 通知应用程序已完成访问 *data_values* 和 *sample_infos* 的合集，这些合集是较早前通过 ***DataReader*** 的 ***read*** 或 ***take*** 方法获得的。

data_values 和 *sample_infos* 必须属于单个相关的“对”，也就是说，它们应该对应于调用 ***read*** 或 ***take*** 方法返回的“对”。*data_values* 和 *sample_infos* 必须从返回它们的同一 ***DataReader*** 中获取。如果未满足以上任何一个条件，方法调用将失败并返回 PRECONDITION_NOT_MET。

return_loan 方法允许 ***read*** 或 ***take*** 方法的具体实现以从 ***DataReader***“借用”缓冲区数据到应用程序层，并以这种方式提供对数据的“零拷贝”访问。在借贷期间，***DataReader*** 将保证不修改数据和样本信息。

应用程序没有必要在调用 ***read*** 或 ***take*** 方法后立即退还贷款。但是由于这些缓冲区是属于 ***DataReader*** 的内部资源，因此应用程序不应无限期地保留它们。

仅当 ***read*** 或 ***take*** 方法“借贷”缓冲区到应用程序时才需要使用 ***return_loan*** 方法。如 2.2.2.5.3.8 中所述，仅当调用 ***read*** 或 ***take*** 方法时 *data_values* 和 *sample_infos* 合集的 *max_len* = 0 才会发生这种情况。应用程序还可以检查合集的“*owns*”属性，从而确定哪些需要偿还贷款。在没有贷款的合集上调用 ***return_loan*** 是安全的，没有副作用。

如果合集有贷款，***return_loan*** 返回时合集的 *max_len* = 0。

与 ***read*** 方法类似，特定应用程序的将要写入数据的数据类型生成的专用类上必须提供此方法实现。

2.2.2.5.3.21 get_liveliness_changed_status

此方法允许访问 LIVELINESS_CHANGED 通信状态。通信状态在 2.2.4.1 中描述。

2.2.2.5.3.22 get_requested_deadline_missed_status

此方法允许访问 REQUESTED_DEADLINE_MISSED 通信状态。通信状态在 2.2.4.1 中描述。

2.2.2.5.3.23 get_requested_incompatible_qos_status

此方法允许访问 REQUESTED_INCOMPATIBLE_QOS 通信状态。通信状态在 2.2.4.1 中描述。

2.2.2.5.3.24 get_sample_lost_status

此方法允许访问 SAMPLE_LOST 通信状态。通信状态在 2.2.4.1 中描述。

2.2.2.5.3.25 get_sample_rejected_status

此方法允许访问 SAMPLE_REJECTED 通信状态。通信状态在 2.2.4.1 中描述。

2.2.2.5.3.26 get_subscription_matched_status

此方法允许访问 SUBSCRIPTION_MATCHED 通信状态。通信状态在 2.2.4.1 中描述。

2.2.2.5.3.27 get_topicdescription

此方法返回与 ***DataReader*** 关联的 ***TopicDescription***，与用于创建 ***DataReader*** 的 ***TopicDescription*** 相同。

2.2.2.5.3.28 get_subscriber

此方法返回 ***DataReader*** 所属的 ***Subscriber***。

2.2.2.5.3.29 get_key_value

此方法可用于获取与 ***instance_handle*** 对应的实例关键字 *ke*，仅填充 ***key_holder*** 实例内组成关键字 *key* 的字段。

如果 ***InstanceHandle_t*** *a_handle* 与 ***DataReader*** 已知的现有数据对象不对应，则此方法可能返回 BAD_PARAMETER。如果 DDS 服务无法检查句柄是否有效，则此情况下的结果未指定。

2.2.2.5.3.30 lookup_instance

此方法将实例 *instance* 作为参数，并返回一个句柄，该句柄可当做后续方法的参数。*instance* 参数仅用于检查定义关键字（*key*）的字段。

此方法不会注册新的实例。如果实例先前未注册过，或者由于其他原因服务无法提供实例句柄，则服务将返回特殊值 HANDLE_NIL。

2.2.2.5.3.31 delete_contained_entities

此方法将删除通过 ***DataReader*** 对象的“创建”方法创建的所有实体。也就是说，它会删除所有包含的 ***QueryCondition*** 和 ***ReadCondition*** 对象。

如果任一包含的实体处于无法删除的状态，则方法将返回 PRECONDITION_NOT_MET。

一旦 ***delete_contained_entities*** 成功返回，应用程序可能会删除 ***DataReader***，因为应用程序知道它没有包含的 ***QueryCondition*** 和 ***ReadCondition*** 对象。

2.2.2.5.3.32 wait_for_historical_data

此方法仅适用于 PERSISTENCE QoS 类型不是 VOLATILE 的 ***DataReader*** 实体。

一旦应用程序启用非 VOLATILE ***DataReader***，它就会开始接收“历史”数据，即 ***DataReader*** 加入域之前写入的数据，以及 ***DataWriter*** 实体写入的任何新数据。在某些情况下，应用程序设计的逻辑可能要求应用程序等待直到收到所有“历史”数据。这便是 ***wait_for_historical_data*** 方法的目的。

wait_for_historical_data 方法阻塞调用线程，直到接收到所有“历史”数据，或者 *max_wait* 参数指定的时间已到期，以先发生者为准。返回值为 OK 表示已收到所有“历史”数据；返回值为 TIMEOUT 表示在确认收到所有数据之前已经超过 *max_wait* 设定的时间周期。

2.2.2.5.3.33 get_matched_publication_data

此方法获取当前与 ***DataReader***“关联”的发布信息，即应用程序尚未指示的具有匹配的 ***Topic*** 和兼容 QoS 的发布应通过 ***DomainParticipant ignore_publication*** 方法进行“忽略”。

publication_handle 必须对应于当前与 ***DataReader*** 关联的发布，否则方法调用将失败并返回 BAD_PARAMETER。***get_matched_publications*** 方法可用于查找当前与 ***DataReader*** 匹配的发布方。

如果底层不包含填写 *publication_data* 所需的信息，则方法也可能失败。在这种情况下，将返回 UNSUPPORTED。

2.2.2.5.3.34 get_matched_publications

此方法获取当前与 ***DataReader*** “关联”的发布列表，即应用程序尚未指示的具有匹配的 ***Topic*** 和兼容 QoS 的发布方应通过 ***DomainParticipant ignore_publication*** 方法进行“忽略”。

“publication_handles” 列表中返回的句柄是 DDS 实现方用于标识本地相应匹配的 ***DataWriter*** 实体的句柄。在读取“DCPSPublications”内置主题时，这些句柄与 ***SampleInfo*** 的“instance_handle”字段的句柄相对应。

如果底层不在本地维护连接信息，则方法可能会失败。

2.2.2.5.4 DataSample 类

DataSample 表示由 ***DataReader*** 的 ***read/take*** 方法返回的数据原子信息（即一个实例的一个值）。它由两部分组成：***SampleInfo*** 和 ***Data***。

2.2.2.5.5 SampleInfo 类

SampleInfo	
attributes	
sample_state	SampleStateKind
view_state	ViewStateKind
instance_state	InstanceStateKind
disposed_generation_count	long

no_writers_generation_count	long
sample_rank	long
generation_rank	long
absolute_generation_rank	long
source_timestamp	Time_t
instance_handle	InstanceHandle_t
publication_handle	InstanceHandle_t
valid_data	boolean
No operations	

SampleInfo 是每个被“读取”或“获取”的样本所附带的信息。它包含以下信息：

•***sample_state***（READ 或 NOT_READ）-代表是否已读取过相应的数据样本。

•***view_state***，（NEW 或 NOT_NEW）-代表 ***DataReader*** 是否已经观察到相关实例最新一代的样本。

•***instance_state***（ALIVE，NOT_ALIVE_DISPOSED 或 NOT_ALIVE_NO_WRITERS）-代表实例当前是否存活，或者该实例已被处理以及它被处理的原因。

•如果此实例当前存活，则为 ALIVE。

•如果此实例由 ***DataWriter*** 处理掉，则为 NOT_ALIVE_DISPOSED。

•如果当前任一“存活”（根据 LIVELINESS QoS 判断）的 ***DataWriter*** 对象都没有写入实例数据，实例已由 ***DataReader*** 处理，则为 NOT_ALIVE_NO_WRITERS。

•***disposed_generation_count***，代表实例在收到样本时实例由 ***DataWriter*** 显式处理后变为存活状态的次数。

•***no_writers_generation_count***，代表实例因为没有写入者而被处理后重新变为存活状态的次数。

•***sample_rank***，代表 ***read*** 或 ***take*** 方法返回的合集中同一个实例的样本数。

•***generation_rank***，代表接收的样本与收到与同一实例集合中最新样本之间的代际差异（实例被处理后再次变为存活状态的次数）。

•***absolute_generation_rank***，代表接收的样本与同一实例最新样本（可能不在返回的合集中）之间的代际差异（实例被处理后再次变为存活状态的次数）。

•***source_timestamp***，代表 ***DataWriter*** 提供的写入数据样本的时间。

•***instance_handle***，用于在本地标识相应的实例。

•***publication_handle***，用于在本地标识修改实例的 ***DataWriter***。***publication_handle*** 与 ***DataReader*** 的 ***get_matched_publications*** 方法返回的 ***InstanceHandle_t*** 相同，也可以用作 ***DataReader*** 的 ***get_matched_publication_data*** 方法的参数。

•***valid_data*** 标志，代表 ***DataSample*** 是否包含数据，或者仅用于传递实例 ***instance_state*** 的更改。

有关这些状态和排序的详细说明，请参阅 2.2.2.5.1 访问数据。

2.2.2.5.6 SubscriberListener 接口

SubscriberListener
no attributes

operations		
on_data_on_readers		void
	the_subscriber	Subscriber

由于***订阅者 Subscriber*** 是一种实体，因此它可以拥有与之关联的监听器。在这种情况下，关联的监听器应该是拥有具体类型的 ***SubscriberListener***。它的定义可以在 2.2.4 监听器，条件和等待集中找到。

2.2.2.5.7 DataReaderListener 接口

DataReaderListener		
no attributes		
operations		
on_data_available		void
	the_reader	DataReader
on_sample_rejected		void
	the_reader	DataReader
	status	SampleRejectedStatus
on_liveliness_changed		void
	the_reader	DataReader
	status	LivelinessChangedStatus
on_requested_deadline_missed		void
	the_reader	DataReader
	status	RequestedDeadlineMissedStatus
on_requested_incompatible_qos		void
	the_reader	DataReader
	status	RequestedIncompatibleQosStatus
on_subscription_matched		void
	the_reader	DataReader
	status	SubscriptionMatchedStatus
on_sample_lost		void
	the_reader	DataReader
	status	SampleLostStatus

由于***数据读取者 DataReader*** 是一种***实体 Entity***，因此它可以拥有与之关联的监听器。在这种情况下，关联的监听器应该是拥有具体类型的 ***DataReaderListener***。它的定义可以在 2.2.4 监听器，条件和等待集中找到。

on_subscription_matched 方法旨在通知应用程序发现与 ***DataReader*** 匹配的 ***DataWriter*** 实体。某些 DDS 服务实现可能不会传递此信息。在这种情况下，DDS 规范不要求此监听器方法被调用。

2.2.2.5.8 ReadCondition 类

ReadCondition 对象是专门用于 ***read*** 方法并附加到 ***DataReader*** 的条件。

ReadCondition		
no attributes		
operations		
get_datareader		DataReader
get_sample_state_mask		SampleStateKind []
get_view_state_mask		ViewStateKind []
get_instance_state_mask		InstanceStateKind []

ReadCondition 对象允许应用程序指定感兴趣的数据样本（通过指定所需的样本状态 *sample-states*，视图状态 *view-states* 和实例状态 *instance-states*）。请参阅 ***DataReader read/take*** 方法的参数定义。）这使得中间件可以在适当的信息可用时启用该条件。它们将作为正常条件与 ***WaitSet*** 一起使用。可以将多个 ***ReadCondition*** 绑定到同一个 ***DataReader***。

2.2.2.5.8.1 get_datareader

此方法返回与 ***ReadCondition*** 关联的 ***DataReader***。请注意每个 ***ReadCondition*** 只有一个关联的 ***DataReader***。

2.2.2.5.8.2 get_sample_state_mask

此方法返回确定 ***ReadCondition*** 的 ***trigger_value*** 值时需要考虑的样本状态集。这些是创建 ***ReadCondition*** 时指定的样本状态。

2.2.2.5.8.3 get_view_state_mask

此方法返回确定 ***ReadCondition*** 的 ***trigger_value*** 值时需要考虑的视图状态集。这些是创建 ***ReadCondition*** 时指定的视图状态。

2.2.2.5.8.4 get_instance_state_mask

此方法返回确定的 ***trigger_value*** 值时需要考虑的实例状态集。这些是创建 ***ReadCondition*** 时指定的实例状态。

2.2.2.5.9 QueryCondition 类

QueryCondition 对象是特殊的 ***ReadCondition*** 对象，允许应用程序为本地可用数据的指定过滤器。

QueryCondition		
no attributes		
operations		
get_query_expression		string
get_query_parameters		ReturnCode_t
	out: query_parameters	string []
set_query_parameters		ReturnCode_t
	query_parameters	string []

查询（***query_expression***）类似于 SQL WHERE 子语句，可以通过 ***set_query_parameters*** 方法动态地更改查询参数。

查询表达式的详细语法介绍可以在附录 B 中找到。

此功能是可选的。在 DDS 服务不支持它的情况下，***DataReader :: create_querycondition*** 将返回一个“nil”值（由平台指定）。

2.2.2.5.9.1 get_query_expression

此方法返回与 ***QueryCondition*** 关联的 ***query_expression***，即创建 ***QueryCondition*** 时指定的查询表达式。

2.2.2.5.9.2 get_query_parameters

此方法返回与 ***QueryCondition*** 关联的 ***query_parameters***，即上次成功调用 ***set_query_parameters*** 时指定的参数，或者从未调用 ***set_query_parameters*** 的情况下创建 ***QueryCondition*** 时指定的参数。

2.2.2.5.9.3 set_query_parameters

此方法更改与 ***QueryCondition*** 关联的 ***query_parameters***。

2.2.3 支持的 QoS

数据分发服务（DDS）依赖于 QoS 的使用。QoS（服务质量）具有控制 DDS 服务某方面行为的特点。QoS 由各个 QoS 策略（继承自 QosPolicy 类型的对象）组成。

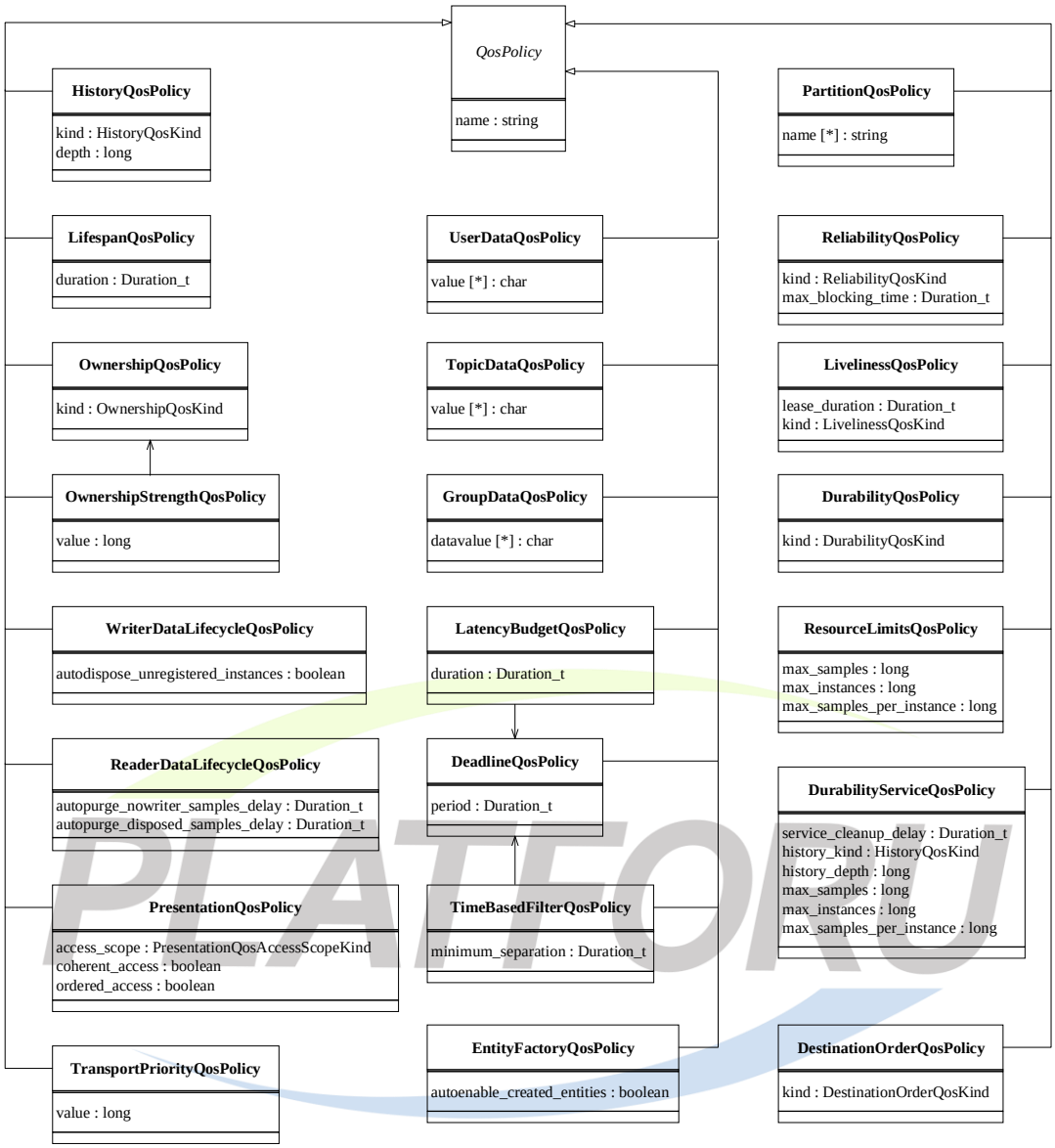

图 2-12 支持的 QoS 策略

QoS（即 ***QosPolicy*** 对象的列表）可以与系统中的所有实体对象相关联，例如 ***Topic***，***DataWriter***，***DataReader***，***Publisher***，***Subscriber*** 和 ***DomainParticipant***。

某些 ***QosPolicy*** 值可能与其他值不一致。下表将详细描述这些场景。当传递一组 ***QosPolicy***（通过 ***set_qos*** 方法）时，在原有的 QoS 上添加新策略所产生的 QoS 集合的一致性会被检查。如果生成的 QoS 无法保持一致，则修改 QoS 的方法将失败，并保留原有值。

在某些情况下，为了正确（或有效）地进行通信，发布方的 ***QosPolicy*** 必须与订阅方的相应策略兼容。例如如果 ***订阅者 Subscriber*** 请求可靠地接收数据，而相应的 ***发布方 Publisher*** 定义为尽力服务的策略，这种情况下通信将不会按请求进行。为解决此问题并尽可能保证发布方和订阅方之间的解耦，***QosPolicy*** 规范遵循 **“订阅者请求，发布者提供”** 模式。在这种模式下，订阅方可以为特定 ***QosPolicy*** 指定“请求”值。发布方为该 ***QosPolicy*** 指定“提供”值。DDS 服务将确定订阅方请求的值是否与发布方提供的值兼容。如果两者兼容，则将建立通信。如果两者不兼容，DDS 服务将不会在两个 ***Entity*** 对象间建立通信，并将通过发布方的 OFFERED_INCOMPATIBLE_QOS 和订阅方的 REQUESTED_INCOMPATIBLE_QOS 记录此结果（请参阅 2.2.4.1）。应用程序可以通过监听器或条件检测到这一结果（请参阅 2.2.4）。

需要在发布者和订阅者之间以兼容方式设置的 ***QosPolicy*** 对象由“RxO”属性的设置指明：

- “RxO”设置为“是”代表可以在发布和订阅两端设置 QoS 策略，并且必须以兼容的方式设置。这种情况明确定义了兼容值。
- “RxO”设置为“否”代表可以在发布和订阅两端设置 QoS 策略，但这两个设置是独立的。也就是说所有值的组合都是兼容的。
- “RxO”设置为“N/A”代表策略只能在发布或订阅端指定，而不能在两端同时指定。因此不需要考虑兼容性。

“可更改”属性确定在启用***实体 Entity*** 后是否可以更改 ***QosPolicy***。换句话说，“可更改”设置为“否”的策略被视为“不可变”，并且只能在***实体 Entity*** 创建时或在***实体 Entity*** 调用 ***enable*** 方法之前指定。

下表给出了支持的 ***QosPolicy*** 列表：它们的名称，语义，可能的值以及它们适用的***实体 Entity***。

QoS 策略	**值**	**含义**	**关联实体**	**RxO**	**可更改**
USER_DATA	八位字节序列：“value”	通过内置主题分发的中间件未知的用户数据（参见 2.2.5）。 默认值为空（零大小）序列。	DomainParticipant, DataReader, DataWriter	否	是
TOPIC_DATA	八位字节序列：“value”	通过内置主题分发的中间件未知的用户数据（参见 2.2.5）。 默认值为空（零大小）序列。	Topic	否	是
GROUP_DATA	八位字节序列：“value”	通过内置主题分发的中间件未知的用户数据（参见 2.2.5）。 默认值为空（零大小）序列。	Publisher, Subscriber	否	是
DURABILITY	名称“kind”取值有以下四种： VOLATILE, TRANSIENT_LOCAL, TRANSIENT, 或者 PERSISTENT	该策略表明数据是否能在 write 方法完成后继续存活	Topic, Publisher, Subscriber	是	否

QoS 策略	值	含义	关联实体	RxO	可更改
	VOLATILE	DDS 服务不需要为以下这种 ***DataReader*** 保留任何数据实例样本，即 ***DataWriter*** 在写入实例数据时未知的 ***DataReader***。也就是 DDS 服务只会尝试向现有的订阅者提供数据。这是默认类型。			
	TRANSIENT_LOCAL, TRANSIENT	DDS 服务将尝试保留一些数据样本，以便将它们发送给潜在的后期加入 DataReader。保留哪些特定样本取决于其他 QoS，例如 HISTORY 和 RESOURCE_LIMITS。 对于 TRANSIENT_LOCAL，服务仅需要将数据保存在写入数据的 *DataWriter* 的内存中，并且数据不要求在 *DataWriter* 失活后继续存活。 对于 TRANSIENT，服务只需要将数据保存在内存中而不是永久存储设备，但是数据并没有与 *DataWriter* 的生命周期联系在一起，通常会一直存活。 支持 TRANSIENT 类型是可选的。			
	PERSISTENT	[可选]数据保存在永久存储器中，因此它们可以比系统会话存活更久。			

QoS 策略	值	含义	关联实体	RxO	可更改
DURABILITY_SERVICE	持续时间：“service_cleanup_delay” HistoryQosPolicy Kind："history_kind" "history_depth" 三个整数：max_samples, max_instances, max_samples_per_instance	指定持久性服务的配置。即服务实现 DURABILITY QoS 的 TRANSIENT 和 PERSISTENT 类型	Topic, DataWriter	No	No
	service_cleanup_delay	控制服务何时能够删除有关数据实例的所有信息。默认情况下为零。			
	history_kind, history_depth	控制隐藏的数据读取者（DataReader）的 HISTORY QoS，该数据读取者将数据存储在持久性服务中（参见 2.2.3.4，DURABILITY）。 默认设置为 history_kind = KEEP_LAST history_depth = 1			
	max_samples, max_instances, max_samples_per_instance	控制隐藏的数据读取者（DataReader）的 RESOURCE_LIMITS QoS，该 DataReader 将数据存储在持久性服务中。 默认情况下它们都是 LENGTH_UNLIMITED。			

