实时发布订阅协议（RTPS）
DDS 互操作有线协议规范

1. 概述

1.1 引言

最近采用的数据分发服务（DDS）规范定义了应用程序级接口和 DDS 的行为，该服务支持实时系统中以数据为中心的发布订阅（DCPS）。DDS 规范使用模型驱动架构（MDA）方法来精确描述以数据为中心的通信模型：

- 应用程序如何为其希望发送和接收的数据建模；
- 应用程序如何与 DCPS 中间件交互并指定其希望发送和接收的数据以及服务质量（QoS）要求；
- 如何发送和接收数据（相对于 QoS 要求），
- 应用程序如何访问数据；
- 应用程序从中间件状态中获得的反馈类型。

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。

DDS 规范还包括特定于平台的 IDL 映射，因此使用 DDS 的应用程序只需重新编译即可在 DDS 实现之间切换。通过这种方式 DDS 解决了“应用程序可移植性”问题。

DDS 规范没有提出中间件通过 TCP/UDP/IP 等传输方式交换消息的协议，因此除非提供特定于供应商的“网桥”，否则 DDS 的不同实现将不会实现彼此之间的互操作。这种情况类似于其他消息传递的 API 标准（如 JMS）。

随着 DDS 在大型分布式系统中的日益普及，需要定义标准的“有线协议”，允许来自多个供应商的 DDS 实现进行互操作。所需的“DDS 有线协议”应该能够利用 DDS 可配置的 QoS 设置来优化其对底层传输能力的使用。特别是所需的有线协议必须能够利用多播，尽力而为和许多 DDS QoS 的无连接的特性。

1.2 DDS 有线协议的要求

在网络通信中，与许多其他工程领域一样，事实上“一种尺寸并不适合所有情况”。工程设计是关于做出正确的权衡取舍，这些权衡必须平衡相互冲突的需求，如通用性，易用性，功能丰富性，性能，内存大小和使用，可扩展性，确定性和健壮性。这些权衡取决于信息流的类型（例如，周期性与突发性，基于状态与基于事件，一对多与请求应答，尽力而为与可

靠，小数据值与大文件等），以及应用程序和执行平台施加的约束。因此，出现了许多成功的协议，例如 HTTP，SOAP，FTP，DHCP，DCE，RTP，DCOM 和 CORBA。这些协议中的每一个都填补了一项空白，为特定目的或应用领域提供了良好的功能。

DDS 的基本通信模型是单向数据交换的一种，其中发布数据的应用程序将相关数据更新“推送”到订阅者的本地高速缓存。此信息流由 DataWriter 和 DataReader 之间隐式建立的 QoS 合约进行管理。DataWriter 在声明其发布数据的意图时指定其 QoS 合约，DataReader 在声明其订阅数据的意图时指定它的 QoS 合约。通信模式通常包括多对多样式配置。部署 DDS 技术的应用程序主要关注的是以最小的开销和有效的方式分发信息。另一个重要的需求是以强大的容错方式扩展到数百或数千个订阅者。

DDS 规范规定了必须要有内置发现服务，允许发布者动态发现订阅者，反之亦然，并且无需联系任何名称服务器即可连续执行此任务。

DDS 规范还规定 DDS 实现不应引入任何单点故障。因此协议不能依赖于集中式名称服务器或集中式信息代理。

部署 DDS 的应用程序的大规模，松耦合，动态特性以及适应新兴传输方式的需求要求数据定义和协议具有一定的灵活性，以便每个都可以发展进化，同时保持与已部署系统的向下兼容性。

1.3 RTPS 有线协议

实时发布订阅（RTPS）协议源于工业自动化，实际上已被 IEC 批准为实时工业以太网套件 IEC-PAS-62030 的一部分。它是一种经过现场验证的技术，目前已在全球数千个工业设备中部署。

RTPS 专门用于支持数据分发系统的独特要求。作为 DDS 所针对的应用领域之一，工业自动化社区定义了与 DDS 匹配的标准发布订阅线协议的需求。在底层的行为架构和 RTPS 的特征方面，DDS 和 RTPS 有线协议之间存在紧密的协同作用。

RTPS 协议旨在能够在多播和无连接的尽力传输方式（例如 UDP/IP）上运行。 RTPS 协议的主要特点包括：

- 性能和服务质量属性，通过标准 IP 网络为实时应用程序提供尽力而为和可靠的发布订阅通信。
- 容错，允许创建没有单点故障的通信网络。
- 可延展性，允许使用新服务来扩展和增强协议，同时不破坏向下兼容性和互操作性。
- 即插即用连接，以便自动发现新的应用程序和服务，应用程序可以随时加入和离开网络，无需重新配置。
- 可配置性，平衡数据传输的可靠性和及时性需求。
- 模块化，允许简单设备实现协议的子集并加入网络。
- 可扩展性，使系统可以扩展到非常大的网络。
- 类型安全，以防止应用程序编程错误影响远程节点的运营。

上述特性使 RTPS 成为 DDS 有线协议的绝佳搭档。鉴于其发布订阅机制的根源，RTPS 专门用于满足 DDS 应用程序域提出的需求类型，这并不是巧合。

此规范定义了消息格式，解释和使用场景，它们是使用 RTPS 协议的应用程序交换传输所有消息的基础。

1.4 RTPS 平台无关模型（PIM）

根据平台无关模型（PIM）和一组 PSM 描述 RTPS 协议。

RTPS PIM 包含四个模块：结构，消息，行为和发现。结构（Structure）模块定义通信端点。 消息（Messages）模块定义这些端点可以交换的消息集合。行为（Behavior）模块定义了合法交互集（消息交换）以及它们如何影响通信端点的状态。换句话说，Structure 模块定义了协议“参与者”，Messages 模块定义了“语法符号”，而 Behavior 模块定义了不同会话的合法语法和语义。发现（Discovery）模块定义如何自动发现和配置实体。

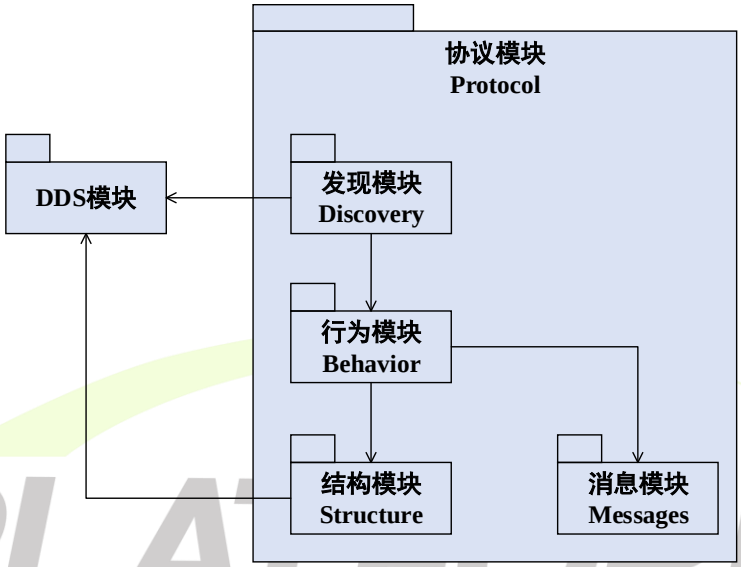


图 1-1 RTPS 模块

在 PIM 中，消息是根据其语义内容定义的。可以将此 PIM 映射到各种平台特定模型（PSM）中，例如普通 UDP 或 CORBA 事件。

1.4.1 结构模块

鉴于其发布订阅的根源，RTPS 自然地映射到多个 DDS 概念中。本规范会使用许多 DDS 规范已经使用的相同核心实体。如图 1.2 所示，所有 RTPS 实体都与 RTPS 域相关联，RTPS 域表示包含一组**参与者（Participants）**的单独通信平面。参与者包含本地**端点（Endpoints）**。有两种类型的端点：**读取者（Readers）**和**写入者（Writers）**。读取者和写入者是通过发送 RTPS 消息来传达信息的参与者。写入者告知数据的存在并将**数据域（Domain）**上的本地可用数据发送给读取者，读取者可以请求和确认数据。

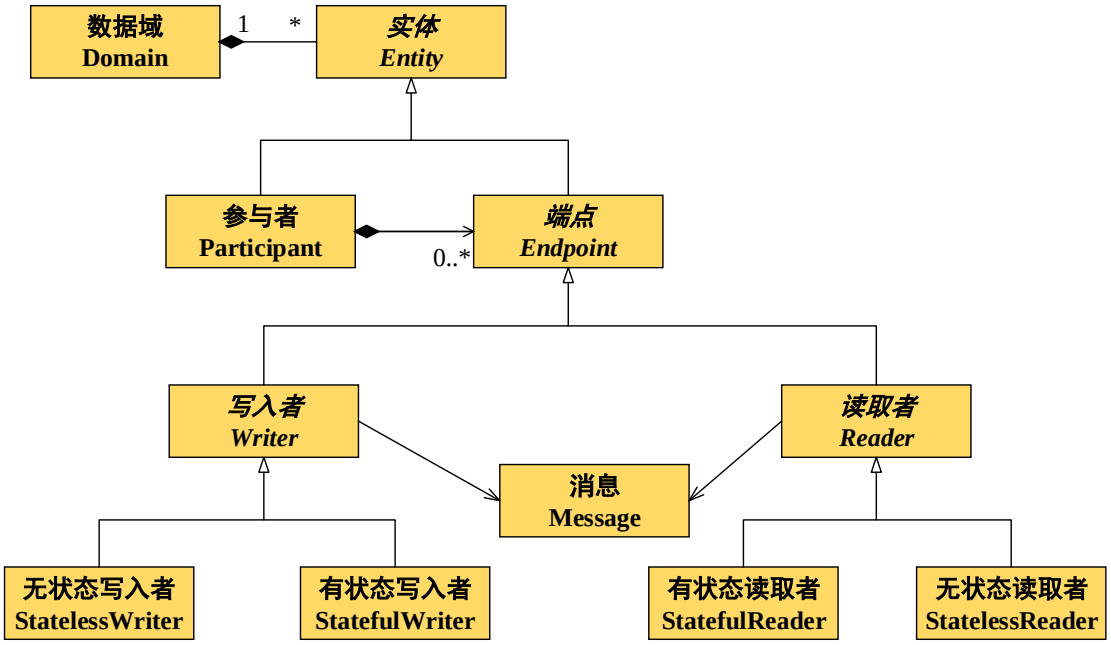


图 1-2 RTPS 结构模块

RTPS 协议中的活动者与 DDS 实体一一对应，这是通信产生的原因。如图 1.3 所示。

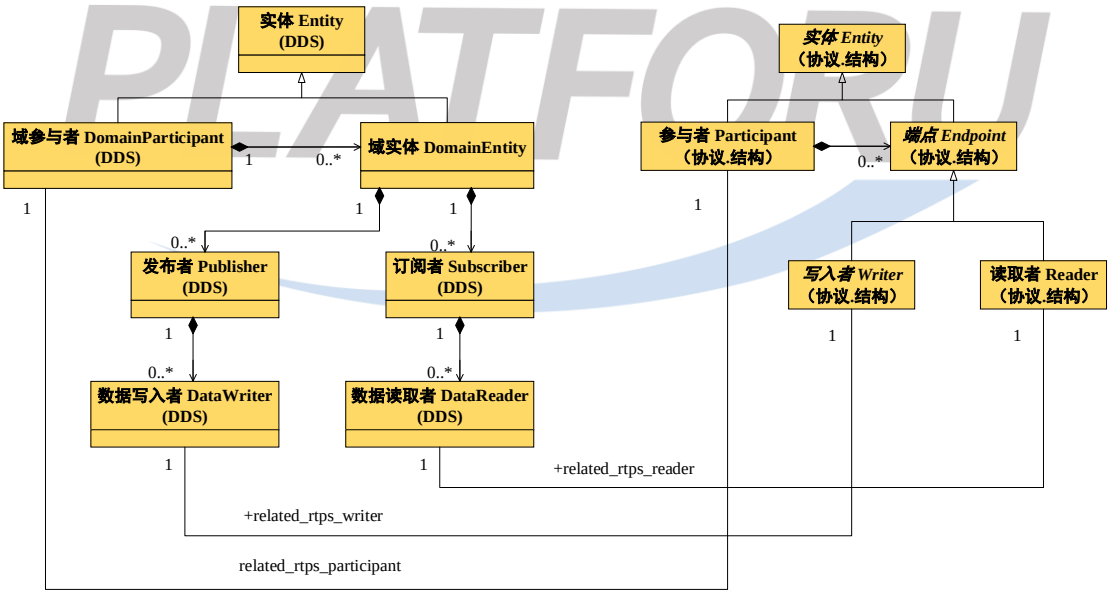


图 1-3 RTPS 与 DDS 对应关系

结构模块在 2.2 中描述。

1.4.2 消息模块

消息模块定义 RTPS 写入者和读取者之间的原子信息交换的内容。**消息（Message）**由一个**报文头（Header）**后跟一些**子消息（Submessage）**组成，如图 2.4 所示。每个消息都是由一系列元素构建的。选择此结构是为了允许扩展子消息的内容和每个子消息的组成，同时

保持向下兼容性。

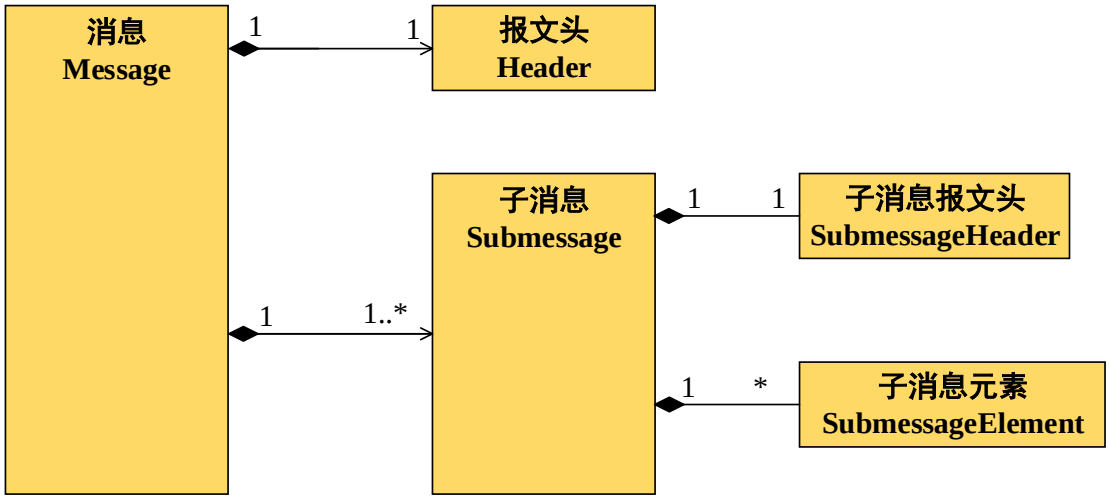


图 1-4 RTPS 消息模块

消息模块将在 2.3 中详细讨论。

1.4.3 行为模块

行为模块描述了可以在 RTPS 写入者和读取者之间交换的消息序列，以及时间和每条消息引起的写入者和读取者状态的变化。

互操作性所需的行为是根据实现必须遵循的最小规则集来描述的，以便实现互操作。实际实现可能表现出超出这些最低要求的不同行为，具体取决于他们希望如何权衡扩展性，内存要求和带宽使用。

为了说明这个概念，行为模块定义了两个参考实现。一个参考实现基于**有状态写入者（StatefulWriters）**和**有状态读取者（StatefulReaders）**，另一个基于**无状态写入者（StatelessWriters）**和**无状态读取者（StatelessReaders）**，如图 2.2 所示。两种参考实现都满足互操作的最低要求，因此可以互操作，但由于它们存储在匹配的远程实体上的信息不同，因此表现出略微不同的行为。 RTPS 协议的实际实现的行为可以是参考实现的完全匹配或组合。

行为模块在 2.4 中描述。

1.4.4 发现模块

发现模块描述了使**参与者（Participants）**能够获取域中所有其他**参与者（Participants）**和**端点（Endpoints）**的存在和属性信息的协议。这种*元通信（metatraffic）*使每个**参与者（Participant）**能够获得域中所有**参与者（Participants）**，**读取者（Readers）**和**写入者（Writers）**的完整信息，并配置本地写入者与远程读取者进行通信，以及本地读取者与远程写入者进行通信。

发现是一个单独的模块。发现的独特需求，即写入者和读取者进行匹配所需的所有信息需要通过透明地即插即用传播，使得单个架构或协议不可能满足各种各样的可扩展性，性能和将部署 DDS 的异构网络的嵌入性需求。从此以后，引入多种发现机制是有意义的，这些机制从简单和有效（但不是很可扩展）到更复杂的分层（但更具可扩展性）机制。

发现模块在 2.5 中描述

1.5 RTPS 平台特定模型（PSM）

特定于平台的模型将 RTPS PIM 映射到特定的底层平台。它定义了所有 RTPS 类型和消息的位和字节的精确表示以及专属于平台的其他信息。

可能支持多个 PSM，但 DDS 的所有实现必须在 UDP/IP 之上实现 PSM，这在第 3 章中介绍。

1.6 RTPS 传输模型

RTPS 支持各种传输方式和传输 QoS。该协议旨在能够在多播和尽力而为的传输方式（例如 UDP/IP）上运行，并且只需要该传输方式提供非常简单的服务。实际上传输方式提供能够最大限度地发送数据包的无连接服务就足够了。也就是说传输方式不需要保证每个数据包会到达其目的地或者数据包按顺序传送。如果需要，RTPS 在传输接口以上实现数据传输和状态的可靠性。这并不排除 RTPS 在可靠的传输方式之上实现。它使得支持更广泛的底层传输方式成为可能。

如果可以的话，RTPS 还可以利用传输机制的多播功能，来自发送方的一条消息可以到达多个接收方。RTPS 旨在促进底层通信机制的确定性。该协议提供了确定性和可靠性之间的公开权衡。

RTPS 对底层传输方式的一般要求可归纳如下：

- 传输具有单播地址（长度应在 16 字节以内）的通用概念。
- 传输具有端口（长度应在 4 字节以内）的通用概念，例如可以是 UDP 端口或者共享存储器段中的偏移等。
- 传输可以将数据报（未解释的八位字节序列）发送到特定的地址/端口。
- 传输可以在特定地址/端口接收数据报。
- 如果传输过程中消息不完整或已损坏，传输将丢弃消息（即 RTPS 假定消息已完成且未损坏）。
- 传输提供推断接收消息大小的方法。

2. 平台无关模型（PIM）

2.1 引言

本节定义了 RTPS 协议的平台无关模型（PIM）。后续小节将 PIM 映射到各种平台，最基本的平台是原生 UDP 数据包。

PIM 以“虚拟机”的形式描述协议。虚拟机的结构是根据 2.2 中描述的类构建的，其中包括**写入者（Writer）**和**读取者（Reader）**端点。这些端点使用 2.3 中描述的消息进行通信。2.4 小节描述了虚拟机的行为，即端点之间发生的消息交换。它列出了互操作性的要求，并使用状态图定义了两个参考实现。2.5 小节定义了用于为虚拟机配置与远程对等方通信所需的信息的发现协议。2.6 小节描述了如何扩展协议以满足未来需求。最后 2.7 小节描述了如

何使用 RTPS 实现 DDS QoS 和一些高级 DDS 功能。

引入 RTPS 虚拟机的唯一目的是以完整且清晰的方式描述协议。该描述的目的是不以任何方式约束内部实现。兼容实现的唯一标准是外部可观察的行为满足互操作性的要求。实现可以基于其他类，并且可以使用除状态机之外的编程结构来实现 RTPS 协议。

2.2 结构模块

本小节描述了作为通信参与者的 RTPS 实体的结构。RTPS 协议使用的主要类如图 2.1 所示。

2.2.1 概述

RTPS 实体是应用程序可见的 DDS 实体用于彼此通信的协议级端点。

每个 RTPS 实体与 DDS 实体一一对应。 ***HistoryCache*** 构成 DDS 实体与其对应的 RTPS 实体之间的接口。例如，DDS ***DataWriter*** 上的 ***write*** 方法会将 ***CacheChange*** 添加到其相应 RTPS ***Writer*** 的 ***HistoryCache*** 中。RTPS ***Writer*** 随后将 CacheChange 传输到所有匹配的 RTPS ***Reader*** 的 ***HistoryCache*** 中。在接收端，RTPS ***Reader*** 通知 DDS ***DataReader*** 新的 CacheChange 已到达 ***HistoryCache***，此时 DDS ***DataReader*** 可以选择使用 DDS ***read*** 或 ***take*** API 来访问它。

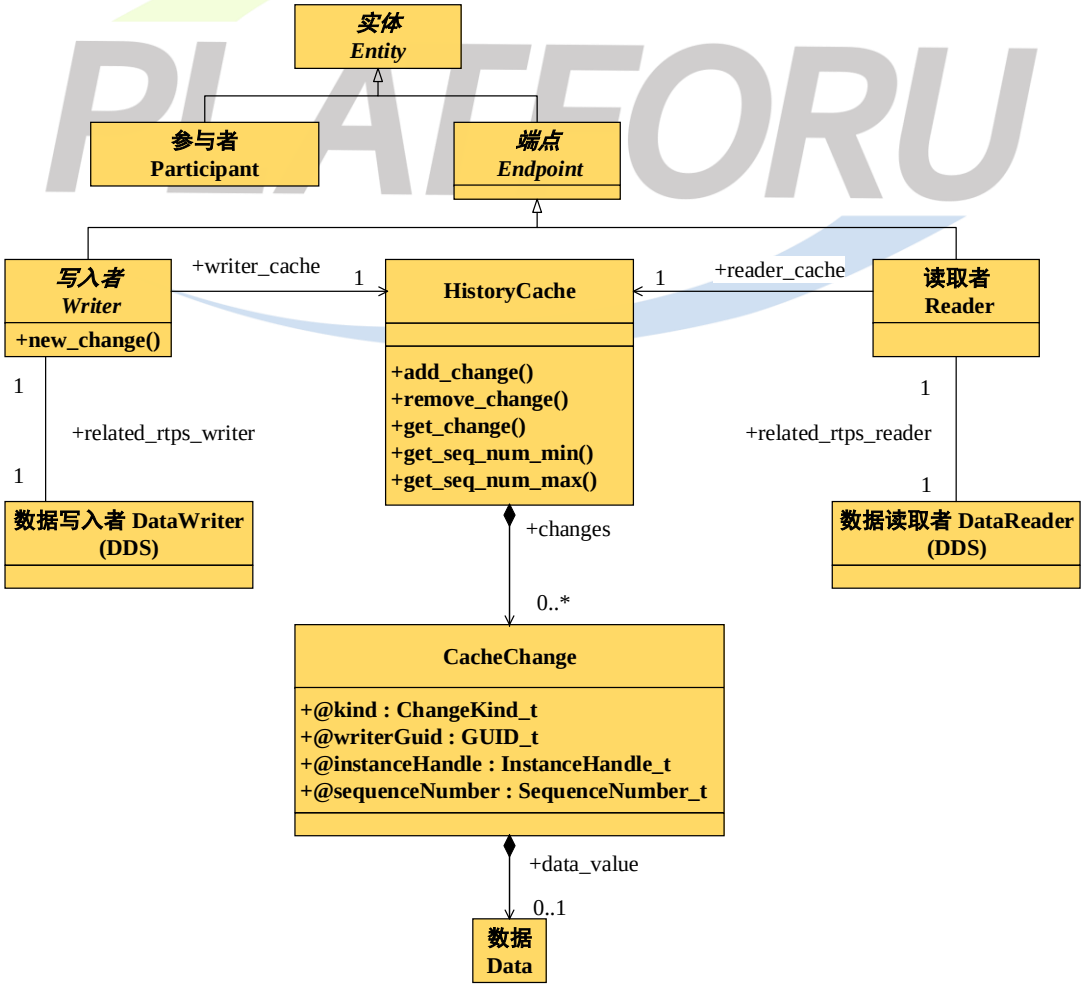


图 2-1 RTPS 结构模块类图

此小节概述了 RTPS 虚拟机使用的主类以及用于描述其属性的类型。后续子小节详细描述了每个类。

2.2.1.1 RTPS 虚拟机使用的类的总结

所有 RTPS 实体都继承自 RTPS 实体类。 表 2.1 列出了 RTPS 虚拟机使用的类。

表 2-1 RTPS 实体和类概述

RTPS 实体和类	
类	**目的**
Entity	所有 RTPS 实体的基类。RTPS ***Entity*** 类表示网络上其他 RTPS 实体可见的对象类。因此，RTPS ***Entity*** 对象具有全局唯一标识符（GUID）并且可以在 RTPS 消息内引用
Endpoint	RTPS ***Entity*** 的特殊化表示可以作为通信端点的对象。即可以是 RTPS 消息的源或目的地址的对象。
Participant	所有 RTPS 实体的容器，它们共享公共属性并位于单个地址空间中
Writer	RTPS ***Endpoint*** 的特殊化，表示可以作为传递 ***CacheChanges*** 的消息源的对象
Reader	RTPS ***Endpoint*** 的特殊化，表示可用于接收传递 ***CacheChanges*** 的消息的对象
HistoryCache	用于临时存储和管理数据对象修改集合的容器类。 在写入者这一端，它包含写入者对数据对象所做更改的历史记录。没有必要保留所有更改的完整历史。而需要的是为现有和未来匹配的 RTPS 读取者端点提供服务所需的部分历史记录。所需的部分历史记录取决于 DDS QoS 以及与匹配的读取者端点的通信状态。 在读取者这一端，它包含匹配的 RTPS ***Writer*** 端点对数据对象所做更改的历史记录。没有必要保留所有更改的完整历史。而需要的是包含根据需要从匹配的写入者接收的数据改变叠加的部分历史，以满足相应的 DDS DataReader 的需要。此叠加的规则和所需的部分历史记录的数量取决于 DDS QoS 以及与匹配的 RTPS Writer 端点的通信状态。
CacheChange	表示对数据对象进行的单个更改。包括数据对象的创建，修改和删除。
Data	表示可能与对数据对象所做的更改相关联的数据。