QoS 策略	值	含义	关联实体	RxO	可更改
PRESENTATION	“access_scope”: INSTANCE,TOPIC,GROUP 两个 bool 变量: “coherent_access” “ordered_access”	指定如何将表示数据实例变化的样本呈现给订阅的应用程序。 此策略会影响应用程序指定和接收一致变化以及查看变化的相对顺序的能力。 *access_scope* 确定可以保留变化顺序和一致性的跨越实体的最大范围。 两个布尔值控制 *access_scope* 范围中是否支持一致访问和有序访问。	Publisher, Subscriber	是	否
	INSTANCE	访问范围仅跨越单个实例。 代表着一个实例的变化不需要与其他实例的变化保持一致性或有序性。即每个实例在自己内部独立维持顺序和一致变化。 这是默认的 *access_scope*。			
	TOPIC	范围跨越同一 ***DataWriter***（或 ***DataReader***）中的所有实例，但不跨越不同 ***DataWriter***（或 ***DataReader***）中的实例。			
	GROUP	[可选]范围跨越属于同一发布者 ***Publisher***（或订阅者 ***Subscriber***）中的 ***DataWriter***（或 ***DataReader***）实体的所有实例。			
	coherent_access	指定支持***一致访问***。即能够在发布端将一组变化作为一个单元分组，使得它们在订阅端作为一个单元被接收。 *coherent_access* 的默认设置为 FALSE。			

QoS 策略	值	含义	关联实体	RxO	可更改
	ordered_access	指定对订阅端收到的样本支持***有序访问***。即订阅者能够以与发布端发生的顺序相同的顺序查看数据变化。*ordered_access* 的默认设置为 FALSE。			
DEADLINE	持续时间：“period”	***DataReader*** 期望在每个截止期限内至少出现一个新的样本，该样本更新一次实例值。 ***DataWriter*** 指示应用程序承诺每个截止期限内至少为 ***DataWriter*** 管理的每个实例写一个新值（使用 ***DataWriter***）。 DataReader 的 DEADLINE *period* 小于 TIME_BASED_FILTER 的 *minimum_separation* 是不一致的情况。 截止期限 ***period*** 默认是无限大的。	Topic, DataReader DataWriter	是	是
LATENCY_BUDGET	持续时间：“duration”	指定从写入数据开始到将数据插入接收端应用程序缓存之间的最大可接受延迟，并通知接收应用程序。此策略是对 DDS 服务的提示，而不是必须监视或强制执行的。服务不需要跟踪或警告用户任何违规行为。 *duration* 的默认值为零，表示应最小化延迟。	Topic, DataReader DataWriter	是	是
OWNERSHIP	“kind”: SHARED EXCLUSIVE	[可选]指定是否允许多个 ***DataWriters*** 写入相同实例的数据；如果允许，应如何仲裁这些修改	Topic, DataReader DataWriter	是	否

QoS 策略	值	含义	关联实体	RxO	可更改
	SHARED	表示每个实例共享所有权。允许多个写入者更新同一实例，并且所有更新都可供读取者使用。即实例没有“所有者”的概念。如果 OWNERSHIP QoS 策略未指定或不支持，这是默认行为。			
	EXCLUSIVE	表示每个实例只能由一个 ***DataWriter*** 拥有，但实例的所有者可以动态更改。 所有者的选择由 OWNERSHIP_STRENGTH QoS 策略的设置控制。 所有者始终设置为当前“活动”（由 LIVELINESS QoS 确定）的强度最高的 ***DataWriter*** 对象。			
OWNERSHIP_STRENGTH	一个整数：“value”	[可选]指定用于在多个 ***DataWriter*** 对象之间进行仲裁的“强度”值，这些 ***DataWriter*** 对象尝试修改相同实例（由 ***Topic*** + ***key*** 标识）的数据对象。 此策略仅适用于 OWNERSHIP QoS 策略为 EXCLUSIVE 的情况。 *ownership_strength* 的默认值为零。	DataWriter	N/A	是

QoS 策略	值	含义	关联实体	RxO	可更改
LIVELINESS	“kind”: AUTOMATIC, MANUAL_BY_PARTICIPANT, MANUAL_BY_TOPIC 持续时间：“leaseduration”	确定应用程序用于确定***实体 Entity*** 是否“活动”（存活）的机制和参数。 ***实体 Entity*** 的“存活性”状态和 OWNERSHIP QoS 策略的设置组合起来用于维护实例所有权。 当实体不再存活时，通过监听器通知应用程序这一状态。 ***DataReader*** 请求通过请求的方式维护写入者的存活性，并且请求在不超过 lease_duration 时间周期内检测到存活性的丢失。 ***DataWriter*** 承诺使用规定的方式以不超过 lease_duration 的间隔传递其存活性。 监听器用于通知 ***DataReader*** 写者存活性的丢失以及某些 ***DataWriter*** 违反存活性合约。 默认类型为 AUTOMATIC，***lease_duration*** 的默认值为无限。	Topic, DataReader DataWriter	是	否
	AUTOMATIC	底层将至少按照 ***lease_duration*** 要求的周期自动发出 ***DataWriters*** 的存活信息			
	MANUAL 模式	用户应用程序负责使用 2.2.3.11 LIVELINESS 中描述的机制当中的一种向服务发出存活信号。在每个 ***lease_duration*** 周期内至少需要声明一次存活，否则 DDS 服务将假定相应的 ***实体 Entity*** 不再“活动/存活”。			

QoS 策略	值	含义	关联实体	RxO	可更改
	MANUAL_BY_PARTICIPANT	服务将假设只要 ***DomainParticipant*** 中的一个***实体 Entity*** 声明其存活性，则同一 ***DomainParticipant*** 中的其他***实体 Entity*** 也是活着的。			
	MANUAL_BY_TOPIC	如果应用程序已声明 ***DataWriter*** 本身的活跃性，则服务仅仅假定该 ***DataWriter*** 是存活的。			
TIME_BASED_FILTER	持续时间：“minimum_separation”	过滤器允许 ***DataReader*** 告知服务它只对（可能的）数据值的子集感兴趣。过滤器声明 ***DataReader*** 不希望在每个 ***minimum_separation*** 内接收多个值，无论数据变化多快。 ***DataReader*** 的 *minimum_separation* 大于其 DEADLINE *period* 是 QoS 不一致的情况。 默认情况下，***minimum_separation*** = 0 表示 ***DataReader*** 可能对所有值感兴趣。	DataReader	N/A	是

QoS 策略	值	含义	关联实体	RxO	可更改
PARTITION	字符串列表： name	一组字符串，用于在发布者和订阅者可见的主题之间引入逻辑分区。如果 ***Publisher*** 和 ***Subscriber*** 具有公共分区名称字符串， ***Publisher*** 的 ***DataWriter*** 与 ***Subscriber*** 的 ***DataReader*** 便可以通信（前提是主题已经匹配并具有兼容的 QoS）。 空字符串（“”）被认为是一种有效的分区，与其他使用相同字符串匹配的规则进行分区命名和使用正则表达式匹配进行分区命名一样（见 2.2.3.13） PARTITION QoS 的默认值是零长度序列。零长度序列被视为一个特殊值，等同于包含由空字符串组成的单个元素的序列。	Publisher, Subscriber	否	是
RELIABILITY	“kind”: RELIABLE, BEST_EFFORT 持续时间 max_blocking_time	表示 DDS 服务提供/请求的可靠性级别。	Topic, DataReader DataWriter	是	否

QoS 策略	值	含义	关联实体	RxO	可更改
	RELIABLE	指定 DDS 服务将尝试传输其历史记录中的所有数据样本。丢失的样本可能会被重新发送。在稳定状态（没有新的需要通过 ***DataWriter*** 传输的修改）条件下，中间件保证 ***DataWriter*** 历史记录中的所有样本最终将被传递给所有 ***DataReader*** 对象。在稳定状态之外，HISTORY 和 RESOURCE_LIMITS 策略将决定样本如何成为历史记录的一部分以及样本是否可以从中丢弃。这是 ***DataWriters*** 的默认值。			
	BEST_EFFORT	表示不重传样本是可以接受的。可假定样本的新值经常产生，因此无需重新发送或确认任何样本。这是 ***DataReaders*** 和 ***Topics*** 的默认值。			
	max_blocking_time	*max_blocking_time* 的值表示如果 *DataWriter* 没有空间来存储写入的值，允许 *DataWriter :: write* 方法阻塞的最长时间。默认的 *max_blocking_time* = 100 ms。			
TRANSPORT_PRIORITY	一个整数：“value”	此策略向底层结构提示如何设置用于发送数据的底层传输的优先级。*transport_priority* 的默认值为零。	Topic, DataWriter	N/A	是
LIFESPAN	持续周期“duration”	指定 ***DataWriter*** 写入的数据的最大有效持续时间。s 生命期限 *duration* 的默认值为无限大。	Topic, DataWriter	N/A	是

QoS 策略	值	含义	关联实体	RxO	可更改
DESTINATION_ ORDER	“kind”: BY_ RECEPTION_ TIMESTAMP, BY_SOURCE_ TIMESTAMP	控制用于确定发布者实体对同一数据实例（有匹配的主题 Topic 和关键字 key）所做的更改之间的逻辑顺序的标准。 默认类型是 BY_RECEPTION_TIMESTAMP。	Topic, DataReader, DataWriter	是	否
	BY_ RECEPTION_ TIMESTAMP	表示根据每个订阅者的接收时间对数据进行排序。由于每个订阅者可能在不同时间接收数据，因此无法保证所有订阅者将以相同的顺序观察到数据的更改。 因此，每个订阅者可能最终得到数据的不同最终值。			
	BY_SOURCE_ TIMESTAMP	表示根据数据源（由 DDS 服务或应用程序）设置的时间戳对数据进行排序。在任何情况下，这都保证了所有订阅者数据最终值的一致性。			
HISTORY	“kind”: KEEP_LAST, KEEP_ALL 一个可选的整数 “depth”	指定在样本值成功传递给一个或多个现有订阅者之前样本值发生变化（一次或多次）的情况下 DDS 服务的行为准则。此 QoS 策略控制 DDS 服务是应仅递送最新值，尝试提供所有中间值，还是在执行两者之间的某些操作。在发布方此策略控制应由为了现存 ***DataReader*** 实体的 ***DataWriter*** 维护的样本。写入样本后发现的 ***DataReader*** 实体的行为由 DURABILITY QoS 策略控制。在订阅方，它控制应该维护的样本数，直到应用程序从 DDS 服务“获取”它们。	Topic, DataReader, DataWriter	否	否

QoS 策略	值	含义	关联实体	RxO	可更改
	KEEP_LAST 和可选整数 “depth”	在发布方，服务将仅尝试保留由 ***DataWriter*** 管理的每个数据实例（由其关键字 key 标识）的最新“*depth*”个样本。 在订阅方，DataReader 将仅尝试保留每个实例（由其关键字 key 标识）接收的最新““*depth*”个样本，直到应用程序通过 ***DataReader*** 的 ***take*** 方法“获取”它们。 KEEP_LAST 是默认类型。***depth*** 的默认值为 1。如果指定的值不是 1，则应与 RESOURCE_LIMITS QoS 策略的设置保持一致。			
	KEEP_ALL	在发布方，服务将尝试保留由 ***DataWriter*** 管理的每个数据实例（由其关键字 key 标识）的所有样本（代表每个写入的值），直到它们可以被传递给所有订阅者。 在订阅方，服务将尝试保留由 ***DataReader*** 管理的每个数据实例（由其关键字 key 标识）的所有样本。服务会保留这些样本，直到应用程序通过执行 ***take*** 方法从服务“获取”它们。***depth*** 设置无效。 其隐含值为LENGTH_UNLIMITED。			
RESOURCE_LIMITS	三个整数：max_samples, max_instances, max_samples_per_instance	指定服务可以使用的资源，以满足请求的 QoS。	Topic, DataReader, DataWriter	否	否

QoS 策略	值	含义	关联实体	RxO	可更改
	max_samples	指定 ***DataWriter***（或 ***DataReader***）可以管理的与其关联的所有实例的样本数之和的最大值。表示中间件可以为任何一个 ***DataWriter***（或 ***DataReader***）存储的最大样本数。 该值小于 *max_samples_per_instance* 是一种 QoS 不一致的情况。 默认情况下，值为 LENGTH_UNLIMITED。			
	max_instances	表示 ***DataWriter***（或 ***DataReader***）可以管理的最大实例数。 默认情况下，值为 LENGTH_UNLIMITED			
	max_samples_per_instance	表示 ***DataWriter***（或 ***DataReader***）可以管理的任一实例的最大样本数。 该值大于 ***max_samples*** 是一种 QoS 不一致的情况。 默认情况下，值为 LENGTH_UNLIMITED。			
ENTITY_FACTORY	一个 bool 值：autoenable_created_entities	作为其他实体的工厂时，控制实体的行为。即配置 create_ *和 delete_ *方法的附带后果。	DomainParticipantFactory, DomainParticipant, Publisher, Subscriber	否	是
	autoenable_created_entities	指定充当工厂的实体是否自动启用它创建的实例。 如果 *autoenable_created_entities* == TRUE，工厂将自动启用每个创建的实体，否则不会。 默认情况下为 TRUE。			
WRITER_DATA_LIFECYCLE	一个 bool 值：autodispose_unregistered_instances	指定 ***DataWriter*** 关于其管理的数据实例的生命周期的行为。	DataWriter	N/A	是

QoS 策略	值	含义	关联实体	RxO	可更改
	autodispose_unregistered_instances	控制 ***DataWriter*** 每次取消注册时是否会自动处理实例。设置 *autodispose_unregistered_instances* = TRUE 表示未注册的实例也将被视为已处置。 默认情况下为 TRUE。			
READER_DATA_LIFECYCLE	两个持续周期： autopurge_nowriter_samples_delay 和 autopurge_disposed_samples_delay	指定 ***DataReader*** 关于其管理的数据实例的生命周期的行为。	DataReader	N/A	是
	autopurge_nowriter_samples_delay	指示 ***DataReader*** 如果某些实例的 *instance_state* 为 NOT_ALIVE_NO_WRITERS，则他们的信息必须保留至少这个时间长度。 默认情况下，无限。			
	autopurge_disposed_samples_delay	指示 ***DataReader*** 如果某些实例的 *instance_state* 为 NOT_ALIVE_DISPOSED，则他们的信息必须保留至少这个时间长度。 默认情况下，无限。			

2.2.3.1 USER_DATA

此 QoS 的目的是允许应用程序将附加信息附加到创建的***实体 Entity*** 对象，以便当远程应用程序发现它们时，它可以访问该信息并将该信息用于其自身目的。此 QoS 的一种可能用途是附加可供远程应用程序使用的安全凭证或一些其他信息，辅助远程应用程序对数据来源进行身份验证。结合 ***ignore_participant***，***ignore_publication***，***ignore_subscription*** 和 ***ignore_topic*** 等方法，这些 QoS 可以帮助应用程序定义和实施自己的安全策略。此 QoS 的使用不仅限于安全性，而是提供简单但灵活的可扩展性机制。

2.2.3.2 TOPIC_DATA

此 QoS 的目的是允许应用程序将附加信息附加到创建的***主题 Topic*** 对象，以便当远程应用程序发现它们存在时，它可以检查信息并以应用程序定义的方式使用它。结合 ***DataReader*** 和 ***DataWriter*** 上的监听器以及 ***ignore_topic*** 等操作，这些 QoS 可以帮助应用程序扩展 DDS 中间件提供的 QoS 功能。

2.2.3.3 GROUP_DATA

此 QoS 的目的是允许应用程序将附加信息附加到创建的***发布者 Publisher*** 或***订阅者 Subscriber***。GROUP_DATA 的值可供 ***DataReader*** 和 ***DataWriter*** 实体上的应用程序使用，并通过内置主题传播。

应用程序与 ***DataReaderListener*** 和 ***DataWriterListener*** 的组合可以使用此 QoS 来实现类似于 PARTITION QoS 的匹配策略，除非可以基于应用程序定义的策略做出决策。

2.2.3.4 DURABILITY

发布/订阅范式实现了 ***DataReader*** 和 ***DataWriter*** 之间的解耦合，这种解耦合的特性允许应用程序在当前网络上没有对应的读取者情况下仍然可以写入数据。此外，后加入网络的 ***DataReader*** 可能有兴趣访问之前已经发布的数据的最新值以及可能的某些历史数据。此 QoS 策略控制服务是否为后加入的读取者提供数据。请注意虽然这个 QoS 与服务相关，但它并不严格控制服务内部维护哪些数据。也就是说，如果 DURABILITY QoS 策略设置为 VOLATILE，则服务可以针对其自身目的（例如流控制）维护一些数据，但不能使这些数据供后期加入的读取者使用。

当且仅当不等式“提供类型 ***kind*** >= 请求类型 ***kind***”为“TRUE”时，提供的值才被认为与所请求的值兼容。出于这种不等式的目的，DURABILITY ***kind*** 的值被认为是有序的，即 VOLATILE <TRANSIENT_LOCAL <TRANSIENT <PERSISTENT。

为了实现 DURABILITY QoS 类型 TRANSIENT 或 PERSISTENT，DDS 服务对于 DURABILITY QoS ***kind*** 为 TRANSIENT 或 PERSISTENT 的每个主题表现出“好像”有相应的“内置”***DataReader*** 和 ***DataWrite***r，这些读取者和写入者配置为相同的 DURABILITY ***kind***。换句话说，“好像”系统中的某个地方（可能在远程节点上）有一个“内置持久性 ***DataReader***”订阅该 ***Topic***，而“内置持久性 ***DataWriter***”发布该 ***Topic***，为后续新加入系统的订阅者提供数据。

对于每个 ***Topic***，内置的虚构“永久服务”***DataReader*** 和 ***DataWriter*** 都根据相应的 ***Topic*** QoS 配置其 QoS。换句话说服务“好像”首先使用 ***find_topic*** 方法来访问 ***Topic***，然后使用 ***Topic*** 的 QoS 来配置虚构的内置实体。

此模型的一个结论是可以在 ***Topic*** 上设置适当的 QoS 来配置瞬态或持久性 QoS 服务。

对于给定的 ***Topic***，通常的请求/提供的语义适用于系统中的任何发送该 ***Topic*** 的 ***DataWriter*** 与订阅该 ***Topic*** 的内置瞬态/持久 ***DataReader*** 之间的匹配；类似地，也适用于发布 ***Topic*** 的内置瞬态/持久 ***DataWriter*** 和订阅 ***Topic*** 的任何 ***DataReader*** 之间的匹配。因此，与 ***Topic*** 指定的 QoS 不兼容的 ***DataWriter*** 将不会将其数据发送到瞬态/持久服务，并且与 ***Topic*** 指定的 QoS 不兼容的 ***DataReader*** 将无法从中获取数据。

本地 ***DataReader*** / ***DataWriter*** 实体与相应的虚构“内置瞬态/持久实体”之间的不兼容性会产生 REQUESTED_INCOMPATIBLE_QOS / OFFERED_INCOMPATIBLE_QOS 的状态更改，相应的 ***Listener*** 调用和/或 ***Condition*** 的信号与 ***WaitSet*** 对象表现地好像和非虚构实体一样。

service_cleanup_delay 参数的设置控制着 TRANSIENT 或 PERSISTENT 服务何时能够删除数据实例的所有信息。满足以下条件便可以不再保持数据实例的相关信息：

1.实例已被明确处理（*instance_state* = NOT_ALIVE_DISPOSED）；

2.当实例处于 NOT_ALIVE_DISPOSED 状态时，系统检测到没有更多“存活”的 ***DataWriter*** 实体写入实例数据，也就是说所有现存的 ***DataWriter*** 要么都取消注册实例（调用取消注册），要么已经不再存活；

3.自服务检测前两个条件已经达到的那一刻起，已经过了 *service_cleanup_delay* 指定的时间。

在应用程序处理实例并完成与处理实例相关的其他任务之前应用程序崩溃的情况下，*service_cleanup_delay* 的作用很明显。重新启动时，应用程序可能会请求初始数据重新获得其状态，并且 *service_cleanup_delay* 引入的延迟将允许重新启动的应用程序接收已处置实例的有关信息并完成中断的任务。

2.2.3.5 DURABILITY_SERVICE

此策略用于配置由“持久性服务”使用的虚构 ***DataReader*** 和 ***DataWriter*** 使用的 HISTORY QoS 和 RESOURCE_LIMITS QoS。“持久性服务”负责实现 DURABILITY ***kind*** 的 TRANSIENT 和 PERSISTENCE（见 2.2.3.4））。

2.2.3.6 PRESENTATION

此 QoS 策略控制相互依赖的数据实例更改的程度，以及服务可以传播和维护的依赖关系的类型。

coherent_access 控制服务是否将通过 ***begin_coherent_change*** 和 ***end_coherent_change*** 方法保留发布端应用程序所做更改的分组。

ordered_access 控制服务是否保留更改顺序。

粒度由 access_scope 的设置控制。

如果设置了 ***coherent_access***，则 ***access_scope*** 控制一致变化的最大范围。表现如下：

• 如果 ***access_scope*** 设置为 INSTANCE，则使用 begin_coherent_change 和 end_coherent_change 不会影响订阅者如何访问数据，因为范围仅限于每个实例，对单独实例的更改被视为独立的，因此无法通过一致性更改进行分组。

•如果 ***access_scope*** 设置为 TOPIC，那么一致的更改（通过 ***begin_coherent_change*** 和 ***end_coherent_change*** 方法的闭环调用表示）将独立地提供给每个远程 ***DataReader***。也就是说对每个单独 ***DataWriter*** 的实例所做的更改将与同一 ***DataWriter*** 的实例的其他更改保持一致，但不会与属于不同 ***DataWriter*** 的实例所做的更改分为一组。

•如果 ***access_scope*** 设置为 GROUP，则通过附属到公共发布者的 ***DataWriter*** 对实例进行的一致性更改将作为一个单元提供给远程订阅者。

如果设置了 ***ordered_access***，则 ***access_scope*** 控制服务保留数据顺序的最大范围。

•如果 ***access_scope*** 设置为 INSTANCE（最低级别），则相对于对任何其他实例的更改，对每个实例的更改将被视为无序。这意味着对两个实例所做的更改（创建，删除，修改）不一定按它们发生的顺序被观察到。即使通过相同的应用程序线程使用相同 ***DataWriter*** 所做的更改也是如此。

•如果 ***access_scope*** 设置为 TOPIC，则单个 ***DataWriter*** 所做的更改（创建，删除，修改）将按其发生的顺序提供给订阅者。通过不同的 ***DataWriter*** 对实例所做的更改不一定按它们发生的顺序被订阅者观察到。即使通过单个应用程序线程中附加到同一 ***Publisher*** 的不同 ***DataWriter*** 所做的更改也是如此。

•最后，如果 ***access_scope*** 设置为 GROUP，附加到同一 ***Publisher*** 对象的 ***DataWriter*** 实体对实例所做的更改将按相同的发生顺序提供给订阅者。

请注意，此 QoS 策略控制订阅者可以使用的更改的范围。这意味着订阅者可以以一致的方式和正确的顺序访问这些更改；但是它并不意味着订阅者确实会以正确的顺序访问更改。为此，订阅端的应用程序必须使用正确的逻辑来读取 ***DataReader*** 对象，如 2.2.2.5.1 访问数据中所述。

当且仅当满足以下条件时，提供的值才被视为与所请求的值兼容：

1.“提供的 ***access_scope*** >= 请求的 ***access_scope***”的不等式为“TRUE”。为了使这一不等式有意义，PRESENTATION ***access_scope*** 的值是有序的， INSTANCE <TOPIC <GROUP。

2.请求的 ***coherent_access*** 为 FALSE，或者提供和请求的 ***coherent_access*** 都为 TRUE。

3.请求的 ***ordered_access*** 为 FALSE，或者提供和请求的 ***ordered_access*** 都为 TRUE。

2.2.3.7 DEADLINE

此策略对于期望 ***Topic*** 定期更新每个实例的场景非常有用。在发布端，此 QoS 设置创建一个约定，应用程序必须满足这一约定。在订阅端，此 QoS 设置为提供数据值的远程发布者建立了最低要求。

当 DDS 服务“匹配”***DataWriter*** 和 ***DataReader*** 时，它检查此 QoS 设置是否兼容（提供的 deadline ***period*** <= 请求的 deadline ***period***）。如果不匹配，则 DDS 服务通知（通过监听器或条件机制）两个实体 QoS 设置不兼容，通信不会发生。

假设读写两端具有兼容的 QoS 设置，则服务将监视此约定的履行情况。当有违约情况发生时，DDS 服务将通过适当的监听器或条件状态通知应用程序。

当且仅当不等式“提供的截止期限<=请求的截止期限”为“TRUE”时，所提供的值被认为与所请求的值相兼容。

DEADLINE 策略的设置必须与 TIME_BASED_FILTER 的设置保持一致。要使这两个政策保持一致，QoS 设置必须是“deadline ***period*** > = ***minimum_separation***”。

2.2.3.8 LATENCY_BUDGET

该策略为应用程序提供了一种告知 DDS 中间件数据通信的“紧迫性”的方法。通过非零的 ***duration***，服务可以优化其内部方法。

这种策略是一种提示。关于服务应如何利用此提示，没有规定的机制。

当且仅当不等式“提供的 ***duration*** <=请求的 ***duration***”为“真”时，提供的值才被认为与所请求的值兼容。

2.2.3.9 OWNERSHIP

此策略控制 DDS 服务是否允许多个 ***DataWriter*** 对象更新数据的同一实例（由“主题+关键字”标识）。

OWNERSHIP QoS 的 ***kind*** 设置有两种取值：SHARED 和 EXCLUSIVE。

2.2.3.9.1 SHARED 类型

此设置表示 DDS 服务不会为每个实例强制实施唯一的所有权。在这种情况下，多个数据写入者可以更新相同的数据对象实例。***Topic*** 的订阅者能获得所有 ***DataWriter*** 对象对数据的修改，但还是会受限于可以过滤特定数据样本的其他 QoS（例如 TIME_BASED_FILTER 或 HISTORY QoS 策略）的设置。在任何情况下，都没有基于 ***DataWriter*** 的身份对数据的修改进行“过滤”。

2.2.3.9.2 EXCLUSIVE 类型

此设置表示数据对象的每个实例只能由一个 ***DataWriter*** 修改。换句话说，在任何时间点只有一个 ***DataWriter*** “拥有”某一实例，并且只有该 ***DataWriter*** 的修改对 ***DataReader*** 对象可见。通过选择具有最高***强度（strength）***值的 ***DataWriter*** 来确定所有者，除此之外该 ***DataWriter*** 还需要满足以下两个条件：由 LIVELINESS QoS 策略定义的状态为“活着”并且没有违反数据实例的 DEADLINE QoS 约定。因此，所有权可以因以下方式而改变：（a）系统中有更高强度值的 ***DataWriter*** 修改该实例，（b）拥有该实例的 ***DataWriter*** 的强度值发生变化，（c）拥有该实例的 ***DataWriter*** 的存活性出现变化，以及（d）拥有该实例的 ***DataWriter*** 超出截止时间还未更新该实例。

系统的行为就好像每个 ***DataReader*** 独立地进行确定一样。每个 ***DataReader*** 都可能在不同的时间检测到所有权的变化。规范不要求该主题的所有 ***DataReader*** 对象在特定时间点对每个实例所有者身份的认知保持一致。

DataWriter 对象也不需要知道它们是否拥有特定实例。当 ***DataWriter*** 修改它当前不拥有的实例时并不会发生错误或者出现异常通知。

这些需求是为了（a）保持发布者和订阅者的解耦，以及（b）有效地实现该策略。

具有相同强度的多个 ***DataWriter*** 对象可能会修改同一实例。如果发生这种情况，服务将选择一个 ***DataWriter*** 对象作为“所有者”。如何选择所有者不做约束，但用于选择“所有者”的策略要保证所有 ***DataReader*** 对象选择相同的 ***DataWriter*** 作为“所有者”。除此之外还要求所有者保持不变，直到以下任一事件发生：

- 所有者的强度值发生改变；
- 所有者的存活性发生改变；
- 所有者超过实例的截止时间未更新数据；
- 具有更高强度值的新 ***DataWriter*** 修改实例；
- 具有相同强度的另一个 ***DataWriter*** 被中间件服务视为可以修改实例的新所有者。

独占的所有权基于逐个实例。也就是说，订阅者可以接收到较低强度值 ***DataWriter*** 写入的数据值，前提是这些较低强度值 ***DataWriter*** 能够影响的实例尚未由较高强度值的 ***DataWriter*** 写入数据值。

所提供的 OWNERSHIP ***kind*** 的值必须与所要求的完全匹配，否则它们被认为是不相容的。

2.2.3.10 OWNERSHIP_STRENGTH

此 QoS 策略应与 OWNERSHIP 策略结合使用。它仅适用于 OWNERSHIP ***kind*** 设置为 EXCLUSIVE 的情况。

OWNERSHIP_STRENGTH 的值用于确定数据实例（由关键字 ***key*** 标识）的所有权。

仲裁由 ***DataReader*** 执行。用于执行仲裁的规则在 2.2.3.9.2 EXCLUSIVE 类型中描述。

2.2.3.11 LIVELINESS

此策略控制中间件服务使用的机制和参数，以确保网络上的特定实体仍处于“存活”状态。存活性还会影响由 OWNERSHIP QoS 策略确定的特定实例的所有权。

此策略具有多个设置以支持定期更新的数据对象以及偶尔更改的数据对象。同时此策略能够根据存活性机制检测到的故障类型为不同的应用程序需求提供定制化服务。

AUTOMATIC 存活性设置最适合仅需要检测出进程级别的故障而不需要检测出进程内逻辑错误的应用程序。服务负责以所需的速率续订租约，因此只要运行 ***DomainParticipant*** 的本地进程以及将其连接到远程参与者的链接保持连接状态，***DomainParticipant*** 中的实体将被视为存活。这种设置需要的开销最低。

MANUAL 存活性设置（MANUAL_BY_PARTICIPANT，MANUAL_BY_TOPIC）要求发布端的应用程序在租约到期之前定期声明其存活性，以表明相应的实体仍然存活。通过调用 ***assert_liveliness*** 方法可以显式地执行，或者通过写入数据来隐式地执行。

两种可能的手动设置控制应用程序声明其存活性的粒度。

- MANUAL_BY_PARTICIPANT 仅要求发布者中的一个实体声明为活动，从而推断同一 ***DomainParticipant*** 中的所有其他 ***Entity*** 对象也是活动的。
- MANUAL_BY_TOPIC 要求声明 DataWriter 中至少有一个实例存活。

当且仅当满足以下条件时，提供的值才被视为与所请求的值兼容：

1.不等式“提供 ***kind*** >= 请求 ***kind***”为“真”。 为了使这一不等式有意义，LIVELINESS ***kind*** 的值被认为是有序的，如下所示：

AUTOMATIC <MANUAL_BY_PARTICIPANT <MANUAL_BY_TOPIC。

2. 不等式“提供的 ***lease_duration*** <= 请求的 ***lease_duration***”为真。

必须由中间件服务检测 LIVELINESS 的变化，检测的时间粒度大于或等于 ***lease_duration***。这确保了在每个 ***lease_duration*** 期间 ***LivelinessChangedStatus*** 的值至少更新一次，并且在 LIVELINESS 更改之后的 ***lease_duration*** 周期中通知相关的***监听器***和 ***WaitSets***。

2.2.3.12 TIME_BASED_FILTER

此策略允许 ***DataReader*** 声明它不一定要收到某个主题下发布的每个实例的所有数据值。相反它希望每个 ***minimum_separation*** 期间最多收到一个数据更新。

TIME_BASED_FILTER 分别应用于每个实例，也就是说 DataReader 不希望每个 ***minimum_separation*** 周期看到单个实例的多个数据样本。

此设置允许 ***DataReader*** 进一步将其自身与 ***DataWriter*** 对象解耦，它可用于保护在异构网络上运行的应用程序，这种网络中的某些节点生成数据的速度比其他节点接收处理数据的速度更快。它还表明这样的事实：对于快速变化的数据，不同的订阅者对于接收最新数据值的频率有不同的需求。

TIME_BASED_FILTER 的设置（即 ***minimum_separation*** 值大于零），与 HISTORY 和 RELIABILITY QoS 的所有设置兼容。TIME_BASED_FILTER 指定 ***DataReader*** 感兴趣的数据样本。HISTORY 和 RELIABILITY QoS 会影响中间件对于已确定为 ***DataReader*** 感兴趣的样本的行为，即它们在应用 TIME_BASED_FILTER 后应用。

在 RELIABILITY QoS 类型为 RELIABLE 并且处于稳定状态的情况下，当 ***DataWriter*** 在比 ***minimum_separation*** 时间更长的周期内不再写入新的数据样本时，系统应保证将最后一个样本传递给 ***DataReader*** 。

TIME_BASED_FILTER ***minimum_separation*** 的设置必须与 DEADLINE ***period*** 一致。要使这两个 QoS 策略保持一致，它们必须确保“***period*** >= ***minimum_separation***”。当通过 ***set_qos*** 方法创建实体时，以不一致的方式设置这些策略将导致方法调用失败。

2.2.3.13 PARTITION

此策略在域定义的“物理”分区内引入逻辑分区概念。

要使 ***DataReader*** 收到 ***DataWriter*** 对实例所做的更改，不仅需要两者的主题匹配，而且它们必须共享公共分区。定义此 QoS 策略的列表中的字符串定义分区名称。分区名称可能包含通配符。共享公共分区意味着其中一个分区名称匹配。

未能匹配分区不会被视为“不兼容”的 QoS，并且不会触发任何监听器或条件。

这项策略是可以改变的。更改此策略可能会修改现有 ***DataReader*** 和 ***DataWriter*** 实体的“匹配”关系。它可以建立之前不存在的新的“匹配”，或者破坏现有的匹配。

PARTITION 名称可以是正则表达式，并包含 POSIX 匹配函数 API（1003.2-1992 B.6 节）定义的通配符。发布者或订阅者可以在分区名称中包含正则表达式，但是两个包含通配符的名称都不会被视为匹配。这意味着虽然可以在发布者和订阅者端使用正则表达式，但中间件服务不会尝试匹配（发布者和订阅者之间的）两个正则表达式。

分区与以不同方式在不同域中创建实体对象不同。首先，属于不同域的实体彼此完全隔离，应用程序或中间件服务本身没有渠道或任何其他方式来查看其不属于的域中的实体；其次，实体只能属于一个域，但实体可以属于多个分区；最后，就 DDS 服务而言，每个唯一数据实例由三元组（***domainId***，***Topic***，***key***）标识。因此不同域中的两个 ***Entity*** 对象不能引用同一个数据实例。另一方面，可以在一个或多个分区上提供（发布）或请求（订阅）相同的数据实例。

2.2.3.14 RELIABILITY

此策略指明 ***DataReader*** 请求或 ***DataWriter*** 提供的可靠性级别。这些级别是有序的，BEST_EFFORT 低于 RELIABLE。提供某一级别的 ***DataWriter*** 隐式提供所有比这一级别低的级别，即 ***DataWriter*** 提供的级别为 RELIABLE 时，默认也提供 BEST_EFFORT 级别。

此策略的设置依赖于 RESOURCE_LIMITS 策略的设置。如果 RELIABILITY ***kind*** 设置为 RELIABLE，则 ***DataWriter*** 上的 ***write*** 方法可能会在以下情况阻塞：数据修改将导致数据丢失或导致超出 RESOURCE_LIMITS QoS 指定的任一限制。在这些情况下，可靠性 ***max_blocking_time*** 配置 ***write*** 方法可能阻塞的最长时间。

如果 RELIABILITY ***kind*** 设置为 RELIABLE，即使 DDS 中间件已经收到新的数据样本，由于通信错误而导致 ***DataReader*** 尚未接收到之前的数据样本，则 ***DataReader*** 无法接收新的数据样本。中间件服务将修复错误并重新传输所需的数据样本，以便在 ***DataReader*** 可以访问数据样本之前重新构建 ***DataWriter*** 历史数据的正确快照。

如果 RELIABILITY ***kind*** 设置为 BEST_EFFORT，则服务不会重新发送丢失的数据样本。但是对于源自任何一个 ***DataWriter*** 的数据样本，服务将确保它们以与它们在 ***DataWriter*** 中发出的顺序相同的顺序存储在 ***DataReader*** 历史记录中。换句话说，***DataReader*** 可能会丢失一些数据样本，但它永远不会看到数据对象的值从较新的值变为旧的值。

当且仅当不等式“提供 ***kind*** > = 请求 ***kind***”为“真”时，提供的值才被认为与所请求的值兼容。为了使这一不等式有意义，RELIABILITY QoS 类型的值被认为是有序的，即 BEST_EFFORT <RELIABLE 。

2.2.3.15 TRANSPORT_PRIORITY

此 QoS 的目的是允许应用程序能够利用发送不同优先级的消息的传输方式。

这项政策被视为一种暗示。策略取决于底层传输的能力，以便为它们发送的消息设置优先级。32 位有符号整数范围内的任何值都可能会被选择为优先级的取值，值越大表示优先级越高。对该策略的任何进一步解释都是针对特定的传输方式和中间服务的特定实现而言的。例如允许特定传输方式将一系列优先级值视为彼此等效。比较理想的情况是在传输配置期间，应用程序提供在 ***DataWriter*** 上设置的 TRANSPORT_PRIORITY 的值与对每种传输方式有意义的值之间的映射。底层实现将使用此映射传输 ***DataWriter*** 写入的数据。

2.2.3.16 LIFESPAN

此 QoS 的目的是避免向应用程序提供“陈旧”数据。

DataWriter 写入的每个数据样本都有一个关联的“到期时间”，超过该时间的数据不应提交给任何应用程序。一旦数据样本到期，数据将从 ***DataReader*** 缓存以及瞬态和持久性信息缓存中删除。

通过将 LIFESPAN QoS 指定的持续时间与***源时间戳（source timestamp）***相加来计算每个样本的“到期时间”。如 2.2.2.4.2.11 和 2.2.2.4.2.12 中所述，每次调用 ***DataWriter*** 的 ***write*** 方法时，***源时间戳（source timestamp）***由中间件服务自动计算或者由应用程序通过 ***write_w_timestamp*** 方法提供。

该 QoS 依赖于发送者和接收端应用程序尽量保持时钟同步。如果无法保证时间同步并

且服务可以检测到这一情况，则允许 ***DataReader*** 在计算“到期时间”时使用接收时间戳而不是源时间戳。

2.2.3.17 DESTINATION_ORDER

此策略控制每个订阅者如何解析由在不同节点上运行的多个 ***DataWriter*** 对象（可能与不同的 ***Publisher*** 对象关联）写入的数据实例的最终值。

假设 OWNERSHIP 策略允许，设置 BY_RECEPTION_TIMESTAMP 意味着实例的最新接收值应该是保留其值的值。这是默认的设置值。

假设 OWNERSHIP 策略允许，设置 BY_SOURCE_TIMESTAMP 意味着应使用在发送源设置的时间戳。在并发的场景下，相同强度的 ***DataWriter*** 对象更新同一实例，这是唯一一种设置确保所有订阅者都具有相同的实例最终值。设置源时间戳的机制取决于中间件的实现。

当且仅当不等式“提供 ***kind*** >= 请求 ***kind***”为“真”时，提供的值被认为与所请求的值兼容。为了使这一不等式有意义，DESTINATION_ORDER ***kind*** 的值被认为是有序的，定义如下：BY_RECEPTION_TIMESTAMP <BY_SOURCE_TIMESTAMP 。

2.2.3.18 HISTORY

1.当实例的值在最终传递给某些现有 ***DataReader*** 实体之前发生更改时，此策略控制服务的行为。

2.如果 ***kind*** 设置为 KEEP_LAST，则服务将仅尝试保留实例的最新值并丢弃旧的值。在这种情况下，深度 ***depth*** 会控制服务维护和交付的数据值（包括最新值）最大个数。深度 ***depth*** 的默认值（最常见的设置）是 1，表示服务只应传递最新值。

3.如果 ***kind*** 设置为 KEEP_ALL，则服务将尝试维护实例的所有值并将其传递给现有订阅者。服务保留这些历史数据的资源受 RESOURCE_LIMITS QoS 的设置限制。如果达到 RESOURCE_LIMITS QoS 的限制，则服务的行为将取决于 RELIABILITY QoS。如果可靠性类型为 BEST_EFFORT，则旧值将被丢弃。如果可靠性是 RELIABLE，那么服务将阻塞 ***DataWriter*** 直到它已经向所有的订阅者提供了需要的旧的数值。

HISTORY ***depth*** 的设置必须与 RESOURCE_LIMITS QoS 的 ***max_samples_per_instance*** 兼容。要使这两个 QoS 保持兼容，它们必须确保 ***depth*** <= ***max_samples_per_instance***。

2.2.3.19 RESOURCE_LIMITS

此策略控制服务可以使用的资源，以满足应用程序和其他 QoS 设置所提出的要求。

如果 ***DataWriter*** 对象的发送数据样本的速度比 ***DataReader*** 对象最终接收数据样本的速度快，那么中间件最终会达到 QoS 设置的某些资源限制。请注意当一个 ***DataReader*** 无法跟上其相应的 ***DataWriter*** 的速度时，便可能会发生这种情况。这种情况下中间件的行为取决于 RELIABILITY QoS 的设置。如果可靠性为 BEST_EFFORT，则允许中间件服务删除样本。如果可靠性是 RELIABLE，中间件服务将阻塞 ***DataWriter*** 或丢弃 ***DataReader*** 上的样本，以免丢失现有数据样本。

常数 LENGTH_UNLIMITED 代表不存在特殊限制。例如将 ***max_samples_per_instance*** 设置为 LENGH_UNLIMITED 将导致中间件不强制执行此特定限制。

RESOURCE_LIMITS ***max_samples*** 的设置必须与 ***max_samples_per_instance*** 兼容。要使这两个值保持兼容，必须确保“***max_samples*** >= ***max_samples_per_instance***”。

RESOURCE_LIMITS ***max_samples_per_instance*** 的设置必须与 HISTORY ***depeh*** 兼容。为了使这两个 QoS 保持兼容，必须确保“***depth*** <= ***max_samples_per_instance***”。

通过 ***set_qos*** 方法创建实体时将此策略设置为不兼容的值将导致方法调用失败。

2.2.3.20 ENTITY_FACTORY

此策略控制实体（***Entity***）作为其他实体的工厂的行为。

此策略仅涉及 ***DomainParticipant***（作为 ***Publisher***，***Subscriber*** 和 ***Topic*** 的工厂），***Publisher***（作为 ***DataWriter*** 的工厂）和 ***Subscriber***（作为 ***DataReader*** 的工厂）。

这项政策是可变的。策略的变化仅影响变化后创建的实体，不影响之前创建的实体。

将 ***autoenable_created_entities*** 设置为 TRUE 表示每次创建新实体时，工厂 ***create_ <entity>*** 方法将自动调用 ***enable*** 方法。因此，***create_ <entity>***方法返回的实体已经启用。设置为 FALSE 表示不会自动启用实体，应用程序需要通过 ***enable*** 方法显式启用它（参见 2.2.2.1.1.7）。

autoenable_created_entities = TRUE 的默认设置意味着默认情况下，不必在新创建的实体上显式调用 ***enable***。

2.2.3.21 WRITER_DATA_LIFECYCLE

此策略控制 ***DataWriter*** 关于其管理的数据实例的生命周期的行为，即使用 *register* 方法显式注册到 ***DataWriter*** 的数据实例（请参阅 2.2.2.4.2.5 和 2.2.2.4.2.6）或通过直接写入数据隐式地注册（见 2.2.2.4.2.11 和 2.2.2.4.2.12）。

autodispose_unregistered_instances 标志控制 ***DataWriter*** 通过 *unregister* 方法取消注册实例的行为（请参阅 2.2.2.4.2.7 和 2.2.2.4.2.8）：

- 设置“***autodispose_unregistered_instances*** = TRUE”会导致 ***DataWriter*** 在每次取消注册时都会处理掉该实例。该行为与在调用 *unregister* 方法之前在实例上显式调用 *dispose* 方法（2.2.2.4.2.13 和 2.2.2.4.2.14）相同。
- 设置“***autodispose_unregistered_instances*** = FALSE”意味着在取消注册时不会自动处理该实例。在取消注册实例之前，应用程序仍然可以调用 *dispose* 方法达到相同的效果。有关处理和取消注册实例的后果的说明，请参阅 2.2.3.23.3。

请注意，删除 ***DataWriter*** 会自动取消注册它管理的所有数据实例（2.2.2.4.1.6）。因此 ***autodispose_unregistered_instances*** 标志的设置将确定在通过 ***Publisher::delete_datawriter*** 方法直接删除 ***DataWriter*** 时是否最终处理实例，或者在包含 ***DataWriter*** 的 ***Publisher*** 或 ***DomainParticipant*** 上调用 ***delete_contained_entities*** 时间接删除实例。

2.2.3.22 READER_DATA_LIFECYCLE

此策略控制 ***DataReader*** 关于其管理的数据实例的生命周期的行为，即已接收的数据实例以及 ***DataReader*** 为其维持着某些内部资源的数据实例。

DataReader 内部维护应用程序未获取（***taken***）的样本，受其他 QoS 策略（如 HISTORY

和 RESOURCE_LIMITS）约束。

DataReader 还维护有关数据实例的特性、***view_state*** 和 ***instance_state*** 的信息，即使在所有样本都被“获取 ***taken***”之后也是如此。这是在未来样本到达时正确计算状态所必需的。

在正常情况下，***DataReader*** 只能回收没有写入者并且所有样本都被“获取”的实例的所有资源。***DataReader*** 对获取的该实例最后一个样本设置 ***instance_state*** 为 NOT_ALIVE_NO_WRITERS 或 NOT_ALIVE_DISPOSED，取决于拥有该实例所有权的最后一位数据写入者是否处理了该实例。有关 ***instance_state*** 可能转换的状态图，请参见图 2.11。如果没有 READER_DATA_LIFECYCLE QoS，应用程序“忘记”“获取”这些样本可能会导致问题。 “未获取”的数据样本将阻止 ***DataReader*** 回收资源，它们将无限期地保留在 ***DataReader*** 中。

autopurge_nowriter_samples_delay 定义 ***DataReader*** 在其 ***instance_state*** 变为 NOT_ALIVE_NO_WRITERS 后将维护有关实例的信息的最长持续时间。经过这段时间后，***DataReader*** 将清除有关该实例的所有内部信息，任何未获取的样本也将丢失。

autopurge_disposed_samples_delay 定义 ***DataReader*** 在其 ***instance_state*** 变为 NOT_ALIVE_DISPOSED 后将为实例维护样本的最长持续时间。经过这段时间后，***DataReader*** 将清除实例的所有样本。

2.2.3.23 注册，LIVELINESS 和 OWNERSHIP 之间的关系

注册/取消注册实例的需求源于两个场景：

- 冗余系统的所有权解决方案。
- 拓扑连接丢失的检测。

这两个场景还说明了 ***DataWriter*** 上的 ***unregister_instance*** 和 ***dispose*** 操作之间的语义差异。

2.2.3.23.1冗余系统的所有权解决方案

可以设想的场景是用户使用 DDS 来建立冗余系统，其中多个 ***DataWriter*** 实体“能够”更新相同的数据实例。在这种情况下，***DataWriter*** 实体配置为：

- 要么所有的 ***DataWriter*** 都在“不断”地更新写入实例；
- 要么他们使用某种机制将彼此分类为“主要”和“次要”，这样“主要”的 ***DataWriter*** 是唯一一个数据写入者，而“次要”的 ***DataWriter*** 监视“主要”的 ***DataWriter***，只有在检测到“主要”的 ***DataWriter*** 不再写入数据时“次要”的 ***DataWriter*** 才写入更新数据。

上述两种情况都使用 OWNERSHIP 策略的 EXCLUSIVE 类型并通过 OWNERSHIP_STRENGTH 进行仲裁。无论方案如何，从 ***DataReader*** 的角度来看的行为是读取者通常接收从 “主要”的 ***DataWriter*** 发出的数据，除非“主要”写入者停止写入数据，在这种情况下，读取者接收从“次要”的 ***DataWriter*** 发出的数据。

这种方法需要一些机制来检测出 ***DataWriter***（主要的）不再按规定“写入”数据的情况。

有以下几个原因导致这种情况发生，这种情况必须被检测出，但不一定要区分具体是什么原因：

1.[崩溃] - 写数据进程不再运行（例如整个应用程序崩溃）。

2.[连接丢失] - 写数据应用程序的连接已丢失（例如网络断开连接）。

3.[应用程序故障] - 正在写入数据的应用程序出现故障，并且已停止调用 ***DataWriter*** 的“***write***”方法。

仲裁选择更高强度的 ***DataWriter*** 很简单，***DataReader*** 可以自主决定。将所有权从更高强度的 ***DataWriter*** 切换到较低强度的 ***DataWriter*** 要求 ***DataReader*** 可以确定更高强度的 ***DataWriter***“不再更新写入实例数据”。