2.2.1.2 用于描述 RTPS 实体和类的类型的总结

虚拟机使用的实体和类都包含一组属性。表 2.2 总结了属性的类型。

表 2-2 RTPS 实体和类中显示的属性类型

RTPS 实体和类中使用的类型	
属性类型	**目的**
GUID_t	用于保存全局唯一 RTPS 实体标识符的类型。这些是用于唯一引用系统中每个 RTPS 实体的标识符。 必须可以使用 16 个八位字节表示。 协议保留以下值：GUID_UNKNOWN
GuidPrefix_t	用于保存全局唯一 RTPS 实体标识符前缀的类型。属于同一参与者的实体的 GUID 都具有相同的前缀（见 2.2.4.3）。 必须可以使用 12 个八位字节表示。 协议保留以下值：GUIDPREFIX_UNKNOWN
EntityId_t	用于保存全局唯一 RTPS 实体标识符的后缀部分的类型。 ***EntityId_t*** 唯一标识**参与者（Participant）**中的**实体（Entity）**。 必须可以使用 4 个八位字节表示。 协议保留以下值：ENTITYID_UNKNOWN 其他预定义值由 2.5 中的发现模块定义。
SequenceNumber_t	用于保存序列号的类型。 必须可以使用 64 bit 字符表示。 协议保留以下值：SEQUENCENUMBER_UNKNOWN
Locator_t	用于表示使用其中一个支持的传输方式将消息发送到 RTPS 端点所需的寻址信息的类型。 应该能够拥有一个鉴别器来识别传输方式的种类，地址，和端口号。必须能够使用 4 个八位字节来分别表示鉴别器和端口号，该地址使用 16 个八位字节。 协议保留以下值： LOCATOR_INVALID LOCATOR_KIND_INVALID LOCATOR_KIND_RESERVED LOCATOR_KIND_UDPv4 LOCATOR_KIND_UDPv6 LOCATOR_ADDRESS_INVALID LOCATOR_PORT_INVALID
TopicKind_t	枚举用于区分主题是否已定义某些字段为主题中的数据实例的“关键字”。有关关键字的更多详细信息，请参阅 DDS 规范。 协议保留以下值： NO_KEY WITH_KEY
ChangeKind_t	枚举用于区分对数据对象所做的更改类型。 包括对数据或数据对象生命周期的更改。 它可以采取以下值： ALIVE NOT_ALIVE_DISPOSED NOT_ALIVE_UNREGISTERED

表 2-2 RTPS 实体和类中显示的属性类型（续）

RTPS 实体和类中使用的类型	
属性类型	**目的**
ReliabilityKind_t	枚举用于指示通信的可靠性级别。 它可以采取以下值： BEST_EFFORT，RELIABLE。
InstanceHandle_t	用于表示数据对象标识的类型，其值的变化由 RTPS 协议传递。
ProtocolVersion_t	用于表示 RTPS 协议版本的类型。该版本由主版本号和次版本号组成。另见 2.6。 协议保留以下值： PROTOCOLVERSION PROTOCOLVERSION_1_0 PROTOCOLVERSION_1_1 PROTOCOLVERSION_2_0 PROTOCOLVERSION_2_1 PROTOCOLVERSION_2_2 在下面这种情况下，PROTOCOLVERSION 是最新版本的别名 PROTOCOLVERSION_2_2
VendorId_t	用于表示实现 RTPS 协议的服务的供应商的类型。 *vendorId* 的可能值由 OMG 分配。 协议保留以下值： VENDORID_UNKNOWN

2.2.1.3 RTPS 实体的配置属性

RTPS 实体由一组属性进行配置。 其中一些属性映射到相应 DDS 实体上设置的 QoS 策略。其他属性表示允许将协议的行为调整到特定传输方式和部署场景的参数。其他属性编码 RTPS 实体的状态，不用于配置行为。

用于配置 RTPS 实体子集的属性如图 2.2 所示。 配置 ***Writer*** 和 ***Reader*** 实体的属性与协议行为密切相关，将在 2.4 中介绍。

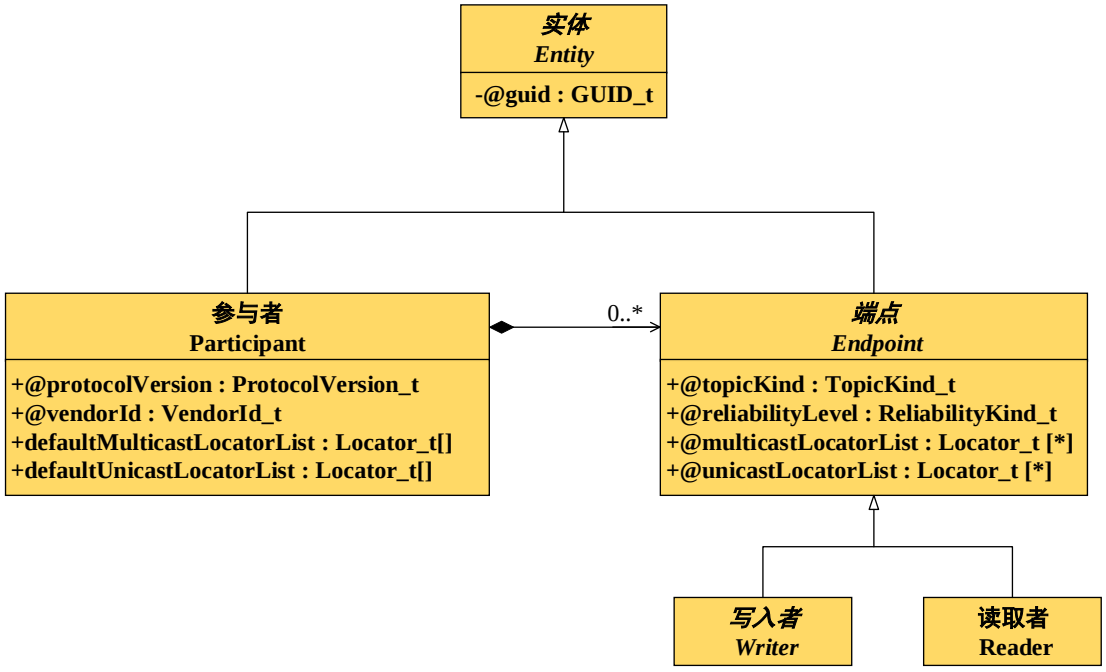


图 2-2 用于配置主 RTPS 实体的属性

本小节的剩余部分更详细地描述了每个 RTPS 实体。

2.2.2 RTPS HistoryCache

HistoryCache 是 DDS 和 RTPS 之间接口的一部分，在读取者和写入者两端扮演着不同的角色。

在写入者这一端 ***HistoryCache*** 包含对相应 DDS 写入者对数据对象更改的部分历史记录，这些更改是为现有和未来匹配的 RTPS 读取者端点提供服务所必需的。所需的部分历史记录取决于 DDS QoS 设置以及与匹配的 RTPS 读取者端点的通信状态。

在读取者端，它包含由所有匹配的 RTPS 写入者端点对数据对象的部分更改的叠加。

“部分”一词用于表示不必保留所有更改的完整历史记录，需要的是满足 RTPS 协议的行为需求和相关 DDS 实体的 QoS 需求所需的历史记录的子集。定义该子集的规则由 RTPS 协议定义，并且取决于通信协议的状态和相关 DDS 实体的 QoS。

HistoryCache 是 DDS 和 RTPS 之间接口的一部分。换句话说，RTPS 实体及其相关的 DDS 实体都能够在其关联的 ***HistoryCache*** 上调用方法。

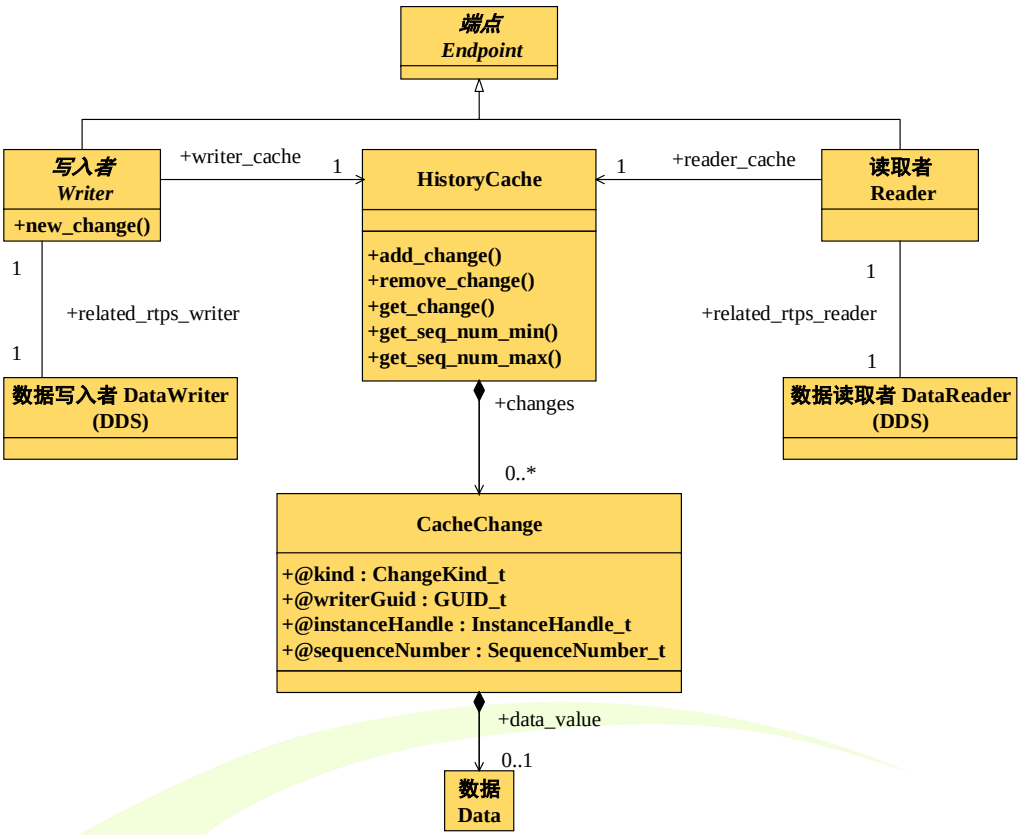


图 2-3 RTPS HistoryCache 类

HistoryCache 属性在表 2.3 中列出。

表 2-3 RTPS HistoryCache 属性

RTPS HistoryCache			
属性	类型	含义	与 DDS 关系
changes	CacheChange[*]	HistoryCache 中包含的 CacheChanges 列表。	N/A.

RTPS 实体和相关的 DDS 实体使用表 2.4 中的方法与 ***HistoryCache*** 交互。

表 2-4 RTPS HistoryCache 方法

RTPS HistoryCache 方法		
方法名称	参数列表	参数类型
new	<return value>	HistoryCache
add_change	<return value>	void
	a_change	CacheChange
remove_change	<return value>	void
	a_change	CacheChange
get_seq_num_min	<return value>	SequenceNumber_t
get_seq_num_max	<return value>	SequenceNumber_t

以下子小节列出了上述方法的详细信息。

2.2.2.1 new

此方法将创建一个新的 RTPS ***HistoryCache***。使用空的修改列表初始化新创建的历史记录缓存。

2.2.2.2 add_change

此方法将 ***CacheChange*** a_change 插入 ***HistoryCache***。

如果没有足够的资源将修改添加到 ***HistoryCache***，则此方法将调用失败。DDS 服务实现负责以与 DDS 实体 RESOURCE_LIMITS QoS 一致的方式配置 ***HistoryCache***，并以 DDS 规范指定的方式将所有错误反馈给 DDS 用户。

此方法执行以下逻辑步骤：

ADD a_change TO this.changes;

2.2.2.3 remove_change

此方法表明先前添加的 ***CacheChange*** 已变得无关紧要，并且无需在 ***HistoryCache*** 中维护有关 ***CacheChange*** 的详细信息。基于与相关 DDS 实体相关联的 QoS 以及 ***CacheChange*** 的确认状态来确定不相关性，这在 2.4.1 中有所描述。

此方法执行以下逻辑步骤：

REMOVE a_change FROM this.changes;

2.2.2.4 get_seq_num_min

此方法获取存储在 ***HistoryCache*** 中的 ***CacheChange*** 中的 CacheChange :: sequenceNumber 属性的最小值。

此方法执行以下逻辑步骤：

min_seq_num := MIN { change.sequenceNumber WHERE (change IN this.changes) }
return min_seq_num;

2.2.2.5 get_seq_num_max

此方法获取存储在 ***HistoryCache*** 中的 ***CacheChange*** 中的 CacheChange :: sequenceNumber 属性的最大值。

此方法执行以下逻辑步骤：

max_seq_num := MAX { change.sequenceNumber WHERE (change IN this.changes) }
return max_seq_num;

2.2.3 RTPS CacheChange

用于表示添加到 ***HistoryCache*** 的每个更改的类。表 2.5 列出了 CacheChange 属性。

表 2-5 RTPS CacheChange 属性

RTPS CacheChange			
属性	类型	含义	与 DDS 关系
kind	ChangeKind_t	确定数据更改的类型。参见表 2.2	DDS 实例状态类型
writerGuid	GUID_t	进行数据更改的 RTPS 写入者的身份标识	N/A.
instanceHandle	InstanceHandle_t	标识此更改适用的数据对象的实例	在 DDS 中，数据中标记为“关键字”的字段的值唯一地标识每个数据对象
sequenceNumber	SequenceNumber_t	由 RTPS 写入者分配的序列号，用于唯一标识更改	N/A.
data_value	Data	与更改关联的数据值。根据 CacheChange 的 kind，可能没有关联的数据。见表 2.2。	N/A.

2.2.4 RTPS 实体（Entity）

RTPS 实体是所有 RTPS 实体的基类，并映射到 DDS 实体。表 2.6 列出了***实体（Entity）***配置属性。

表 2-6 RTPS 实体（Entity）属性

RTPS Entity			
属性	类型	含义	与 DDS 关系
guid	GUID_t	全局且唯一地标识 DDS 域内的 RTPS ***实体（Entity）***	映射到 DDS BuiltinTopicKey_t 的值，用于描述相应的 DDS 实体。更多详细信息，请参阅 DDS 规范。

2.2.4.1 识别 RTPS 实体：GUID

GUID（Globally Unique Identifier，全局唯一标识符）是所有 RTPS 实体都具有的属性，并唯一标识 DDS 域内的实体。

GUID 构建为组合 ***GuidPrefix***_t *前缀（prefix）*和 ***EntityId_t*** *entityId* 的二元组< prefix，entityId>。

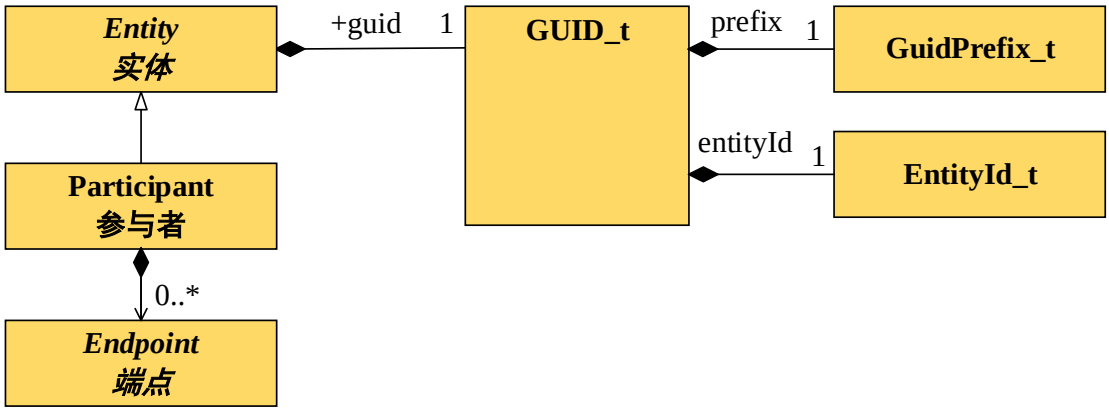


图 2-4 RTPS GUID_t 唯一标识实体，由前缀和后缀组成

表 2-7 GUID_t 的结构

字段	类型	含义
prefix	GuidPrefix_t	唯一标识域内的参与者
entityId	EntityId_t	唯一标识参与者中的实体

2.2.4.2 RTPS 参与者（Participants）的 GUID

每个参与者都具有 GUID< prefix, ENTITYID_PARTICIPANT>，其中常量 ENTITYID_PARTICIPANT 是由 RTPS 协议定义的特殊值。它的实际值取决于 PSM。

只要确保域中的每个参与者都具有唯一的 GUID，该实现就可以自由选择前缀。

2.2.4.3 参与者中 RTPS 端点的 GUID

具有 GUID <participantPrefix，ENTITYID_PARTICIPANT>的***参与者（Participant）***包含的***端点（Endpoint）***具有 GUID <participantPrefix，*entityId*>。*entityId* 是端点相对于参与者的唯一标识。这会产生以下几种结果：

- ***参与者（Participant）***中所有***端点（Endpoint）***的 GUID 具有相同的前缀。
- 一旦知道***端点（Endpoint）***的 GUID，包含端点的***参与者（Participant）***的 GUID 也是已知的。
- 任何***端点（Endpoint）***的 GUID 都可以根据其所属的***参与者（Participant）***的 GUID 及其 *entityId* 推断出来。

每个 RTPS 实体的 *entityId* 的选择取决于 PSM。

2.2.5 RTPS 参与者（Participant）

RTPS ***参与者（Participant）***是 RTPS ***端点（Endpoint）***实体的容器，并映射到 DDS DomainParticipant。此外 RTPS ***参与者（Participant）***促进了单个 RTPS ***参与者（Participant）***中的 RTPS ***端点（Endpoint）***实体共享共同属性。

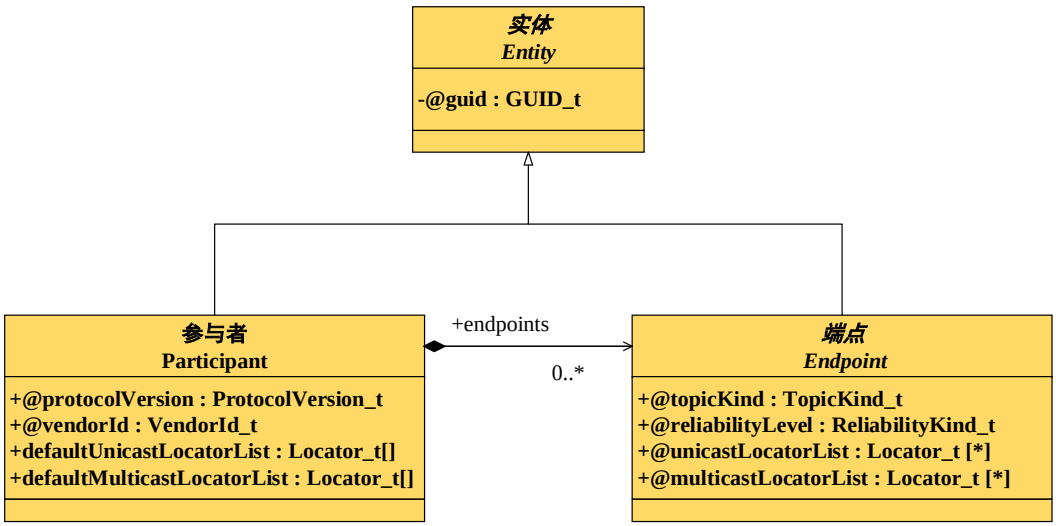


图 2-5 RTPS Participant

RTPS ***参与者（Participant）***包含表 2.8 中显示的属性。

表 2-8 RTPS Participant 属性

RTPS Participant : RTPS Entity			
属性	类型	含义	与 DDS 关系
defaultUnicastLocatorList	Locator_t[*]	可用于将消息发送到参与者中包含的端点的单播定位器（传输方式，地址，端口组合）的默认列表； 这些是单播定位器，将在端点未指定其自定义定位器集的情况下使用。	N/A. 通过发现配置
defaultMulticastLocatorList	Locator_t[*]	可用于将消息发送到参与者中包含的端点的多播定位器（传输方式，地址，端口组合）的默认列表； 这些是多播定位器，将在端点未指定其自定义定位器集的情况下使用。	N/A. 通过发现配置
protocolVersion	ProtocolVersion_t	标识参与者用于通信的 RTPS 协议的版本。	N/A. 由每个版本的协议指定。
vendorId	VendorId_t	标识包含参与者的 RTPS 中间件的供应商。	N/A. 由每个供应商设置

2.2.6 RTPS 端点（Endpoint）

从 RTPS 协议的角度来看，RTPS ***端点（Endpoint）***表示可能的通信端点。有两种 RTPS ***端点（Endpoint）***实体：***Writer*** 端点和 ***Reader*** 端点。

RTPS ***Writer*** 端点将 ***CacheChange*** 消息发送到 RTPS ***Reader*** 端点，并可能接收它们发送的更改的确认。RTPS ***Reader*** 端点从 ***Writer*** 端点接收 ***CacheChange*** 和更改可用性声明，并可能确认更改和/或请求丢失的更改。

RTPS ***端点（Endpoint）***包含表 2.9 中显示的属性。

表 2-9 RTPS Endpoint 配置属性

RTPS Endpoint : RTPS Entity			
属性	类型	含义	与 DDS 关系
unicastLocatorList	Locator_t[*]	可用于将消息发送到 ***端点（Endpoint）***的单播定位器列表（传输方式，地址，端口组合）。该列表可能为空。	N/A. 通过发现配置
multicastLocatorList	Locator_t[*]	可用于将消息发送到 ***端点（Endpoint）***的多播定位器列表（传输方式，地址，端口组合）。该列表可能为空。	N/A. 通过发现配置
reliabilityLevel	ReliabilityKind_t	***端点（Endpoint）***支持的可靠性级别。	映射到 RELIABILITY QoS “kind”
topicKind	TopicKind_t	用于指示 ***端点（Endpoint）***是否与将某些字段定义为包含 DDS 关键字的 DataType 相关联。	由与 RTPS ***端点（Endpoint）***相关的 DDS 主题关联的数据类型定义

2.2.7 RTPS 写入者（Writer）

RTPS ***Writer*** 是 RTPS ***端点（Endpoint）***的特殊实例，负责将 ***CacheChange*** 消息发送到匹配的 RTPS ***Reader*** 端点。它的作用是将其 ***HistoryCache*** 中的所有 ***CacheChange*** 更改传输到匹配的远程 RTPS ***Reader*** 的 ***HistoryCache***。

配置 RTPS ***Writer*** 的属性与协议行为密切相关，将在行为模块（2.4）中介绍。

2.2.8 RTPS 读取者（Reader）

RTPS ***Reader*** 是 RTPS ***端点（Endpoint）***的特殊实例，负责从匹配的 RTPS ***Writer*** 端点接收 ***CacheChange*** 消息。

配置 RTPS ***Reader*** 的属性与协议行为密切相关，将在行为模块（2.4）中介绍。

2.2.9 与 DDS 实体的关系

如 2.2.2 中所述，***HistoryCache*** 构成 DDS 实体与其对应的 RTPS 实体之间的接口。例如 DDS ***DataWriter*** 通过公共 ***HistoryCache*** 将数据传递给匹配的 RTPS ***Writer***。

DDS 实体与 ***HistoryCache*** 的确切交互方式是特定于实现的，而不是由 RTPS 协议建模。相反 RTPS 协议的行为模块仅指定如何将 ***CacheChange*** 更改从 RTPS ***Writer*** 的 ***HistoryCache*** 传输到每个匹配的 RTPS ***Reader*** 的 ***HistoryCache***。

尽管它不是 RTPS 协议的一部分，但了解 DDS 实体如何与 ***HistoryCache*** 交互以获得对协议的完整理解非常重要。本主题构成本小节的主题。

使用 UML 状态图描述上述交互。用于指代 DDS 和 RTPS 实体的缩写列于下表 2.10 中。

表 2-10 序列图和状态图中使用的缩写

首字母缩写词	含义	用法示例
DW	DDS DataWriter	DW::write
DR	DDS DataReader	DR::read
W	RTPS Writer	W::heartbeatPeriod
R	RTPS Reader	R::heartbeatResponseDelay
WHC	RTPS Writer 的 HistoryCache	WHC::changes
RHC	RTPS Reader 的 HistoryCache	RHC::changes

2.2.9.1 DDS 数据写入者（DataWriter）

DDS ***DataWriter*** 上的 ***write*** 方法作将 ***CacheChange*** 更改添加到其关联的 RTPS ***Writer*** 的 ***HistoryCache*** 中。因此 ***HistoryCache*** 包含最新写入的更改的历史记录。更改数量由 DDS ***DataWriter*** 上的 QoS 设置确定，如 HISTORY 和 RESOURCE_LIMITS QoS。

默认情况下，***HistoryCache*** 中的所有更改都被认为与每个匹配的远程 RTPS ***Reader*** 相关。也就是说，***Writer*** 应该尝试将 ***HistoryCache*** 中的所有更改发送到匹配的远程 ***Readers***。如何做到这一点是 RTPS 协议行为模块的主题。

由于以下两个原因，可能无法将更改发送到远程 ***Reader***：

1. 这些更改已被 DDS ***DataWriter*** 从 ***HistoryCache*** 中删除，不再可用。
2. 这些更改被认为与此 ***Reader*** 无关。

由于多种原因，DDS ***DataWriter*** 可能决定从 ***HistoryCache*** 中删除更改。如可能需要根据 HISTORY QoS 设置存储有限数量的更改。或者由于 LIFESPAN QoS 的配置，数据样本可能已过期。使用严格的可靠通信时，只有在所有读取者都已确认收到更改且 DDS ***DataWriter*** 仍处于活动和存活状态时，才能删除更改。

并非所有更改都可能与每个匹配的远程 ***Reader*** 相关，例如通过 TIME_BASED_FILTER QoS 或通过使用 DDS 内容过滤主题进行确定。在这种情况下，必须由每个读取者确定更改是否相关。可能的话在 ***Writer*** 端进行过滤，DDS 实现可以优化带宽和/或 CPU 使用。这是否可行取决于 DDS 实现是否追踪每个远程读取者的状态以及适用于此 ***Reader*** 的 QoS 和过滤器。***Reader*** 本身将始终过滤。

本文档使用 **DDS_FILTER（reader，change）**表示基于 QoS 或内容的过滤，这是一种反映过滤依赖于读取者的符号。根据写入者存储的特定读取者信息，DDS_FILTER 可能是空的。对于基于内容的过滤，RTPS 规范允许在每次更改时发送信息，列出已应用于更改的过滤器以及它传递的过滤器。如果可用，读取者可以使用此信息过滤更改，而无需调用 DDS_FILTER。这种方法通过在 ***Writer*** 端过滤样本来节省 CPU 周期，而不是在每个 ***Reader*** 上过滤。