2.2.3.23.1.1 数据周期性更新的场景

当以某种速率周期性写入数据时，这种数据实例的所有权确定相当简单。***DataWriter*** 简单地声明其提供的 DEADLINE（两次数据更新之间的最大间隔），***DataReader*** 自动监视 ***DataWriter*** 每个截止时间内是否至少更新一次实例。如果截止时间内 ***DataWriter*** 没有更新过数据，***DataReader*** 会认为 ***DataWriter***“不存活”，并自动将所有权分配给存活的最高强度的 ***DataWriter***。

2.2.3.23.1.2 数据非周期性更新的场景

DataWriter 非周期写入数据也是一种非常重要的场景。由于实例不会在固定期间内更新数据，因此“截止时间”机制不能用于确定所有权。存活性在种情况下便可解决问题。***DataWriter*** 为“存活”状态时保持实例的所有权，***DataWriter*** 为了处于存活状态必须履行其“LIVELINESS”QoS 的合约。每次写入数据时隐含的更新组合起来便会更新存活性（自动，手动），这些不同方法会处理上述[崩溃]，[连接丢失]和[应用程序故障]的三种场景。请注意要处理[应用程序故障] ，LIVELINESS 必须设置为 MANUAL_BY_TOPIC。***DataWriter*** 可以通过定期写入数据或者在没有数据需要写入时调用 ***assert_liveliness*** 来保留所有权。如果只需要防止[崩溃]或[连接丢失]，则只需要写入者进程上的某些任务定期写入数据或者调用 ***DomainParticipant*** 的 ***assert_liveliness*** 方法即可。

然而此方案要求 ***DataReader*** 知道 ***DataWriter*** 正在“写入”哪些数据实例。这是 ***DataReader*** 从 ***DataWriter*** 仍然“存活”的事实中推断出特定实例的所有权的唯一方法。因此写入者需要“注册”和“注销”实例。虽然 ***DataWriter*** 第一次写实例时可以懒惰地“注册”，但“注销”通常不能。类似的原因导致注销也需要将消息发送给读取者。

2.2.3.23.2拓扑连接丢失的检测

有些应用程序的设计使得它们的正确操作需要最小拓扑连接，即写入者需要具有最少数量的读取者，或者读取者必须具有最少数量的写入者。

一种常见的情况是，应用程序在知道某些特定写入者具有最小配置读取者（例如告警监视器已启动）之前不会开始执行其逻辑。

更常见的情况是应用程序逻辑将等待直到某些写入者出现，这些写入者可以提供一些所需的信息源（例如必须处理的原始传感器数据）。

此外一旦应用程序运行，则要求监视（来自数据源的）最小连接，并且如果该连接丢失则通知应用程序。对于周期性写入数据的情况，DEADLINE QoS 和 ***on_deadline_missed*** 监听器提供通知。数据非周期更新的情况需要将 LIVELINESS 与注册/注销实例结合使用以检测“连接”是否已丢失，并且通过“NO_WRITERS”视图状态提供通知。

就所需机制而言，该场景与维护所有权的情况非常相似。在这两种情况下，读取者都需要知道写入者是否仍在“管理实例的当前值”，即使写入者没有在不断地写入数据。这需要写入者保持其存活性并具有了解写入者当前管理哪些实例（即注册的实例）的方法。

2.2.3.23.3unregister_instance 和 dispose 之间的语义差异

DataWriter 的 ***dispose*** 方法在语义上与 ***unregister_instance*** 方法不同。***dispose*** 方法表明数据实例不再存在（例如已消失的踪迹，已销毁的模拟实体，已删除的记录条目等），而 ***unregister_instance*** 方法表明写入者不再负责更新实例的值。

删除 ***DataWriter*** 等同于注销它正在写入更新的所有实例，但与“处理”所有实例不同。

对于具有 EXCLUSIVE OWNERSHIP 的主题，如果实例的当前所有者处理了该实例，访问该实例的读取者将看到 ***instance_state*** 为“DISPOSED”而且不会看到所有权强度值较弱的写入者编写的值（即使在所有权较强的写入者处理了实例之后）。这是因为拥有该实例的 ***DataWriter*** 说该实例不再存在（例如数据库的所有人说该记录已被删除），读取者应该这样理解。

对于具有 EXCLUSIVE OWNERSHIP 的主题，如果实例的当前所有者注销掉该实例，那么它将放弃实例的所有权，因此读取者可能会收到由另一个写入者（新的所有者）更新的值。这是因为旧的所有权强度最大的所有者说它不再为实例提供值，因此另一个写入者可以获得所有权并提供这些数据值。

2.2.4 Listeners, Conditions 和 Wait-sets

监听器和条件（与等待集一起使用）是两种可选机制，允许应用程序了解 DCPS 通信状态的变化。

2.2.4.1 通信状态

依赖于实体 ***Entity*** 可以将通信状态的改变传递给应用程序。下表显示了每个实体相关的状态。

实体 Entity	**状态名称**	**含义**
主题（Topic）	INCONSISTENT_TOPIC	与另一个主题同名但具有不同的特征
订阅者（Subscriber）	DATA_ON_READERS	有新信息
数据读取者（DataReader）	SAMPLE_REJECTED	（已接收的）数据样本被拒绝
	LIVELINESS_CHANGED	更新（通过 ***DataReader*** 读取）实例的一个或多个 ***DataWriter*** 的存活性已发生变化。某些 ***DataWriter*** 已变为“活动”或“非活动”
	REQUESTED_DEADLINE_MISSED	对于特定实例，***DataReader*** 通过其 DEADLINE QosPolicy 设置的期望截止时间未得到遵守
	REQUESTED_INCOMPATIBLE_QOS	需要的 QoS 策略与提供的 QoS 策略不匹配
	DATA_AVAILABLE	有新信息

	SAMPLE_LOST	（未接收的）数据样本丢失
	SUBSCRIPTION_MATCHED	*DataReader* 发现有 *DataWriter* 与他的 *Topic* 匹配并具有兼容的 QoS;或者 *DataReader* 已经不再与之前认为匹配的 *DataWriter* 匹配
数据写入者（DataWriter）	LIVELINESS_LOST	***DataWriter*** 通过 LIVELINESS QosPolicy 所声明的存活性并未得到满足; ***DataReader*** 实体将把 ***DataWriter*** 视为不再“活跃”
	OFFERED_DEADLINE_MISSED	***DataWriter*** 通过 DEADLINE QosPolicy 设定的针对特定实例的截止时间未得到满足
	OFFERED_INCOMPATIBLE_QOS	提供的 QoS 策略与需要的 QoS 策略不匹配
	PUBLICATION_MATCHED	*DataWriter* 发现有 *DataReader* 与他的 *Topic* 匹配并具有兼容的 QoS；或者 *DataWriter* 已经不再与之前被认为匹配的 *DataReader* 匹配

这些状态可分为：

•读取通信状态：与数据到达相关的状态，即 DATA_ON_READERS 和 DATA_AVAILABLE。

•普通的通信状态：所有其他状态。

读取通信状态的处理方式与其他状态略有不同，因为它们不会独立更改。换句话说，至少两种更改（DATA_ON_READERS＋DATA_AVAILABLE）将同时出现，甚至后一种更改也可能是该集合的一部分。必须将此“分组”传达给应用程序。如何做到这一点将在以下两个子章节中讨论。

对于每个普通通信状态，存在保持状态值的相应结构。这些值包含与状态变化相关的信息，以及与状态本身相关的信息（例如包含累积计数）。它们与以下子章节中解释的两种不同机制一起使用。

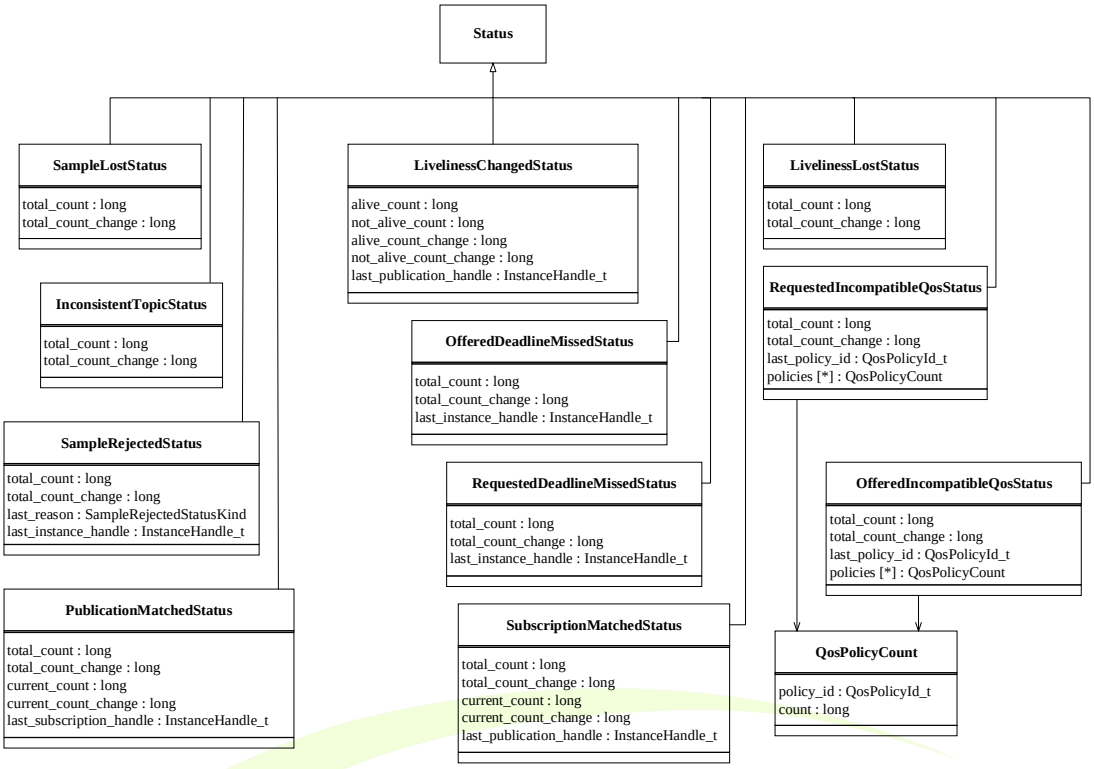

图 2-13 状态值

下表中提供了每个状态值的属性解释。

SampleLostStatus	属性含义
total_count	所有在某一主题下发布的所有数据实例中丢失的样本累积计数
total_count_change	自上次调用监听器或读取状态以来样本丢失的增加量
SampleRejectedStatus	属性含义
total_count	DataReader 拒绝的样本累计总数
total_count_change	自上次调用监听器或读取状态以来样本拒绝的增加量
last_reason	拒绝被拒绝的最后一个样本的原因，如果没有样本被拒绝，原因是特殊值 NOT_REJECTED
last_instance_handle	被拒绝的最后一个样本正在更新的实例的句柄
InconsistentTopicStatus	属性含义
total_count	发现的主题的累计总数，其名称与此状态附加的主题相匹配，类型与主题不一致。
total_count_change	自上次调用监听器或读取状态以来发现的不一致主题的增加量
LivelinessChangedStatus	属性含义
alive_count	当前活动的 DataWriters 的总数，这些 DataWriters 写入供 DataReader 读取的 Topic 数据。当新匹配的 DataWriter 第一次声明其存活性或者之前被认为失活的 DataWriter 重新确认其存活性时，此计数会增加。

	当被认为活着的 DataWriter 无法声明其存活性并且变得不活跃时，计数会减少，无论是因为它被正常删除还是出于其他原因
not_alive_count	当前不再声明其存活性的 DataWriters 的总数，这些 DataWriters 写入供 DataReader 读取的 Topic 数据。当一个被认为活着的 DataWriter 无法断言其存活性并且由于某种原因而不是正常删除该 DataWriter 而变得不再存活时，此计数会增加。当先前不存活的 DataWriter 重新声明其存活性或正常删除时，它会减少
alive_count_change	自上次调用监听器或读取状态以来 alive_count 的更改
not_alive_count_change	自上次调用监听器或读取状态以来 not_alive_count 的更改
last_publication_handle	最后一个存活性改变导致此状态改变的 DataWriter 的句柄
RequestedDeadlineMissedStatus	属性含义
total_count	检测到的任一为 DataReader 读取的实例的错过截止时间的累计次数。错过的截止时间会积攒; 也就是说对于没有收到数据的每个实例，每个截止时间周期 *total_count* 将增加 1
total_count_change	自上次调用监听器或读取状态以来增加的检测到错过截止时间的次数
last_instance_handle	DataReader 中检测到截止时间丢失的最后一个实例的句柄
RequestedIncompatibleQosStatus	属性含义
total_count	相关 DataReader 针对同一主题发现 DataWriter 的总累计次数，该 DataWriter 提供的 QoS 与 DataReader 请求的 QoS 不兼容
total_count_change	自上次调用监听器或读取状态以来 total_count 的更改
last_policy_id	上次检测到 QoS 不兼容时发现的不兼容 QoS 的 QosPolicyId_t
policies	包含每个策略的列表，表示相关 DataReader 针对同一主题发现 DataWriter 的总次数，该 DataWriter 提供的 QoS 与 DataReader 请求的 QoS 不兼容
LivelinessLostStatus	属性含义
total_count	由于未能在其提供的存活周期内主动发出存活信号，之前存活的 DataWriter 变为非存活的总累计次数。当一个已经不存活的 DataWriter 在另一个存活周期内仍然没有变为存活时，这个计数不会改变
total_count_change	自上次调用监听器或读取状态以来 total_count 的更改
OfferedDeadlineMissedStatus	属性含义
total_count	提供的截止时间超期的总累积次数，在该周期内 DataWriter 没有写入数据。错过的截止时间积攒，也就是说，每出现一次截止时间超期，total_count 将增加 1

total_count_change	自上次调用监听器或读取状态以来 total_count 的更改
last_instance_handle	DataWriter 中错过提供的截止时间的最后一个实例的句柄
OfferedIncompatibleQosStatus	属性含义
total_count	相关 DataWriter 针对同一主题发现 DataReader 的总累计次数，该 DataReader 请求的 QoS 与 DataWriter 提供的 QoS 不兼容
total_count_change	自上次调用监听器或读取状态以来 total_count 的更改
last_policy_id	上次检测到 QoS 不兼容时发现的不兼容 QoS 的 QosPolicyId_t
policies	包含每个策略的列表，表示相关 DataWriter 针对同一主题发现 DataReader 的总次数，该 DataReader 请求的 QoS 与 DataWriter 提供的 QoS 不兼容
PublicationMatchedStatus	属性含义
total_count	相关 DataWriter 发现一个“匹配”的 DataReader 的总累计次数。也就是说针对同一主题该 DataWriter 找到了一个 DataReader，其请求的 QoS 与 DataWriter 提供的 QoS 兼容
total_count_change	自上次调用监听器或读取状态以来 total_count 的更改
last_subscription_handle	导致状态发生变化的与 DataWriter 匹配的最后一个 DataReader 的句柄
current_count	当前与相关 DataWriter 匹配的 DataReader 的数目
current_count_change	自上次调用监听器或读取状态以来 current_count 的更改
SubscriptionMatchedStatus	属性含义
total_count	相关 DataReader 发现一个“匹配”的 DataWriter 的总累计次数。也就是说针对同一主题该 DataReader 找到了一个 DataWriter，其提供的 QoS 与 DataReader 请求的 QoS 兼容
total_count_change	自上次调用监听器或读取状态以来 total_count 的更改
last_publication_handle	导致状态发生变化的与 DataReader 匹配的最后一个 DataWriter 的句柄
current_count	当前与相关 DataReader 匹配的 DataWriter 的数目
current_count_change	自上次调用监听器或读取状态以来 current_count 的更改

2.2.4.2 状态更改

与实体的每个通信状态相关联的是一个逻辑的 ***StatusChangedFlag***。此标志代表自上次应用程序“读取”状态以来该特定通信状态是否已更改。对于普通通信状态和读取通信状态，状态更改的方式略有不同。

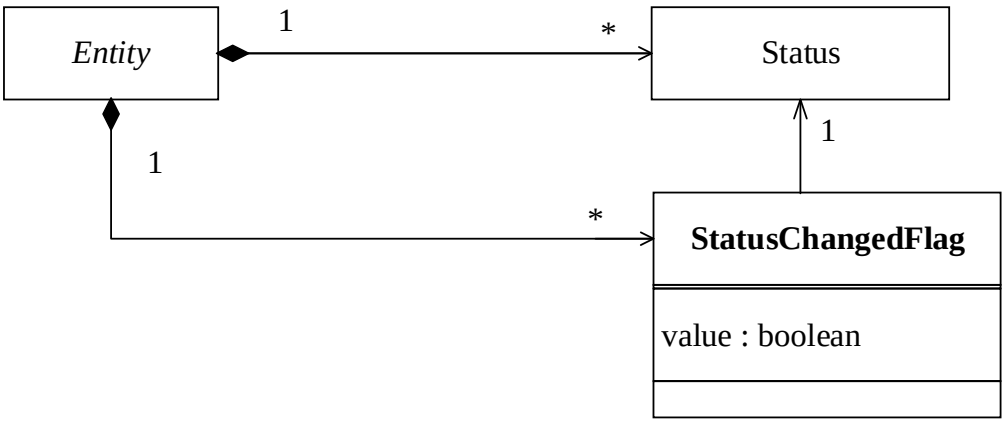

图 2-14 StatusChangedFlag 代表状态是否发生改变

请注意，图 2.14 仅是概念性的，它只是表示实体知道哪些特定状态已更改，并不意味着以布尔值这种方式完成特定的 ***StatusChangedFlag*** 实现。

2.2.4.2.1 普通通信状态的变化

对于普通通信状态，***StatusChangedFlag*** 标志最初设置为 FALSE。每当普通通信状态改变时它变为 TRUE，并且每当应用程序通过实体的 ***get_ <plain communication status>***方法访问普通通信状态时，它变为 FALSE。

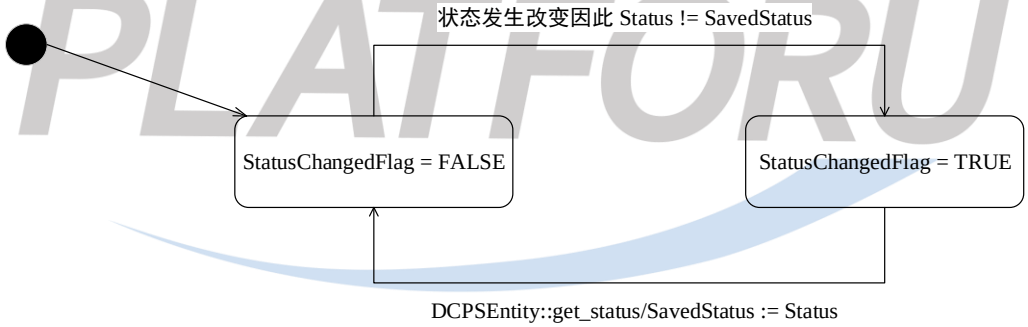

图 2-15 普通通信状态 StatusChangedFlag 标志改变活动图

每当调用关联的监听器方法时，通信状态也会重置为 FALSE，因为监听器会隐式访问作为参数传递给方法的状态。在调用监听器之前重置状态的事实意味着如果应用程序从监听器内部调用 ***get_ <plain communication status>***，它将看到状态已经重置。

此规则的一个例外是关联的监听器是“nil”监听器。如 2.2.4.3.1 中所述，“nil”监听器被作为 NOOP 处理，并且调用“nil”监听器的行为不会重置通信状态。

例如每次截止时间到期时，与 REQUESTED_DEADLINE_MISSED 状态关联的 ***StatusChangedFlag*** 的值将变为 TRUE（这会增加 ***RequestedDeadlineMissedStatus*** 中的 ***total_count*** 字段）。当应用程序通过适当的实体上的相应 ***get_requested_deadline_missed_status*** 方法访问状态时，该值更改为 FALSE。

2.2.4.2.2 读取通信状态的变化

对于读取通信状态，***StatusChangedFlag*** 标志最初设置为 FALSE。

当数据样本到达时，***StatusChangedFlag*** 变为 TRUE，或者任何现有样本的 ***ViewState***，

SampleState 或 ***InstanceState*** 因调用 ***DataReader*** :: **read**，***DataReader*** :: **take** 或其变体之外的任何原因而更改，***StatusChangedFlag*** 也会变为 TRUE。具体而言，以下任一事件都将导致 ***StatusChangedFlag*** 变为 TRUE：

- 新数据的到来。
- 包含实例的 ***InstanceState*** 的更改。这可能是由以下原因引起的：
  - 实例已被处理的通知到达，处理该实例的 ***DataWriter*** 为以下两种之一：
    - ***OWNERSHIP QoS kind***= EXLUSIVE，则为拥有该实例的 DataWriter；
    - ***OWNERSHIP QoS kind***=SHARED 的任何 DataWriter。
  - 只有一个 ***DataWriter*** 的实例，该 ***DataWriter*** 的存活性丢失。
  - 实例已由唯一已知正在更新写入实例数据的 ***DataWriter*** 注销。

根据 ***StatusChangedFlag*** 的类型，以下情况导致标志变为 FALSE：

- 当调用相应的监听器方法（***on_data_available***）或在关联的 ***DataReader*** 上调用 ***read*** 或 ***take*** 方法（或其变体）时，DATA_AVAILABLE ***StatusChangedFlag*** 变为 FALSE。
- 发生以下任一事件时，DATA_ON_READERS ***StatusChangedFlag*** 变为 FALSE：
  - 调用相应的监听器操作（***on_data_on_readers***）。
  - 在属于订阅者 ***Subscriber*** 的任一 ***DataReader*** 上调用 ***on_data_available*** 监听器方法。
  - 在属于订阅者 ***Subscriber*** 的任一 ***DataReader*** 上调用 ***read*** 或 ***take*** 方法（或其变体）。

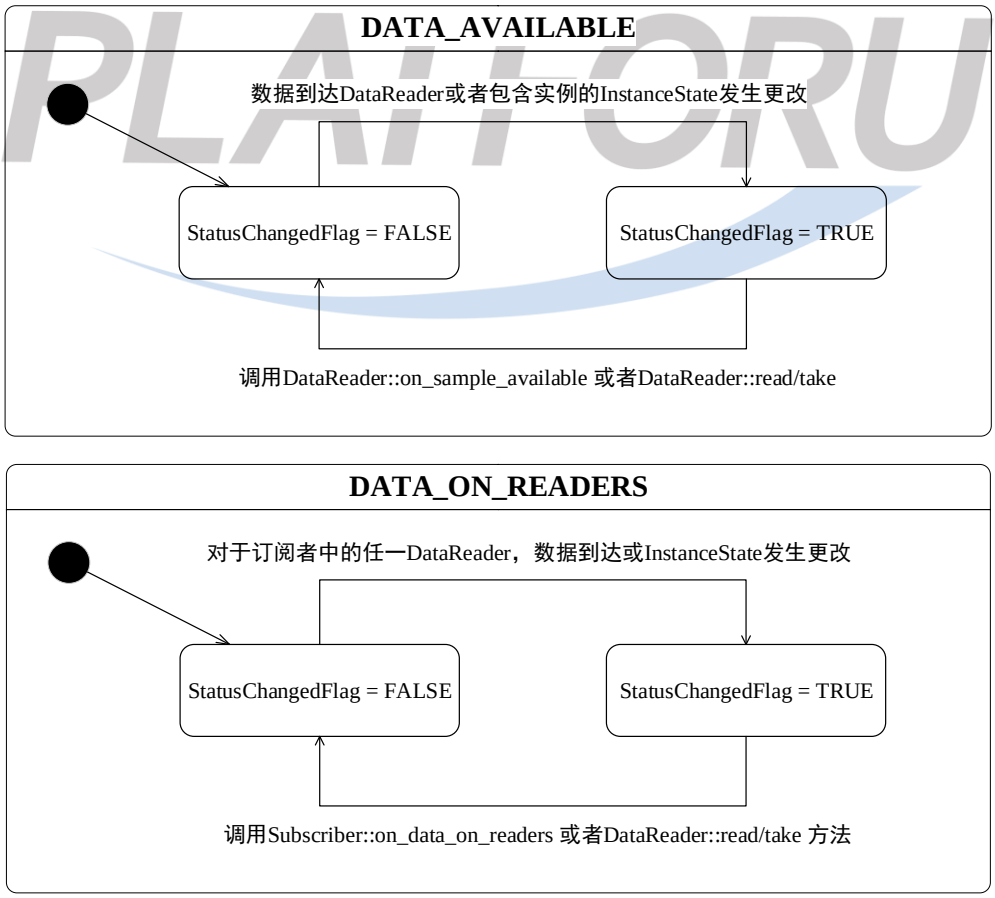

图 2-16 读取通信状态 StatusChangedFlag 标志改变活动图

2.2.4.3 通过监听器访问

监听器为中间件提供了一种机制，使得中间件可以异步地向应用程序发出相关状态更改的警报。

所有实体（***Entity***）都支持一个监听器，该监听器的类型专用于相关实体的特定类型（例如 ***DataReader*** 的 ***DataReaderListener***）。监听器是应用程序必须实现的接口。每个专用监听器呈现对应于相关通信状态变化的方法列表（即应用程序可以对其作出反应）。

图 2.17 中列出了所有监听器，这些监听器与参与此机制的 DCPS 构造相关联（请注意仅显示相关方法）。

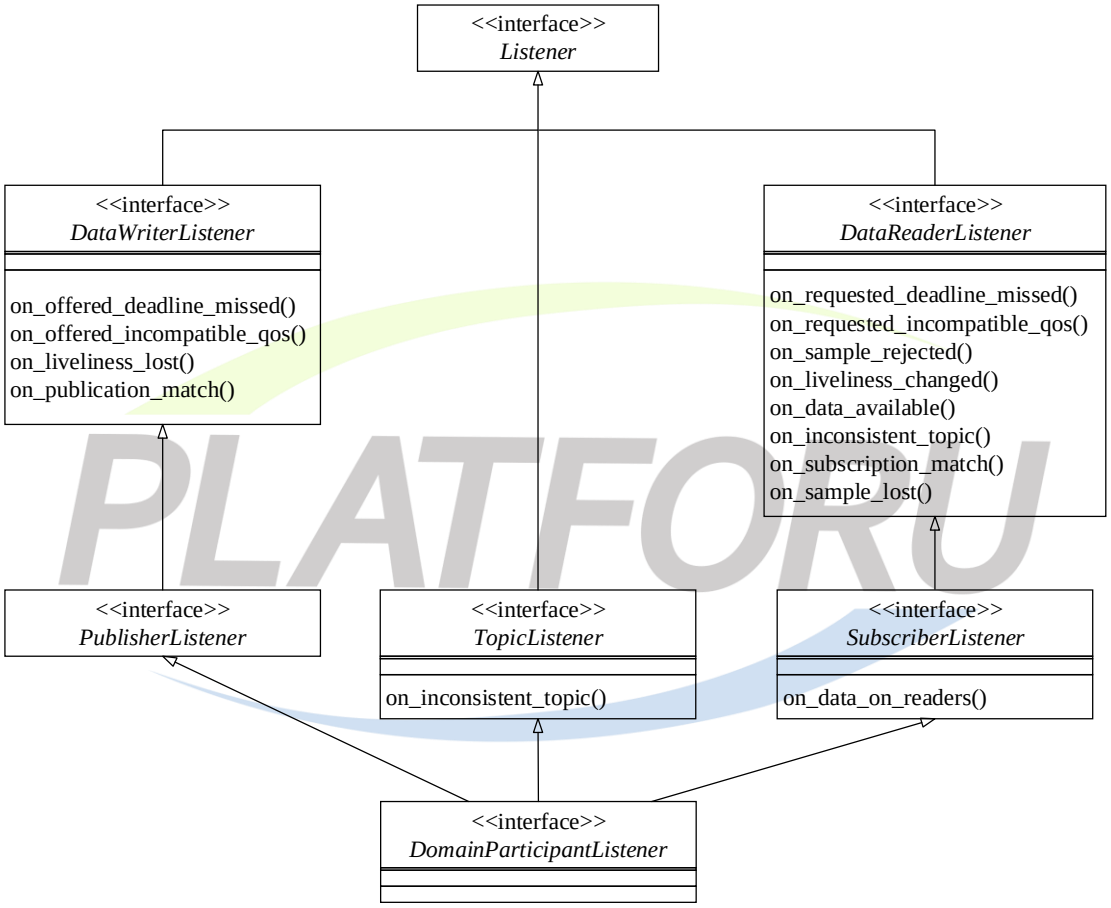

图 2-17 支持的 DCPS 监听器

监听器是无状态的，因此可以在所有 ***DataReader*** 对象上共享相同的 ***DataReaderListener*** 实例（假设它们将对类似的状态更改做出类似的反应）。 因此提供的参数包含对相关实际实体的引用。

2.2.4.3.1 监听器访问普通通信状态

2.2.4.1 中解释的普通通信状态与监听器方法之间的一般映射如下：

•对于每种通信状态，都有一个名称为 ***on_<communication_status>***的相应方法，该方法采用 2.2.4.1 通信状态中列出的***<communication_status>***类型的参数。

•***on_<communication_status>***在相关实体以及嵌入它的实体上可用，如下图所示：

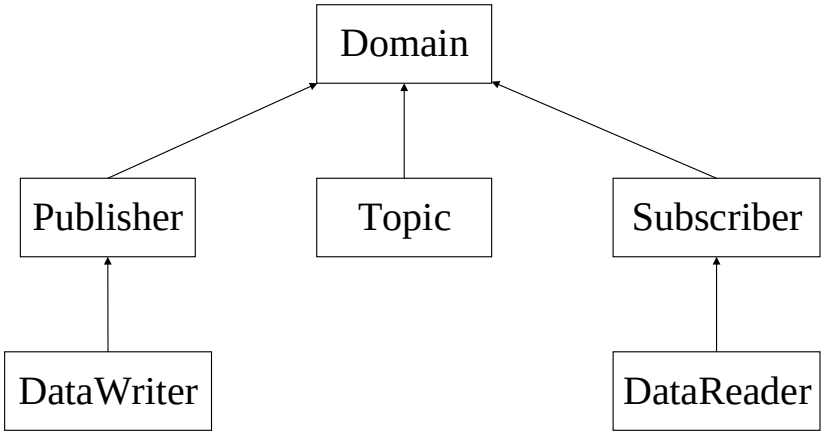

图 2-18 监听器访问普通通信状态实体关系图

•当应用程序在实体上附加监听器时，它必须设置一个掩码，向中间件指示在此监听器中启用了哪些方法（请参阅 ***Entity :: set_listener*** 方法）。

•当普通通信状态发生变化时，中间件会触发已启用的最“特定”相关的监听器方法。如果最特定的相关监听器方法对应于应用程序安装的“nil”监听器，则方法将被视为由 NO-OP 方法处理。

此行为允许应用程序设置默认行为（例如在与 ***DomainParticipant*** 关联的监听器中）并仅在需要时设置专用行为。

2.2.4.3.2 监听器访问读取通信状态

与数据到达相关的两种状态的处理方式略有不同，由于它们构成了数据分发服务（DDS）的真实目的，因此没有必要提供普通通信状态的默认机制。更重要的是如 2.2.4.1 通信状态所述，其中一些状态可能需要作为一个整体来对待。

规则如下。每次读取通信状态发生改变时：

•首先，中间件尝试使用相关订阅者 ***Subscriber*** 的参数触发 ***SubscriberListener*** 的 ***on_data_on_readers*** 方法;

•如果上述操作不成功（没有监听器或方法未启用），它会尝试在所有相关 ***DataReaderListener*** 对象上调用 ***on_data_available***，并使用相关 ***DataReader*** 作为参数。

理由是应用程序对数据到达的关系感兴趣并且必须使用第一个选项（通过调用相关 ***Subscriber*** 上的 ***get_datareaders*** 获取相应的 ***DataReader*** 对象，然后通过调用返回的 ***DataReader*** 上的 ***read / take*** 方法来获取数据对象）；如果它想要完全独立地处理每个 ***DataReader***，它可以选择第二个选项（通过调用相关 ***DataReader*** 上的 ***read / take*** 来获取数据）。

请注意，如果调用 ***on_data_on_readers***，则中间件不会尝试调用 ***on_data_available***，但是应用程序可以通过 ***notify_datareaders*** 方法强制调用具有数据的 ***DataReader*** 对象。

在监听器的调用中没有隐含的“事件排队”，即如果相同类型的状态变化顺序发生，则 DCPS 实现不必在每个“单元”变化传递一个监听器回调。例如可能会出现 DCPS 实现发现 ***DataReader*** 的存活性已经发生变化，因为出现了几个匹配的 ***DataWriter*** 实体;在这种情况下，只要提供给监听器的 ***LivelinessChangedStatus*** 对应于最新的一个，DCPS 实现可以选择仅调用 ***DataReaderListener*** 上的 ***on_liveliness_changed*** 方法一次。

2.2.4.4 条件和等待集

如前所述，条件（与等待集一起）提供了一种替代机制，允许中间件将通信状态更改（包括数据到达）传递给应用程序。

图 2.19-等待集和条件显示该机制中涉及的所有 DCPS 组成部分（请注意仅显示相关方法）。

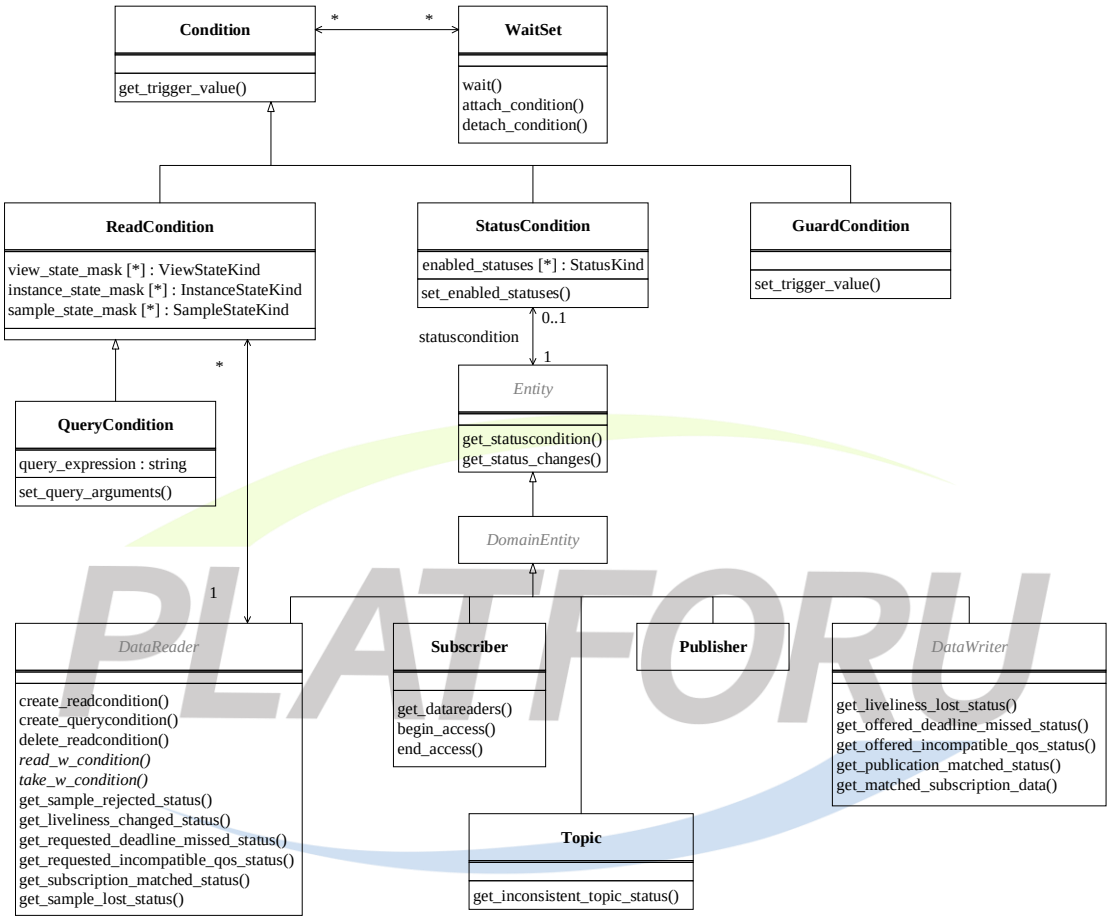

图 2-19 条件和等待集

这种机制是基于等待的。其一般使用模式如下：

- 应用程序通过创建 ***Condition*** 对象（***StatusCondition***，***ReadCondition*** 或 ***QueryCondition***）并将它们添加到 ***WaitSet***，通过以上方法指示它想要获取哪些相关信息。
- 然后等待该 ***WaitSet***，直到一个或多个 ***Condition*** 对象的 ***trigger_value*** 变为 ***TRUE***。
- 然后使用等待的结果（即具有 ***trigger_value == TRUE*** 的 ***Condition*** 对象列表）来实际获取信息，通过调用以下方法来实现：
  - ***get_status_changes*** 以及相关实体的 ***get_ <communication_status>***。如果条件是 ***StatusCondition*** 并且状态改变了，参考普通通信状态。
  - ***get_status_changes*** 以及相关订阅者的 ***get_datareaders***。如果条件是 ***StatusCondition*** 并且状态改变了，参考 DATA_ON_READERS。
  - ***get_status_changes*** 然后调用相关 ***DataReader*** 的 ***read***/***take*** 方法。如果条件是 ***StatusCondition*** 并且状态改变了，参考 DATA_AVAILABLE。
- 如果是 ***ReadCondition*** 或 ***QueryCondition***，则直接调用 ***DataReader*** 的

read_w_condition/take_w_condition 方法，并将 ***Condition*** 作为参数。

通常，第一步是在初始化阶段完成，而其他步骤则放在应用程序主循环中。

由于在等待返回时没有额外的信息从中间件传递到应用程序（除了触发的 ***Condition*** 对象列表），因此 ***Condition*** 对象嵌入了条件触发时应用程序做出正确响应所需的所有信息。特别地，与实体相关的条件仅仅与一个实体相关并且不能共享。

WaitSet 的阻塞行为如图 2.20 所示。***wait*** 方法的结果取决于 ***WaitSet*** 的状态，而 ***WaitSet*** 的状态又取决于至少一个附加的 ***Condition*** 的 ***trigger_value*** 是否为 TRUE。如果在状态为 BLOCKED 的 ***WaitSet*** 上调用 ***wait*** 方法，它将阻塞调用线程。如果在状态为 UNBLOCKED 的 ***WaitSet*** 上调用 ***wait***，它将立即返回。此外，当 ***WaitSet*** 从 BLOCKED 转换为 UNBLOCKED 时，它会唤醒任一调用 ***wait*** 的线程。

类似于监听器的调用，在 ***WaitSet*** 的唤醒中没有隐含的“事件排队”，即如果附加到 ***WaitSet*** 的几个条件的 ***trigger_value*** 按顺序转换为 TRUE，则 DCPS 实现只需要解除一次对 ***WaitSet*** 的阻塞。

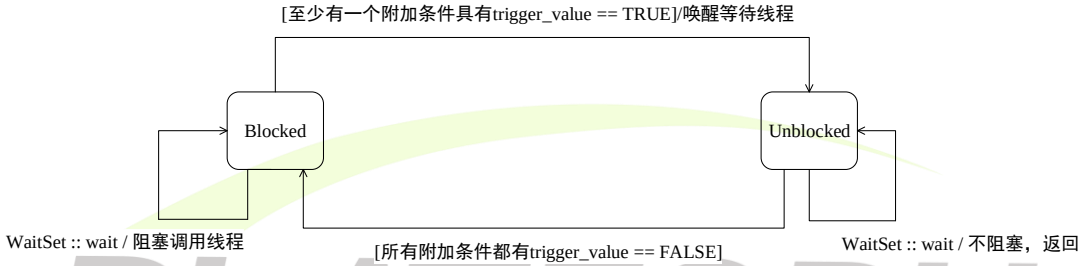

图 2-20 WaitSet 的阻塞行为

Condition/WaitSet 机制的一个关键方面是设置每个 ***Condition*** 的 ***trigger_value***。

2.2.4.4.1 StatusCondition 的触发状态

StatusCondition 的 ***trigger_value*** 是对所有通信状态敏感的 ***ChangedStatusFlag*** 的布尔值。也就是说，仅当 ***ChangedStatusFlags*** 的所有值都为 FALSE 时，***trigger_value*** == FALSE。

StatusCondition 对特定通信状态的敏感性由通过 ***set_enabled_statuses*** 方法在条件上设置的 ***enabled_statuses*** 列表控制。

2.2.4.4.2 ReadCondition 的触发状态

与 ***StatusCondition*** 类似，***ReadCondition*** 也有一个 ***trigger_value***，用于确定附加到的 ***WaitSet*** 是 BLOCKED 还是 UNBLOCKED。与 ***StatusCondition*** 不同的是 ***ReadCondition*** 的 ***trigger_value*** 与至少一个由中间件服务管理的样本相关联，该样本的 ***SampleState***，***ViewState*** 和 ***InstanceState*** 与 ***ReadCondition*** 的样本匹配。此外为了使 ***QueryCondition*** 的 ***trigger_value*** == TRUE，与样本关联的数据必须使 ***query_expression*** 为 TRUE。

ReadCondition 的 ***trigger_value*** 依赖于相关 ***DataReader*** 上样本的存在这一事实意味着简单的 ***take*** 方法可能会改变多个 ***ReadCondition*** 或 ***QueryCondition*** 的 ***trigger_value***。例如，如果所有数据样本被获取，则任一之前 ***trigger_value***== TRUE 的与 ***DataReader*** 关联的 ***ReadCondition*** 和 ***QueryCondition*** 都会将 ***trigger_value*** 更改为 FALSE。请注意这并不能保证独立地附加到这些条件的 ***WaitSet*** 对象不会被唤醒。一旦我们在某个条件下触发了 ***trigger_value*** == TRUE，它就可以唤醒附加的 ***WaitSet***，转换为 trigger_value == FALSE 的条

件不一定会“取消唤醒”***WaitSet***，因为“取消唤醒”通常是不可能的。结果是在 ***WaitSet*** 上阻塞的应用程序可能会从等待中返回一个条件列表，其中一些条件不再“活动”。如果多个线程同时等待单独的 ***WaitSet*** 对象并获取与同一个 ***DataReader*** 实体相关的数据，上述情况则为不可避免的。

为了进一步详细说明，请考虑以下示例：具有 ***sample_state_mask*** = {NOT_READ}的 ***ReadCondition*** 将在新样本到达时 ***trigger_value*** 为 TRUE，一旦所有新到达的样本被读取（它们的状态更改为 READ）或获取（它们不再由服务管理），则 ***ReadCondition*** 的 ***trigger_value*** 将立即转换为 FALSE。但是如果相同的 ***ReadCondition*** 的 ***sample_state_mask*** = {READ，NOT_READ}，那么只有所有新到达的样本被获取后，***trigger_value*** 才变为 FALSE（这不足以读取它们，因为这只会将 ***SampleState*** 更改为 READ，重叠 ***ReadCondition*** 上的掩码）。

2.2.4.4.3 GuardCondition 的触发状态

GuardCondition 的 ***trigger_value*** 完全由应用程序通过 ***set_trigger_value*** 方法控制。

2.2.4.5 组合

这两种机制可以在应用程序中组合（例如使用等待集和条件来访问数据，使用监听器以异步地警告错误的通信状态）。

应用程序可能会为每个特定的通信状态选择一种或另一种机制（而不是两者都采用）。但是，如果同时启用了这两种机制，则首先使用监听器机制，然后发出 ***WaitSet*** 对象的信号。

2.2.5 内置主题（Built-in Topics）

作为其方法的一部分，中间件必须发现并尽可能跟踪远程实体（例如域中的新参与者）的存在。此信息对于应用程序也很重要，应用程序可能希望对此发现做出反应，或者根据需要对其进行访问。

为了使应用程序可以访问此信息，DCPS 规范引入了一组内置主题和相应的 ***DataReader*** 对象，应用程序可以使用这些对象获取所需信息，就好像它是正常的应用程序数据一样。此方法避免引入新 API 来访问这些信息，并允许应用程序通过 2.2.4 监听器，条件和等待集中提供的机制获知这些值中的任一更改。

内置数据读取者都属于内置订阅者（***Subscriber***）。可以使用 ***DomainParticipant*** 提供的 ***get_builtin_subscriber*** 方法获取此订阅者对象。可以使用 ***lookup_datareader*** 方法查找内置 ***DataReader*** 对象，使用 ***Subscriber*** 对象和主题名称作为参数。

内置 ***Subscriber*** 和 ***DataReader*** 对象的 QoS 如下表所示：

USER_DATA	<未指定>
TOPIC_DATA	<未指定>
GROUP_DATA	<未指定>
DURABILITY	TRANSIENT_LOCAL
DURABILITY_SERVICE	不适用，因为 DURABILITY 是 TRANSIENT_LOCAL
PRESENTATION	access_scope = TOPIC coherent_access = FALSE

	ordered_access = FALSE
DEADLINE	Period = 无限
LATENCY_BUDGET	duration = <未指定>
OWNERSHIP	SHARED
LIVELINESS	kind = AUTOMATIC lease_duration = <未指定>
TIME_BASED_FILTER	minimum_separation = 0
PARTITION	<未指定>
RELIABILITY	kind = RELIABLE max_blocking_time = 100 milliseconds
DESTINATION_ORDER	BY_RECEPTION_TIMESTAMP
HISTORY	kind = KEEP_LAST depth = 1
RESOURCE_LIMITS	All LENGTH_UNLIMITED
READER_DATA_LIFECYCLE	autopurge_nowriter_samples_delay = 无限 autopurge_disposed_samples_delay = 无限
ENTITY_FACTORY	autoenable_created_entities = TRUE

内置实体也具有默认监听器设置。内置的订阅者及其所有内置主题都有 nil 个侦听器，其所有状态都出现在其监听器掩码中。内置的 ***DataReaders*** 具有 nil 监听器，其掩码中没有状态。

通过内置主题可访问的有关远程实体的信息包括远程实体的使用的所有 QoS 策略。此 QoS 策略在通过内置主题读取的数据内显示为正常的“数据”字段。提供附加信息以识别实体并帮助实现应用逻辑。

从给定参与者获得的内置 ***DataReader*** 将不会提供与通过同一参与者创建的实体有关的数据，服务默认这些实体已为创建它们的应用程序所知。

下表列出了内置主题，其名称以及除了应用于远程实体的 QoS 策略之外的其他信息，这些信息显示在与内置主题关联的数据中。

主题名称	**字段名称**	**类型**	**含义**
DCPSParticipant（创建 ***DomainParticipant*** 对象时创建的条目）	key	BuiltinTopicKey_t	区分条目的 DCPS 关键字
	user_data	UserDataQosPolicy	相应 *DomainParticipant* 的策略
DCPSTopic（创建 ***Topic*** 对象时创建的条目）	key	BuiltinTopicKey_t	区分条目的 DCPS 关键字
	name	string	主题的名称
	type_name	string	主题附带类型的名称
	durability	DurabilityQosPolicy	相应 *Topic* 的策略
	durability_service	DurabilityServiceQosPolicy	相应 *Topic* 的策略
	deadline	DeadlineQosPolicy	相应 *Topic* 的策略
	latency_budget	LatencyBudgetQosPolicy	相应 *Topic* 的策略
	liveliness	LivelinessQosPolicy	相应 *Topic* 的策略
	reliability	ReliabilityQosPolicy	相应 *Topic* 的策略
	transport_priority	TransportPriorityQosPolicy	相应 *Topic* 的策略

	lifespan	LifespanQosPolicy	相应 *Topic* 的策略
	destination_order	DestinationOrderQosPolicy	相应 *Topic* 的策略
	history	HistoryQosPolicy	相应 *Topic* 的策略
	resource_limits	ResourceLimitsQosPolicy	相应 *Topic* 的策略
	ownership	OwnershipQosPolicy	相应 *Topic* 的策略
	topic_data	TopicDataQosPolicy	相应 *Topic* 的策略
DCPSPublication（创建与其 ***Publisher*** 关联的 ***DataWriter*** 对象时创建的条目）	key	BuiltinTopicKey_t	区分条目的 DCPS 关键字
	participant_key	BuiltinTopicKey_t	*DataWriter* 所属参与者的 DCPS 关键字
	topic_name	string	关联 *Topic* 的名称
	type_name	string	关联主题附带类型的名称
	durability	DurabilityQosPolicy	相应 *DataWriter* 的策略
	durability_service	DurabilityServiceQosPolicy	相应 *DataWriter* 的策略
	deadline	DeadlineQosPolicy	相应 *DataWriter* 的策略
	latency_budget	LatencyBudgetQosPolicy	相应 *DataWriter* 的策略
	liveliness	LivelinessQosPolicy	相应 *DataWriter* 的策略
	reliability	ReliabilityQosPolicy	相应 *DataWriter* 的策略
	lifespan	LifespanQosPolicy	相应 *DataWriter* 的策略
	user_data	UserDataQosPolicy	相应 *DataWriter* 的策略
	ownership	OwnershipQosPolicy	相应 *DataWriter* 的策略
	ownership_strength	OwnershipStrengthQosPolicy	相应 *DataWriter* 的策略
	destination_order	DestinationOrderQosPolicy	相应 *DataWriter* 的策略
	presentation	PresentationQosPolicy	*DataWriter* 所属 *Publisher* 的策略
	partition	PartitionQosPolicy	*DataWriter* 所属 *Publisher* 的策略
	topic_data	TopicDataQosPolicy	关联 *Topic* 的策略
	group_data	GroupDataQosPolicy	*DataWriter* 所属 *Publisher* 的策略
DCPSSubscription（创建与其 ***Subscriber*** 关联的 ***DataReader*** 对象时创建的条目）	key	BuiltinTopicKey_t	区分条目的 DCPS 关键字
	participant_key	BuiltinTopicKey_t	*DataReader* 所属参与者的 DCPS 关键字
	topic_name	string	关联 *Topic* 的名称
	type_name	string	关联主题附带类型的名称
	durability	DurabilityQosPolicy	相应 *DataReader* 的策略
	deadline	DeadlineQosPolicy	相应 *DataReader* 的策略
	latency_budget	LatencyBudgetQosPolicy	相应 *DataReader* 的策略
	liveliness	LivelinessQosPolicy	相应 *DataReader* 的策略
	reliability	ReliabilityQosPolicy	相应 *DataReader* 的策略
	ownership	OwnershipQosPolicy	相应 *DataReader* 的策略
	destination_order	DestinationOrderQosPolicy	相应 *DataReader* 的策略
	user_data	UserDataQosPolicy	相应 *DataReader* 的策略
	time_based_filter	TimeBasedFilterQosPolicy	相应 *DataReader* 的策略

	presentation	PresentationQosPolicy	*DataReader* 所属 *Subscriber* 的策略
	partition	PartitionQosPolicy	*DataReader* 所属 *Subscriber* 的策略
	topic_data	TopicDataQosPolicy	关联 *Topic* 的策略
	group_data	GroupDataQosPolicy	*DataReader* 所属 *Subscriber* 的策略