以下状态图说明了 DDS ***DataWriter*** 如何向 ***HistoryCache*** 添加更改。

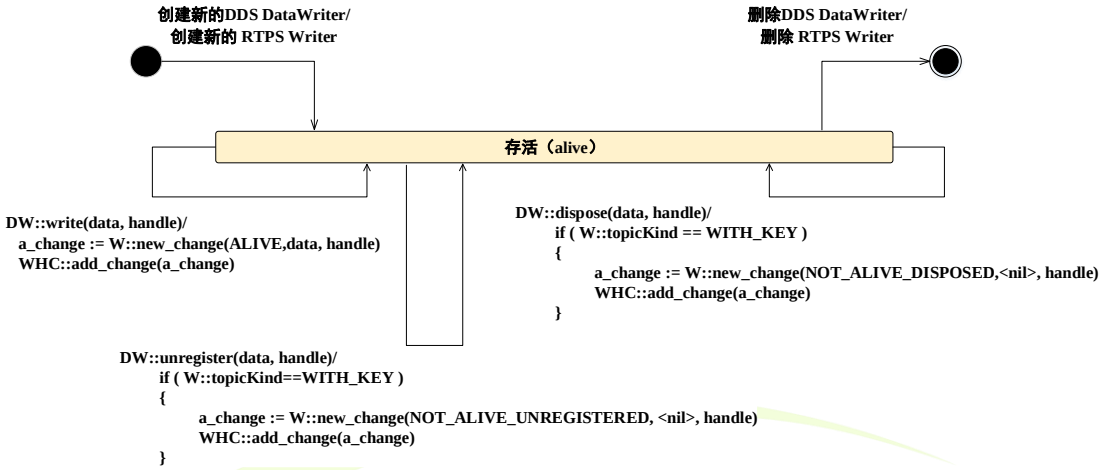


图 2-6 DDS DataWriter 向 HistoryCache 添加更改状态图

表 2-11 DDS DataWriter 向 HistoryCache 添加更改转换表

转换	状态	事件	下一状态
T1	初始状态（initial）	创建新的 DDS DataWriter	存活（alive）
T2	存活（alive）	DDS DataWriter::write	存活（alive）
T3	存活（alive）	DDS DataWriter::dispose	存活（alive）
T4	存活（alive）	DDS DataWriter::unregister	存活（alive）
T5	存活（alive）	删除 DDS DataWriter	结束（final）

2.2.9.1.1转换 T1

通过创建 DDS DataWriter“the_dds_writer”来触发此转换。

转换在虚拟机中执行以下逻辑动作：

the_rtps_writer = new RTPS::Writer;

the_dds_writer.related_rtps_writer := the_rtps_writer;

2.2.9.1.2转换 T2

这种转换是由使用 DDS DataWriter“the_dds_writer”写入数据的行为触发的。DataWriter :: write()方法将数据“data”和用于区分不同的数据对象的 InstanceHandle_t 类型的“handle”作为参数。

转换在虚拟机中执行以下逻辑动作：

the_rtps_writer := the_dds_writer.related_rtps_writer;

a_change := the_rtps_writer.new_change(ALIVE, data, handle);

the_rtps_writer.writer_cache.add_change(a_change);

转换后，以下后置条件成立：

the_rtps_writer.writer_cache.get_seq_num_max() == a_change.sequenceNumber

2.2.9.1.3转换 T3

这种转换是由处理先前使用 DDS DataWriter“the_dds_writer”写入的数据对象的行为触发的。DataWriter :: dispose()方法将用于区分不同的数据对象的 InstanceHandle_t 类型的“handle”作为参数。

如果 topicKind == NO_KEY，则此方法无效。

转换在虚拟机中执行以下逻辑动作：

```
the_rtps_writer := the_dds_writer.related_rtps_writer;
if (the_rtps_writer.topicKind == WITH_KEY)
{
    a_change := the_rtps_writer.new_change(NOT_ALIVE_DISPOSED, <nil>, handle);
    the_rtps_writer.writer_cache.add_change(a_change);
}
```

转换后，以下后置条件成立：

```
if (the_rtps_writer.topicKind == WITH_KEY) then
    the_rtps_writer.writer_cache.get_seq_num_max() == a_change.sequenceNumber
```

2.2.9.1.4转换 T4

这种转换是通过取消注册先前使用 DDS DataWriter“the_dds_writer”写入的数据对象的行为触发的。DataWriter :: unregister()方法将用于区分不同的数据对象的 InstanceHandle_t 类型的“handle”作为参数。

如果 topicKind == NO_KEY，则此方法无效。

转换在虚拟机中执行以下逻辑动作：

```
the_rtps_writer := the_dds_writer.related_rtps_writer;
if (the_rtps_writer.topicKind == WITH_KEY)
{
    a_change := the_rtps_writer.new_change(NOT_ALIVE_UNREGISTERED,
                                           <nil>, handle);
    the_rtps_writer.writer_cache.add_change(a_change);
}
```

转换后，以下后置条件成立：

```
if (the_rtps_writer.topicKind == WITH_KEY) then
    the_rtps_writer.writer_cache.get_seq_num_max() == a_change.sequenceNumber
```

2.2.9.1.5转换 T5

这种转换是由 DDS DataWriter“the_dds_writer”的析构引发的。

转换在虚拟机中执行以下逻辑动作：

```
delete the_dds_writer.related_rtps_writer;
```

2.2.9.2 DDS 数据读取者（DataReader）

DDS ***DataReader*** 从相应 RTPS ***Reader*** 的 ***HistoryCache*** 获取其接收的数据。存储在 ***HistoryCache*** 中的更改数由 QoS 设置（如 HISTORY 和 RESOURCE_LIMITS QoS）确定。

每个匹配的 ***Writer*** 将尝试将所有相关数据样本从其 ***HistoryCache*** 传输到 ***Reader*** 的 ***HistoryCache*** 中。在 DDS ***DataReader*** 上调用 ***read*** 或 ***take*** 方法访问 ***HistoryCache***。返回给用户的更改是 ***HistoryCache*** 中的更改，它们通过了 ***Reader*** 设置的所有过滤器（如果有）限制。

Reader 过滤器同样地由 **DDS_FILTER（reader,change）**表示。如上所述，DDS 实现可能在 ***Writer*** 端执行大部分过滤。在这种情况下，被过滤的数据样本永远不会被发送（因此不会出现在 ***Reader*** 的 ***HistoryCache*** 中），或者包含应用于何处的过滤器及其对应的结果信息（基于内容的过滤）。

DDS ***DataReader*** 还可以决定从 ***HistoryCache*** 中删除更改以满足 TIME_BASED_FILTER 等 QoS 的设置。这种确切的行为仍然是特定于实现的，并且不由 RTPS 协议建模。

以下状态图说明了 DDS ***DataReader*** 如何访问 ***HistoryCache*** 中的更改。

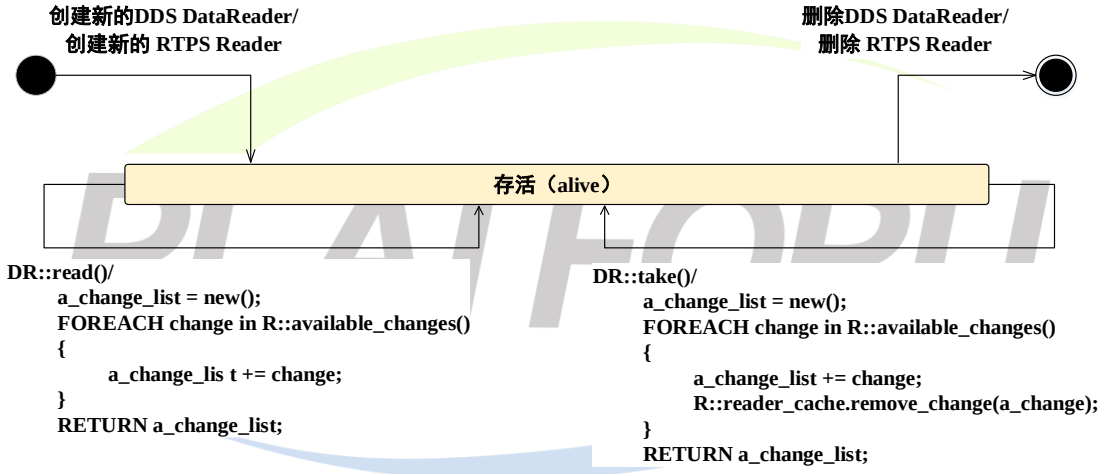


图 2-7 DDS DataReader 访问 HistoryCache 状态图

表 2-12 DDS DataReader 访问 HistoryCache 转换表

转换	状态	事件	下一状态
T1	初始状态（initial）	创建新的 DDS DataReader	存活（alive）
T2	存活（alive）	DDS DataReader::read	存活（alive）
T3	存活（alive）	DDS DataReader::take	存活（alive）
T4	存活（alive）	删除 DDS DataReader	结束（final）

2.2.9.2.1转换 T1

这种转换是由创建 DDS DataReader“the_dds_reader”触发的。

转换在虚拟机中执行以下逻辑动作：

the_rtps_reader = new RTPS::Reader;

the_dds_reader.related_rtps_reader := the_rtps_reader;

2.2.9.2.2转换 T2

这种转换是通过 DDS DataReader“the_dds_reader”调用“***read***”方法读取数据触发的。返回到应用程序的更改将保留在 RTPS ***Reader*** 的 ***HistoryCache*** 中，后续的 ***read*** 或 ***take*** 方法可以再次找到它们。

转换在虚拟机中执行以下逻辑动作：

```
the_rtps_reader := the_dds_reader.related_rtps_reader;
a_change_list := new();
FOREACH change IN the_rtps_reader.reader_cache.changes
{
    if DDS_FILTER(the_rtps_reader, change) ADD change TO a_change_list;
}
RETURN a_change_list;
```

DDS_FILTER()方法反映了 DDS DataReader API 根据 ***CacheChange :: kind***，QoS，内容过滤器和其他机制选择更改子集的功能。请注意上面的逻辑动作仅反映行为，而不一定反映协议的实际实现。

2.2.9.2.3转换 T3

这种转换是通过 DDS DataReader“the_dds_reader”调用“***take***”方法读取数据触发的。返回到应用程序的更改将从 RTPS ***Reader*** 的 ***HistoryCache*** 中删除，后续的 ***read*** 或 ***take*** 方法找不到相同的更改。

转换在虚拟机中执行以下逻辑动作：

```
the_rtps_reader := the_dds_reader.related_rtps_reader;
a_change_list := new();
FOREACH change IN the_rtps_reader.reader_cache.changes
{
    if DDS_FILTER(the_rtps_reader, change)
    {
        ADD change TO a_change_list;
    }
    the_rtps_reader.reader_cache.remove_change(a_change);
}
RETURN a_change_list;
```

DDS_FILTER()方法反映了 DDS DataReader API 根据 ***CacheChange :: kind***，QoS，内容过滤器和其他机制选择更改子集的功能。请注意上面的逻辑动作仅反映行为，而不一定反映协议的实际实现。

转换后，以下后置条件成立：

```
FOREACH change IN a_change_list
    change BELONGS_TO the_rtps_reader.reader_cache.changes == FALSE
```

2.2.9.2.4转换 T4

这种转换是由 DDS DataReader“the_dds_reader”的析构触发的。

转换在虚拟机中执行以下逻辑动作：

delete the_dds_reader.related_rtps_reader;

2.3 消息模块

消息模块描述了在 RTPS ***Writer*** 端点和 RTPS ***Reader*** 端点之间交换的消息的总体结构和逻辑内容。RTPS 消息是模块化设计，可以轻松扩展以支持标准协议功能添加以及供应商专属内容的扩展。

2.3.1 概述

消息模块章节的组织结构如下：

•2.3.2 介绍了后续子章节中定义 RTPS 消息所需的所有其他类型。

•2.3.3 描述了适用于所有 RTPS 消息的通用结构。所有 RTPS 消息都包含一个报文头（Header）以及一系列子消息（Submessages）。可以在单个 RTPS 消息中发送的子消息的数量受底层传输方式支持的最大消息大小的限制。

•某些子消息可能会影响同一 RTPS 消息中后续子消息的解析。解析 Submessages 的上下文由 RTPS 消息接收器维护，并在 2.3.4 中描述。

•2.3.5 列出了用于创建子消息（Submessages）的基础构建块，也称为子消息元素（SubmessageElements），其中包括序列号集，时间戳，标识符等。

•2.3.6 描述了 RTPS 报文头的结构。固定大小的 RTPS 报文头用于标识 RTPS 消息。

•2.3.7 详细介绍了所有可用的子消息（Submessages）。对于每个 Submessage，规范定义其内容、何时被认为是有效的以及它如何影响 RTPS 消息接收器的状态。PSM 将在 3.4.5 中定义每个 Submessage 到线路传输上的位和字节的实际映射。

2.3.2 类型定义

除了 2.2.1.2 中定义的类型之外，消息模块还使用表 2.13 中列出的类型。

表 2-13 用于定义 RTPS 消息的类型表

用来定义 RTPS 消息的类型	
类型	**目的**
ProtocolId_t	用于标识协议的枚举类型。 协议保留以下值： PROTOCOL_RTPS
SubmessageFlag	用于指定 Submessage 标志的类型。 Submessage 标志采用布尔值并影响接收器对 Submessage 的解析。
SubmessageKind	用于标识子消息的枚举类型。 此版本的协议保留以下值：

用来定义 RTPS 消息的类型	
类型	**目的**
	DATA，GAP，HEARTBEAT，ACKNACK，PAD，INFO_TS，INFO_REPLY， INFO_DST， INFO_SRC， DATA_FRAG， NACK_FRAG，HEARTBEAT_FRAG
Time_t	用于保存时间戳的类型。 应至少具有纳秒级分辨率。 协议保留以下值： TIME_ZERO TIME_INVALID TIME_INFINITE
Count_t	用于封装单调递增的计数的类型，用于标识消息重复。
ParameterId_t	用于唯一标识参数列表中的参数的类型。 发现模块广泛使用，主要用于定义 QoS 参数。为协议定义的参数保留一系列值，而另一个范围可用于供应商定义的参数，见 2.3.5.9。
FragmentNumber_t	用于保存分片编号的类型。 必须可以使用 32 位表示。

2.3.3 RTPS 消息的总体结构

RTPS 消息的整体结构包括固定大小的前导 RTPS **报文头（Header）**，后跟数量不定的 RTPS **子消息（Submessage）**部分。每个**子消息（Submessage）**依次由**子消息报文头（SubmessageHeader）**和数量不定的**子消息元素（SubmessageElements）**组成。 如图 2.8 所示。

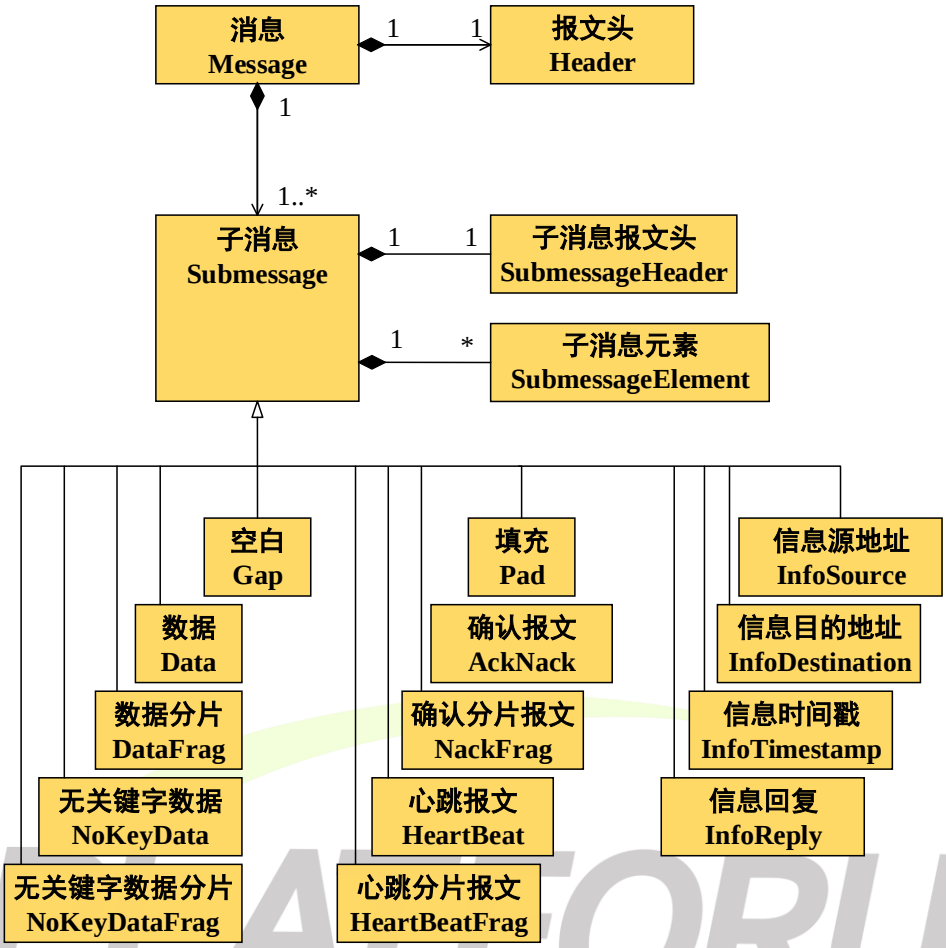


图 2-8 RTPS 消息结构

RTPS 协议发送的每条消息都有一个有限的长度。该长度不是由 RTPS 协议显式发送的，而是发送 RTPS 消息的底层传输方式的一部分。在面向分组的传输方式（如 UDP/IP）情况下，消息的长度已由传输方式封装提供。面向流的传输方式（如 TCP）需要在消息之前插入长度，以便识别 RTPS 消息的边界。

2.3.3.1 RTPS 报文头（Header）结构

RTPS **报文头（Header）**必须出现在每条消息的开头。

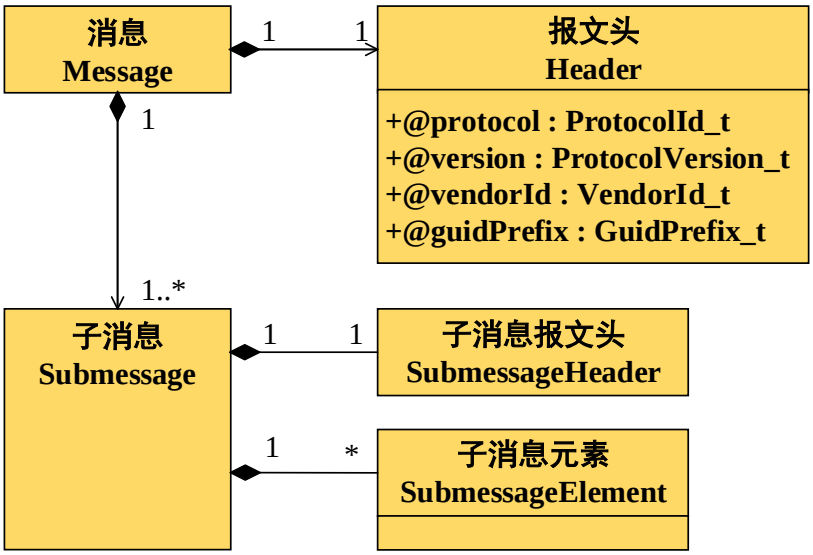


图 2-9 RTPS 消息报文头结构

报文头（Header）将消息标识为属于 RTPS 协议。**报文头（Header）**标识协议的版本和发送消息的供应商。**报文头（Header）**包含表 2.14 中列出的字段。

表 2-14 RTPS 消息报文头（Header）结构

字段	类型	含义
protocol	ProtocolId_t	将消息标识为 RTPS 消息
version	ProtocolVersion_t	标识 RTPS 协议的版本
vendorId	VendorId_t	表示提供 RTPS 协议实现的供应商
guidPrefix	GuidPrefix_t	定义消息中所有 GUID 使用的默认前缀

在协议的主要版本（2）中不能更改 RTPS **报文头（Header）**的结构。

2.3.3.1.1 协议（protocol）

*协议（protocol）*字段将消息标识为 RTPS 消息。该值设置为 PROTOCOL_RTPS。

2.3.3.1.2 版本（version）

*版本(version)*字段标识 RTPS 协议的版本。本文档之后的实现定义协议版本为 2.2（major = 2，minor = 2），并将此字段设置为 PROTOCOLVERSION。

2.3.3.1.3 供应商 ID（vendorId）

*供应商 ID（vendorId）*标识了实现 RTPS 协议的中间件供应商 ID，并允许该供应商添加协议的特定扩展。*vendorId* 不是指包含 RTPS 中间件的设备或产品的供应商 ID。*vendorId* 由 OMG 分配。

该协议保留以下值：

VENDORID_UNKNOWN

供应商 ID 只能由承诺遵守当前主要协议版本的实现者持有。为了促进增量进化，供应商 ID 列表与此规范分开管理。该列表在 OMG DDS Wiki 中维护，可从以下网址访问：http://www.omgwiki.org/dds/content/page/dds-rtps-vendor-and product-ids。

新供应商 ID 的请求应通过电子邮件发送至 dds@omg.org。

2.3.3.1.4GUID 前缀（guidPrefix）

*GUID 前缀（guidPrefix）*定义了一个默认前缀，可用于重建消息中包含的子消息（Submessages）中出现的全局唯一标识符（GUID）。*guidPrefix* 允许子消息（Submessages）只包含 GUID 的 EntityId 部分，因此不必在每个 GUID 上重复出现公共前缀（见 2.2.4.1）。

2.3.3.2 子消息（Submessage）结构

每个 RTPS 消息由可变数量的 RTPS **子消息（Submessage）**部分组成。所有 RTPS 子消息都具有相同的结构，如图 2.10 所示。

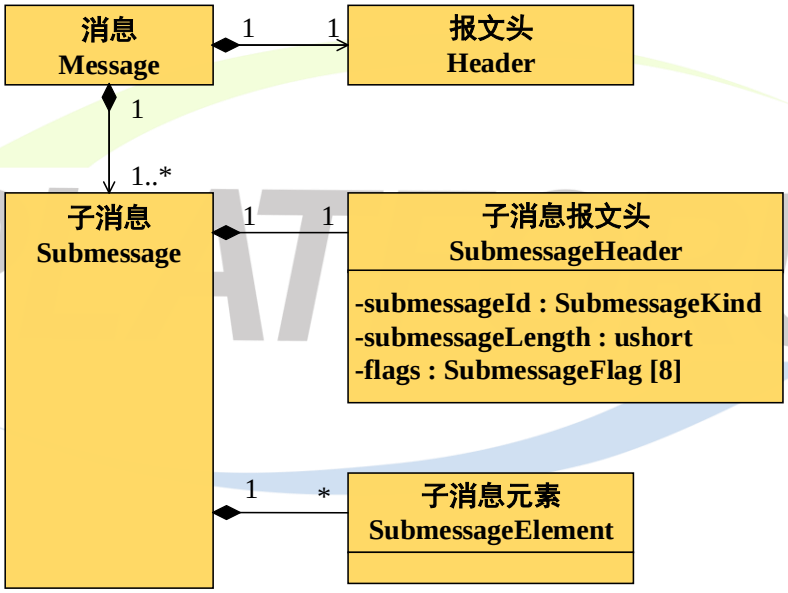


图 2-10 RTPS 子消息（Submessage）结构

所有 **Submessages** 都以 **SubmessageHeader** 部分开头，后跟 **SubmessageElement** 部分的串联。**SubmessageHeader** 标识 **Submessage** 的种类以及 **Submessage** 中的可选元素。**SubmessageHeader** 包含表 2.15 中列出的字段。

表 2-15 子消息报文头（SubmessageHeader）结构

字段	类型	含义
submessageId	SubmessageKind	标识子消息（Submessage）的类型。可能的 Submessages 类型在 2.3.7 中描述
flags	SubmessageFlag[8]	标识用于封装子消息（Submessage）的字节序、子消息（Submessage）中可选元素的存在，并可能修改子消息（Submessage）的解析。有 8 个可能的标志。第一个标志（索引 0）标

		识用于封装子消息（Submessage）的字节序。其余标志根据 Submessage 的类型进行不同的解释，并为每种 Submessage 单独描述
submessageLength	ushort	子消息（Submessage）的长度。RTPS 消息由子消息的串联组成，Submessage 长度可用于跳到下一个 Submessage。

在协议的主要版本（2）中不能更改 RTPS **子消息（Submessage）**的结构。

2.3.3.2.1 子消息 ID（submessageId）

*子消息 ID（submessageId）*标识**子消息（Submessage）**的种类。有效 ID 由 SubmessageKind 的可能值枚举定义（参见表 2.13）。

在此主要版本（2）中无法修改子消息 ID 的含义。可以在较高的次要版本中添加其他子消息类型。为了保持与未来版本的互操作，平台特定映射应保留一系列用于协议扩展的值以及为特定于供应商的子消息保留的值，这些值将永远不会被未来版本的 RTPS 协议使用。

2.3.3.2.2 标志（flags）

Submessage 报文头中的*标志（flags）*包含 8 个布尔值。第一个标志 *EndiannessFlag* 并位于所有子消息的相同位置，表示对 **Submessage** 中的信息进行编码的字节序。文字“E”通常用于表示 *EndiannessFlag*。

如果 *EndiannessFlag* 设置为 FALSE，则 **Submessage** 以大端格式编码，*EndiannessFlag* 设置为 TRUE 表示小端编码。

其他标志的解释取决于 **Submessage** 的类型

2.3.3.2.3 子消息长度（submessageLength）

本字段代表**子消息（Submessage）**的长度（不包括 Submessage **报文头 Header**）。

如果 *submessageLength*> 0，则它是

- 从 **Submessage** 内容开始到下一个 **Submessage** 报文头（Header）开始的长度（如果 **Submessage** 不是消息中的最后一个 **Submessage**）。
- 或者是剩余的消息长度（如果 **Submessage** 是消息中的最后一个 **Submessage**）。

Message 的解释器可以区分这两种情况，因为它知道 **Message** 的总长度。

如果 *submessageLength* == 0，则 **Submessage** 是 **Message** 中的最后一个 **Submessage**，并一直延伸到 **Message** 的末尾。这使得可以发送大于 64KB（可以存储在 *submessageLength* 字段中的最大长度）的子消息，前提是它们是 **Message** 中的最后一个 **Submessage**。

2.3.4 RTPS 消息接收器（RTPS Message Receiver）

消息中的子消息的解释和含义可能取决于同一消息中包含的先前子消息。因此，**Message** 的接收者必须记录同一 **Message** 中先前反序列化 **Submessage** 的状态。每次处理新

消息时重置 RTPS 接收器的状态，并为每个 **Submessage** 的解析提供上下文环境。RTPS 接收器如图 2.11 所示。表 2.16 列出了用于表示 RTPS 接收器状态的属性。

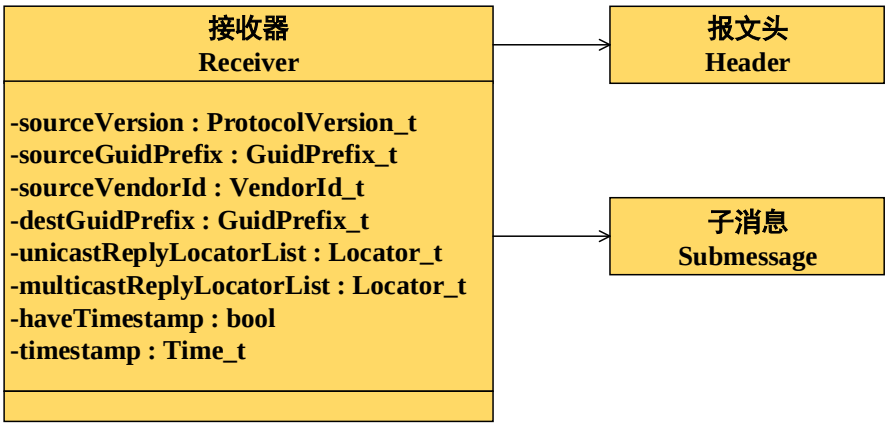


图 2-11 RTPS 消息接收器

对于每个新**消息（Message）**，接收器的状态将重置并初始化，如下所示。

表 2-16 接收器初始状态

名称	初始值
sourceVersion	PROTOCOLVERSION
sourceVendorId	VENDORID_UNKNOWN
sourceGuidPrefix	GUIDPREFIX_UNKNOWN
destGuidPrefix	接收消息的参与者的 GUID 前缀
UnicastReplyLocatorList	该列表初始化为包含单个 Locator_t，其中包含下面指定的 **LocatorKind**，**Address** 和 **Port** 字段： •**LocatorKind** 设置为标识接收消息的传输方式的类型（例如，LOCATOR_KIND_UDPv4）。 •**Address** 设置为消息来源的地址，假设使用的传输方式支持此功能（例如，对于 UDP，源地址是 UDP 报头的一部分）。否则，它将设置为 LOCATOR_ADDRESS_INVALID。 •**Port** 设置为 LOCATOR_PORT_INVALID。
multicastReplyLocatorList	该列表初始化为包含单个 Locator_t，其中包含下面指定的 **LocatorKind**，**Address** 和 **Port** 字段： •**LocatorKind** 设置为标识接收传输的传输方式的类型（例如，LOCATOR_KIND_UDPv4）。 •**Address** 设置为 LOCATOR_ADDRESS_INVALID。 •**Port** 设置为 LOCATOR_PORT_INVALID。
haveTimestamp	FALSE
timestamp	TIME_INVALID

2.3.4.1 消息接收器遵循的规则

以下算法概述了任一**消息（Message）**接收器必须遵循的规则：

1.如果无法读取完整的 **Submessage** 报文头，则该消息的剩余部分将被视为无效。

2. *submessageLength* 字段定义下一个 **Submessage** 的开始位置或指示 **Submessage** 延伸到 **Message** 的末尾，如 2.3.3.2.3 中所述。如果此字段无效，则消息的其余部分无效。

3.必须忽略具有未知 SubmessageId 的 **Submessage**，并且必须继续解析下一个 Submessage。具体来说：RTPS 2.2 的实现必须忽略任何 ID 不在 RTPS 2.2 版规范定义的 SubmessageKind 集合当中的 Submessages。*vendorId* 不在供应商特定范围中的 SubmessageIds 也必须被忽略，并且必须继续解析下一个 Submessage。

4.子消息标志（Submessage flags）。Submessage 的接收器应该忽略未知标志。RTPS 2.2 的实现应该跳过协议中标记为“X”（未使用）的所有标志。

5.必须始终使用有效的 *submessageLength* 字段来查找下一个 Submessage，即使对于具有已知 ID 的 Submessages 也是如此。

6.已知但无效的 Submessage 使消息的其余部分无效。2.3.7 描述了每个已知的 Submessage 以及何时应被视为无效。

接收有效报文头和/或 Submessage 有两个影响：

1.它可以改变接收器的状态; 此状态会影响消息中后续 **Submessages** 的解释方式。2.3.7 小节讨论了每个 **Submessage** 的状态如何变化。在此版本的协议中，只有 **Header** 和 **Submessages InfoSource**，**InfoReply**，**InfoDestination** 和 **InfoTimestamp** 会更改接收器的状态。

2.它可能影响消息目的端点的行为。这适用于基本的 RTPS 消息：**Data**，**DataFrag**，**HeartBeat**，**AckNack**，**Gap**，**HeartbeatFrag**，**NackFrag**。

2.3.7 描述了**报文头（Header）**和每个**子消息（SubMessage）**的详细解释

2.3.5 RTPS 子消息元素（SubmessageElements）

每个 RTPS 消息包含可变数量的 RTPS 子消息。每个 RTPS 子消息都是由一组名为**子消息元素（SubmessageElements）**的预定义原子块构建的。RTPS 2.2 定义以下 **Submessage** 元素：**GuidPrefix**，**EntityId**，**SequenceNumber**，**SequenceNumberSet**，**FragmentNumber**，**FragmentNumberSet**，**VendorId**，**ProtocolVersion**，**LocatorList**，**Timestamp**，**Count**，**SerializedData** 和 **ParameterList**。

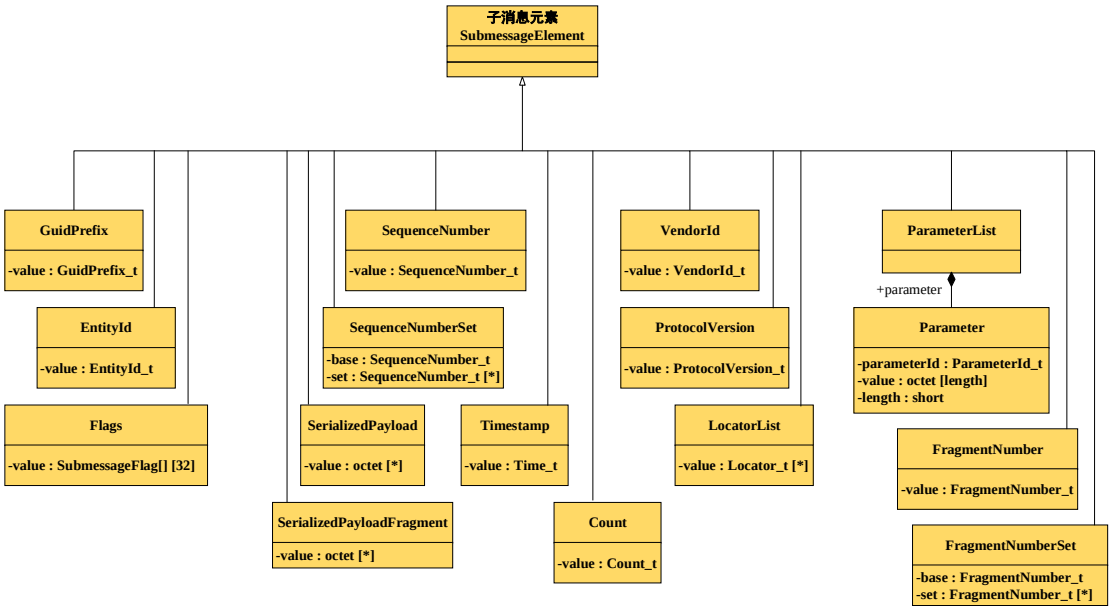


图 2-12 RTPS 子消息元素（SubmessageElements）

2.3.5.1 GuidPrefix 和 EntityId

这些子消息元素（SubmessageElements）用于封装 Submessages 中 GUID_t（在 2.2.4.1 中定义）的 **GuidPrefix_t** 和 **EntityId_t** 部分。

表 2-17 GuidPrefix 子消息元素结构

字段	类型	含义
value	GuidPrefix_t	标识作为消息源或目标的实体 GUID_t 的 GuidPrefix_t 部分。

表 2-18 EntityId 子消息元素结构

字段	类型	含义
value	EntityId_t	标识作为消息源或目标的实体 GUID_t 的 EntityId_t 部分。

2.3.5.2 VendorId

VendorId 标识实现 RTPS 协议的中间件的供应商，并允许该供应商添加协议的特定扩展。供应商 ID 不是指包含 DDS 中间件的设备或产品的供应商。

表 2-19 VendorId 子消息元素结构

字段	类型	含义
value	VendorId_t	标识实现 RTPS 协议中间件的供应商。

协议保留以下值：

VENDORID_UNKNOWN

其他值必须由 OMG 分配。

2.3.5.3 ProtocolVersion

ProtocolVersion 定义了 RTPS 协议的版本。

表 2-20 ProtocolVersion 子消息元素结构

字段	类型	含义
value	ProtocolVersion_t	标识 RTPS 协议的主要版本和次要版本。

RTPS 协议版本 2.2 定义了以下特殊值：

PROTOCOLVERSION_1_0
PROTOCOLVERSION_1_1
PROTOCOLVERSION_2_0
PROTOCOLVERSION_2_1
PROTOCOLVERSION_2_2
PROTOCOLVERSION

2.3.5.4 SequenceNumber

序列号是 64 位有符号整数，可以取以下范围内的值：-2 ^ 63 <= N <= 2 ^ 63-1。选择 64 位作为序列号的表现形式可确保序列号永不缠绕。序号从 1 开始。

表 2-21 SequenceNumber 子消息元素结构

字段	类型	含义
value	SequenceNumber_t	提供 64 位序列号的值。

该协议保留以下值：

SEQUENCENUMBER_UNKNOWN

2.3.5.5 SequenceNumberSet

SequenceNumberSet 子消息元素用作多个消息的一部分，以提供某个范围内各个序列号的二进制信息。**SequenceNumberSet** 中表示的序列号被限制为属于范围不大于 256 的区间。换句话说，有效的 **SequenceNumberSet** 必须确保：

maximum（**SequenceNumberSet**） - minimum（**SequenceNumberSet**）<256
minimum（**SequenceNumberSet**）> = 1

上述限制允许使用位映像以高效且紧凑的方式表示 **SequenceNumberSet**。

SequenceNumberSet 子消息元素用于（例如）选择性地请求重新发送一组序列号。

表 2-22 SequenceNumberSet 子消息元素结构

字段	类型	含义
base	SequenceNumber_t	标识集合中的第一个序列号。
set	SequenceNumber_t[*]	一组序列号，每个序列号确保： base <= element（set）<= base + 255

2.3.5.6 FragmentNumber

分片编号是 32 位无符号整数，**Submessages** 使用它来识别分段序列化数据中的特定片段。分片编号从 1 开始。

表 2-23 FragmentNumber 子消息元素结构

字段	类型	含义
value	FragmentNumber_t	提供 32 位分片编号的值。

2.3.5.7 FragmentNumberSet

FragmentNumberSet 子消息元素用于提供某个范围内各个分片编号的二进制信息。**FragmentNumberSet** 中表示的分片编号限制为属于范围不大于 256 的区间。换句话说，有效的 **FragmentNumberSet** 必须确保：

maximum（**FragmentNumberSet**） - minimum（**FragmentNumberSet**）<256

minimum（**FragmentNumberSet**）> = 1

上述限制允许使用位映射以高效且紧凑的方式表示 **FragmentNumberSet**。

FragmentNumberSet 子消息元素用于（例如）有选择地请求重新发送一组分片数据。

表 2-24 FragmentNumberSet 子消息元素结构

字段	类型	含义
base	FragmentNumber_t	标识集合中的第一个分片编号。
set	FragmentNumber_t [*]	一组分片编号，每个分片编号确保： base <= element（set）<= base + 255

2.3.5.8 Timestamp

时间戳（Timestamp）用于表示时间，应该能够具有纳秒或更高的精确度来表示时间。

表 2-25 Timestamp 子消息元素结构

字段	类型	含义
value	Time_t	提供时间戳的值。

协议使用了三个特殊值：

TIME_ZERO

TIME_INVALID

TIME_INFINITE

2.3.5.9 ParameterList

ParameterList 用作若干消息的一部分，以封装可能影响消息解析的 QoS 参数。参数的封装遵循允许扩展 QoS 而不破坏向下兼容的机制。

表 2-26 ParameterList 子消息元素结构

字段	类型	含义
parameter	Parameter[*]	参数列表

表 2-27 ParameterList 子消息元素中每个参数的结构

字段	类型	含义
parameterId	ParameterId_t	唯一标识参数
length	short	参数值的长度
value	octet[length]	参数值

为每个 PSM 定义的 **ParameterList** 的实际表示。为了支持 PSM 之间的互操作或桥接并允许保留向下兼容的扩展，所有 PSM 使用的表示必须符合以下规则：

- ParameterId_t *parameterId* 的值不得超过 2 ^ 16。
- 为协议定义的参数保留了 2 ^ 15 个值的范围。2.2 版本的协议定义的所有 parameter_id 值以及相同主要版本的所有未来版本都必须使用此范围内的值。
- 为供应商定义的参数保留了 2 ^ 15 个值的范围。协议的 2.2 版本以及对应于相同主要版本的协议的任何未来版本都不允许使用此范围内的值。
- 任一参数的最大长度限制为 2 ^ 16 个八位字节。

根据上述约束，不同的 PSM 可以为 ParameterId_t 选择不同的表示。例如，PSM 可以使用短整数表示 *parameterId*，而另一个 PSM 可以使用字符串。

2.3.5.10 Count

Count 由多个 **Submessages** 使用，使接收器能够检测出重复的 **Submessage**。

表 2-28 Count 子消息元素结构

字段	类型	含义
value	Count_t	Count 值

2.3.5.11 LocatorList

LocatorList 用于指定定位器列表。

表 2-29 LocatorList 子消息元素结构

字段	类型	含义
value	Locator_t[*]	定位器列表

2.3.5.12 SerializedData

SerializedData 包含数据对象值的序列化表示。RTPS 协议不解析序列化数据流。因此，它表示为不透明数据。有关数据封装的其他信息，请参见第 4 章“数据封装”。

表 2-30 SerializedData 子消息元素结构

字段	类型	含义
value	octet[*]	序列化数据流

2.3.5.13 SerializedDataFragment

SerializedDataFragment 包含已分片的数据对象的序列化表示。 与未分片的 **SerializedData** 一样，RTPS 协议不解析分片的序列化数据流。因此，它表示为不透明数据。有关数据封装的其他信息，请参见第 4 章“数据封装”。

表 2-31 SerializedDataFragment 子消息元素结构

字段	类型	含义
value	octet[*]	序列化数据流分片

2.3.6 RTPS 报文头（Header）

如 2.3.3 中所述，每个 RTPS 消息必须以报文头（Header）开头。

2.3.6.1 目的

报文头（Header）用于将消息标识为属于 RTPS 协议，标识所使用的 RTPS 协议的版本，并提供消息中包含的子消息可能会用到的上下文信息。

2.3.6.2 内容

构成**报文头（Header）**结构的元素在 2.3.3.1 中描述。只有在协议的主要版本也发生变化时，才能更改**报文头（Header）**的结构。

2.3.6.3 有效性

如果满足以下任一条件，则**报文头（Header）**无效：

- Message 拥有的位数小于完整**报文头（Header）**所需的八位字节数。所需数量由 PSM 定义。
- *protocol* 值与 PROTOCOL_RTPS 的值不匹配。
- 主要协议版本大于实现支持的主要协议版本。

2.3.6.4 接收器状态的变化

接收器的初始状态在 2.3.4 中描述。此处描述新消息的**报文头（Header）**如何影响接收器的状态。

Receiver.sourceGuidPrefix = Header.guidPrefix
Receiver.sourceVersion = Header.version
Receiver.sourceVendorId = Header.vendorId
Receiver.haveTimestamp = false

2.3.6.5 逻辑解释

无。

2.3.7 RTPS 子消息（Submessages）

RTPS 协议版本 2.2 定义了几种 Submessages。它们分为两组：实体子消息和解释器子消息。实体子消息以 RTPS 实体（***Entity***）为目标。解释器子消息修改 RTPS 接收器状态并提供有助于处理后续实体子消息的上下文信息。

实体子消息主要有以下几类：

•数据（Data）：包含有关应用程序数据对象值的信息。**数据（Data）**子消息由写入者（**NO_KEY Writer** 或 **WITH_KEY Writer**）发送给读取者（**NO_KEY Reader** 或 **WITH_KEY Reader**）。

•数据分片（DataFrag）：等效于**数据（Data）**，但仅包含新值的一部分（一个或多个分片）。允许将数据作为多个分片传输，以克服传输消息大小限制。

•心跳报文（Heartbeat）：描述 **Writer** 中可用的信息。**心跳报文（Heartbeat）**消息由写入者（**NO_KEY Writer** 或 **WITH_KEY Writer**）发送到一个或多个读取者（**NO_KEY Reader** 或 **WITH_KEY Reader**）。

•心跳分片报文（HeartbeatFrag）：对于分片数据，**心跳分片报文（HeartbeatFrag）**描述 **Writer** 中可用的分片。**心跳分片报文（HeartbeatFrag）**消息由写入者（**NO_KEY Writer** 或 **WITH_KEY Writer**）发送到一个或多个读取者（**NO_KEY Reader** 或 **WITH_KEY Reader**）。

•空白（Gap）：描述与读取者不再相关的信息。**空白（Gap）**消息由写入者发送给一个或多个读取者。

•确认报文（AckNack）：向 **Writer** 提供有关 **Reader** 状态的信息。**确认报文（AckNack）**消息由 **Reader** 发送给一个或多个 **Writer**。

•确认分片报文（NackFrag）：向 **Writer** 提供有关 **Reader** 状态的信息，更具体地说，**Reader** 仍然缺少哪些分片。**确认分片报文（NackFrag）**消息由 **Reader** 发送给一个或多个 **Writer**。

解释器子消息分为以下几类：

•信息源地址（InfoSource）：提供有关后续实体子消息来源的信息。该子消息主要用于转发 RTPS 子消息。**这在当前的规范中没有讨论**。

•信息目的地址（InfoDestination）：提供有关后续实体子消息的最终目的地的信息。该子消息主要用于转发 RTPS 子消息。**这在当前的规范中没有讨论**。

•信息回复（InfoReply）：提供有关在后续子消息中显示的给实体回复的位置信息。

•信息时间戳（InfoTimestamp）：为后续实体子消息提供源时间戳。

•**填充（Pad）**：用于在内存对齐需要时为消息添加填充。

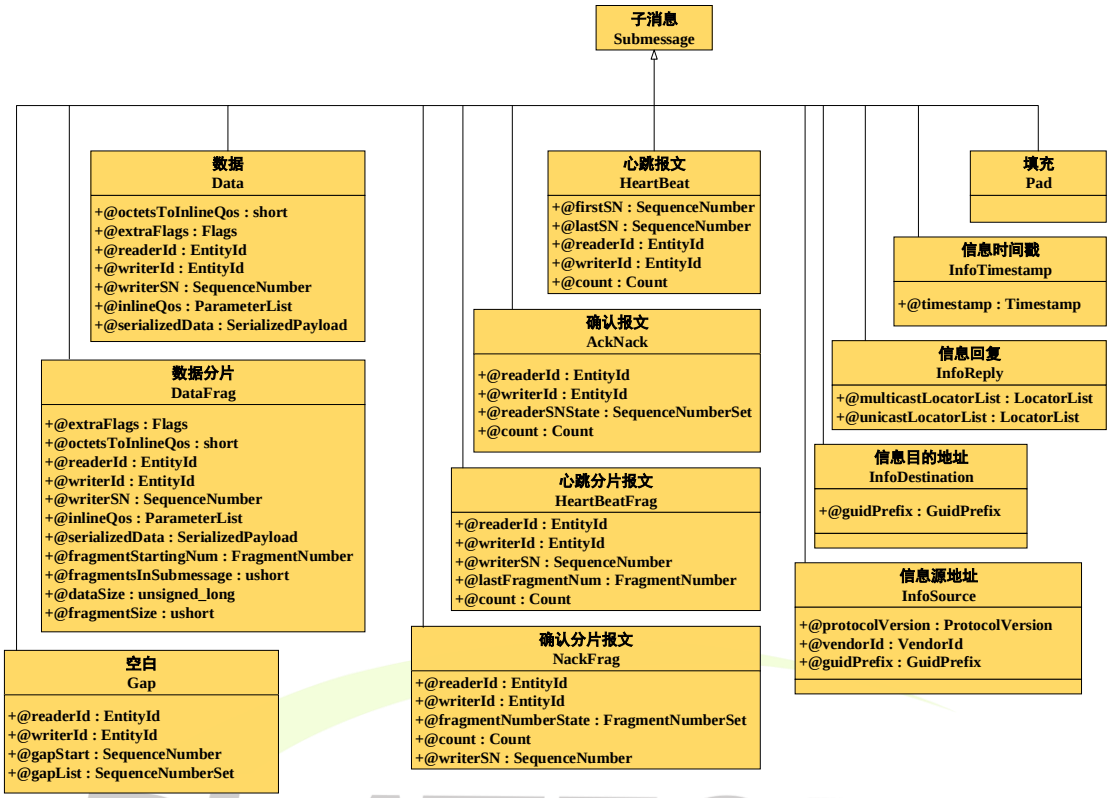


图 2-13 RTPS 子消息（Submessage）

本小节描述了每个子消息及其解释。每个子消息的详细描述标题与表 2.32 中的第一列描述的方式一致。

表 2-32 用于描述每个子消息的格式

标题	含义
目的	**子消息（Submessage）**主要目的的进一步描述
内容	**SubmessageHeader**（*SubmessageId* 和 *flags*）的描述。 可以出现在 **Submessage** 中的 **SubmessageElements** 的描述。
有效性	**Submessage** 必须满足的约束才能使其有效
接收器状态的变化	消息中的子消息的解释和含义可能取决于同一消息中的先前子消息内容。如第 2.3.4 节所述，此上下文内容被建模为 **Receiver** 对象的状态
逻辑解释	描述 **Submessage** 应如何解释。

2.3.7.1 确认报文（AckNack）

2.3.7.1.1 目的

此种子消息用于将 ***Reader*** 的状态传达给 ***Writer***，允许 ***Reader*** 向 ***Writer*** 通知它已收到的序列号以及它仍然未收到的序列号。此种子消息可用于做出正面和负面的确认。

2.3.7.1.2 内容

构成确认报文（AckNack）消息结构的元素在下表中描述。

表 2-33 确认报文（AckNack）结构

元素	类型	含义
EndiannessFlag	SubmessageFlag	出现在子消息报文头标志中。表示字节序。
FinalFlag	SubmessageFlag	出现在子消息报文头标志中。向 Writer 指示是否必需有所响应。
readerId	EntityId	标识确认收到某些序列号和/或请求某些序列号的的 **Reader** 实体。
writerId	EntityId	标识作为确认报文（AckNack）目标的 Writer 实体。这是被要求重新发送一些序列号或被告知某些序列号已被接收的写入者实体。
readerSNState	SequenceNumberSet	将读取者的状态传达给写入者。 读取者确认已经接收到所有 readerSNState.base 之前的序列号。 出现在集合中的序列号表示读取者侧缺少的序列号。未出现在集合中的那些未确定（可能已经接收到或没接收到）。
count	Count	每次发送新的确认报文（AckNack）时递增的计数器。 为 **Writer** 提供检测可能由于存在冗余通信路径而导致的接收到重复确认报文（AckNack）的方法

2.3.7.1.3 有效性

如果以下任一条件成立，则此类子消息无效：

- 子消息报文头中的 *submessageLength* 太小。
- *readerSNState* 无效（如 2.3.5.5 中所定义）。

2.3.7.1.4 接收器状态的变化

无。

2.3.7.1.5 逻辑解释

Reader 将**确认报文（AckNack）**发送给 ***Writer***，以便根据 ***Writer*** 使用的序列号传达其接收状态。

Writer 由其 GUID 唯一标识。Writer GUID 是使用接收器（Receiver）的状态获得的：

writerGUID = {Receiver.destGuidPrefix，AckNack.writerId}

Reader 由 GUID 唯一标识。使用接收器（Receiver）的状态获取 Reader GUID：

readerGUID = {Receiver.sourceGuidPrefix，AckNack.readerId}

该消息同时有两个目的：

•子消息（Submessage）确认所有序列号，包括 **SequenceNumberSet** 中最低序列号之前的序列号（即 readerSNState.base -1）。

•子消息（Submessage）负面确认（即请求）显式出现在集合中的序列号。

明确表示序列号的机制取决于 PSM。通常使用紧凑表示（例如位映射）。

FinalFlag 指示 ***Reader*** 是否期望 ***Writer*** 的响应，或者决定权是否留给 ***Writer***。8.4 中描述了该标志的使用。

2.3.7.2 数据（Data）

此类子消息从 RTPS ***Writer***（**NO_KEY** 或 **WITH_KEY**）发送到 RTPS ***Reader***（**NO_KEY** 或 **WITH_KEY**）。

2.3.7.2.1 目的

此类子消息通知 RTPS ***Reader*** 对属于 RTPS ***Writer*** 的数据对象的更改。可能的更改包括值的更改以及数据对象生命周期的更改。

2.3.7.2.2 内容

构成数据（Data）消息结构的元素在下表中描述。

表 2-34 数据（Data）子消息结构

元素	类型	含义
EndiannessFlag	SubmessageFlag	出现在子消息报文头标志中。表示字节序。
InlineQosFlag	SubmessageFlag	出现在子消息报文头标志中。 向 Reader 指示存在 ParameterList，该列表包含用于解释消息的 QoS 参数。
DataFlag	SubmessageFlag	出现在子消息报文头标志中。 向 Reader 指示 dataPayload 子消息元素包含数据对象的序列化值。
KeyFlag	SubmessageFlag	出现在子消息报文头标志中。 向 Reader 指示 dataPayload 子消息元素包含数据对象的关键字的序列化值。
readerId	EntityId	标识被通知存在数据对象更改的 RTPS ***Reader*** 实体。
writerId	EntityId	标识对数据对象进行更改的 RTPS ***Writer*** 实体。
writerSN	SequenceNumber	唯一标识由 writerGuid 标识的 RTPS ***Writer*** 所做的所有更改和更改的相对顺序。 每次更改都会获得一个连续的序列号。每个 RTPS ***Writer*** 维护自己的序列号。
inlineQos	ParameterList	仅在子消息报文头中设置 InlineQosFlag 时有内容。 包含可能影响消息解释的 QoS。

元素	类型	含义
serializedPayload	SerializedPayload	仅在子消息报文头中设置 DataFlag 或 KeyFlag 时显示。 如果设置了 DataFlag，则它包含更改后数据对象新的值的封装。 如果设置了 KeyFlag，则它包含消息所引用的数据对象的关键字的封装。