2.2.6 交互模型

这里显示了两个交互模型来说明 DCPS 的行为。第一个是发布，第二个是订阅。

应该注意的是，这些模型并不是要解释服务的实施方式。特别地，在图片的右侧发生的事情（例如哪些组件实际发送通知）应该被理解为它***可能*** 如何工作而不是它***实际*** 如何工作（如图中的内部引用）。

2.2.6.1 发布视图

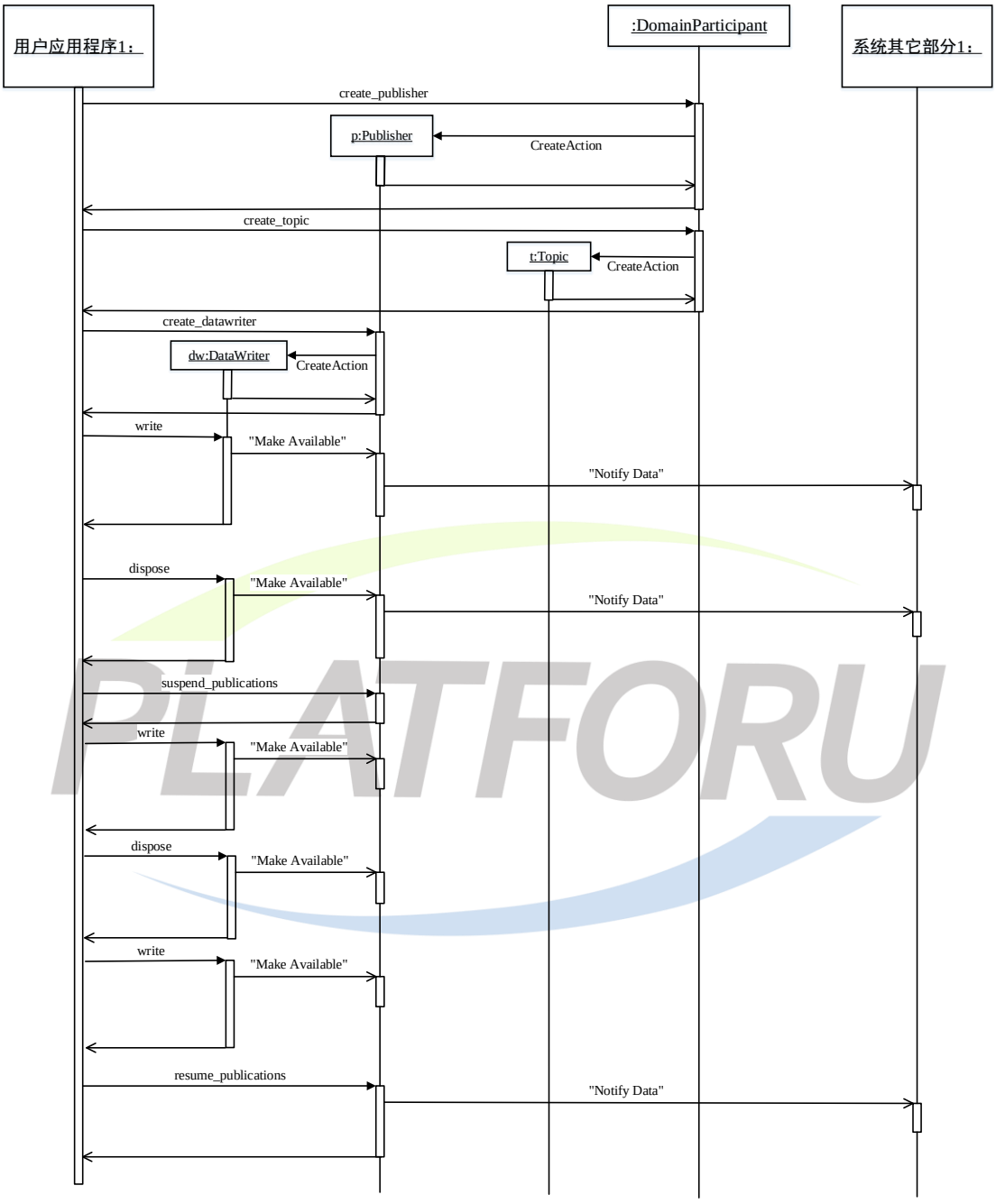

图 2-21 DCPS 交互模型时序图（发布）

图 2.21 的第一部分显示了 ***Publisher*** 的创建。第二部分显示必须在 DataWriter 引用主题之前创建主题。应该注意的是主题创建可以由不同于之后使用它的角色完成请求（在这种情况下它必须通过 ***TopicFactory***:: ***get_topic*** 搜索）。

图 2.21 的第三部分显示了 ***DataWriter*** 的创建。在 ***DataWriter*** 上调用 ***write*** 和 ***dispose*** 方法会立即通知发布者。由于应用程序尚未在发布者上调用 ***suspend_publications*** 方法，因此将根据当前发布者有关发送数据的策略传播相应的通知。

图 2.21 的最后一部分显示了嵌入到一对 ***suspend_publications***/***resume_publications*** 中的

相同类型的交互。重要的是需要考虑相应的通知现在被延迟到最后一次调用 ***resume_publications*** 才完成。还应注意尽管该时序图仅显示一个 ***DataWriter***，多个 ***DataWriter*** 也可以被包括在暂停/恢复对中。

2.2.6.2 订阅视图

在订阅端，给出了两个图表。第一个（见图 2.22）显示了在使用监听器时它是如何工作的，而第二个（见图 2.23）显示了条件和等待集的使用。

2.2.6.2.1 通过监听器通知

图 2.22 的第一部分显示了 ***Subscriber*** 和 ***DataReader*** 通过 ***DomainParticipant*** 的创建。

第二部分显示了 ***SubscriberListener*** 的使用：首先必须创建它并将其附加到 ***Subscriber***（通过 ***set_listener*** 方法）。当数据到达时，它可供每个相关的 ***DataReader*** 使用，之后触发 ***SubscriberListener***（***on_data_on_readers***）。应用程序必须获取受影响的 DataReader 对象列表（***get_datareaders***），然后它可以直接在这些对象上读取/获取数据。

或者，该图的第三部分显示了首先创建并附加到读取者的 ***DataReaderListener*** 对象的使用。当数据在 ***DataReader*** 上可用时，将触发相关的监听器并读取（读取/读取）数据。应该注意的是在这种情况下，不能获取读取者之间的关系。

注意：设置两种监听器时，***SubscriberListener*** 将取代 ***DataReaderListener***。

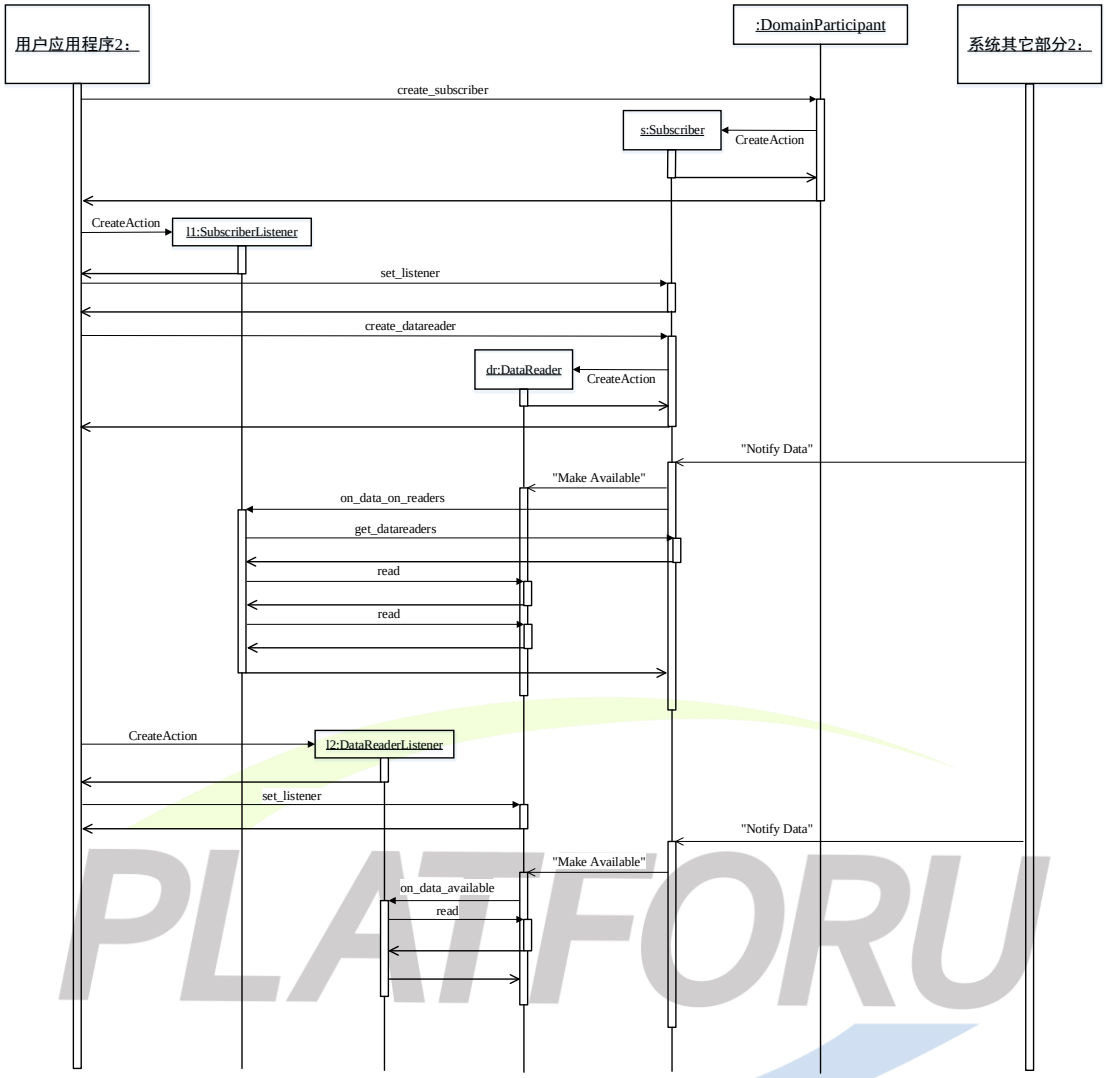

图 2-22 DCPS 交互模型时序图（使用监听器机制的订阅）

2.2.6.2.2 通过条件和等待集通知

图 2.23 的第一部分显示了通过 ***DomainParticipant*** 的创建 ***Subscriber*** 和 ***DataReader***。

第二部分显示了 ***WaitSet*** 和 ***ReadCondition*** 的创建，后者与前者的连接，以及对 ***WaitSet*** :: ***wait*** 方法的调用。请注意可能会创建几个条件（***ReadCondition*** 还有 ***StatusCondition***）并将这几个条件附加到同一个 ***WaitSet***。

该图的第三部分显示了收到数据时的信息流。请注意，***wait*** 方法会在一个数据到达周期内返回所有已启用条件的列表。如果多个 ***DataReader*** 对象接收到可用数据，则会同时启用多个条件并且应用程序将相应地执行多个 ***read*** 方法。

注意：对于条件和等待集，***read*** 方法在用户上下文中自然执行。

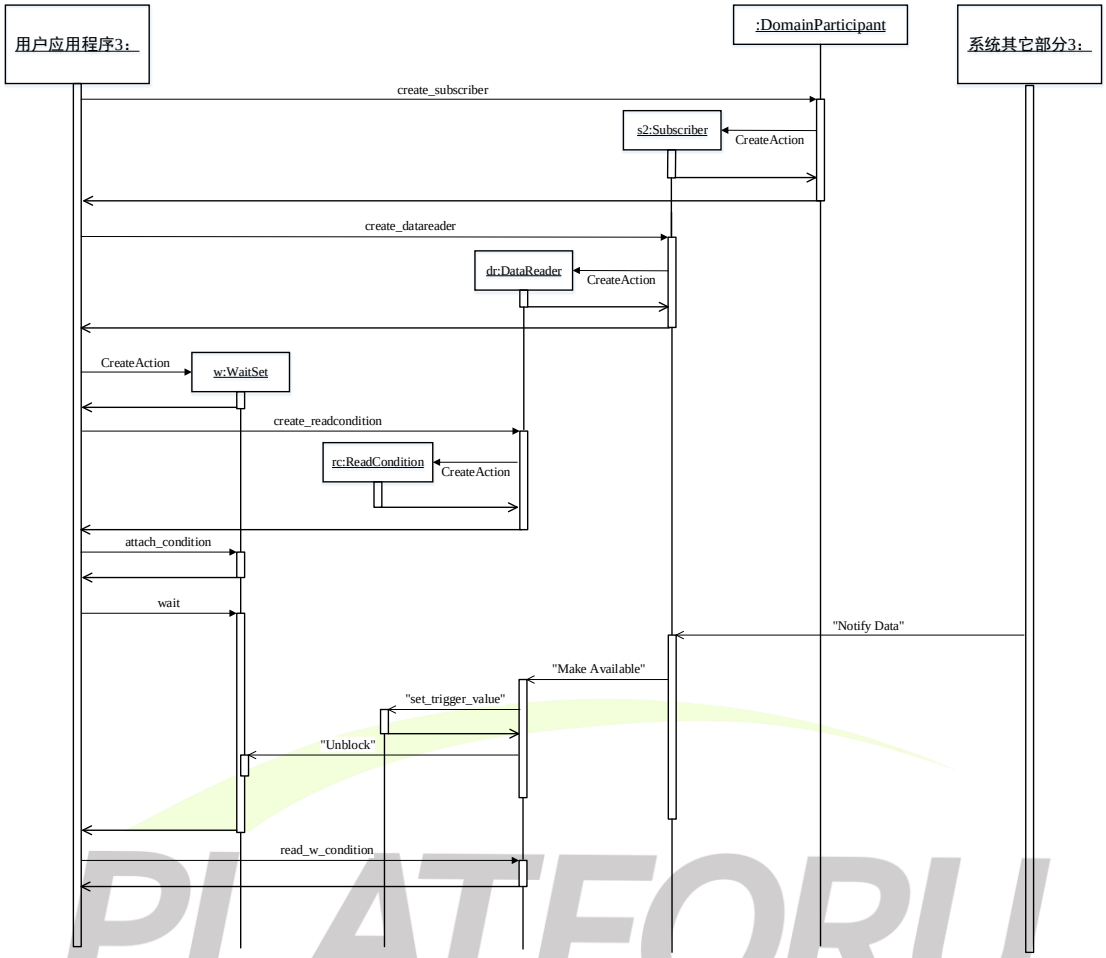

图 2-23 DCPS 交互模型时序图（使用条件机制的订阅）

2.3 OMG IDL 平台特定模型（PSM）

2.3.1 引言

OMG IDL PSM 通过 IDL 提供，IDL 定义应用程序可用于与服务交互的接口。

2.3.2 PIM 到 PSM 映射规则

接口开发中的关键问题是性能。这是数据分发服务（DDS）所针对的应用程序空间的后果。

PIM 中的“Out”参数通常映射到 PSM 中的“inout”参数，以便最小化分配服务执行所需的内存，同时提供更有效的实现。设想的场景是这种方法的调用者应该提供一个对象作为“容器”，方法将适当地“填充”该对象的状态。

PIM 到 PSM 映射将 UML 接口和类接口映射到 IDL 接口。普通数据类型映射到结构中。

IDL 接口不支持重载。基类或接口具有必须由专用类或接口重新定义的抽象方法的情景已映射到基本 IDL 接口，其中抽象方法出现在注释中。这是为了提醒所有专用化数据必须实现该方法。

枚举已映射到 IDL“枚举”或手工挑选的 IDL“long”类型数值，这些数值对应于增加 2 的幂（即 0x01,0x02,0x04 等）。当需要将多个值作为函数参数的部分引用时，选择后一种方式（2 的整数幂）。这允许使用“long”作为掩码来指示一组枚举值。此选择仅影响 PIM 的“状态类型”值，即：***StatusKind***，***SampleStateKind***，***ViewStateKind*** 和 ***InstanceStateKind***。掩码类型的名称是通过将“Kind”替换为单词“Mask”而形成的，如 ***StatusMask***，***SampleStateMask*** 等。

集合参数已映射到 IDL 序列。唯一的例外适用于集合元素是手工挑选的 IDL“long”类型的情况。在这种情况下，集合被映射到 IDL“long”类型并被解释为掩码。

每个 QosPolicy 都已映射为 IDL 结构。适用于每个实体的策略集合已建模为另一个 IDL 结构，其中包含与应用于此实体的策略相对应的属性。这种方法有几个优点。首先，它提供了在特定实体上设置特定 QosPolicy 的适用性编译时检查。第二个优点是它不需要使用增加代码大小并且在“C”中使用不太自然的 “any” 类型。其他方法不那么有吸引力。 IDL 接口不合适，因为 QosPolicy 集合作为多个方法的参数出现，需要“按值”传递。IDL“valuetype”也被考虑过但最终被否决，因为它不受普遍支持并强制通过一个方法专门访问一个属性。

错误返回值已映射到相应函数的普通返回码。原因是 DCPS 将“C”作为关键部署语言之一，并且返回代码在“C”中使用更为自然。

与数据访问方法（***read*** 和 ***take***）返回的 ***SampleInfo*** 和 ***Data*** 集合相关联的 ***DataSample*** 类尚未显式映射到 IDL。集合本身已被映射到序列中。每个 ***Data*** 和 ***SampleInfo*** 之间的对应关系通过使用相同的索引来访问每个集合上的相应元素来表示。预计可以在每种目标语言上提供附加的数据访问 API，以使该方法尽可能自然且有效。原因是访问数据是数据分发服务的主要目的，而 IDL 映射提供了一种编程语言中立表示，无法利用每种特定语言的优势。

没有工厂方法的类，即 ***WaitSet*** 和 ***GuardCondition***，将映射到 IDL 接口。目的是将它们作为每种实现语言的原生类实现，并且将使用该语言的“new”运算符构造它们。此外，实现语言映射应至少提供一个不带参数的构造函数，以便应用程序可以跨该映射的不同供应商实现可移植性。

DomainParticipantFactory 接口的语言实现应具有 2.2.2.2.2 中描述的静态方法 **get_instance**。此方法不会出现在 IDL 接口 DomainParticipantFactory 中，因为静态方法无法在 IDL 中表示。

用于表示时间的两种类型：***Duration_t*** 和 ***Time_t*** 被映射到包含秒和纳秒字段的结构。这些类型被进一步约束为始终使用“标准化”表示，即 nanosec 字段必须验证 0 <= nanosec <1000000000。

IDL PSM 引入了许多旨在以原生方式定义的类型。由于这些类型是不透明的，因此类型的实际定义不会影响可移植性并且依赖于此实现。为了完整性，类型的名称在 IDL 中显示为 typedef，带有后缀“_TYPE_NATIVE”的#define 用作实际类型的占位符。通过这种方式在 IDL 中使用的类型不是标准化的，并且允许实现使用任何其他类型，包括非标量（即结构化类型）。