2.3.7.2.3有效性

如果以下任一条件成立，则此类子消息无效：

•子消息报文头中的 *submessageLength* 太小。

•*writerSN.value* 不是正整数（1,2，...）或 SEQUENCENUMBER_UNKNOWN。

•*inlineQos* 无效。

2.3.7.2.4接收器状态的变化

无。

2.3.7.2.5逻辑解释

RTPS ***Writer*** 将**数据（Data）**子消息发送到 RTPS ***Reader***，以便对写入者内的数据对象进行通信。更改包括值的更改以及数据对象生命周期的更改。

通过 *serializedPayload* 的来传输对数据对象值的更改。如果 *serializedPayload* 有内容则 *serializedPayload* 将被解析为数据对象的值或者被解析为从已注册对象集中唯一标识数据对象的关键字。

•如果设置了 *DataFlag* 且未设置 *KeyFlag*，则 *serializedPayload* 元素将被解释为数据对象的值。

•如果设置了 *KeyFlag* 且未设置 *DataFlag*，则 *serializedPayload* 元素将被解释为标识数据对象的已注册实例的关键字的值。

如果设置了 *InlineQosFlag*，则 *inlineQos* 元素包含覆盖 RTPS ***Writer*** 的 QoS 值的 QoS，应该用于处理更新。有关可能的内联 QoS 参数的完整列表，请参见表 2.80。

Writer 由其 GUID 唯一标识。 Writer GUID 是使用接收器的状态获得的：

writerGUID = { Receiver.sourceGuidPrefix，Data.writerId }

Reader 由 GUID 唯一标识。使用接收器的状态获取 Reader GUID：

readerGUID = { Receiver.destGuidPrefix，Data.readerId }

Data.readerId 可以是 ENTITYID_UNKNOWN，在这种情况下，数据适用于 ***GuidPrefix_t*** Receiver.destGuidPrefix 标识的 ***Participant*** 中该 *writerGUID* 的所有读取者。

2.3.7.3 数据分片（DataFrag）

此类子消息从 RTPS ***Writer***（NO_KEY 或 WITH_KEY）发送到 RTPS ***Reader***（NO_KEY

或 WITH_KEY）。

2.3.7.3.1 目的

DataFrag 子消息通过对 *serializedData* 进行分片并作为多个 **DataFrag** 子消息发送来扩展 **Data** 子消息。RTPS ***Reader*** 重新汇总 **DataFrag** 子消息中包含的分片。

除了扩展 Data 子消息发送报文大小之外，定义单独的 DataFrag 子消息还具有以下优点：

- 它将每个子消息的内容和结构的变化保持在最低限度。这使得能够更有效地实现协议，因为简化了网络分组的解析。
- 它避免了必须在数据子消息中添加碎片信息作为内联 QoS 参数。这可能不仅会降低性能，还会使线上调试变得更加困难，因为数据是否拥有碎片以及哪些消息包含哪些分片已不再明显。

2.3.7.3.2 内容

构成**数据分片（DataFrag）**消息结构的元素在下表中描述。

表 2-35 数据分片（DataFrag）子消息结构

元素	类型	含义
EndiannessFlag	SubmessageFlag	出现在子消息报文头标志中。表示字节序。
InlineQosFlag	SubmessageFlag	出现在子消息报文头标志中。 向 Reader 指示存在 ParameterList，该列表包含用于解释消息的 QoS 参数。
readerId	EntityId	标识被通知存在数据对象更改的 RTPS ***Reader*** 实体。
writerId	EntityId	标识对数据对象进行更改的 RTPS ***Writer*** 实体。
writerSN	SequenceNumber	唯一标识由 writerGuid 标识的 RTPS ***Writer*** 所做的所有更改和更改的相对顺序。 每次更改都会获得一个连续的序列号。每个 RTPS ***Writer*** 维护自己的序列号。
fragmentStartingNum	FragmentNumber	表示 *serializedData* 中一系列分片的起始分片。 分片编号从 1 开始。
fragmentsInSubmessage	ushort	子消息中包含的连续分片数，从 *fragmentStartingNum* 开始
dataSize	ulong	分片之前原始数据的总大小（以字节为单位）
fragmentSize	ushort	单个分片的大小（以字节为单位）。 最大分片大小为 64KB
inlineQos	ParameterList	仅在子消息报文头中设置 InlineQosFlag 时有内容。 包含可能影响消息解释的 QoS。

元素	类型	含义
serializedPayload	SerializedPayload	仅当标头中设置了 DataFlag 时才显示。 封装一系列连续分片数据，从 *fragmentStartingNum* 开始，共有 *fragmentInSubmessage* 个分片。 表示更改后数据对象新值的一部分。仅在子消息报文头中设置 DataFlag 或 KeyFlag 时显示。 •如果设置了 DataFlag，则它包含一组连续的分片，这些分片封装了更改后数据对象的新值。 •如果设置了 KeyFlag，则它包含一组连续的分片，用于封装消息所引用的数据对象的关键字。 在任何一种情况下，连续的分片集都包含 *fragmentInSubmessage* 个分片，并以 *fragmentStartingNum* 标识的分片开始。

2.3.7.3.3 有效性

如果以下任一条件成立，则此类子消息无效：

- 子消息报文头中的 *submessageLength* 太小。
- *writerSN.value* 不是正整数（1,2，...）或 SEQUENCENUMBER_UNKNOWN。
- *fragmentStartingNum.value* 不是正整数（1,2，...）或超过分片总数（见下文）。
- *fragmentSize* 超过 *dataSize*。
- *serializedData* 的大小超过 *fragmentsInSubmessage* * *fragmentSize*。
- *inlineQos* 无效。

2.3.7.3.4 接收器状态的变化

无。

2.3.7.3.5 逻辑解释

DataFrag 子消息将 *serializedData* 进行分片并作为多个 DataFrag 子消息发送来扩展 **Data** 子消息。一旦通过 RTPS *Reader* 重新组装 *serializedData*，DataFrag 子消息的解析与 Data 子消息的解析相同。

如何使用 **DataFrag** 子消息中的信息重新组装 *serializedData* 如下所述。

要重新组装的数据的总大小由 dataSize 给出。每个 **DataFrag** 子消息在其 *serializedData* 元素中包含此数据的连续段。段的大小由 *serializedData* 元素的大小决定。在重新组装期间，每个段的偏移量由以下方法确定：

（fragmentStartingNum - 1）* fragmentSize

收到所有分片后，数据完全重新组装。分片总数等于：

（dataSize / fragmentSize）+（（dataSize％fragmentSize）？1：0）

请注意，每个 **DataFrag** 子消息可能包含多个分片。RTPS ***Writer*** 将根据所有底层传输方式支持的最小消息大小选择 *fragmentSize*。如果可以通过支持更大消息的传输方式来访问某些 RTPS ***Reader***，则 RTPS ***Writer*** 可以将多个分片打包到单个 **DataFrag** 子消息中，甚至不再需要分片，直接发送常规数据子消息。有关 **DataFrag** 更多详细信息，请参见 2.4.14.1。

2.3.7.4 空白（Gap）

2.3.7.4.1 目的

此类子消息从 RTPS ***Writer*** 发送到 RTPS ***Reader***，告知 RTPS ***Reader*** 一系列序列号不再相关。该集合可以是连续的序列号范围或一组特定的序列号。

2.3.7.4.2 内容

构成**空白（Gap）**消息结构的元素在下表中描述。

表 2-36 空白（Gap）子消息结构

元素	类型	含义
EndiannessFlag	SubmessageFlag	出现在子消息报文头标志中。表示字节序。
readerId	EntityId	标识正在被告知一组序列号无关的 RTPS ***Reader*** 实体。
writerId	EntityId	标识序列号范围适用的写入者实体。
gapStart	SequenceNumber	标识无关序列号区间中的第一个序列号。
gapList	SequenceNumberSet	有两个目的： （1）标识无关序列号间隔中的最后一个序列号。 （2）标识不相关的序列号的附加列表

2.3.7.4.3 有效性

如果以下任一条件成立，则此类子消息无效：

- 子消息报文头中的 *submessageLength* 太小。
- *gapStart* 为零或负数。
- *gapList* 无效（如 2.3.5.5 中所定义）。

2.3.7.4.4 接收器状态的变化

无。

2.3.7.4.5 逻辑解释

RTPS ***Writer*** 将 Gap 消息发送到 RTPS ***Reader*** 以通知某些序列号不再相关。这通常是由样本的 ***Writer*** 端过滤引起的（内容过滤的主题，基于时间的过滤）。在这种情况下，新数据值可能会替换 Gap 消息中无关的序列号表示的数据对象的旧值。

由 Gap 消息传递的无关序列号由两组组成：

1. gapStart <= sequence_number <= gapList.base -1 范围内的所有序列号
2.出现在 gapList 中明确列出的所有序列号。

该集合将被称为 Gap :: irrelevant_sequence_number_list。

写入者由其 GUID 唯一标识。使用接收器的状态获取 Writer GUID：
writerGUID = {Receiver.sourceGuidPrefix，Gap.writerId}
读取者由其 GUID 唯一标识。使用接收器的状态获取 Reader GUID：
readerGUID = {Receiver.destGuidPrefix，Gap.readerId}

2.3.7.5 心跳报文（Heartbeat）

2.3.7.5.1 目的

此消息从 RTPS ***Writer*** 发送到 RTPS ***Reader***，以传达 **Writer** 可用的更改的序列号。

2.3.7.5.2 内容

构成**心跳报文（Heartbeat）**消息结构的元素在下表中描述。

表 2-37 心跳报文（Heartbeat）子消息结构

元素	类型	含义
EndiannessFlag	SubmessageFlag	出现在子消息报文头标志中。表示字节序。
FinalFlag	SubmessageFlag	出现在子消息报文头标志中。 指示 **Reader** 是否需要响应 Heartbeat 或者它是否只是一个咨询心跳。
LivelinessFlag	SubmessageFlag	出现在子消息报文头标志中。 表示与消息的 RTPS ***Writer*** 关联的 DDS DataWriter 已手动声明其存活性
readerId	EntityId	标识读取者实体，这些实体被告知某组数据序列号的可用性。 可以设置为 ENTITYID_UNKNOWN 指明为发送消息的写入者匹配的的所有读取者。
writerId	EntityId	标识序列号范围适用的写入者实体。
firstSN	SequenceNumber	标识写入者中可用的第一个（最低）序列号。
lastSN	SequenceNumber	标识写入者中可用的最后（最高）序列号。
count	Count	每次发送新的心跳报文时递增的计数器。 为读取者提供检测可能由于存在冗余通信路径而导致的重复心跳报文的方法。

2.3.7.5.3有效性

如果以下任一条件成立，则此类子消息无效：

- 子消息报文头中的 *submessageLength* 太小。
- *firstSN.value* 为零或负数。
- *lastSN.value* 为零或负数。
- *lastSN.value* < *firstSN.value*。

2.3.7.5.4接收器状态的变化

无。

2.3.7.5.5逻辑解释

心跳报文（Heartbeat）报文消息有两个用途：

1.它将写入者 ***HistoryCache*** 中可用的序列号告知读取者，以便读取者可以请求（使用 **AckNack**）任意丢失的序列号。

2.它要求读取者发送已经存入读取者 ***HistoryCache*** 中的 ***CacheChange*** 更改的确认，以便写入者知道读取者的状态。

所有**心跳报文（Heartbeat）**消息都有第一个用途。也就是说读取者最终都会获取到写入者 ***HistoryCache*** 的状态，并可能会请求它丢失的数据。通常，RTPS ***Reader*** 只有在丢失 ***CacheChange*** 时才会发送 **AckNack** 消息。

写入者使用 *FinalFlag* 请求读取者发送已收到的序列号的确认。如果**心跳报文（Heartbeat）**设置了 *FinalFlag*，那么读取者不需要发回 **AckNack** 消息。但是如果未设置 *FinalFlag*，则读取者必须发送 **AckNack** 消息，表明它已收到哪些 ***CacheChange*** 更改，即使 **AckNack** 内容代表它已收到写入者 ***HistoryCache*** 中的所有 ***CacheChange*** 更改。

写入者设置 *LivelinessFlag* 以指明与消息的 RTPS ***Writer*** 关联的 DDS DataWriter 已使用适当的 DDS 方法手动声明其活跃性（请参阅 DDS 规范）。因此，RTPS ***Reader*** 应该更新相应远程 DDS DataWriter 的存活租期。

写入者由其 GUID 唯一标识。使用接收器的状态获取 Writer GUID：

writerGUID = {Receiver.sourceGuidPrefix，Heartbeat.writerId}

读取者由其 GUID 唯一标识。使用接收器的状态获取 Reader GUID：

readerGUID = {Receiver.destGuidPrefix，Heartbeat.readerId}

Heartbeat.readerId 可以是 ENTITYID_UNKNOWN，在这种情况下，**心跳报文（Heartbeat）**适用于 ***Participant*** 中该 writerGUID 的所有读取者。

2.3.7.6 心跳分片报文（HeartbeatFrag）

2.3.7.6.1目的

在对数据进行分片并且直到所有分片都可用时，**心跳分片（HeartbeatFrag）**子消息从

RTPS ***Writer*** 发送到 RTPS ***Reader*** 以传达 Writer 可用的分片。这实现了分片级别的可靠通信。

一旦所有分片都可用，就会使用常规的**心跳报文（Heartbeat）**消息。

2.3.7.6.2 内容

构成**心跳分片报文（HeartbeatFrag）**消息结构的元素在下表中描述。

表 2-38 心跳分片报文（HeartbeatFrag）子消息结构

元素	类型	含义
EndiannessFlag	SubmessageFlag	出现在子消息报文头标志中。表示字节序。
readerId	EntityId	标识读取者实体，这些实体被告知分片的可用性。 可以设置为 ENTITYID_UNKNOWN 指明为发送消息的写入者匹配的的所有读取者。
writerId	EntityId	标识发送子消息的写入者实体。
writerSN	SequenceNumber	标识可用分片的数据更改的序列号。
lastFragmentNum	FragmentNumber	所有分片（包括最后一个（最高）分片）在写入者端可用于通过 *writerSN* 识别的更改。
count	Count	每次发送新的心跳分片报文时递增的计数器。 为读取者提供检测可能由于存在冗余通信路径而导致的重复心跳分片报文的方法。

2.3.7.6.3 有效性

如果以下任一条件成立，则此类子消息无效：

- 子消息报文头中的 *submessageLength* 太小。
- *writerSN.value* 为零或负数。
- *lastFragmentNum.value* 为零或负数。

2.3.7.6.4 接收器状态的变化

无。

2.3.7.6.5 逻辑解释

心跳分片报文（HeartbeatFrag）报文消息与常规**心跳报文（Heartbeat）**消息的用途相同，但它不是指明一系列序列号的可用性，而是指示序列号为 *WriterSN* 的数据更改的一系列分片的可用性。

RTPS ***Reader*** 将通过发送 **NackFrag** 消息进行响应，前提是它缺少可用的分片。

写入者由其 GUID 唯一标识。使用接收器的状态获取 Writer GUID：

writerGUID = {Receiver.sourceGuidPrefix，Heartbeat.writerId}

读取者由其 GUID 唯一标识。使用接收器的状态获取 Reader GUID：

readerGUID = {Receiver.destGuidPrefix，Heartbeat.readerId}

HeartbeatFrag.readerId 可以是 ENTITYID_UNKNOWN，在这种情况下，**心跳分片报文（HeartbeatFrag）**适用于 ***Participant*** 中该 Writer GUID 的所有读取者。

2.3.7.7 信息目的地址（InfoDestination）

2.3.7.7.1 目的

此消息从 RTPS ***Writer*** 发送到 RTPS ***Reader*** 以修改 GuidPrefix，该 GuidPrefix 用于解析出现在后面子消息中的 ***Reader*** entityId。

2.3.7.7.2 内容

构成**信息目的地址（InfoDestination）**消息结构的元素在下表中描述。

表 2-39 信息目的地址（InfoDestination）子消息结构

元素	类型	含义
EndiannessFlag	SubmessageFlag	出现在子消息报文头标志中。表示字节序。
guidPrefix	GuidPrefix	提供 GuidPrefix，该 GuidPrefix 应该用于重建所有 RTPS ***Reader*** 的 GUID，这些 RTPS ***Reader*** 的 EntityIds 出现在后面子消息中。

2.3.7.7.3 有效性

如果以下任一条件成立，则此类子消息无效：

- 子消息报文头中的 *submessageLength* 太小。

2.3.7.7.4 接收器状态的变化

```
if (InfoDestination.guidPrefix != GUIDPREFIX_UNKNOWN)
{
    Receiver.destGuidPrefix = InfoDestination.guidPrefix
} else
{
    Receiver.destGuidPrefix = < 接收该消息的参与者 GuidPrefix_t >
}
```

2.3.7.7.5 逻辑解释

无。

2.3.7.8 信息回复（InfoReply）

2.3.7.8.1 目的

此消息从 RTPS ***Reader*** 发送到 RTPS ***Writer***。它包含向哪里发送回复报文的明确信息，该明确信息在同一消息中的后续部分。

2.3.7.8.2 内容

构成**信息回复（InfoReply）**消息结构的元素在下表中描述。

表 2-40 信息回复（InfoReply）子消息结构

元素	类型	含义
EndiannessFlag	SubmessageFlag	出现在子消息报文头标志中。表示字节序。
MulticastFlag	SubmessageFlag	出现在子消息报文头标志中。 指明子消息是否还包含多播地址。
unicastLocatorList	LocatorList	表示写入者在回复后面的子消息时可供选择的用来访问读取者的另一组单播地址。
multicastLocatorList	LocatorList	表示写入者在回复后面的子消息时可供选择的用来访问读取者的另一组多播地址。 仅在设置 *MulticastFlag* 时出现。

2.3.7.8.3 有效性

如果以下任一条件成立，则此类子消息无效：

- 子消息报文头中的 *submessageLength* 太小。

2.3.7.8.4 接收器状态的变化

```
Receiver.unicastReplyLocatorList = InfoReply.unicastLocatorList
if ( MulticastFlag )
{
    Receiver.multicastReplyLocatorList = InfoReply.multicastLocatorList
} else
{
    Receiver.multicastReplyLocatorList = <empty>
}
```

2.3.7.8.5 逻辑解释

无。

2.3.7.9 信息源地址（InfoSource）

2.3.7.9.1 目的

此消息修改后面子消息的逻辑源地址。

2.3.7.9.2 内容

构成**信息源地址（InfoSource）**消息结构的元素在下表中描述。

表 2-41 信息源地址（InfoSource）子消息结构

元素	类型	含义
EndiannessFlag	SubmessageFlag	出现在子消息报文头标志中。表示字节序。
protocolVersion	ProtocolVersion	表示用于封装后续子消息的协议版本。
vendorId	VendorId	表示封装后续子消息的供应商的 VendorId。
guidPrefix	GuidPrefix	标识作为 RTPS Writer 实体容器的参与者，它是后面子消息的来源。

2.3.7.9.3 有效性

如果以下任一条件成立，则此类子消息无效：

- 子消息报文头中的 *submessageLength* 太小。

2.3.7.9.4 接收器状态的变化

Receiver.sourceGuidPrefix = InfoSource.guidPrefix
Receiver.sourceVersion = InfoSource.protocolVersion
Receiver.sourceVendorId = InfoSource.vendorId
Receiver.unicastReplyLocatorList = { LOCATOR_INVALID }
Receiver.multicastReplyLocatorList = { LOCATOR_INVALID }
haveTimestamp = false

2.3.7.9.5 逻辑解释

无。

2.3.7.10 信息时间戳（InfoTimestamp）

2.3.7.10.1 目的

此类子消息用于发送时间戳，该时间戳适用于同一消息中的子消息。

2.3.7.10.2 内容

构成**信息时间戳（InfoTimestamp）**消息结构的元素在下表中描述。

表 2-42 信息时间戳（InfoTimestamp）子消息结构

元素	类型	含义
EndiannessFlag	SubmessageFlag	出现在子消息报文头标志中。表示字节序。
InvalidateFlag	SubmessageFlag	指明后续子消息是否应被视为具有时间戳。
timestamp	Timestamp	仅在报文头中未设置 *InvalidateFlag* 时显示。 包含应该用于解析后续子消息的时间戳。

2.3.7.10.3 有效性

如果以下任一条件成立，则此类子消息无效：

•子消息报文头中的 *submessageLength* 太小。

2.3.7.10.4 接收器状态的变化

```
if ( !InfoTimestamp.InvalidateFlag )
{
    Receiver.haveTimestamp = true
    Receiver.timestamp = InfoTimestamp.timestamp
} else
{
    Receiver.haveTimestamp = false
}
```

2.3.7.10.5 逻辑解释

无。

2.3.7.11 确认分片报文（NackFrag）

2.3.7.11.1 目的

确认分片报文（NackFrag）子消息用于将 ***Reader*** 的状态传递给 ***Writer***。当数据更改作为一系列分片发送时，**确认分片报文（NackFrag）**子消息允许 ***Reader*** 通知 ***Writer*** 它仍然缺少的特定分片号。

此子消息只能包含否定确认。请注意，这与 **AckNack** 子消息不同，后者包括正面和负面的确认。这种方法的优点包括：

•它消除了 **AckNack** 子消息引入的窗口限制。鉴于 *SequenceNumberSet* 的大小限制为 256，AckNack 子消息仅限于确认那些序列号不超过第一个丢失的样本编号 256 的

样本。确认第一个丢失样本之后的任意样本。

另一方面，**确认分片报文（NackFrag）**子消息可用于确认任何分片数，甚至超过较早的 AckNack 子消息中确认的 256 个分片数。当处理含有大量分片的数据样本时，这变得很重要。

•可以按任何顺序对分片进行否定确认。

2.3.7.11.2 内容

构成**确认分片报文（NackFrag）**消息结构的元素在下表中描述。

表 2-43 确认分片报文（NackFrag）子消息结构

元素	类型	含义
EndiannessFlag	SubmessageFlag	出现在子消息报文头标志中。表示字节序。
readerId	EntityId	标识请求接收特定分片的 **Reader** 实体。
writerId	EntityId	标识作为 NackFrag 消息目标的 **Writer** 实体。 这是被要求重新发送部分分片的 **Writer** 实体。
writerSN	SequenceNumber	标识丢失分片的序列号。
fragmentNumberState	FragmentNumberSet	将读取者的状态传递给写入者。 出现在集合中的分片编号表示读取者缺少的分片。未出现在集合中的未确定（可能已收到或未收到）。
count	Count	每次发送新的确认分片报文时递增的计数器。 为写入者提供检测可能由于存在冗余通信路径而导致的重复确认分片报文的方法。

2.3.7.11.3 有效性

如果以下任一条件成立，则此类子消息无效：

•子消息报文头中的 *submessageLength* 太小。

•*writerSN.value* 为零或负数。

•*fragmentNumberState* 无效（如 2.3.5.7 中所定义）。

2.3.7.11.4 接收器状态的变化

无。

2.3.7.11.5 逻辑解释

Reader 将 **NackFrag** 消息发送给 ***Writer***，以从 ***Writer*** 请求分片数据。

写入者由其 GUID 唯一标识。使用接收器的状态获取 Writer GUID：

writerGUID = {Receiver.destGuidPrefix，NackFrag.writerId}

读取者由其 GUID 唯一标识。使用接收器的状态获取 Reader GUID：

readerGUID = {Receiver.sourceGuidPrefix，NackFrag.readerId}

请求分片的序列号由 writerSN 给出。明确表示分片编号的机制取决于 PSM。通常，使用紧凑表示（例如位图）。

2.3.7.12 填充报文（Pad）

2.3.7.12.1 目的

此子消息的目的是允许引入任何必要的填充以满足任何所需的内存对齐要求。它没有其他意义。

2.3.7.12.2 内容

该子消息没有内容。它仅使用子消息的报文头部分来实现其目的。填充量由 *submessageLength* 的值确定。

2.3.7.12.3 有效性

该子消息始终有效。

2.3.7.12.4 接收器状态的变化

无。

2.3.7.12.5 逻辑解释

无。

2.4 行为模块

该模块描述了 RTPS 实体的动态行为。它描述了 RTPS ***Writer*** 端点和 RTPS ***Reader*** 端点之间的有效消息交换序列以及这些消息的时序约束。

2.4.1 概述

将 RTPS ***Writer*** 与 RTPS ***Reader*** 匹配后，它们都负责确保将 ***Writer*** 的 HistoryCache 中存在的 CacheChange 更改传递到 ***Reader*** 的 HistoryCache 中。

行为模块描述了匹配的 RTPS 写入者和读取者对必须进行的行为，以便传递 ***CacheChange*** 更改。该行为是通过使用 2.3 中定义的 RTPS 消息进行消息交换来定义的。

行为模块主要内容如下：

•2.4.2 列出了行为方面，RTPS 协议的所有实现必须满足的要求。满足这些要求的实现被认为是兼容的，并且可以与其他兼容的实现互操作。

•如上所述，多个实现有可能满足最低要求，其中每个实现可能会在内存占用，带宽使用率，可伸缩性和效率之间选择不同的折衷方案。RTPS 规范不要求具有相应行为的单个实现。相反，它定义了互操作的最低要求，并提供两种参考实现，即无状态参考实现和有状态参考实现，如第 2.4.3 节所述。

•协议行为取决于诸如可靠性 QoS 之类的设置以及是否使用关键字主题。2.4.4 讨论了可能的组合。

•2.4.5 和 2.4.6 定义符号约定，并定义此模块中使用的任何新类型。

•2.4.7 至 2.4.12 为两个参考实现建模。

•2.4.14 讨论了一些可选行为，包括对碎片数据的支持。

•最后，2.4.15 提供了实际实现的指南。

请注意如 2.2.9 所述，行为模块不对 DDS 实体与其相应的 RTPS 实体之间的交互进行建模。例如，仅假设 DDS DataWriter 在其 RTPS ***Writer*** 的 HistoryCache 中添加和删除 ***CacheChange*** 更改。DDS DataWriter 更改添加为 ***write*** 方法的一部分，并在不再需要时将其删除。重要的是要意识到 DDS DataWriter 可以在将 CacheChange 传递到一个或多个匹配的 RTPS ***Reader*** 端点之前将其删除。RTPS ***Writer*** 无法控制何时从 ***Writer*** 的 HistoryCache 中删除 ***CacheChange***。DDS DataWriter 负责删除那些根据通信状态和 DDS DataWriter 的 QoS 确定可以删除的 ***CacheChange*** 更改。例如 KEEP_LAST 的 HISTORY QoS 设置（深度为 1）允许 DataWriter 删除 ***CacheChange***（如果最近的更改替换了同一数据对象的值）。

2.4.1.1 示例行为

本小节的内容不是协议正式规范的一部分。本小节的目的是提供对该协议的直观理解。

图 2.14 列出了一个典型的时序图，说明了 RTPS ***Writer*** 和匹配的 RTPS ***Reader*** 之间的交互。在这种情况下，示例时序图使用有状态参考实现。

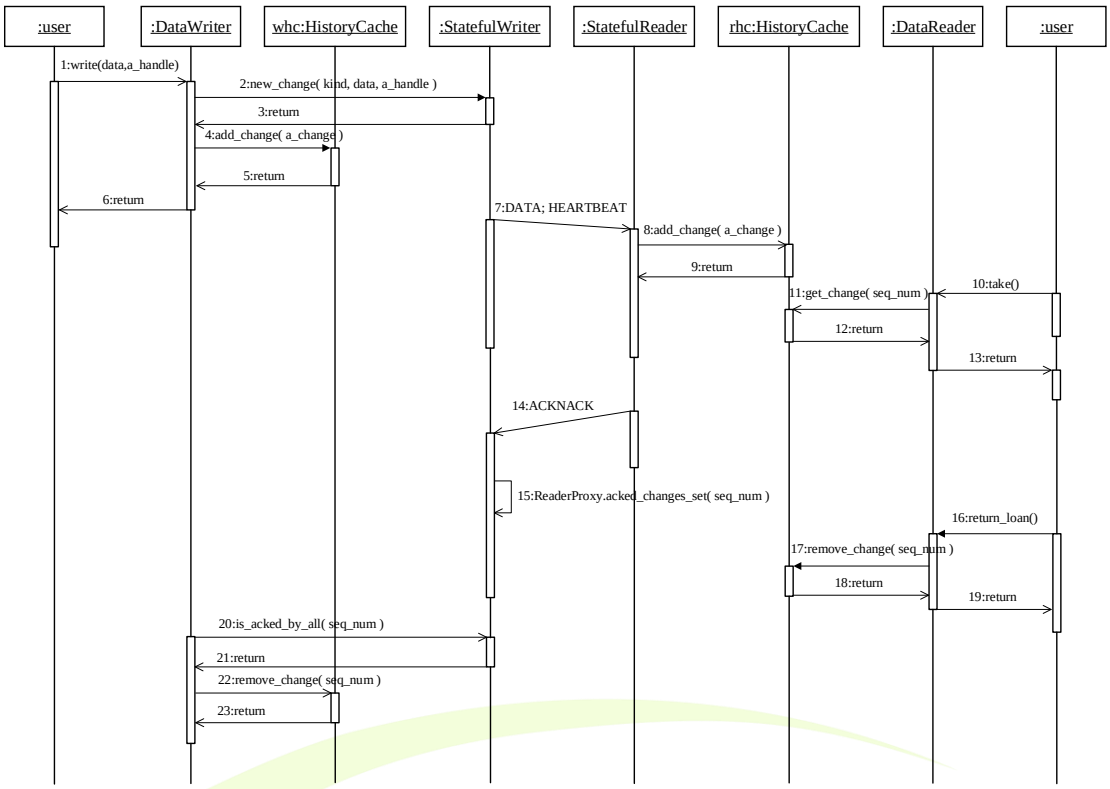


图 2-14 示例行为时序图

交互步骤解释如下：

1. DDS 用户通过在 DDS ***DataWriter*** 上调用 write 方法来写数据。

2. DDS ***DataWriter*** 在 RTPS ***Writer*** 上调用 **new_change** 方法创建一个新的 ***CacheChange***。每个 ***CacheChange*** 由 ***SequenceNumber*** 唯一标识。

3. **new_change** 方法返回。

4. DDS ***DataWriter*** 使用 **add_change** 方法将 ***CacheChange*** 存储到 RTPS ***Writer*** 的 ***HistoryCache*** 中。

5. **add_change** 方法返回。

6. **write** 方法返回，用户完成写入数据的操作。

7. RTPS ***Writer*** 使用数据（**Data**）子消息将 ***CacheChange*** 更改的内容发送到 RTPS ***Reader***，并通过发送心跳子消息来请求确认信息。

8. RTPS ***Reader*** 接收数据（**Data**）消息，并在资源限制允许的情况下，使用 **add_change** 方法将 ***CacheChange*** 放入 ***Reader*** 的 ***HistoryCache*** 中。

9. **add_change** 方法返回。***CacheChange*** 对 DDS ***DataReader*** 和 DDS 用户可见。此条件取决于 RTPS ***Reader*** 的可靠性级别属性。

   a. 对于 RELIABLE DDS ***DataReader***，仅当所有以前的更改（即序列号较小的更改）可见时，用户应用程序才能获取其 RTPS ***Reader*** 的 ***HistoryCache*** 中的更改。

   b. 对于 BEST_EFFORT DDS ***DataReader***，仅当没有收到未来的更改（如 RTPS 层接收者的 ***HistoryCache*** 中没有更高序号的更改）时，用户才能获取其 RTPS ***Reader*** 的 ***HistoryCache*** 中的更改。

10. 通过 DDS 规范描述的机制通知 DDS 用户（例如通过监听器或 ***WaitSet***），并通过调用 DDS ***DataReader*** 上的 **take** 方法来启动数据读取。

11. DDS ***DataReader*** 使用 ***HistoryCache*** 上的 **get_change** 方法访问更改。

12. **get_change** 方法将 ***CacheChange*** 返回到 DataReader。

13. **take** 方法将数据返回给 DDS 用户。

14. RTPS ***Reader*** 发送一条 **AckNack** 消息，表明 ***CacheChange*** 已放入 ***Reader*** 的 ***HistoryCache*** 中。**AckNack** 消息包含 RTPS ***Reader*** 的 ***GUID*** 和更改的序列号。此操作独立于通知 DDS 用户以及 DDS 用户的数据读取。它可能在此之前或同时发生。

15. ***StatefulWriter*** 记录 RTPS ***Reader*** 已收到 ***CacheChange***，并使用 **acked_changes_set** 方法将其添加到 ***ReaderProxy*** 维护的 *acked_changes* 集。

16. DDS 用户在 DataReader 上调用 **return_loan** 操作，以表明它不再使用先前通过 **take** 方法获取的数据。该行为独立于 DDS 用户引起的写入端的行为。

17. DDS ***DataReader*** 使用 **remove_change** 方法从 HistoryCache 中删除数据。

18. **remove_change** 方法返回

19. **return_loan** 方法返回

20. DDS ***DataWriter*** 使用 **is_acked_by_all** 方法来确定与 ***StatefulWriter*** 匹配的所有 RTPS ***Reader*** 端点已接收到哪些 ***CacheChanges***。

21. **is_acked_by_all** 返回并表明所有 RTPS ***Reader*** 端点已确认收到具有指定“seq_num” 序列号的更改。

22. DDS DataWriter 使用 **remove_change** 方法从 RTPS ***Writer*** 的 ***HistoryCache*** 中删除与“seq_num”关联的更改。在执行此方法时，DDS DataWriter 还需考虑其他 DDS QoS（例如 DURABILITY）。

23.操作 **remove_change** 返回。

上面的描述没有对 DDS ***DataReader*** 和 RTPS ***Reader*** 之间的某些交互进行建模。例如 RTPS ***Reader*** 用来提醒 DataReader 应该调用 **read** 或 **take** 方法来检查是否已接收到新更改的机制（即导致执行步骤 10 的原因）。

DDS ***DataWriter*** 和 RTPS ***Writer*** 之间的某些交互也是未建模的。例如 RTPS ***Writer*** 用来提醒 DataWriter 的机制，该机制检查是否已完全确认特定更改，以便可以将其从 ***HistoryCache*** 中删除（即导致上述步骤 20 触发的原因）。

前述交互未建模，因为它们是中间件实现的内部组成部分，并且对 RTPS 协议没有影响。

2.4.2 互操作所需的行为

本小节描述了 RTPS 协议的所有实现必须满足的要求，以便：

- 符合协议规范，
- 与其他实现可互操作。

这些要求的范围仅限于不同供应商在 RTPS 实现之间进行消息交换。

对于同一供应商的实现之间的消息交换，供应商可以选择不兼容的实现，也可以使用专有协议。

2.4.2.1 一般要求

以下要求适用于所有 RTPS 实体。

2.4.2.1.1 所有通信都必须使用 RTPS 消息进行

除 2.3 中定义的 RTPS 消息外，不能使用其他类型消息。RTPS 规范定义了每个消息的所需内容，有效性和解释。

供应商可以使用协议提供的扩展机制来扩展针对特定供应商需求的消息（请参见 2.6）。这不会影响互操作性。

2.4.2.1.2 所有实现都必须实现 RTPS 消息接收器

实现必须实现 2.3.4 引入的 RTPS 消息接收器（**Message Receiver**）遵循的规则，以解释 RTPS 消息内的子消息并维护消息接收器的状态。

此要求还包括在正确解析实体子消息时，在前面的实体子消息和解释器子消息之间进行适当的消息格式化，如 2.3.7 所定义。

2.4.2.1.3 所有实现的时序特征必须可调

根据应用程序要求、部署配置和底层传输，终端用户可能希望调整 RTPS 协议的时序特性。

因此，在对协议行为的要求允许延迟响应或指定周期性事件的地方，实现必须允许终端用户调整那些时序特性。

2.4.2.1.4 实现必须实现简单参与者和端点发现协议

实现必须实现简单参与者和端点发现协议以启用对远程端点的发现（请参阅 2.5）。

RTPS 允许根据应用程序的部署需求使用不同的参与者和端点发现协议。为了实现互操作，实现必须至少实现简单参与者发现协议和简单端点发现协议（请参阅 2.5.1）。

2.4.2.2 所需的 RTPS Writer 行为

以下要求仅适用于 RTPS ***Writer***。除非另有说明，否则要求同时适用于可靠写入者和尽力而为写入者。

2.4.2.2.1 写入者不得乱序发送数据

Writer 必须按照将其添加到 ***HistoryCache*** 的顺序发送数据样本。

2.4.2.2.2 如果读取者需要，写入者必须包括内联 QoS 值

Writer 必须满足 ***Reader*** 接收具有内联 QoS 的数据消息的请求。

2.4.2.2.3 编写者必须定期发送 HEARTBEAT 消息（仅针对可靠写入者）

Writer 必须通过发送包括可用样本序列号的周期性的 HEARTBEAT 消息，定期向每个匹配的可靠 ***Reader*** 通知数据样本的可用性。如果没有可用的样本，则无需发送任何 HEARTBEAT 消息。

为了进行严格可靠的通信，***Writer*** 必须持续地向 ***Reader*** 发送 HEARTBEAT 消息，直到 ***Reader*** 确认已收到所有可用样本或消失为止。在所有其他情况下，发送的 HEARTBEAT 消息的数量可能是特定于实现的，并且可能是有限的。

2.4.2.2.4 写入者最终必须对否定的回应做出回应（仅针对可靠写入者）

当收到表明 ***Reader*** 缺少某些数据样本的 ACKNACK 消息时，***Writer*** 必须通过以下方式做出响应：发送缺少的数据样本，在样本无关时发送 GAP 消息；或在样本不再可用时发送 HEARTBEAT 消息。

Writer 可以立即做出响应，也可以选择将响应安排在将来的某个时间。它还可以合并相关的响应，因此 ACKNACK 消息和 ***Writer*** 的响应不必一一对应。 这些决定和时序特征是特定于实现的。

2.4.2.3 所需的 RTPS Reader 行为

尽力而为的 ***Reader*** 是完全被动的，因为它仅接收数据而自身不发送消息。因此，以下要求仅适用于可靠的 ***Reader***。

2.4.2.3.1 读取者在收到未设置最终标志的 HEARTBEAT 消息之后必须做出回应

收到未设置最终标志的 HEARTBEAT 消息后，***Reader*** 必须以 ACKNACK 消息作为响应。

ACKNACK 消息可能确认已收到所有数据样本，或者可能指示某些数据样本已丢失。

响应可能会延迟以避免消息风暴。

2.4.2.3.2 读取者在收到表明样本缺失的 HEARTBEAT 消息之后必须做出回应

收到 HEARTBEAT 消息后，缺少某些数据样本的 ***Reader*** 必须以 ACKNACK 消息作为响应，以指示缺少哪些数据样本。仅当 ***Reader*** 可以将这些丢失的样本容纳在其缓存中时，此要求才适用，并且此要求与 HEARTBEAT 消息中的最终标志的设置无关。

响应可能会延迟以避免消息风暴。

当存活性 HEARTBEAT 同时设置了存活性标志和最终标志以指示它仅仅是活动性消息时，则不需要响应。

2.4.2.3.3一旦确认，就始终确认

Reader 使用 ACKNACK 消息肯定地确认接收到样本后，便不可在以后否定地确认相同的样本。

一旦 ***Writer*** 收到所有 ***Reader*** 的肯定确认，***Writer*** 就可以收回该数据样本关联的任一资源。但是如果 ***Writer*** 收到对先前已肯定确认的样本的否定确认，并且写入者仍可以处理该请求，则写入者应发送该样本。

2.4.2.3.4读取者只能发送 ACKNACK 消息作为对 HEARTBEAT 消息的响应

在稳定状态下，ACKNACK 消息只能作为对 ***Writer*** 的 HEARTBEAT 消息的响应而发送。ACKNACK 消息可以在读取者首次发现 ***Writer*** 时作为一种优化从 ***Reader*** 发送。***Writer*** 不需要响应这些先发的 ACKNACK 消息。

2.4.3 实现 RTPS 协议

RTPS 规范指出，该协议的兼容实现仅需要满足 2.4.2 中提出的要求。因此，实际实现的行为可能会因每个实现所考虑的设计权衡而有所不同。

RTPS 规范的行为模块定义了两个参考实现：

•无状态参考实现：

无状态参考实现针对可扩展性进行了优化。它在远程实体上几乎不保持任何状态，因此可以在大型系统上很好地扩展。这涉及一个权衡，因为可扩展性的提高和内存的减少使用可能需要额外的带宽使用。无状态参考实现非常适合使用多播的尽力而为通信。

•有状态参考实现：

有状态参考实现在远程实体上维护完整状态。此方法可最大程度地减少带宽使用，但需要更多内存，并且可能意味着可扩展性的降低。与无状态参考实现相反，它可以保证严格可靠的通信，并且能够在 ***Writer*** 端应用基于 QoS 或基于内容的筛选。

这两个参考实现在后面的小节中都有详细描述。

实际实现不必一定遵循参考实现。取决于维持多少状态，实现可以是参考实现的组合。

例如无状态参考实现在远程实体上维护的信息和状态最少。因此，它无法在 ***Writer*** 端执行基于时间的筛选，因为这需要跟踪每个远程 ***Reader*** 及其属性。它也无法在 ***Reader*** 端丢弃乱序样本，因为这需要跟踪从每个远程 ***Writer*** 接收到的最大数据样本序列号。一些实现可以模仿无状态参考实现，但也可以存储足够的附加状态以能够避免上述某些限制。可以以永久方式存储所需的附加信息，在这种情况下，实现将接近有状态参考实现，或者可以根据需要缓慢地进行老化和保留，如果状态保持不变以尽可能近似地将导致的行为。

无论实际实现如何，为了保证互操作性，重要的是所有实现（包括两个参考实现）都必须满足 2.4.2 中的要求。

2.4.4 写入者对每个匹配读取者的行为

对于每个匹配的 ***Reader***，RTPS ***Writer*** 的行为取决于：

•RTPS ***Writer*** 和 RTPS ***Reader*** 中设置可靠性级别属性。这决定了使用尽力而为协议还是可靠协议。

•RTPS ***Writer*** 和 RTPS ***Reader*** 中 *topicKind* 属性的设置。这控制着正在通信的数据是否对应于已为其定义了关键字的 DDS 主题。

并非可靠性级别和 *topicKind* 属性的所有可能组合都是可能的。除非满足以下两个条件，否则 RTPS ***Writer*** 不能与 RTPS ***Reader*** 匹配：

1. RTPS ***Writer*** 和 ***Reader*** 都必须具有相同的 *topicKind* 属性值。这是因为它们都与同一个 DDS 主题相关，该主题将为 WITH_KEY 或 NO_KEY。

2.或者将 RTPS ***Writer*** 的可靠性级别设置为 RELIABLE，或者将 RTPS ***Writer*** 和 RTPS ***Reader*** 的可靠性级别设置为 BEST_EFFORT。这是因为 DDS 规范指出，BEST_EFFORT DDS DataWriter 只能与 BEST_EFFORT DDS DataReader 匹配，而 RELIABLE DDS DataWriter 可以与 RELIABLE 和 BEST_EFFORT DDS DataReader 匹配。

如 2.4.3 所述，***Writer*** 是否可以与 ***Reader*** 匹配并不取决于两者是否都使用 RTPS 协议的相同实现。也就是说有状态写入者可以与无状态读取者进行通信，反之亦然。

表 2-44 总结了该协议支持的合法组合。后续子小节描述了列出的每种组合的行为。

表 2-44 匹配的 RTPS Writer 和 RTPS Reader 属性的可能组合

Writer 属性	Reader 属性	组合名称
topicKind = WITH_KEY reliabilityLevel = BEST_EFFORT 或 reliabilityLevel = RELIABLE	topicKind = WITH_KEY reliabilityLevel = BEST_EFFORT	WITH_KEY Best-Effort
topicKind = NO_KEY reliabilityLevel = BEST_EFFORT 或 reliabilityLevel = RELIABLE	topicKind = NO_KEY reliabilityLevel = BEST_EFFORT	NO_KEY Best-Effort
topicKind = WITH_KEY reliabilityLevel = RELIABLE	topicKind = WITH_KEY reliabilityLevel = RELIABLE	WITH_KEY Reliable
topicKind = NO_KEY reliabilityLevel = RELIABLE	topicKind = NO_KEY reliabilityLevel = RELIABLE	NO_KEY Reliable

2.4.5 符号约定

使用 UML 时序图和状态图描述了参考实现。这些图使用一些缩写来表示 RTPS 实体。表 2-45 中列出了使用的缩写。

表 2-45 行为模块的时序图和状态图中使用的缩写

缩写	含义	用法示例
DW	DDS DataWriter	DW::write
DR	DDS DataReader	DR::read

缩写	含义	用法示例
W	RTPS Writer	W::heartbeatPeriod
RP	RTPS ReaderProxy	RP::unicastLocatorList
RL	RTPS ReaderLocator	RL::locator
R	RTPS Reader	R::heartbeatResponseDelay
WP	RTPS WriterProxy	WP::remoteWriterGuid
WHC	HistoryCache of RTPS Writer	WHC::changes
RHC	HistoryCache of RTPS Reader	RHC::changes

2.4.6 类型定义

行为模块引入了以下其他类型。

表 2-46 行为模块的类型定义

RTPS 模型类中使用的类型	
属性类型	**目的**
Duration_t	用于保存时间差的类型。 应至少具有纳秒级的分辨率
ChangeForReaderStatusKind	用于指明 ***ChangeForReader*** 状态的枚举类型。 可以取以下值： UNSENT, UNACKNOWLEDGED, REQUESTED, ACKNOWLEDGED, UNDERWAY
ChangeFromWriterStatusKind	用于指明 ***ChangeFromWriter*** 状态的枚举类型。 可以取以下值： LOST, MISSING, RECEIVED, UNKNOWN
InstanceHandle_t	用于表示数据对象标识的类型，其值的更改通过 RTPS 协议进行通信。
ParticipantMessageData	用于保存参与者之间交换的数据的类型。这种类型最值得注意的用途是用于 ***Writer*** 存活协议。

2.4.7 RTPS Writer 参考实现

RTPS Writer 参考实现基于 2.2 中首次引入的 RTPS ***Writer*** 类的专业化。本小节描述了 RTPS ***Writer*** 以及用于建模 RTPS ***Writer*** 参考实现的所有其他类。实际行为在 2.4.8 和 2.4.9 中描述.

2.4.7.1 RTPS Writer

RTPS ***Writer*** 是 RTPS ***Endpoints*** 的特殊化，并承担着将 ***CacheChange*** 消息发送到匹配的 RTPS ***Reader*** 端点的职能。参考实现 ***StatelessWriter*** 和 ***StatefulWriter*** 进一步特殊化 RTPS ***Writer***，它们对匹配的 ***Reader*** 端点的维护信息上有所不同。

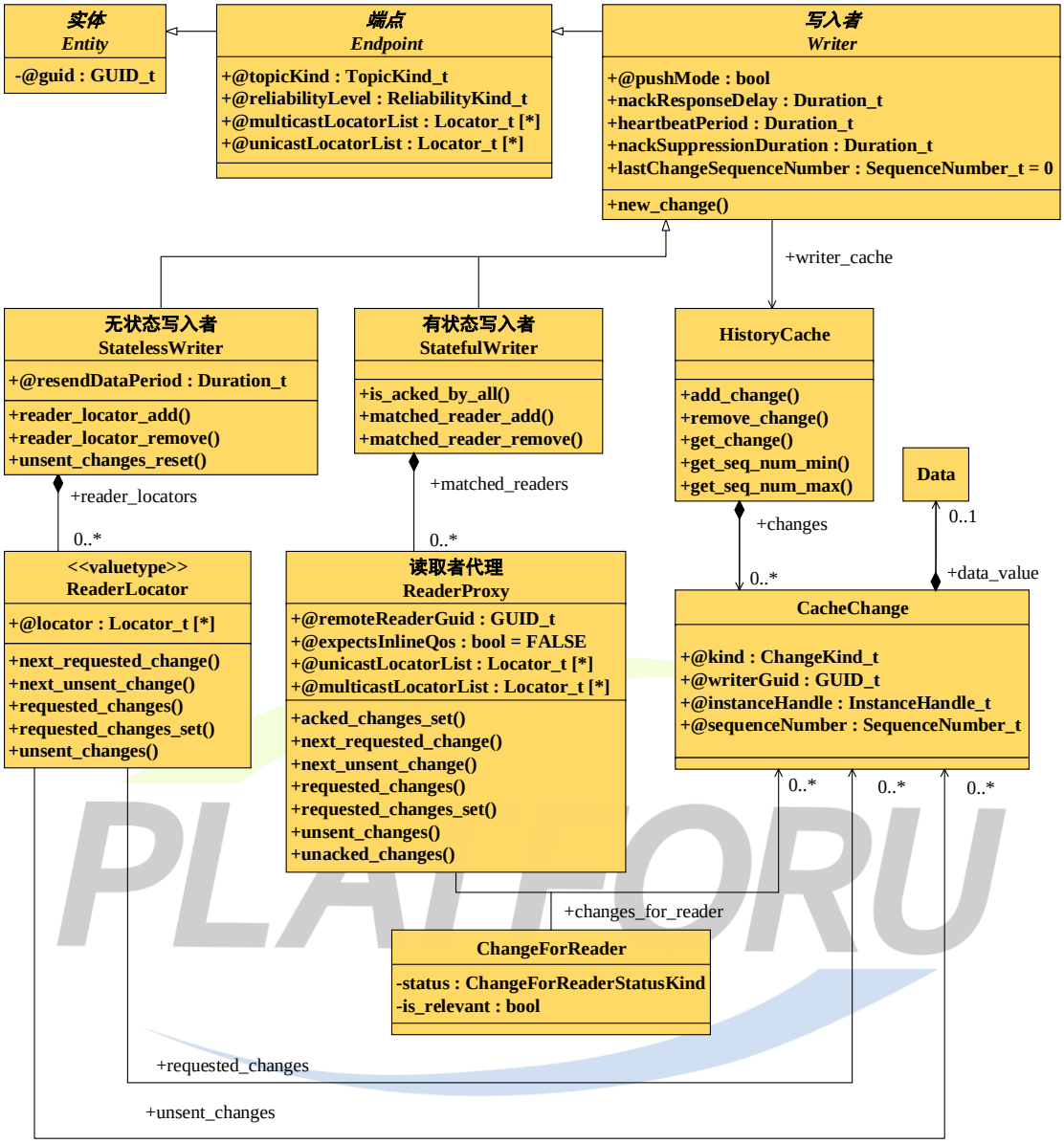


图 2-15 RTPS 写入者端点

表 2-47 描述了 RTPS ***Writer*** 的属性。

表 2-47 RTPS Writer 属性

RTPS Writer : RTPS Endpoint			
属性	**类型**	**含义**	**与 DDS 关系**
pushMode	bool	配置 Writer 操作的模式。如果 pushMode == true，则 Writer 会将更改推送到 Reader。如果 pushMode == false，则更改仅通过心跳宣布，并且仅作为对 Reader 请求的响应	N/A（自动配置）

RTPS Writer : RTPS Endpoint			
属性	**类型**	**含义**	**与 DDS 关系**
		发送。	
heartbeatPeriod	Duration_t	协议调整参数，允许 RTPS ***Writer*** 通过发送心跳消息来反复声明数据的可用性。	N/A（自动配置）
nackResponseDelay	Duration_t	协议调整参数，允许 RTPS ***Writer*** 延迟对来自否定确认的数据请求的响应	N/A（自动配置）
nackSuppressionDuration	Duration_t	协议调整参数，允许 RTPS ***Writer*** 忽略来自否定确认的数据请求，这些否定确认在发送相应更改后“很快”到达。	N/A（自动配置）
lastChangeSequenceNumber	SequenceNumber_t	内部计数器，用于为写入者所做的每次更改分配递增的序列号。	N/A（用作虚拟机逻辑的一部分）
writer_cache	HistoryCache	包含此 ***Writer*** 的 ***CacheChange*** 更改的历史记录。	N/A

RTPS ***Writer*** 的属性允许对协议行为进行微调。表 2-48 描述了 RTPS ***Writer*** 的方法。

表 2-48 RTPS Writer 方法

RTPS Writer 方法		
方法名	*参数列表*	*类型*
new	<return value>	Writer
	attribute_values	Writer 和所有父类所需的属性值集。
new_change	<return value>	CacheChange
	kind	ChangeKind_t
	data	Data
	handle	InstanceHandle_t

以下子节提供了有关方法的详细信息。

2.4.7.1.1 默认时序相关值

以下与时序相关的值用作默认值，以促进不同实现之间的“开箱即用”互操作性。

nackResponseDelay.sec = 0;

nackResponseDelay.nanosec = 200 * 1000 * 1000; //200 milliseconds

nackSuppressionDuration.sec = 0;

nackSuppressionDuration.nanosec = 0;

2.4.7.1.2 new 方法

此方法将创建一个新的 RTPS ***Writer***。

新创建的写入者“this”的初始化方式如下：

this.guid := <as specified in the constructor>;

this.unicastLocatorList := <as specified in the constructor>;

this.multicastLocatorList := <as specified in the constructor>;

this.reliabilityLevel := <as specified in the constructor>;

this.topicKind := <as specified in the constructor>;

this.pushMode := <as specified in the constructor>;

this.heartbeatPeriod := <as specified in the constructor>;

this.nackResponseDelay := <as specified in the constructor>;

this.nackSuppressionDuration := <as specified in the constructor>;

this.lastChangeSequenceNumber := 0;

this.writer_cache := new HistoryCache;