2.3.3 DCPS PSM ： IDL

```
#define DOMAINID_TYPE_NATIVE        long
#define HANDLE_TYPE_NATIVE          long
#define HANDLE_NIL_NATIVE           0
#define BUILTIN_TOPIC_KEY_TYPE_NATIVE      long
```

```
#define TheParticipantFactory
#define PARTICIPANT_QOS_DEFAULT
#define TOPIC_QOS_DEFAULT
#define PUBLISHER_QOS_DEFAULT
#define SUBSCRIBER_QOS_DEFAULT
#define DATAWRITER_QOS_DEFAULT
#define DATAREADER_QOS_DEFAULT
#define DATAWRITER_QOS_USE_TOPIC_QOS
#define DATAREADER_QOS_USE_TOPIC_QOS

//------------------start of module DDS---------------------------------
module DDS
{
  typedef   DOMAINID_TYPE_NATIVE        DomainId_t;
  typedef   HANDLE_TYPE_NATIVE          InstanceHandle_t;

  struct BuiltinTopicKey_t
  {
    BUILTIN_TOPIC_KEY_TYPE_NATIVE value[3];
  };

  typedef sequence<InstanceHandle_t> InstanceHandleSeq;
  typedef long ReturnCode_t;
  typedef long QosPolicyId_t;
  typedef sequence<string> StringSeq;

  struct Duration_t
  {
    long sec;
    unsigned long nanosec;
  };

  struct Time_t
  {
    long sec;
    unsigned long nanosec;
  };

  // -----------------------------------------------------------------
  // Pre-defined values
  // -----------------------------------------------------------------
  const   InstanceHandle_t   HANDLE_NIL                 =   HANDLE_NIL_NATIVE;

  const   long               LENGTH_UNLIMITED           =   -1;
```

```
const   long            DURATION_INFINITE_SEC     =   0x7fffffff;
const   unsigned long   DURATION_INFINITE_NSEC    =   0x7fffffff;

const   long            DURATION_ZERO_SEC         =   0;
const   unsigned long   DURATION_ZERO_NSEC        =   0;

const   long            TIME_INVALID_SEC          =   -1;
const   unsigned long   TIME_INVALID_NSEC         =   0xffffffff;

// -----------------------------------------------------------------------
// Return codes
// -----------------------------------------------------------------------
const   ReturnCode_t    RETCODE_OK                        =   0;
const   ReturnCode_t    RETCODE_ERROR                     =   1;
const   ReturnCode_t    RETCODE_UNSUPPORTED               =   2;
const   ReturnCode_t    RETCODE_BAD_PARAMETER             =   3;
const   ReturnCode_t    RETCODE_PRECONDITION_NOT_MET      =   4;
const   ReturnCode_t    RETCODE_OUT_OF_RESOURCES          =   5;
const   ReturnCode_t    RETCODE_NOT_ENABLED               =   6;
const   ReturnCode_t    RETCODE_IMMUTABLE_POLICY          =   7;
const   ReturnCode_t    RETCODE_INCONSISTENT_POLICY       =   8;
const   ReturnCode_t    RETCODE_ALREADY_DELETED           =   9;
const   ReturnCode_t    RETCODE_TIMEOUT                   =   10;
const   ReturnCode_t    RETCODE_NO_DATA                   =   11;
const   ReturnCode_t    RETCODE_ILLEGAL_OPERATION         =   12;

// -----------------------------------------------------------------------
// Status to support listeners and conditions
// -----------------------------------------------------------------------
typedef   unsigned long     StatusKind;
typedef   unsigned long     StatusMask;     // bit-mask StatusKind
const   StatusKind   INCONSISTENT_TOPIC_STATUS              = 0x0001 << 0;
const   StatusKind   OFFERED_DEADLINE_MISSED_STATUS         = 0x0001 << 1;
const   StatusKind   REQUESTED_DEADLINE_MISSED_STATUS       = 0x0001 << 2;
const   StatusKind   OFFERED_INCOMPATIBLE_QOS_STATUS        = 0x0001 << 5;
const   StatusKind   REQUESTED_INCOMPATIBLE_QOS_STATUS      = 0x0001 << 6;
const   StatusKind   SAMPLE_LOST_STATUS                     = 0x0001 << 7;
const   StatusKind   SAMPLE_REJECTED_STATUS                 = 0x0001 << 8;
const   StatusKind   DATA_ON_READERS_STATUS                 = 0x0001 << 9;
const   StatusKind   DATA_AVAILABLE_STATUS                  = 0x0001 << 10;
const   StatusKind   LIVELINESS_LOST_STATUS                 = 0x0001 << 11;
const   StatusKind   LIVELINESS_CHANGED_STATUS              = 0x0001 << 12;
const   StatusKind   PUBLICATION_MATCHED_STATUS             = 0x0001 << 13;
```

```
const    StatusKind    SUBSCRIPTION_MATCHED_STATUS            = 0x0001 << 14;
struct InconsistentTopicStatus
{
   long        total_count;
   long        total_count_change;
};

struct SampleLostStatus
{
   long        total_count;
   long        total_count_change;
};

enum SampleRejectedStatusKind
{
   NOT_REJECTED,
   REJECTED_BY_INSTANCES_LIMIT,
   REJECTED_BY_SAMPLES_LIMIT,
   REJECTED_BY_SAMPLES_PER_INSTANCE_LIMIT
};

struct SampleRejectedStatus
{
   long                        total_count;
   long                        total_count_change;
   SampleRejectedStatusKind    last_reason;
   InstanceHandle_t            last_instance_handle;
};

struct LivelinessLostStatus
{
   long        total_count;
   long        total_count_change;
};

struct LivelinessChangedStatus
{
   long                alive_count;
   long                not_alive_count;
   long                alive_count_change;
   long                not_alive_count_change;
   InstanceHandle_t    last_publication_handle;
};
```

```
struct OfferedDeadlineMissedStatus
{
   long                 total_count;
   long                 total_count_change;
   InstanceHandle_t     last_instance_handle;
};

struct RequestedDeadlineMissedStatus
{
   long                 total_count;
   long                 total_count_change;
   InstanceHandle_t     last_instance_handle;
};

struct QosPolicyCount
{
   QosPolicyId_t        policy_id;
   long                 count;
};

typedef   sequence<QosPolicyCount>    QosPolicyCountSeq;

struct OfferedIncompatibleQosStatus
{
   long                       total_count;
   long                       total_count_change;
   QosPolicyId_t              last_policy_id;
   QosPolicyCountSeq          policies;
};

struct RequestedIncompatibleQosStatus
{
   long                       total_count;
   long                       total_count_change;
   QosPolicyId_t              last_policy_id;
   QosPolicyCountSeq          policies;
};

struct PublicationMatchedStatus
{
   long                 total_count;
   long                 total_count_change;
   long                 current_count;
```

```
    long                current_count_change;
    InstanceHandle_t    last_subscription_handle;
};

struct SubscriptionMatchedStatus
{
    long                total_count;
    long                total_count_change;
    long                current_count;
    long                current_count_change;
    InstanceHandle_t    last_publication_handle;
};

// -----------------------------------------------------------------------
// Listeners
// -----------------------------------------------------------------------
interface     Listener;
interface     Entity;
interface     TopicDescription;
interface     Topic;
interface     ContentFilteredTopic;
interface     MultiTopic;
interface     DataWriter;
interface     DataReader;
interface     Subscriber;
interface     Publisher;

typedef    sequence<DataReader>    DataReaderSeq;

interface Listener {};

interface    TopicListener : Listener
{
    void   on_inconsistent_topic(in    Topic    the_topic,
                                 in    InconsistentTopicStatus    status);
};

interface    DataWriterListener : Listener
{
    void    on_offered_deadline_missed(
                in DataWriter writer,
                in OfferedDeadlineMissedStatus status);
    void    on_offered_incompatible_qos(
                in DataWriter writer,
```

```
                in OfferedIncompatibleQosStatus status);
    void    on_liveliness_lost(
                in DataWriter writer,
                in LivelinessLostStatus status);
    void    on_publication_matched(
                in DataWriter writer,
                in PublicationMatchedStatus status);
};

interface PublisherListener : DataWriterListener
{
};

interface DataReaderListener : Listener
{
    void    on_requested_deadline_missed(
                in DataReader the_reader,
                in RequestedDeadlineMissedStatus status);
    void    on_requested_incompatible_qos(
                in DataReader the_reader,
                in RequestedIncompatibleQosStatus status);
    void    on_sample_rejected(
                in DataReader the_reader,
                in SampleRejectedStatus status);
    void    on_liveliness_changed(
                in DataReader the_reader,
                in LivelinessChangedStatus status);
    void    on_data_available(
                in DataReader the_reader);
    void    on_subscription_matched(
                in DataReader the_reader,
                in SubscriptionMatchedStatus status);
    void    on_sample_lost(
                in DataReader the_reader,
                in SampleLostStatus status);
};

interface    SubscriberListener : DataReaderListener
{
    void    on_data_on_readers(
                in Subscriber the_subscriber);
};
interface    DomainParticipantListener :    TopicListener,
                                            PublisherListener,
```

```
{
};

// ---------------------------------------------------------------------
// Conditions
// ---------------------------------------------------------------------
interface Condition
{
  boolean get_trigger_value();
};

typedef sequence<Condition> ConditionSeq;

interface WaitSet
{
  ReturnCode_t  wait (
        inout ConditionSeq active_conditions,
        in Duration_t timeout);
  ReturnCode_t  attach_condition (
        in Condition cond);
  ReturnCode_t  detach_condition (
        in Condition cond);
  ReturnCode_t  get_conditions (
        inout ConditionSeq attached_conditions);
};

interface GuardCondition : Condition
{
  ReturnCode_t  set_trigger_value (
        in boolean value);
};

interface StatusCondition : Condition
{
  StatusMask    get_enabled_statuses();
  ReturnCode_t  set_enabled_statuses (
              in StatusMask mask);
  Entity  get_entity();
};

// Sample states to support reads
typedef  unsigned long        SampleStateKind;
const  SampleStateKind        READ_SAMPLE_STATE        = 0x0001 << 0;
```

```
const    SampleStateKind        NOT_READ_SAMPLE_STATE   = 0x0001 << 1;
// This is a bit-mask SampleStateKind
typedef    unsigned long        SampleStateMask;
const    SampleStateMask        ANY_SAMPLE_STATE        = 0xffff;

// View states to support reads
typedef unsigned long ViewStateKind;
const    ViewStateKind          NEW_VIEW_STATE          = 0x0001 << 0;
const    ViewStateKind          NOT_NEW_VIEW_STATE      = 0x0001 << 1;

// This is a bit-mask ViewStateKind
typedef    unsigned long        ViewStateMask;
const    ViewStateMask          ANY_VIEW_STATE          = 0xffff;

// Instance states to support reads
typedef    unsigned long        InstanceStateKind;
const    InstanceStateKind      ALIVE_INSTANCE_STATE    = 0x0001 << 0;
const    InstanceStateKind      NOT_ALIVE_DISPOSED_INSTANCE_STATE      =
                                                        0x0001 << 1;
const    InstanceStateKind      NOT_ALIVE_NO_WRITERS_INSTANCE_STATE  =
                                                        0x0001 << 2;

// This is a bit-mask InstanceStateKind
typedef    unsigned long InstanceStateMask;
const    InstanceStateMask      ANY_INSTANCE_STATE          = 0xffff;
const    InstanceStateMask      NOT_ALIVE_INSTANCE_STATE    = 0x006;

interface ReadCondition : Condition
{
   SampleStateMask       get_sample_state_mask();
   ViewStateMask         get_view_state_mask();
   InstanceStateMask     get_instance_state_mask();
   DataReader            get_datareader();
};

interface QueryCondition : ReadCondition
{
   string                get_query_expression();
   ReturnCode_t          get_query_parameters(
            inout StringSeq   query_parameters);
   ReturnCode_t          set_query_parameters(
            in StringSeq query_parameters);
};
```

```
// --------------------------------------------------------------------
// Qos
// --------------------------------------------------------------------
const   string   USERDATA_QOS_POLICY_NAME              = "UserData";
const   string   DURABILITY_QOS_POLICY_NAME            = "Durability";
const   string   PRESENTATION_QOS_POLICY_NAME          = "Presentation";
const   string   DEADLINE_QOS_POLICY_NAME              = "Deadline";
const   string   LATENCYBUDGET_QOS_POLICY_NAME         = "LatencyBudget";
const   string   OWNERSHIP_QOS_POLICY_NAME             = "Ownership";
const   string   OWNERSHIPSTRENGTH_QOS_POLICY_NAME
                                                       = "OwnershipStrength";
const   string   LIVELINESS_QOS_POLICY_NAME            = "Liveliness";
const   string   TIMEBASEDFILTER_QOS_POLICY_NAME       = "TimeBasedFilter";
const   string   PARTITION_QOS_POLICY_NAME             = "Partition";
const   string   RELIABILITY_QOS_POLICY_NAME           = "Reliability";
const   string   DESTINATIONORDER_QOS_POLICY_NAME      = "DestinationOrder";
const   string   HISTORY_QOS_POLICY_NAME               = "History";
const   string   RESOURCELIMITS_QOS_POLICY_NAME        = "ResourceLimits";
const   string   ENTITYFACTORY_QOS_POLICY_NAME         = "EntityFactory";
const   string   WRITERDATALIFECYCLE_QOS_POLICY_NAME
                                                       = "WriterDataLifecycle";
const   string   READERDATALIFECYCLE_QOS_POLICY_NAME
                                                       = "ReaderDataLifecycle";
const   string   TOPICDATA_QOS_POLICY_NAME             = "TopicData";
const   string   GROUPDATA_QOS_POLICY_NAME             = "TransportPriority";
const   string   LIFESPAN_QOS_POLICY_NAME              = "Lifespan";
const   string   DURABILITYSERVICE_POLICY_NAME         = "DurabilityService";

const   QosPolicyId_t   INVALID_QOS_POLICY_ID                  = 0;
const   QosPolicyId_t   USERDATA_QOS_POLICY_ID                 = 1;
const   QosPolicyId_t   DURABILITY_QOS_POLICY_ID               = 2;
const   QosPolicyId_t   PRESENTATION_QOS_POLICY_ID             = 3;
const   QosPolicyId_t   DEADLINE_QOS_POLICY_ID                 = 4;
const   QosPolicyId_t   LATENCYBUDGET_QOS_POLICY_ID            = 5;
const   QosPolicyId_t   OWNERSHIP_QOS_POLICY_ID                = 6;
const   QosPolicyId_t   OWNERSHIPSTRENGTH_QOS_POLICY_ID        = 7;
const   QosPolicyId_t   LIVELINESS_QOS_POLICY_ID               = 8;
const   QosPolicyId_t   TIMEBASEDFILTER_QOS_POLICY_ID          = 9;
const   QosPolicyId_t   PARTITION_QOS_POLICY_ID                = 10;
const   QosPolicyId_t   RELIABILITY_QOS_POLICY_ID              = 11;
const   QosPolicyId_t   DESTINATIONORDER_QOS_POLICY_ID         = 12;
const   QosPolicyId_t   HISTORY_QOS_POLICY_ID                  = 13;
const   QosPolicyId_t   RESOURCELIMITS_QOS_POLICY_ID           = 14;
```

```
const   QosPolicyId_t   ENTITYFACTORY_QOS_POLICY_ID             = 15;
const   QosPolicyId_t   WRITERDATALIFECYCLE_QOS_POLICY_ID       = 16;
const   QosPolicyId_t   READERDATALIFECYCLE_QOS_POLICY_ID       = 17;
const   QosPolicyId_t   TOPICDATA_QOS_POLICY_ID                 = 18;
const   QosPolicyId_t   GROUPDATA_QOS_POLICY_ID                 = 19;
const   QosPolicyId_t   TRANSPORTPRIORITY_QOS_POLICY_ID         = 20;
const   QosPolicyId_t   LIFESPAN_QOS_POLICY_ID                  = 21;
const   QosPolicyId_t   DURABILITYSERVICE_QOS_POLICY_ID         = 22;

struct UserDataQosPolicy
{
  sequence<octet>   value;
};

struct TopicDataQosPolicy
{
  sequence<octet>   value;
};

struct GroupDataQosPolicy
{
  sequence<octet>   value;
};

struct TransportPriorityQosPolicy
{
  long              value;
};
struct LifespanQosPolicy
{
  Duration_t      duration;
};

enum DurabilityQosPolicyKind
{
        VOLATILE_DURABILITY_QOS,
        TRANSIENT_LOCAL_DURABILITY_QOS,
        TRANSIENT_DURABILITY_QOS,
        PERSISTENT_DURABILITY_QOS
};

struct DurabilityQosPolicy
{
  DurabilityQosPolicyKind    kind;
```

```
};
enum PresentationQosPolicyAccessScopeKind
{
    INSTANCE_PRESENTATION_QOS,
    TOPIC_PRESENTATION_QOS,
    GROUP_PRESENTATION_QOS
};
struct PresentationQosPolicy
{
  PresentationQosPolicyAccessScopeKind    access_scope;
  boolean                                 coherent_access;
  boolean                                 ordered_access;
};

struct DeadlineQosPolicy
{
  Duration_t      period;
};

struct LatencyBudgetQosPolicy
{
  Duration_t      duration;
};

enum OwnershipQosPolicyKind
{
    SHARED_OWNERSHIP_QOS,
    EXCLUSIVE_OWNERSHIP_QOS
};

struct OwnershipQosPolicy
{
  OwnershipQosPolicyKind     kind;
};

struct OwnershipStrengthQosPolicy
{
        long        value;
};
enum LivelinessQosPolicyKind
{
    AUTOMATIC_LIVELINESS_QOS,
    MANUAL_BY_PARTICIPANT_LIVELINESS_QOS,
    MANUAL_BY_TOPIC_LIVELINESS_QOS
```

```
};
struct LivelinessQosPolicy
{
    LivelinessQosPolicyKind     kind;
    Duration_t                  lease_duration;
};

struct TimeBasedFilterQosPolicy
{
    Duration_t                  minimum_separation;
};
struct PartitionQosPolicy
{
  StringSeq                     name;
};

enum ReliabilityQosPolicyKind
{
    BEST_EFFORT_RELIABILITY_QOS,
    RELIABLE_RELIABILITY_QOS
};

struct ReliabilityQosPolicy
{
    ReliabilityQosPolicyKind    kind;
    Duration_t                  max_blocking_time;
};

enum DestinationOrderQosPolicyKind
{
    BY_RECEPTION_TIMESTAMP_DESTINATIONORDER_QOS,
    BY_SOURCE_TIMESTAMP_DESTINATIONORDER_QOS
};

struct DestinationOrderQosPolicy
{
  DestinationOrderQosPolicyKind kind;
};

enum HistoryQosPolicyKind
{
    KEEP_LAST_HISTORY_QOS,
    KEEP_ALL_HISTORY_QOS
};
```

```
struct HistoryQosPolicy
{
   HistoryQosPolicyKind          kind;
   long                          depth;
};

struct ResourceLimitsQosPolicy
{
      long                       max_samples;
      long                       max_instances;
      long                       max_samples_per_instance;
};

struct EntityFactoryQosPolicy
{
      boolean                    autoenable_created_entities;
};

struct WriterDataLifecycleQosPolicy
{
      boolean                    autodispose_unregistered_instances;
};

struct ReaderDataLifecycleQosPolicy
{
      Duration_t                 autopurge_nowriter_samples_delay;
      Duration_t                 autopurge_disposed_samples_delay;
};

struct DurabilityServiceQosPolicy
{
      Duration_t                 service_cleanup_delay;
      HistoryQosPolicyKind       history_kind;
      long                       history_depth;
      long                       max_samples;
      long                       max_instances;
      long                       max_samples_per_instance;
};

struct DomainParticipantFactoryQos
{
      EntityFactoryQosPolicy     entity_factory;
};
```

```
struct DomainParticipantQos
{
    UserDataQosPolicy           user_data;
    EntityFactoryQosPolicy      entity_factory;
};

struct TopicQos
{
    TopicDataQosPolicy          topic_data;
    DurabilityQosPolicy         durability;
    DurabilityServiceQosPolicy  durability_service;
    DeadlineQosPolicy           deadline;
    LatencyBudgetQosPolicy      latency_budget;
    LivelinessQosPolicy         liveliness;
    ReliabilityQosPolicy        reliability;
    DestinationOrderQosPolicy   destination_order;
    HistoryQosPolicy            history;
    ResourceLimitsQosPolicy     resource_limits;
    TransportPriorityQosPolicy  transport_priority;
    LifespanQosPolicy           lifespan;
    OwnershipQosPolicy          ownership;
};

struct DataWriterQos
{
    DurabilityQosPolicy         durability;
    DurabilityServiceQosPolicy  durability_service;
    DeadlineQosPolicy           deadline;
    LatencyBudgetQosPolicy      latency_budget;
    LivelinessQosPolicy         liveliness;
    ReliabilityQosPolicy        reliability;
    DestinationOrderQosPolicy   destination_order;
    HistoryQosPolicy            history;
    ResourceLimitsQosPolicy     resource_limits;
    TransportPriorityQosPolicy  transport_priority;
    LifespanQosPolicy           lifespan;
    UserDataQosPolicy           user_data;
    OwnershipQosPolicy          ownership;
    OwnershipStrengthQosPolicy  ownership_strength;
    WriterDataLifecycleQosPolicy writer_data_lifecycle;
};

struct PublisherQos
```

```
{
    PresentationQosPolicy           presentation;
    PartitionQosPolicy              partition;
    GroupDataQosPolicy              group_data;
    EntityFactoryQosPolicy          entity_factory;
};

struct DataReaderQos
{
    DurabilityQosPolicy             durability;
    DeadlineQosPolicy               deadline;
    LatencyBudgetQosPolicy          latency_budget;
    LivelinessQosPolicy             liveliness;
    ReliabilityQosPolicy            reliability;
    DestinationOrderQosPolicy       destination_order;
    HistoryQosPolicy                history;
    ResourceLimitsQosPolicy         resource_limits;
    UserDataQosPolicy               user_data;
    OwnershipQosPolicy              ownership;
    TimeBasedFilterQosPolicy        time_based_filter;
    ReaderDataLifecycleQosPolicy    reader_data_lifecycle;
};

struct SubscriberQos
{
    PresentationQosPolicy           presentation;
    PartitionQosPolicy              partition;
    GroupDataQosPolicy              group_data;
    EntityFactoryQosPolicy          entity_factory;
};

// -----------------------------------------------------------------
struct ParticipantBuiltinTopicData
{
    BuiltinTopicKey_t               key;
    UserDataQosPolicy               user_data;
};

struct TopicBuiltinTopicData
{
    BuiltinTopicKey_t               key;
    string                          name;
    string                          type_name;
    DurabilityQosPolicy             durability;
```

```
    DurabilityServiceQosPolicy    durability_service;
    DeadlineQosPolicy             deadline;
    LatencyBudgetQosPolicy        latency_budget;
    LivelinessQosPolicy           liveliness;
    ReliabilityQosPolicy          reliability;
    TransportPriorityQosPolicy    transport_priority;
    LifespanQosPolicy             lifespan;
    DestinationOrderQosPolicy     destination_order;
    HistoryQosPolicy              history;
    ResourceLimitsQosPolicy       resource_limits;
    OwnershipQosPolicy            ownership;
    TopicDataQosPolicy            topic_data;
};

struct PublicationBuiltinTopicData
{
    BuiltinTopicKey_t             key;
    BuiltinTopicKey_t             participant_key;
    string                        topic_name;
    string                        type_name;
    DurabilityQosPolicy           durability;
    DurabilityServiceQosPolicy    durability_service;
    DeadlineQosPolicy             deadline;
    LatencyBudgetQosPolicy        latency_budget;
    LivelinessQosPolicy           liveliness;
    ReliabilityQosPolicy          reliability;
    LifespanQosPolicy             lifespan;
    UserDataQosPolicy             user_data;
    OwnershipQosPolicy            ownership;
    OwnershipStrengthQosPolicy    ownership_strength;
    DestinationOrderQosPolicy     destination_order;
    PresentationQosPolicy         presentation;
    PartitionQosPolicy            partition;
    TopicDataQosPolicy            topic_data;
    GroupDataQosPolicy            group_data;
};

struct SubscriptionBuiltinTopicData
{
    BuiltinTopicKey_t             key;
    BuiltinTopicKey_t             participant_key;
    string                        topic_name;
    string                        type_name;
    DurabilityQosPolicy           durability;
```

```
        DeadlineQosPolicy               deadline;
        LatencyBudgetQosPolicy          latency_budget;
        LivelinessQosPolicy             liveliness;
        ReliabilityQosPolicy            reliability;
        OwnershipQosPolicy              ownership;
        DestinationOrderQosPolicy       destination_order;
        UserDataQosPolicy               user_data;
        TimeBasedFilterQosPolicy        time_based_filter;
        PresentationQosPolicy           presentation;
        PartitionQosPolicy              partition;
        TopicDataQosPolicy              topic_data;
        GroupDataQosPolicy              group_data;
};

// -----------------------------------------------------------------------
interface Entity
{
// ReturnCode_t set_qos(
// in EntityQos qos);
// ReturnCode_t get_qos(
// inout EntityQos qos);
// ReturnCode_t set_listener(
// in Listener l,
// in StatusMask mask);
// Listener get_listener();
        ReturnCode_t    enable();
        StatusCondition    get_statuscondition();
        StatusMask    get_status_changes();
        InstanceHandle_t    get_instance_handle();
};

// -----------------------------------------------------------------------
interface DomainParticipant : Entity
{
        // Factory interfaces
        Publisher    create_publisher(
                in PublisherQos qos,
                in PublisherListener a_listener,
                in StatusMask mask);

        ReturnCode_t    delete_publisher(
                in Publisher p);

        Subscriber    create_subscriber(
```

```
        in SubscriberQos qos,
        in SubscriberListener a_listener,
        in StatusMask mask);

ReturnCode_t    delete_subscriber(
        in Subscriber s);

Subscriber    get_builtin_subscriber();
Topic    create_topic(
        in string topic_name,
        in string type_name,
        in TopicQos qos,
        in TopicListener a_listener,
        in StatusMask mask);

ReturnCode_t    delete_topic(
        in Topic a_topic);

Topic    find_topic(
        in string topic_name,
        in Duration_t timeout);

TopicDescription    lookup_topicdescription(
        in string name);

ContentFilteredTopic    create_contentfilteredtopic(
        in string name,
        in Topic related_topic,
        in string filter_expression,
        in StringSeq expression_parameters);

ReturnCode_t    delete_contentfilteredtopic(
        in ContentFilteredTopic a_contentfilteredtopic);

MultiTopic    create_multitopic(
        in string name,
        in string type_name,
        in string subscription_expression,
        in StringSeq expression_parameters);

ReturnCode_t    delete_multitopic(
        in MultiTopic a_multitopic);

ReturnCode_t    delete_contained_entities();
```

```
ReturnCode_t    set_qos(
        in DomainParticipantQos qos);

ReturnCode_t    get_qos(
        inout DomainParticipantQos qos);

ReturnCode_t    set_listener(
        in DomainParticipantListener a_listener,
        in StatusMask mask);

DomainParticipantListener    get_listener();

ReturnCode_t    ignore_participant(
        in InstanceHandle_t handle);

ReturnCode_t    ignore_topic(
        in InstanceHandle_t handle);

ReturnCode_t    ignore_publication(
        in InstanceHandle_t handle);

ReturnCode_t    ignore_subscription(
        in InstanceHandle_t handle);

DomainId_t    get_domain_id();

ReturnCode_t    assert_liveliness();

ReturnCode_t    set_default_publisher_qos(
        in PublisherQos qos);

ReturnCode_t    get_default_publisher_qos(
        inout PublisherQos qos);

ReturnCode_t    set_default_subscriber_qos(
        in SubscriberQos qos);

ReturnCode_t    get_default_subscriber_qos(
        inout SubscriberQos qos);

ReturnCode_t    set_default_topic_qos(
        in TopicQos qos);
```

```
        ReturnCode_t    get_default_topic_qos(
                inout TopicQos qos);

        ReturnCode_t    get_discovered_participants(
                inout InstanceHandleSeq participant_handles);

        ReturnCode_t    get_discovered_participant_data(
                inout ParticipantBuiltinTopicData participant_data,
                in InstanceHandle_t participant_handle);

        ReturnCode_t    get_discovered_topics(
                inout InstanceHandleSeq topic_handles);

        ReturnCode_t    get_discovered_topic_data(
                inout TopicBuiltinTopicData topic_data,
                in InstanceHandle_t topic_handle);

        boolean    contains_entity(
                in InstanceHandle_t a_handle);

        ReturnCode_t    get_current_time(
                inout Time_t current_time);
};

interface    DomainParticipantFactory
{
        DomainParticipant    create_participant(
                in DomainId_t domain_id,
                in DomainParticipantQos qos,
                in DomainParticipantListener a_listener,
                in StatusMask mask);

        ReturnCode_t    delete_participant(
                in DomainParticipant a_participant);

        DomainParticipant    lookup_participant(
                in DomainId_t domain_id);

        ReturnCode_t    set_default_participant_qos(
                in DomainParticipantQos qos);

        ReturnCode_t    get_default_participant_qos(
                inout DomainParticipantQos qos);
```

```
        ReturnCode_t    set_qos(
                in DomainParticipantFactoryQos qos);

        ReturnCode_t    get_qos(
                inout DomainParticipantFactoryQos qos);
};

interface TypeSupport
{
        // ReturnCode_t register_type(
        // in DomainParticipant domain,
        // in string type_name);
        // string get_type_name();
};

// -----------------------------------------------------------------------
interface TopicDescription
{
        string                  get_type_name();
        string                  get_name();
        DomainParticipant       get_participant();
};

interface Topic : Entity, TopicDescription
{
        ReturnCode_t    set_qos(
                in TopicQos qos);

        ReturnCode_t    get_qos(
                inout TopicQos qos);

        ReturnCode_t    set_listener(
                in TopicListener a_listener,
                in StatusMask mask);

        TopicListener   get_listener();

        // Access the status
        ReturnCode_t    get_inconsistent_topic_status(
        inout InconsistentTopicStatus a_status);
};

interface    ContentFilteredTopic : TopicDescription
{
```

```
        string    get_filter_expression();
        ReturnCode_t    get_expression_parameters(
                inout StringSeq);

        ReturnCode_t    set_expression_parameters(
                in StringSeq expression_parameters);

        Topic get_related_topic();
};

interface MultiTopic : TopicDescription
{
        string    get_subscription_expression();

        ReturnCode_t    get_expression_parameters(
                inout StringSeq);

        ReturnCode_t    set_expression_parameters(
                in StringSeq expression_parameters);
};
// ---------------------------------------------------------------------
interface Publisher : Entity
{
        DataWriter    create_datawriter(
                in Topic a_topic,
                in DataWriterQos qos,
                in DataWriterListener a_listener,
        in StatusMask mask);

        ReturnCode_t    delete_datawriter(
                in DataWriter a_datawriter);

        DataWriter    lookup_datawriter(
                in string topic_name);

        ReturnCode_t    delete_contained_entities();

        ReturnCode_t    set_qos(
                in PublisherQos qos);

        ReturnCode_t    get_qos(
                inout PublisherQos qos);

        ReturnCode_t    set_listener(
```

```
            in PublisherListener a_listener,
            in StatusMask mask);

        PublisherListener    get_listener();

        ReturnCode_t    suspend_publications();

        ReturnCode_t    resume_publications();

        ReturnCode_t    begin_coherent_changes();

        ReturnCode_t    end_coherent_changes();

        ReturnCode_t    wait_for_acknowledgments(
            in Duration_t max_wait);

        DomainParticipant    get_participant();

        ReturnCode_t    set_default_datawriter_qos(
            in DataWriterQos qos);

        ReturnCode_t get_default_datawriter_qos(
            inout DataWriterQos qos);

        ReturnCode_t copy_from_topic_qos(
            inout DataWriterQos a_datawriter_qos,
            in TopicQos a_topic_qos);
};

interface DataWriter : Entity
{
        // InstanceHandle_t    register_instance(
        //             in Data instance_data);
        // InstanceHandle_t    register_instance_w_timestamp(
        //             in Data instance_data,
        //             in Time_t source_timestamp);
        // ReturnCode_t    unregister_instance(
        //             in Data instance_data,
        //             in InstanceHandle_t handle);
        // ReturnCode_t    unregister_instance_w_timestamp(
        //             in Data instance_data,
        //             in InstanceHandle_t handle,
        //             in Time_t source_timestamp);
        // ReturnCode_t    write(
```

```
//              in Data instance_data,
//              in InstanceHandle_t handle);
// ReturnCode_t    write_w_timestamp(
//              in Data instance_data,
//              in InstanceHandle_t handle,
//              in Time_t source_timestamp);
// ReturnCode_t    dispose(
//              in Data instance_data,
//              in InstanceHandle_t instance_handle);
// ReturnCode_t    dispose_w_timestamp(
//              in Data instance_data,
//              in InstanceHandle_t instance_handle,
//              in Time_t source_timestamp);
// ReturnCode_t    get_key_value(
//              inout Data key_holder,
//              in InstanceHandle_t handle);
// InstanceHandle_t    lookup_instance(
//              in Data instance_data);
ReturnCode_t    set_qos(
        in DataWriterQos qos);

ReturnCode_t    get_qos(
        inout DataWriterQos qos);

ReturnCode_t    set_listener(
        in DataWriterListener a_listener,
        in StatusMask mask);

DataWriterListener    get_listener();

Topic    get_topic();

Publisher    get_publisher();

ReturnCode_t    wait_for_acknowledgments(
        in Duration_t max_wait);

// Access the status
ReturnCode_t    get_liveliness_lost_status(
        inout LivelinessLostStatus);

ReturnCode_t    get_offered_deadline_missed_status(
        inout OfferedDeadlineMissedStatus status);
```

```
        ReturnCode_t    get_offered_incompatible_qos_status(
                inout OfferedIncompatibleQosStatus status);

        ReturnCode_t    get_publication_matched_status(
                inout PublicationMatchedStatus status);

        ReturnCode_t    assert_liveliness();

        ReturnCode_t    get_matched_subscriptions(
                inout InstanceHandleSeq subscription_handles);

        ReturnCode_t    get_matched_subscription_data(
                inout SubscriptionBuiltinTopicData subscription_data,
                in InstanceHandle_t subscription_handle);
};

// ---------------------------------------------------------------------
interface Subscriber : Entity
{
        DataReader    create_datareader(
                in TopicDescription a_topic,
                in DataReaderQos qos,
                in DataReaderListener a_listener,
                in StatusMask mask);

        ReturnCode_t    delete_datareader(
                in DataReader a_datareader);

        ReturnCode_t    delete_contained_entities();

        DataReader    lookup_datareader(
                in string topic_name);

        ReturnCode_t    get_datareaders(
                inout DataReaderSeq readers,
                in SampleStateMask sample_states,
                in ViewStateMask view_states,
                in InstanceStateMask instance_states);

        ReturnCode_t    notify_datareaders();

        ReturnCode_t    set_qos(
                in SubscriberQos qos);
```

```
        ReturnCode_t    get_qos(
                inout SubscriberQos qos);

        ReturnCode_t    set_listener(
                in SubscriberListener a_listener,
                in StatusMask mask);

        SubscriberListener    get_listener();

        ReturnCode_t    begin_access();

        ReturnCode_t    end_access();

        DomainParticipant    get_participant();

        ReturnCode_t    set_default_datareader_qos(
                in DataReaderQos qos);

        ReturnCode_t    get_default_datareader_qos(
                inout DataReaderQos qos);

        ReturnCode_t    copy_from_topic_qos(
                inout DataReaderQos a_datareader_qos,
                in TopicQos a_topic_qos);
};

interface DataReader : Entity
{
        // ReturnCode_t    read(
        //      inout DataSeq data_values,
        //      inout SampleInfoSeq sample_infos,
        //      in long max_samples,
        //      in SampleStateMask sample_states,
        //      in ViewStateMask view_states,
        //      in InstanceStateMask instance_states);

        // ReturnCode_t    take(
        //      inout DataSeq data_values,
        //      inout SampleInfoSeq sample_infos,
        //      in long max_samples,
        //      in SampleStateMask sample_states,
        //      in ViewStateMask view_states,
        //      in InstanceStateMask instance_states);
```

```
// ReturnCode_t    read_w_condition(
//         inout DataSeq data_values,
//         inout SampleInfoSeq sample_infos,
//         in long max_samples,
//         in ReadCondition a_condition);

// ReturnCode_t    take_w_condition(
//         inout DataSeq data_values,
//         inout SampleInfoSeq sample_infos,
//         in long max_samples,
//         in ReadCondition a_condition);

// ReturnCode_t    read_next_sample(
//         inout Data data_values,
//         inout SampleInfo sample_info);

// ReturnCode_t    take_next_sample(
//         inout Data data_values,
//         inout SampleInfo sample_info);

// ReturnCode_t    read_instance(
//         inout DataSeq data_values,
//         inout SampleInfoSeq sample_infos,
//         in long max_samples,
//         in InstanceHandle_t a_handle,
//         in SampleStateMask sample_states,
//         in ViewStateMask view_states,
//         in InstanceStateMask instance_states);

// ReturnCode_t    take_instance(
//         inout DataSeq data_values,
//         inout SampleInfoSeq sample_infos,
//         in long max_samples,
//         in InstanceHandle_t a_handle,
//         in SampleStateMask sample_states,
//         in ViewStateMask view_states,
//         in InstanceStateMask instance_states);

//ReturnCode_t    read_next_instance(
//         inout DataSeq data_values,
//         inout SampleInfoSeq sample_infos,
//         in long max_samples,
//         in InstanceHandle_t previous_handle,
```

```
//        in SampleStateMask sample_states,
//        in ViewStateMask view_states,
//        in InstanceStateMask instance_states);

// ReturnCode_t    take_next_instance(
//        inout DataSeq data_values,
//        inout SampleInfoSeq sample_infos,
//        in long max_samples,
//        in InstanceHandle_t previous_handle,
//        in SampleStateMask sample_states,
//        in ViewStateMask view_states,
//        in InstanceStateMask instance_states);

// ReturnCode_t    read_next_instance_w_condition(
//        inout DataSeq data_values,
//        inout SampleInfoSeq sample_infos,
//        in long max_samples,
//        in InstanceHandle_t previous_handle,
//        in ReadCondition a_condition);

// ReturnCode_t    take_next_instance_w_condition(
//        inout DataSeq data_values,
//        inout SampleInfoSeq sample_infos,
//        in long max_samples,
//        in InstanceHandle_t previous_handle,
//        in ReadCondition a_condition);

// ReturnCode_t    return_loan(
//        inout DataSeq data_values,
//        inout SampleInfoSeq sample_infos);

// ReturnCode_t    get_key_value(
//        inout Data key_holder,
//        in InstanceHandle_t handle);

// InstanceHandle_t    lookup_instance(
//        in Data instance_data);

ReadCondition    create_readcondition(
          in SampleStateMask sample_states,
          in ViewStateMask view_states,
          in InstanceStateMask instance_states);

QueryCondition    create_querycondition(
```

```
        in SampleStateMask sample_states,
        in ViewStateMask view_states,
        in InstanceStateMask instance_states,
        in string query_expression,
        in StringSeq query_parameters);

ReturnCode_t    delete_readcondition(
        in ReadCondition a_condition);

ReturnCode_t    delete_contained_entities();

ReturnCode_t    set_qos(
        in DataReaderQos qos);

ReturnCode_t    get_qos(
        inout DataReaderQos qos);

ReturnCode_t    set_listener(
        in DataReaderListener a_listener,
        in StatusMask mask);

DataReaderListener    get_listener();

TopicDescription    get_topicdescription();

Subscriber    get_subscriber();

ReturnCode_t    get_sample_rejected_status(
        inout SampleRejectedStatus status);

ReturnCode_t    get_liveliness_changed_status(
        inout LivelinessChangedStatus status);

ReturnCode_t    get_requested_deadline_missed_status(
        inout RequestedDeadlineMissedStatus status);

ReturnCode_t    get_requested_incompatible_qos_status(
        inout RequestedIncompatibleQosStatus status);

ReturnCode_t    get_subscription_matched_status(
        inout SubscriptionMatchedStatus status);

ReturnCode_t    get_sample_lost_status(
        inout SampleLostStatus status);
```

```
        ReturnCode_t    wait_for_historical_data(
                in Duration_t max_wait);

        ReturnCode_t    get_matched_publications(
                inout InstanceHandleSeq publication_handles);

        ReturnCode_t    get_matched_publication_data(
                inout PublicationBuiltinTopicData publication_data,
                in InstanceHandle_t publication_handle);
    };

    struct SampleInfo
    {
        SampleStateKind         sample_state;
        ViewStateKind           view_state;
        InstanceStateKind       instance_state;
        Time_t                  source_timestamp;
        InstanceHandle_t        instance_handle;
        InstanceHandle_t        publication_handle;
        long                    disposed_generation_count;
        long                    no_writers_generation_count;
        long                    sample_rank;
        long                    generation_rank;
        long                    absolute_generation_rank;
        boolean                 valid_data;
    };

    typedef    sequence<SampleInfo>    SampleInfoSeq;
};
//-------------------------end of module DDS---------------------------------

// Implied IDL for type "Foo"
// Example user defined structure
struct Foo
{
        long    dummy;
};

typedef    sequence<Foo>    FooSeq;

#include "dds_dcps.idl"

interface    FooTypeSupport : DDS::TypeSupport
```

```
{
    DDS::ReturnCode_t    register_type(
                in DDS::DomainParticipant participant,
                in string type_name);

    string    get_type_name();
};

interface    FooDataWriter : DDS::DataWriter
{
    DDS::InstanceHandle_t    register_instance(
                in Foo instance_data);

    DDS::InstanceHandle_t    register_instance_w_timestamp(
                in Foo instance_data,
                in DDS::Time_t source_timestamp);

    DDS::ReturnCode_t    unregister_instance(
                in Foo instance_data,
                in DDS::InstanceHandle_t handle);

    DDS::ReturnCode_t    unregister_instance_w_timestamp(
                in Foo instance_data,
                in DDS::InstanceHandle_t handle,
                in DDS::Time_t source_timestamp);

    DDS::ReturnCode_t    write(
                in Foo instance_data,
                in DDS::InstanceHandle_t handle);

    DDS::ReturnCode_t    write_w_timestamp(
                in Foo instance_data,
                in DDS::InstanceHandle_t handle,
                in DDS::Time_t source_timestamp);

    DDS::ReturnCode_t    dispose(
                in Foo instance_data,
                in DDS::InstanceHandle_t instance_handle);

    DDS::ReturnCode_t    dispose_w_timestamp(
                in Foo instance_data,
                in DDS::InstanceHandle_t instance_handle,
                in DDS::Time_t source_timestamp);
```

```
        DDS::ReturnCode_t    get_key_value(
                    inout Foo key_holder,
                    in DDS::InstanceHandle_t handle);

        DDS::InstanceHandle_t    lookup_instance(
                    in Foo key_holder);
};

interface FooDataReader : DDS::DataReader
{
        DDS::ReturnCode_t    read(
                    inout FooSeq data_values,
                    inout DDS::SampleInfoSeq sample_infos,
                    in long max_samples,
                    in DDS::SampleStateMask sample_states,
                    in DDS::ViewStateMask view_states,
                    in DDS::InstanceStateMask instance_states);

        DDS::ReturnCode_t    take(
                    inout FooSeq data_values,
                    inout DDS::SampleInfoSeq sample_infos,
                    in long max_samples,
                    in DDS::SampleStateMask sample_states,
                    in DDS::ViewStateMask view_states,
                    in DDS::InstanceStateMask instance_states);

        DDS::ReturnCode_t    read_w_condition(
                    inout FooSeq data_values,
                    inout DDS::SampleInfoSeq sample_infos,
                    in long max_samples,
                    in DDS::ReadCondition a_condition);

        DDS::ReturnCode_t    take_w_condition(
                    inout FooSeq data_values,
                    inout DDS::SampleInfoSeq sample_infos,
                    in long max_samples,
                    in DDS::ReadCondition a_condition);

        DDS::ReturnCode_t    read_next_sample(
                    inout Foo data_value,
                    inout DDS::SampleInfo sample_info);
```

```
    DDS::ReturnCode_t    take_next_sample(
                inout Foo data_value,
                inout DDS::SampleInfo sample_info);

    DDS::ReturnCode_t    read_instance(
                inout FooSeq data_values,
                inout DDS::SampleInfoSeq sample_infos,
                in long max_samples,
                in DDS::InstanceHandle_t a_handle,
                in DDS::SampleStateMask sample_states,
                in DDS::ViewStateMask view_states,
                in DDS::InstanceStateMask instance_states);

    DDS::ReturnCode_t    take_instance(
                inout FooSeq data_values,
                inout DDS::SampleInfoSeq sample_infos,
                in long max_samples,
                in DDS::InstanceHandle_t a_handle,
                in DDS::SampleStateMask sample_states,
                in DDS::ViewStateMask view_states,
                in DDS::InstanceStateMask instance_states);

    DDS::ReturnCode_t    read_next_instance(
                inout FooSeq data_values,
                inout DDS::SampleInfoSeq sample_infos,
                in long max_samples,
                in DDS::InstanceHandle_t previous_handle,
                in DDS::SampleStateMask sample_states,
                in DDS::ViewStateMask view_states,
                in DDS::InstanceStateMask instance_states);

    DDS::ReturnCode_t    take_next_instance(
                inout FooSeq data_values,
                inout DDS::SampleInfoSeq sample_infos,
                in long max_samples,
                in DDS::InstanceHandle_t previous_handle,
                in DDS::SampleStateMask sample_states,
                in DDS::ViewStateMask view_states,
                in DDS::InstanceStateMask instance_states);
}
```