2.4.7.1.3 new_change 方法

此方法将创建一个新的 ***CacheChange***，以添加到 RTPS ***Writer*** 的 ***HistoryCache*** 中。***CacheChange*** 的序列号自动设置为上一个更改的序列号加 1。

此方法将返回新的更改。

此方法执行以下逻辑步骤：

++this.lastChangeSequenceNumber;

a_change := new CacheChange(kind, this.guid, this.lastChangeSequenceNumber,
data, handle);

RETURN a_change;

2.4.7.2 RTPS StatelessWriter

RTPS StatelessWriter 是用于无状态参考实现的 RTPS ***Writer*** 的特殊化。RTPS StatelessWriter 不清楚匹配的 ***Reader*** 的数量，也不为每个匹配的 RTPS ***Reader*** 端点维护任何状态。 RTPS StatelessWriter 仅维护用于向匹配的 ***Reader*** 发送信息的 RTPS Locator_t 列表。

表 2-49 RTPS StatelessWriter 属性

RTPS StatelessWriter : RTPS Writer			
属性	**类型**	**含义**	**与 DDS 关系**
resendDataPeriod	Duration_t	协议调优参数，指示 StatelessWriter 将写程序 HistoryCache 中的所有更改重新发送给所有 Locator，每个周期重新发送一次。	N/A（自动配置）
reader_locators	ReaderLocator[*]	StatelessWriter 维护的 CacheChanges 的目的定位器列表。此列表可能包括单播和多播定位器。	N/A（自动配置）

RTPS StatelessWriter 在以下情况下很有用：（a）写入者的 ***HistoryCache*** 很小，或者（b）通信为尽力而为的方式，或者（c）写入者通过多播与大量读取者进行通讯。

虚拟机使用表 2-50 中的方法与 ***StatelessWriter*** 进行交互。

表 2-50 StatelessWriter 方法

RTPS StatelessWriter 方法		
方法名	*参数列表*	*类型*
new	<return value>	StatelessWriter
	attribute_values	StatelessWriter 和所有父类所需的属性值集。
reader_locator_add	<return value>	void
	a_locator	Locator_t
reader_locator_remove	<return value>	void
	a_locator	Locator_t
unsent_changes_reset	<return value>	void

2.4.7.2.1 new 方法

此方法将创建一个新的 RTPS StatelessWriter。

除了在 RTPS ***Writer*** 父类（2.4.7.1.2）上执行的初始化外，新创建的 StatelessWriter“this”还按以下方式初始化：

this.readerlocators := <empty>;

this.resendDataPeriod := <as specified in the constructor>;

2.4.7.2.2 reader_locator_add

此方法将 ***Locator_t*** a_locator 添加到 ***StatelessWriter*** :: reader_locators。

ADD a_locator TO {this.reader_locators};

2.4.7.2.3 reader_locator_remove

此方法从 ***StatelessWriter*** :: reader_locators 中删除 ***Locator_t*** a_locator

REMOVE a_locator FROM {this.reader_locators};

2.4.7.2.4 unsent_changes_reset

此方法为 ***StatelessWriter*** :: reader_locators 中的所有 ***ReaderLocator*** 修改“unsent_changes”集。重置未发送的更改列表，以匹配写入者 HistoryCache 中可用的完整更改列表。

FOREACH readerLocator in {this.reader_locators} DO

readerLocator.unsent_changes := {this.writer_cache.changes}

2.4.7.3 RTPS ReaderLocator

RTPS ***StatelessWriter*** 使用的 Valuetype 跟踪所有匹配的远程 ***Reader*** 的定位器。

表 2-51 RTPS ReaderLocator 属性

RTPS ReaderLocator			
属性	**类型**	**含义**	**与 DDS 关系**
requested_changes	CacheChange[*]	远程 Reader 在此 ReaderLocator 上请求的 Writer 的 HistoryCache 更改列表。	N/A（自动配置）
unsent_changes	CacheChange[*]	Writer 的 HistoryCache 中尚未发送到此 ReaderLocator 的更改列表。	N/A（自动配置）
locator	Locator_t	单播或多播定位器，可以通过它到达此 ReaderLocator 代表的 Reader。	N/A（自动配置）
expectsInlineQos	bool	指定此 ReaderLocator 代表的 Reader 是否希望与每个数据消息一起发送内联 QoS。	

虚拟机使用表 2-52 中的方法与 ReaderLocator 进行交互。

表 2-52 ReaderLocator 方法

RTPS ReaderLocator 方法		
方法名	*参数列表*	*类型*
new	<return value>	ReaderLocator
	attribute_values	ReaderLocator 所需的属性值集。
next_requested_change	<return value>	ChangeForReader
next_unsent_change	<return value>	ChangeForReader
requested_changes	<return value>	CacheChange[*]
requested_changes_set	<return value>	void
	req_seq_num_set	SequenceNumber_t[*]
unsent_changes	<return value>	CacheChange[*]

2.4.7.4 RTPS StatefulWriter

RTPS ***StatefulWriter*** 是用于有状态参考实现的 RTPS ***Writer*** 的特殊化。使用所有匹配的 RTPS ***Reader*** 端点的信息配置 RTPS ***StatefulWriter***，并维护每个匹配的 RTPS ***Reader*** 端点的状态。

通过维护每个匹配的 RTPS ***Reader*** 端点上的状态，RTPS ***StatefulWriter*** 可以确定是否所有匹配的 RTPS ***Reader*** 端点都已接收到特定的 ***CacheChange***，并且可以避免向已接收到 ***Writer*** 的 HistoryCache 中所有更改的 ***Reader*** 发送通知，从而在网络带宽使用方面达到最佳状态。它维护的信息还简化了 ***Writer*** 端基于 QoS 的过滤。表 2-53 中描述了 ***StatefulWriter*** 特有的属性。

表 2-53 RTPS StatefulWriter 属性

RTPS StatefulWriter			
属性	**类型**	**含义**	**与 DDS 关系**
matched_readers	ReaderProxy[*]	StatefulWriter 跟踪与之匹配的所有 RTPS Reader。每个匹配的 Reader 由 ReaderProxy 类的一个实例表示。	N/A（自动配置）

虚拟机使用表 2-54 中的方法与 ***StatefulWriter*** 进行交互。

表 2-54 StatefulWriter 方法

StatefulWriter 方法		
方法名	*参数列表*	*类型*
new	<return value>	StatefulWriter
	attribute_values	StatefulWriter 及所有父类所需的属性值集。
matched_reader_add	<return value>	void
	a_reader_proxy	ReaderProxy
matched_reader_remove	<return value>	void
	a_reader_proxy	ReaderProxy
matched_reader_lookup	<return value>	ReaderProxy
	a_reader_guid	GUID_t
is_acked_by_all	<return value>	bool
	a_change	CacheChange

2.4.7.4.1 new 方法

此方法将创建一个新的 RTPS ***StatefulWriter***。除了在 RTPS Writer 父类（2.4.7.1.2）上执行的初始化外，新创建的 ***StatefulWriter*** “this”还按以下方式初始化：

this.matched_readers := <empty>;

2.4.7.4.2 is_acked_by_all

此方法将 ***CacheChange*** a_change 作为参数，并确定是否所有 ***ReaderProxy*** 都已确认收到该 CacheChange。如果所有 ***ReaderProxy*** 均已确认相应的 CacheChange，则该方法将返回 true，否则返回 false。

return true IF and only IF
FOREACH proxy IN this.matched_readers
IF change IN proxy.changes_for_reader THEN
change.is_relevant == TRUE AND change.status == ACKNOWLEDGED

2.4.7.4.3 matched_reader_add

此方法将 ***ReaderProxy*** a_reader_proxy 添加到 ***StatefulWriter*** :: matched_readers 集合中。

ADD a_reader_proxy TO {this.matched_readers};

2.4.7.4.4 matched_reader_remove

此方法从 ***StatefulWriter*** :: matched_readers 集合中删除 ***ReaderProxy*** a_reader_proxy。

```
REMOVE a_reader_proxy FROM {this.matched_readers};
delete proxy;
```

2.4.7.4.5 matched_reader_lookup

此方法从 ***StatefulWriter*** :: matched_readers 集合中找到具有 GUID_t a_reader_guid 的 ***ReaderProxy***。

```
FIND proxy IN this.matched_readers SUCH-THAT
    (proxy.remoteReaderGuid == a_reader_guid);
return proxy;
```

2.4.7.5 RTPS ReaderProxy

RTPS ***ReaderProxy*** 类代表着 RTPS ***StatefulWriter*** 为每个匹配的 RTPS Reader 维护的信息。表 2-55 描述了 RTPS ***ReaderProxy*** 的属性。

表 2-55 RTPS ReaderProxy 属性

RTPS ReaderProxy			
属性	**类型**	**含义**	**与 DDS 关系**
remoteReaderGuid	GUID_t	标识由 ReaderProxy 表示的远程匹配的 RTPS Reader。	N/A（通过发现配置）
unicastLocatorList	Locator_t[*]	可用于将消息发送到匹配的 RTPS Reader 的单播定位器列表（传输，地址，端口组合）。该列表可能为空。	N/A（通过发现配置）
multicastLocatorList	Locator_t[*]	可用于将消息发送到匹配的 RTPS Reader 的多播定位器列表（传输，地址，端口组合）。该列表可能为空。	N/A（通过发现配置）
changes_for_reader	CacheChange[*]	与匹配的 RTPS ***Reader*** 相关的 ***CacheChange*** 更改列表。	N/A。用于实现 RTPS 协议的行为。
expectsInlineQos	bool	指明远程匹配的 RTPS ***Reader*** 是否希望将内联 QoS 与任一数据一起发送。	
isActive	bool	指定远程 ***Reader*** 是否对 ***Writer*** 作出响应。	N/A

RTPS ***StatefulWriter*** 与 RTPS ***Reader*** 的匹配意味着 RTPS ***StatefulWriter*** 会将 ***Writer*** 的 ***HistoryCache*** 中的 CacheChange 更改发送到由 ***ReaderProxy*** 表示的匹配的 RTPS ***Reader***。匹配是相应 DDS 实体匹配的结果。也就是说，DDS DataWriter 按主题匹配 DDS DataReader，具有兼容的 QoS，并且使用 DDS 的应用程序不会将其明确忽略。

虚拟机使用表 2-56 中的方法与 ***ReaderProxy*** 进行交互。

表 2-56 ReaderProxy 方法

ReaderProxy 方法		
方法名	*参数列表*	*类型*
new	<return value>	ReaderProxy
	attribute_values	ReaderProxy 所需的属性值集。
acked_changes_set	<return value>	void
	committed_seq_num	SequenceNumber_t
next_requested_change	<return value>	ChangeForReader
next_unsent_change	<return value>	ChangeForReader
unsent_changes	<return value>	ChangeForReader[*]
requested_changes	<return value>	ChangeForReader[*]
requested_changes_set	<return value>	void
	req_seq_num_set	SequenceNumber_t[*]
unacked_changes	<return value>	ChangeForReader[*]

2.4.7.5.1 new 方法

此方法将创建一个新的 RTPS ***ReaderProxy***。 新创建的读取者代理“this”的初始化如下：

```
this.attributes := <as specified in the constructor>;
this.changes_for_reader := RTPS::Writer.writer_cache.changes;
FOR_EACH change IN (this.changes_for_reader) DO
{
    IF ( DDS_FILTER(this, change) )
        THEN change.is_relevant := FALSE;
    ELSE change.is_relevant := TRUE;

    IF ( RTPS::Writer.pushMode == true)
        THEN change.status := UNSENT;
    ELSE change.status := UNACKNOWLEDGED;
}
```

上面的逻辑表明，新创建的 ***ReaderProxy*** 初始化其“changes_for_reader”集合，以包含 ***Writer*** 的 ***HistoryCache*** 中的所有 ***CacheChanges***。

如果任一应用程序的 DDS-DataReader 过滤器表明更改与该特定读取者都不相关，则该更改被标记为“不相关”。DDS 规范指明 DataReader 可以提供基于时间的过滤器以及基于内容的过滤器。应该以与 DDS 规范一致的方式应用这些过滤器，以选择与 DataReader 不相关的任一更改。

根据 RTPS ***Writer*** 属性“pushMode”的值来设置状态。

2.4.7.5.2 acked_changes_set

此方法将更改由 ***ReaderProxy*** “the_reader_proxy”表示的读取者的一组更改的

ChangeForReader 状态。序列号小于或等于值“committed_seq_num”的一组更改将其状态更改为 ACKNOWLEDGED。

```
FOR_EACH change in this.changes_for_reader
        SUCH-THAT (change.sequenceNumber <= committed_seq_num) DO
    change.status := ACKNOWLEDGED;
```

2.4.7.5.3 next_requested_change

此方法返回状态为“REQUESTED”的更改中具有最低序列号的 ***ReaderProxy*** 的 ***ChangeForReader***。这表示下一个修复数据包，应响应先前的 **AckNack** 消息发送给 ***ReaderProxy*** 代表的 RTPS ***Reader***。（参见 2.3.7.1）

```
next_seq_num := MIN
{
    change.sequenceNumber SUCH-THAT change IN this.requested_changes()
}

return change IN this.requested_changes() SUCH-THAT (change.sequenceNumber
==next_seq_num);
```

2.4.7.5.4 next_unsent_change

此方法返回状态为“UNSENT”的更改中序列号最低的 ***ReaderProxy*** 的 ***CacheChange***。这表示下一个更改应发送到 ***ReaderProxy*** 代表的 RTPS ***Reader***。

```
next_seq_num := MIN
{
    change.sequenceNumber SUCH-THAT change IN this.unsent_changes()
};

return change IN this.unsent_changes() SUCH-THAT (change.sequenceNumber
==next_seq_num);
```

2.4.7.5.5 requested_changes

此方法返回状态为“REQUESTED”的 ***ReaderProxy*** 所需的更改子集。这表示由 ***ReaderProxy*** 表示的 RTPS ***Reader*** 使用 **ACKNACK** 消息请求的更改集。

```
return change IN this.changes_for_reader SUCH-THAT (change.status == REQUESTED);
```

2.4.7.5.6 requested_changes_set

此方法修改了由 ***ReaderProxy*** “this”代表的 RTPS ***Reader*** 的一组更改的 ***ChangeForReader*** 状态。序列号为“req_seq_num_set”的一组更改的状态更改为 REQUESTED。

```
FOR_EACH seq_num IN req_seq_num_set DO
    FIND change_for_reader IN this.changes_for_reader
            SUCH-THAT (change_for_reader.sequenceNumber==seq_num)
    change_for_reader.status := REQUESTED;
```

END

2.4.7.5.7 unsent_changes

此方法返回状态为“UNSENT”的 ***ReaderProxy*** 的更改子集。这表示尚未发送到 ***ReaderProxy*** 表示的 RTPS ***Reader*** 的一组更改。

return change IN this.changes_for_reader SUCH-THAT (change.status == UNSENT);

2.4.7.5.8 unacked_changes

此方法返回状态为“UNACKNOWLEDGED”的 ***ReaderProxy*** 的更改子集。这表示由 ***ReaderProxy*** 表示的 RTPS ***Reader*** 尚未确认的一组更改。

return change IN this.changes_for_reader
SUCH-THAT (change.status == UNACKNOWLEDGED);

2.4.7.6 RTPS ChangeForReader

RTPS ***ChangeForReader*** 是一个关联类，用于维护 RTPS ***Writer HistoryCache*** 中 ***CacheChange*** 的信息，因为它与 ***ReaderProxy*** 代表的 RTPS ***Reader*** 有关。表 2-57 描述了 RTPS ***ChangeForReader*** 的属性。

表 2-57 RTPS ChangeForReader 属性

RTPS ChangeForReader			
属性	**类型**	**含义**	**与 DDS 关系**
status	ChangeForReaderStatusKind	表明与 ReaderProxy 代表的 RTPS Reader 有关的 CacheChange 的状态。	N/A。被协议使用
isRelevant	bool	表明更改是否与 ReaderProxy 代表的 RTPS Reader 相关。	不相关更改的确定受 DDS DataReader TIME_BASED_FILTER QoS 以及 DDS ContentFilteredTopics 的使用影响。

2.4.8 RTPS StatelessWriter 行为

2.4.8.1 尽力而为 StatelessWriter 行为

图 2-16 说明了“尽力而为带关键字”RTPS ***StatelessWriter*** 相对于每个 ***ReaderLocator*** 的行为。

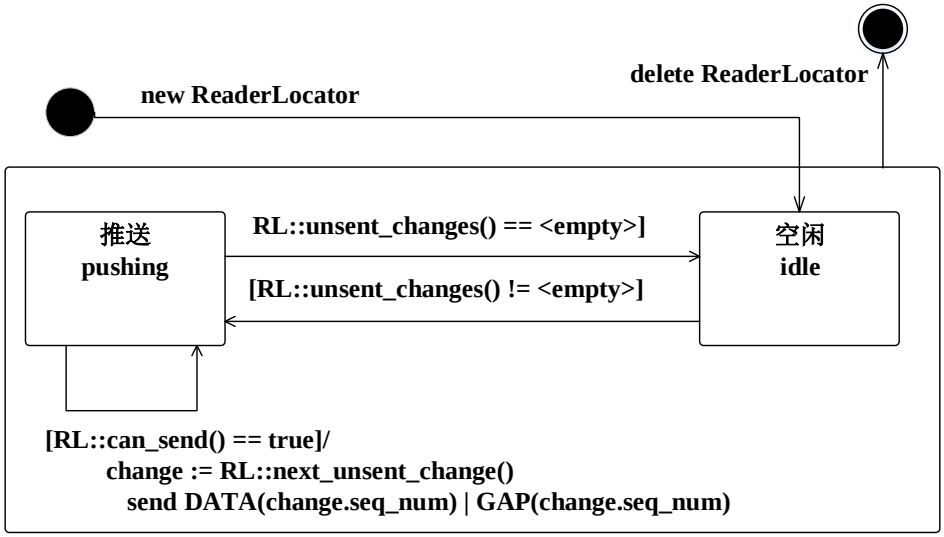


图 2-16 尽力而为带关键字 StatelessWriter 相对于每个 ReaderLocator 的行为

表 2-58 列出了状态机转换场景。

表 2-58 关于每个 ReaderLocator 的尽力而为 StatelessWriter 行为的转换场景

转换	状态	事件	下一状态
T1	初始（initial）	RTPS Writer 配置有 ReaderLocator	空闲（idle）
T2	空闲（idle）	GuardCondition: RL::unsent_changes() != <empty>	推送（pushing）
T3	推送（pushing）	GuardCondition: RL::unsent_changes() == <empty>	空闲（idle）
T4	推送（pushing）	GuardCondition: RL::can_send() == true	推送（pushing）
T5	任一状态	RTPS Writer 配置为不再具有 ReaderLocator	终止（final）

2.4.8.1.1 转换场景 T1

通过使用 RTPS ReaderLocator 配置 RTPS 尽力而为 ***StatelessWriter***“the_rtps_writer”来触发此转换。作为发现与“the_rtps_writer”相关的 DDS DataWriter 匹配的 DDS DataReader 的结果，通过发现协议（2.5）完成此配置。

发现协议为 ***ReaderLocator*** 构造函数参数提供值。

转换在虚拟机中执行以下逻辑操作：

a_locator := new ReaderLocator(locator, expectsInlineQos);

the_rtps_writer.reader_locator_add(a_locator);

2.4.8.1.2 转换场景 T2

此转换由保护条件[RL :: unsent_changes（）！= <empty>]触发，该保护条件指示 RTPS ***Writer HistoryCache*** 中的某些更改尚未发送到 RTPS ***ReaderLocator***。

此转换在虚拟机中不执行任何逻辑操作。

2.4.8.1.3转换场景 T3

此转换由保护条件[RL :: unsent_changes（）== <empty>]触发，该保护条件表示 RTPS ***Writer HistoryCache*** 中的所有更改均已发送到 RTPS ***ReaderLocator***。请注意，这并不意味 ***Reader*** 已收到更改，而只是尝试发送它们。

此转换在虚拟机中不执行任何逻辑操作。

2.4.8.1.4转换场景 T4

此转换由保护条件[RL :: can_send（）== true]触发，表明 RTPS ***Writer*** “the_writer”具有将更改发送到 RTPS ***ReaderLocator***“the_reader_locator”所需的资源。

此转换在虚拟机中执行以下逻辑操作：

```
a_change := the_reader_locator.next_unsent_change();
DATA = new DATA(a_change);
IF (the_reader_locator.expectsInlineQos)
{
    DATA.inlineQos := the_writer.related_dds_writer.qos;
}
DATA.readerId := ENTITYID_UNKNOWN;
sendto the_reader_locator.locator, DATA;
```

转换后，将保留以下后置条件：

```
( a_change BELONGS-TO the_reader_locator.unsent_changes() ) == FALSE
```


2.4.8.1.5转换场景 T5

这种转换是由 RTPS ***Writer*** “the_rtps_writer” 不再发送给 RTPS ***ReaderLocator*** “the_reader_locator”的配置触发的。作为打破了 DDS 的预先存在的匹配带来的结果，该匹配存在于与“the_rtps_writer”相关的 DDS DataWriter 和 DDS DataReader 之间，此配置由发现协议（2.5）完成。

转换在虚拟机中执行以下逻辑操作：

```
the_rtps_writer.reader_locator_remove(the_reader_locator);
delete the_reader_locator;
```

2.4.8.2 可靠 StatelessWriter 行为

图 2-17 描述了“可靠带关键字”的 RTPS ***StatelessWriter*** 相对于每个 ReaderLocator 的行为。对于“可靠不带关键字”***StatelessWriter***，协议保持相同的行为。

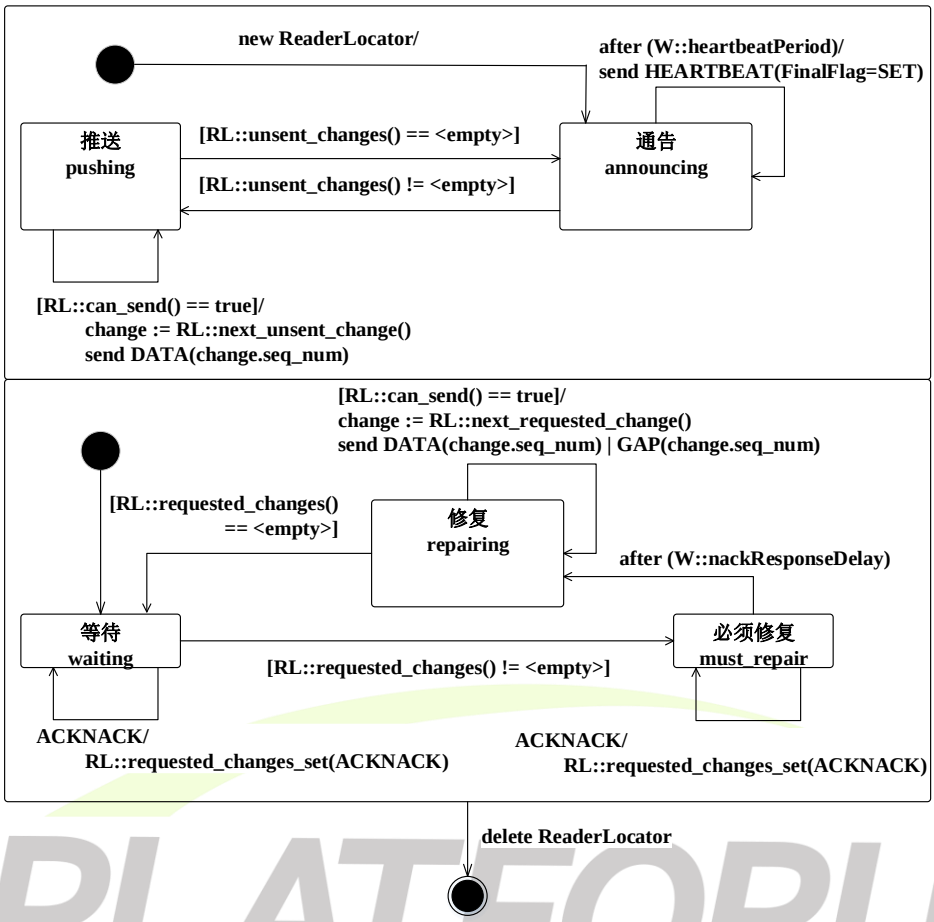


图 2-17 可靠带关键字 StatelessWriter 相对于每个 ReaderLocator 的行为

表 2-59 列出了状态机转换场景。

表 2-59 关于每个 ReaderLocator 的可靠 StatelessWriter 行为的转换场景

转换	状态	事件	下一状态
T1	初始（initial）	RTPS Writer 配置有 ReaderLocator	通告（announcing）
T2	通告（announcing）	GuardCondition: RL::unsent_changes() != <empty>	推送（pushing）
T3	推送（pushing）	GuardCondition: RL::unsent_changes() == <empty>	通告（announcing）
T4	推送（pushing）	GuardCondition: RL::can_send() == true	推送（pushing）
T5	通告（announcing）	after(W::heartbeatPeriod)	通告（announcing）
T6	等待（waiting）	收到 ACKNACK 消息	等待（waiting）
T7	等待（waiting）	GuardCondition: RL::requested_changes() != <empty>	必须修复（must_repair）
T8	必须修复（must_repair）	收到 ACKNACK 消息	必须修复（must_repair）
T9	必须修复（must_repair）	after(W::nackResponseDelay)	修复（repairing）

转换	状态	事件	下一状态
T10	修复（repairing）	GuardCondition: RL::can_send() == true	修复（repairing）
T11	修复（repairing）	GuardCondition: RL::requested_changes() == <empty>	等待（waiting）
T12	任一状态	RTPS Writer 配置为不再具有 Reader Locator	终止（final）

2.4.8.2.1 转换场景 T1

通过使用 RTPS ReaderLocator 配置 RTPS 可靠 ***StatelessWriter***“the_rtps_writer”来触发此转换。作为发现与“the_rtps_writer”相关的 DDS DataWriter 匹配的 DDS DataReader 的结果，通过发现协议（2.5）完成此配置。

发现协议为 ***ReaderLocator*** 构造函数参数提供值。

转换在虚拟机中执行以下逻辑操作：

```
a_locator := new ReaderLocator( locator, expectsInlineQos );
the_rtps_writer.reader_locator_add( a_locator );
```