附件 A 合规要点

该规范包括以下合规性配置文件。

•最小配置文件：此配置文件仅包含 DCPS 的强制功能。没有包含任何可选功能。

•内容订阅配置文件：此配置文件添加可选类：***ContentFilteredTopic***，***QueryCondition***，***MultiTopic***。此配置文件允许按内容进行订阅。请参见 2.2.2.3 主题定义模块。

•持久性配置文件：此配置文件添加了可选的 QoS 策略 DURABILITY_SERVICE 以及 DURABILITY QoS 策略 ***kind*** 的可选设置“TRANSIENT”和“PERSISTENT”。此配置文件使得 DDS 中间件可以将数据保存到 TRANSIENT 内存或永久存储中，以便它可以在 ***DataWriter*** 和系统外的生命周期中存活。见 2.2.3.4 持久性。

•所有权配置文件：此配置文件添加了两项内容，首先是 OWNERSHIP ***kind*** 的可选设置“EXCLUSIVE”；其次包括对可选 OWNERSHIP_STRENGTH 策略的支持。最后提供为 HISTORY QoS 策略设置 ***depth*** > 1 的能力。

•组访问配置文件：此配置文件包括对 PRESENTATION ***access_scope*** 设置为“GROUP”的支持（参见 2.2.3.6 PRESENTATION）。

附件 B 查询和过滤器的语法

B.1 引言

SQL 语法的一个子集应用于此规范的几个部分：

- ***ContentFilteredTopic*** 中的 ***filter_expression***（参见 2.2.2.3.3 ContentFilteredTopic 类）。
- ***MultiTopic*** 中的 ***topic_expression***（参见 2.2.2.3.4 MultiTopic 类[可选]）。
- ***QueryReadCondition*** 中的 ***query_expression***（参见 2.2.2.5.9 QueryCondition 类）。

这些表达式可以使用 SQL 的一个子集，可以在 SQL 表达式中使用程序变量。使用下面的 BNF 语法定义允许使用的 SQL 表达式。

B.2 BNF 中的 SQL 语法

```
Expression          ::=   FilterExpression
                    |     TopicExpression
                    |     QueryExpression
                    .
FilterExpression          ::=  Condition
TopicExpression           ::=  SelectFrom {Where } ';'
QueryExpression           ::=  {Condition}{'ORDER BY' (FIELDNAME // ',') }
                               .
SelectFrom                ::=  'SELECT' Aggregation 'FROM' Selection
                               .
Aggregation               ::=  '*'
                               |    (SubjectFieldSpec // ',')
                               .
SubjectFieldSpec          ::=  FIELDNAME
                               | FIELDNAME 'AS' FIELDNAME
                               | FIELDNAME FIELDNAME
                               .
Selection           ::=   TOPICNAME
                    |     TOPICTNAME NaturalJoin JoinItem
                    .
JoinItem            ::=   TOPICNAME
                    |     TOPICNAME NaturalJoin JoinItem
                    |     '(' TOPICNAME NaturalJoin JoinItem ')'
                    .
NaturalJoin ::= 'INNER NATURAL JOIN'
| 'NATURAL JOIN'
```

```
| 'NATURAL INNER JOIN'
.
Where               ::=  'WHERE' Condition
                         .
Condition           ::=  Predicate
                         |    Condition 'AND' Condition
                         |    Condition 'OR' Condition
                         |    'NOT' Condition
                         |    '(' Condition ')'
                         .
Predicate           ::=  ComparisonPredicate
                         |    BetweenPredicate
                         .
ComparisonPredicate      ::=  FIELDNAME RelOp Parameter
                              |    Parameter RelOp FIELDNAME
                              |    FIELDNAME RelOp FIELDNAME
                              .
BetweenPredicate         ::=  FIELDNAME 'BETWEEN' Range
                              |    FIELDNAME 'NOT BETWEEN' Range
                              .
RelOp               ::=  '=' | '>' | '>=' | '<' | '<=' | '<>' | like
.
Range               ::=  Parameter 'AND' Parameter
                         .
Parameter           ::=  INTEGERVALUE
                         |    CHARVALUE
                         |    FLOATVALUE
                         |    STRING
                         |    ENUMERATEDVALUE
                         |    PARAMETER
                         .
```

注意：INNER NATURAL JOIN，NATURAL JOIN 和 NATURAL INNER JOIN 都是别名，因为它们具有相同的语义。它们都受支持，因为它们都是 SQL 标准的一部分。

标记表达式

SQL 语法中使用的标记的语法和含义描述如下：

• FIELDNAME：对数据结构中字段的引用。点“.”用于浏览嵌套结构。可以在 FIELD-NAME 中使用的点数是无限的。FIELDNAME 可以引用数据结构中任何深度的字段。字段的名称是在相应结构的 IDL 定义中指定的名称，其可以与结构的特定语言（例如 C / C ++，Java）映射上出现的字段名称匹配或不匹配。

• TOPICNAME ：主题名称是主题的标识符，定义为任意字符“a”，…，“z”，“A”，…，“Z”，“0”，…，“9”组成的序列，但不能以数字开头。

• INTEGERVALUE：任意一数字序列，前面可以可选地带有加号或减号，表示系统范围内的十进制整数值。十六进制数字前面带有 0x，必须是有效的十六进制表达式。

• CHARVALUE：单引号之间包含的单个字符。

• FLOATVALUE：任何一系列数字，前面可以可选地带有加号或减号，并且可选地包括一个浮点（“.”）。十次幂表达式可以以后缀形式表示，其语法为 ***en***，其中 ***n*** 是数字，前面可以可选地带有加号或减号。

• STRING：用单引号封装的任何字符（除了换行符和右引号）序列。字符串以左引号或右引号开头，以右引号结尾。

• ENUMERATEDVALUE：枚举值是对枚举中声明的值的引用。枚举值由用单引号括起的枚举标签的名称组成。用于枚举标签的名称必须与枚举的 IDL 定义中指定的标签名称相对应。

• PARAMETER：参数的形式为 ***%n***，其中 ***n*** 表示小于 100 的自然数（包括零）。它指的是给定上下文中的第 n + 1 个参数。