2.4.8.2.2 转换场景 T2

此转换由保护条件[RL::unsent_changes（）！= <empty>]触发，表明 RTPS ***Writer HistoryCache*** 中的某些更改尚未发送到 ***ReaderLocator***。此转换在虚拟机中不执行任何逻辑操作。

2.4.8.2.3 转换场景 T3

此转换由保护条件[RL :: unsent_changes == <empty>]触发，该保护条件表示 RTPS ***Writer HistoryCache*** 中的所有更改都已发送到 ***ReaderLocator***。 请注意，这并不意味 ***Reader*** 已收到更改，而只是尝试发送它们。此转换在虚拟机中不执行任何逻辑操作。

2.4.8.2.4 转换场景 T4

此转换由保护条件[RL :: can_send（）== true]触发，表明 RTPS ***Writer***“the_writer”具有将更改发送到 RTPS ***ReaderLocator***“the_reader_locator”所需的资源。

转换在虚拟机中执行以下逻辑操作：

```
a_change := the_reader_locator.next_unsent_change();
DATA = new DATA(a_change);
IF (the_reader_locator.expectsInlineQos)
{
    DATA.inlineQos := the_writer.related_dds_writer.qos;
}
DATA.readerId := ENTITYID_UNKNOWN;
sendto the_reader_locator.locator, DATA;
```

转换后，将保留以下后置条件：

```
( a_change BELONGS-TO the_reader_locator.unsent_changes() ) == FALSE
```

2.4.8.2.5 转换场景 T5

此转换由配置周期为 W :: heartbeatPeriod 的定期计时器的触发来触发。

转换将在虚拟机中为 ***Writer***“the_rtps_writer”和 ***ReaderLocator*** “the_reader_locator”执行以下逻辑操作。

```
seq_num_min := the_rtps_writer.writer_cache.get_seq_num_min();
seq_num_max := the_rtps_writer.writer_cache.get_seq_num_max();
HEARTBEAT := new HEARTBEAT(the_rtps_writer.writerGuid, seq_num_min,
                                        seq_num_max);
HEARTBEAT.FinalFlag := SET;
HEARTBEAT.readerId := ENTITYID_UNKNOWN;
sendto the_reader_locator, HEARTBEAT;
```

2.4.8.2.6 转换场景 T6

接收到发给某个 RTPS ***Reader*** 的 RTPS ***StatelessWriter***“the_rtps_writer”的 ACKNACK 消息即可触发此转换。

转换在虚拟机中执行以下逻辑操作：

```
FOREACH reply_locator_t IN { Receiver.unicastReplyLocatorList,
                             Receiver.multicastReplyLocatorList }
    reader_locator := the_rtps_writer.reader_locator_lookup(reply_locator_t);
    reader_locator.requested_changes_set(ACKNACK.readerSNState.set);
```

请注意，此消息的处理使用 RTPS ***Receiver*** 中的答复定位符。这是 ***StatelessWriter*** 决定将答复发送到何处的唯一信息源。协议的正常运行需要 RTPS ***Reader*** 在 **AckNack** 之前插入一个 **InfoReply** 子消息，以便正确设置这些字段。

2.4.8.2.7 转换场景 T7

此转换由保护条件[RL :: requested_changes（）! =<空>]触发，该保护条件表明某些 RTPS ***Reader*** 请求的更改可通过 RTPS ***ReaderLocator*** 访问获取。此转换在虚拟机中不执行任何逻辑操作。

2.4.8.2.8 转换场景 T8

RTPS ***StatelessWriter***“the_rtps_writer”接收到来自某个 RTPS ***Reader*** 的 ACKNACK 消息即可触发此转换。转换执行与转换 T6（2.4.8.2.6）相同的逻辑操作。

2.4.8.2.9 转换场景 T9

触发计时器触发此转换，该计时器指示自进入状态 **must_repair** 以来 **W :: nackResponseDelay** 的持续时间已过去。转换在虚拟机中不执行任何逻辑操作。

2.4.8.2.10 转换场景 T10

此转换由保护条件[RL :: can_send（）== true）触发，表明 RTPS ***Writer*** 'the_writer'具有将更改发送给 RTPS ***ReaderLocator*** 'the_reader_locator'所需的资源。该转换在虚拟机中执行以下逻辑操作。

```
a_change := the_reader_locator.next_requested_change();
IF a_change IN the_writer.writer_cache.changes
{
    DATA = new DATA(a_change);
    IF (the_reader_locator.expectsInlineQos)
    {
        DATA.inlineQos := the_writer.related_dds_writer.qos;
    }
    DATA.readerId := ENTITYID_UNKNOWN;
    sendto the_reader_locator.locator, DATA;
}
ELSE
{
    GAP = new GAP(a_change.sequenceNumber);
    GAP.readerId := ENTITYID_UNKNOWN;
    sendto the_reader_locator.locator, GAP;
}
```

转换后，以下后置条件成立：

```
( a_change BELONGS-TO the_reader_locator.requested_changes() ) == FALSE
```

请注意，DDS ***DataWriter*** 可能已将请求的更改从 ***HistoryCache*** 中删除。在这种情况下，***StatelessWriter*** 发送一个 **GAP** 消息。

2.4.8.2.11 转换场景 T11

此转换由保护条件[RL :: requested_changes（）== <empty>]触发，该保护条件指明 RTPS ***ReaderLocator*** 上可达的 RTPS ***Reader*** 不再请求任何更改。该转换在虚拟机中不执行任何逻辑操作。

2.4.8.2.12 转换场景 T12

这种转换是由 RTPS ***Writer*** “the_rtps_writer ”不再发送更改给 RTPS ***Reader*** “the_reader_locator”的配置触发的。这个配置是由发现协议(2.5)完成的，因为它打破了 DDS ***DataReader*** 与“the_rtps_writer”相关的 DDS ***DataWriter*** 之间预先存在的匹配。

转换在虚拟机中执行以下逻辑操作。

```
the_rtps_writer.reader_locator_remove(the_reader_locator);
delete the_reader_locator;
```

2.4.9 RTPS StatefulWriter 行为

2.4.9.1 尽力而为 StatefulWriter 行为

图 2-18 描述了“尽力而为带关键字”的 RTPS ***StatefulWriter*** 相对于每个匹配的 RTPS ***Reader*** 的行为。“尽力而为不带关键字”RTPS ***StatefulWriter*** 的行为是相同的。

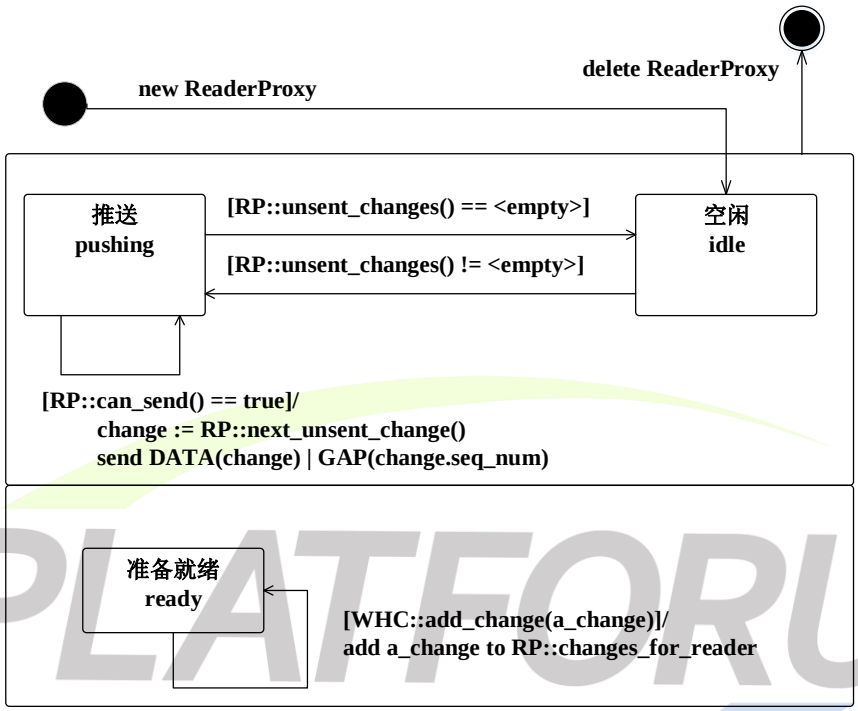


图 2-18 尽力而为带关键字 StatefulWriter 相对于每个匹配的 Reader 的行为

表 2-60 中列出了状态机转换场景。

表 2-60 针对每个匹配 Reader 的尽力而为 StatefulWriter 行为的转换场景

转换	状态	事件	下一状态
T1	初始（initial）	RTPS Writer 配置有一个匹配的 RTPS Reader	空闲（idle）
T2	空闲（idle）	GuardCondition: RP::unsent_changes() != <empty>	推送（pushing）
T3	推送（pushing）	GuardCondition: RP::unsent_changes() == <empty>	空闲（idle）
T4	推送（pushing）	GuardCondition: RP::can_send() == true	推送（pushing）
T5	准备就绪（ready）	RTPS ***Writer*** 的 ***HistoryCache*** 中添加了新的更改	准备就绪（ready）
T6	任一状态	RTPS ***Writer*** 配置为不再与 RTPS ***Reader*** 匹配	终止（final）

2.4.9.1.1转换场景 T1

通过配置具有匹配 RTPS ***Reader*** 的 RTPS ***Writer***“the_rtps_writer”来触发此转换。作为发现与“the_rtps_writer”相关的 DDS DataWriter 匹配的 DDS DataReader 的结果，通过发现协议（2.5）完成此配置。

发现协议为 ***ReaderProxy*** 构造函数参数提供值。

转换在虚拟机中执行以下逻辑操作：

```
a_reader_proxy := new ReaderProxy( remoteReaderGuid,
                                   expectsInlineQos,
                                   unicastLocatorList,
                                   multicastLocatorList);
the_rtps_writer.matched_reader_add(a_reader_proxy);
```

如 2.4.7.5 中所述初始化 ReaderProxy“a_reader_proxy”。这包括初始化未发送的更改集，并将 DDS_FILTER 应用于每个更改。

2.4.9.1.2转换场景 T2

此转换由保护条件[RP :: unsent_changes（）！= <empty>]触发，该保护条件指明 RTPS ***Writer HistoryCache*** 中的某些更改尚未发送到 ***ReaderProxy*** 代表的 RTPS ***Reader***。

请注意，对于尽力而为的 ***Writer***，W :: pushMode == true，因为没有确认。因此，***Writer*** 总是在数据可用时推送数据。

转换在虚拟机中不执行任何逻辑操作。

2.4.9.1.3转换场景 T3

此转换由保护条件[RP :: unsent_changes（）== <empty>]触发，该保护条件指明 RTPS ***Writer HistoryCache*** 中的所有更改都已发送给 ***ReaderProxy*** 代表的 RTPS ***Reader***。请注意，这并不表示已接收到更改，仅表示已尝试发送更改。

转换在虚拟机中不执行任何逻辑操作。

2.4.9.1.4转换场景 T4

此转换由保护条件[RP :: can_send（）== true）触发，表明 RTPS ***Writer***“the_rtps_writer”具有将更改发送到由 ***ReaderProxy***“the_reader_proxy”代表的 RTPS ***Reader*** 所需的资源。

转换在虚拟机中执行以下逻辑操作：

```
a_change := the_reader_proxy.next_unsent_change();
a_change.status := UNDERWAY;
if (a_change.is_relevant)
{
    DATA = new DATA(a_change);
    IF (the_reader_proxy.expectsInlineQos)
    {
        DATA.inlineQos := the_rtps_writer.related_dds_writer.qos;
    }
```

```
        DATA.readerId := ENTITYID_UNKNOWN;
        send DATA;
    }
```

上面的逻辑并不意味着每个 DATA 子消息都是在单独的 RTPS 消息中发送的。而是可以将多个子消息组合为单个 RTPS 消息。

转换后，将保留以下后置条件：

```
( a_change BELONGS-TO the_reader_proxy.unsent_changes() ) == FALSE
```

2.4.9.1.5转换场景 T5

相应的 DDS DataWriter 向 RTPS ***Writer***“the_rtps_writer”的 ***HistoryCache*** 中添加了新的 ***CacheChange***“a_change”，从而触发了此转换。更改是否与 ***ReaderProxy***“the_reader_proxy”代表的 RTPS ***Reader*** 相关，由 DDS_FILTER 决定。

转换在虚拟机中执行以下逻辑操作：

```
ADD a_change TO the_reader_proxy.changes_for_reader;
IF (DDS_FILTER(the_reader_proxy, change))
    THEN change.is_relevant := FALSE;
ELSE
    change.is_relevant := TRUE;
IF (the_rtps_writer.pushMode == true)
    THEN change.status := UNSENT;
ELSE
    change.status := UNACKNOWLEDGED;
```

2.4.9.1.6转换场景 T6

这种转换是由 RTPS ***Writer***“the_rtps_writer” 不再与 ***ReaderProxy***“the_reader_proxy”代表的 RTPS Reader 匹配的配置触发的。由于打破了 DDS DataReader 与“the_rtps_writer”相关的 DDS DataWriter 的预先存在的匹配关系，因此该配置由发现协议（2.5）完成。

此转换在虚拟机中执行以下逻辑操作：

```
the_rtps_writer.matched_reader_remove(the_reader_proxy);
delete the_reader_proxy;
```

2.4.9.2 可靠 StatefulWriter 行为

图 2-19 中描述了“可靠带关键字”RTPS ***StatefulWriter*** 相对于每个匹配的 RTPS ***Reader*** 的行为。“可靠不带关键字”的 RTPS ***StatefulWriter*** 的行为是相同的。

使用子消息代替数据子消息。

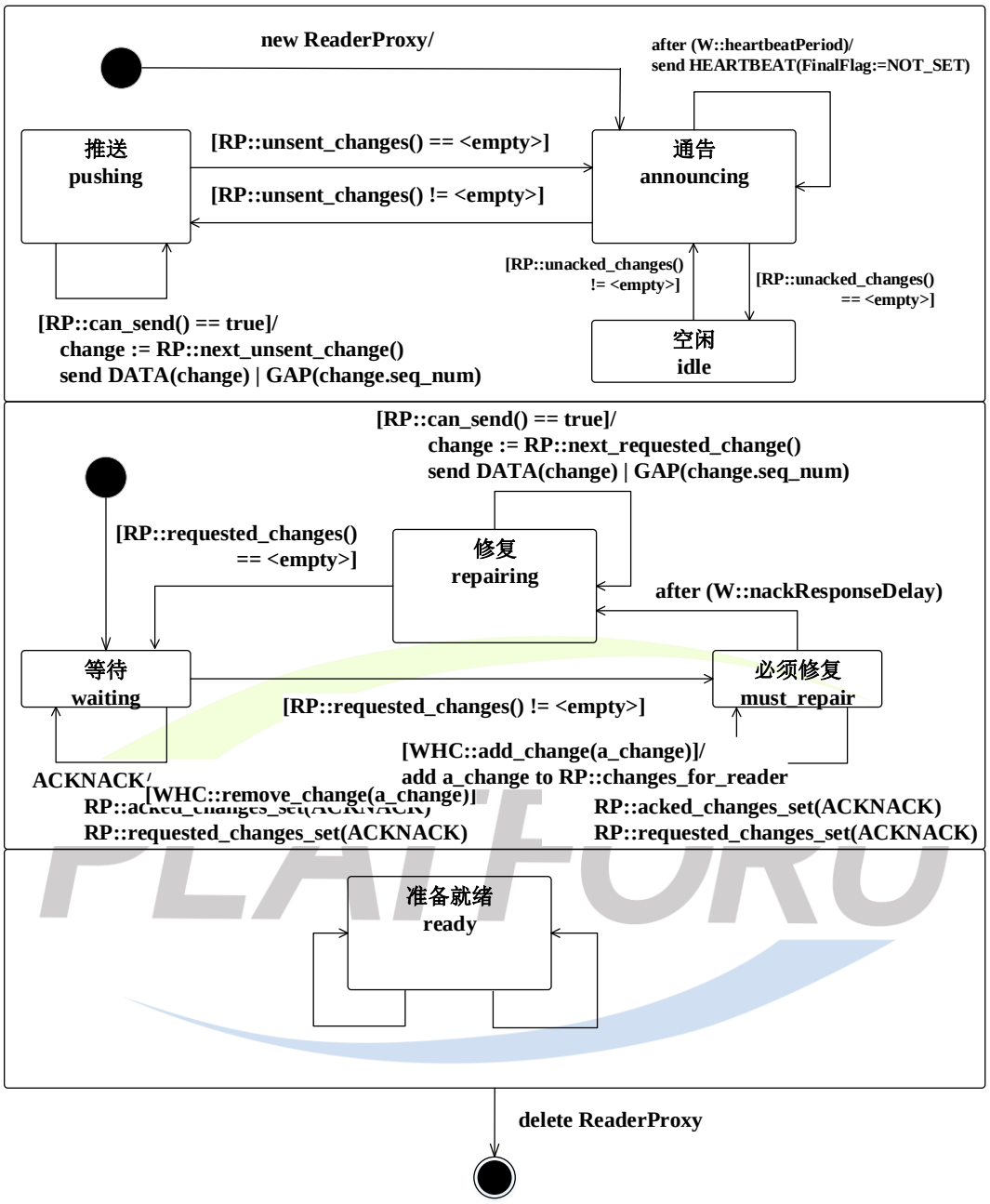


图 2-19 可靠带关键字 StatelfulWriter 相对于每个匹配的 Reader 的行为

表 2-61 列出了状态机转换场景。

表 2-61 针对每个匹配 Reader 的可靠 StatefulWriter 行为的转换

转换	状态	事件	下一状态
T1	初始（initial）	RTPS Writer 配置有一个匹配的 RTPS Reader	通告（announcing）
T2	通告（announcing）	GuardCondition: RP::unsent_changes() != <empty>	推送（pushing）
T3	推送（pushing）	GuardCondition: RP::unsent_changes() == <empty>	通告（announcing）

转换	状态	事件	下一状态
T4	推送（pushing）	GuardCondition: RP::can_send() == true	推送（pushing）
T5	通告（announcing）	GuardCondition: RP::unacked_changes() == <empty>	空闲（idle）
T6	空闲（idle）	GuardCondition: RP::unacked_changes() != <empty>	通告（announcing）
T7	通告（announcing）	经过 (W::heartbeatPeriod)时间	通告（announcing）
T8	等待（waiting）	收到 ACKNACK 消息	等待（waiting）
T9	等待（waiting）	GuardCondition: RP::requested_changes() != <empty>	必须修复 （must_repair）
T10	必须修复 （must_repair）	收到 ACKNACK 消息	必须修复 （must_repair）
T11	必须修复 （must_repair）	经过 (W::heartbeatPeriod)时间	修复（repairing）
T12	修复（repairing）	GuardCondition: RP::can_send() == true	修复（repairing）
T13	修复（repairing）	GuardCondition: RP::requested_changes() == <empty>	等待（waiting）
T14	准备就绪（ready）	RTPS Writer 的 HistoryCache 中添加了新的更改	准备就绪（ready）
T15	准备就绪（ready）	更改已从 RTPS Writer 的 HistoryCache 中删除	准备就绪（ready）
T16	任一状态	RTPS ***Writer*** 配置为不再与 RTPS ***Reader*** 匹配	终止（final）

2.4.9.2.1转换场景 T1

通过配置具有匹配 RTPS ***Reader*** 的 RTPS 可靠 ***StatefulWriter***“the_rtps_writer”来触发此转换。通过发现与“the_rtps_writer”相关的 DDS DataWriter 匹配的 DDS DataReader，由发现协议（2.5）完成此配置。

发现协议提供 ***ReaderProxy*** 构造函数参数的值。

转换在虚拟机中执行以下逻辑操作：

```
a_reader_proxy := new ReaderProxy( remoteReaderGuid,
                                   expectsInlineQos,
                                   unicastLocatorList,
                                   multicastLocatorList);
the_rtps_writer.matched_reader_add(a_reader_proxy);
```

如 2.4.7.5 中所述初始化 ***ReaderProxy*** “a_reader_proxy”。这包括初始化未发送的更改集，并对每个更改应用过滤器。

2.4.9.2.2转换场景 T2

此转换由保护条件[RP :: unsent_changes（）！= <empty>]触发，该保护条件指明 RTPS ***Writer HistoryCache*** 中的某些更改尚未发送到 ***ReaderProxy*** 代表的 RTPS ***Reader***。

此转换在虚拟机中不执行任何逻辑操作。

2.4.9.2.3转换场景 T3

此转换由保护条件[RP :: unsent_changes（）== <empty>]触发，该保护条件指明 RTPS ***Writer HistoryCache*** 中的所有更改都已发送给 ***ReaderProxy*** 代表的 RTPS ***Reader***。请注意，这并不表示接收方已收到更改，仅表示已尝试发送更改。

此转换在虚拟机中不执行任何逻辑操作。

2.4.9.2.4转换场景 T4

此转换由保护条件[RP :: can_send（）== true）触发，表明 RTPS ***Writer*** “the_rtps_writer”具有将更改发送到由 ***ReaderProxy***“the_reader_proxy”代表的 RTPS ***Reader*** 所需的资源。

转换在虚拟机中执行以下逻辑操作：

```
a_change := the_reader_proxy.next_unsent_change();
a_change.status := UNDERWAY;
if (a_change.is_relevant)
{
    DATA = new DATA(a_change);
    IF (the_reader_proxy.expectsInlineQos)
    {
        DATA.inlineQos := the_rtps_writer.related_dds_writer.qos;
    }
    DATA.readerId := ENTITYID_UNKNOWN;
    send DATA;
}
else
{
    GAP = new GAP(a_change.sequenceNumber);
    GAP.readerId := ENTITYID_UNKNOWN;
    send GAP;
}
```

上面的逻辑并不意味着每个 DATA 或 GAP 子消息都是在单独的 RTPS 消息中发送的。而是可以将多个子消息组合为单个 RTPS 消息。

上面的逻辑给出了 GAP 子消息包括单个序列号的简化情况。在与 ***Reader*** 无关的更改是紧邻的序号的情况下可能会产生多个 GAP 子消息。高效的实现会将尽可能多的“无关”序列号组合到单个 GAP 消息中。

转换后，将保留以下后置条件：

```
( a_change BELONGS-TO the_reader_proxy.unsent_changes() ) == FALSE
```

2.4.9.2.5 转换场景 T5

此转换由保护条件[RP :: unacked_changes（）== <empty>]触发，该保护条件表示 RTPS ***Writer HistoryCache*** 中的所有更改已由 ***ReaderProxy*** 代表的 RTPS ***Reader*** 确认。

此转换在虚拟机中不执行任何逻辑操作。

2.4.9.2.6 转换场景 T6

此转换由保护条件[RP :: unacked_changes（）！= <empty>]触发，该保护条件表示 RTPS ***Writer HistoryCache*** 中的更改尚未由 ***ReaderProxy*** 代表的 RTPS ***Reader*** 确认。

此转换在虚拟机中不执行任何逻辑操作。

2.4.9.2.7 转换场景 T7

此转换由周期配置为 W :: heartbeatPeriod 的计时器的触发来触发。

转换会对虚拟机中的 ***StatefulWriter***“the_rtps_writer”执行以下逻辑操作：

```
seq_num_min := the_rtps_writer.writer_cache.get_seq_num_min();
seq_num_max := the_rtps_writer.writer_cache.get_seq_num_max();
HEARTBEAT := new HEARTBEAT(the_rtps_writer.writerGuid, seq_num_min,
                                        seq_num_max);
HEARTBEAT.FinalFlag := NOT_SET;
HEARTBEAT.readerId := ENTITYID_UNKNOWN;
send HEARTBEAT;
```


2.4.9.2.8 转换场景 T8

接收到发往 RTPS ***StatefulWriter***“the_rtps_writer”的 ACKNACK 消息触发此转换，该消息源自 ***ReaderProxy***“the_reader_proxy”代表的 RTPS ***Reader***。

转换在虚拟机中执行以下逻辑操作：

```
the_rtps_writer.acked_changes_set(ACKNACK.readerSNState.base - 1);
the_reader_proxy.requested_changes_set(ACKNACK.readerSNState.set);
```

转换后，以下后置条件成立：

```
MIN { change.sequenceNumber IN the_reader_proxy.unacked_changes() } >=
                                        ACKNACK.readerSNState.base - 1
```

2.4.9.2.9 转换场景 T9

此转换由保护条件[RP :: requested_changes（）！= <empty>]触发，该保护条件表示存在由 ***ReaderProxy*** 表示的 RTPS ***Reader*** 请求的更改。

此转换在虚拟机中不执行任何逻辑操作。

2.4.9.2.10 转换场景 T10

通过接收发往 RTPS ***StatefulWriter***“the_writer”的 ACKNACK 消息触发此转换，该消息源自 ***ReaderProxy***“the_reader_proxy”代表的 RTPS ***Reader***。

转换执行与转换 T8（2.4.9.2.8）相同的逻辑动作。

2.4.9.2.11 转换场景 T11

触发计时器触发此转换，该计时器表示自进入必须修复（must_repair）状态以来已经过去 W :: nackResponseDelay 的时间。

此转换在虚拟机中不执行任何逻辑操作。

2.4.9.2.12 转换场景 T12

此转换由保护条件[RP :: can_send（）== true）触发，表明 RTPS ***Writer***“the_rtps_writer”具有将更改发送到由 ***ReaderProxy***“the_reader_proxy 代表的 RTPS ***Reader***”所需的资源。

转换在虚拟机中执行以下逻辑操作：

```
a_change := the_reader_proxy.next_requested_change();
a_change.status := UNDERWAY;
if (a_change.is_relevant)
{
    DATA = new DATA(a_change, the_reader_proxy.remoteReaderGuid);
    IF (the_reader_proxy.expectsInlineQos)
    {
        DATA.inlineQos := the_rtps_writer.related_dds_writer.qos;
    }
    send DATA;
}
else
{
    GAP= new GAP(a_change.sequenceNumber, the_reader_proxy.remoteReaderGuid);
    send GAP;
}
```

以上逻辑并不意味着每个 DATA 或 GAP 子消息都是在单独的 RTPS 消息中发送的。而是可以将多个子消息组合为单个 RTPS 消息。

上面给出了 GAP 子消息包括单个序列号的简化情况。在与 ***Reader*** 无关的更改是紧邻的序号的情况下可能会产生多个 GAP 子消息。高效的实现会将尽可能多的“无关”序列号组合到单个 GAP 消息中。

转换后，以下后置条件成立：

(a_change BELONGS-TO the_reader_proxy.requested_changes()) == FALSE

2.4.9.2.13 转换场景 T13

此转换由保护条件[RP :: requested_changes（）== <empty>]触发，该保护条件表示由 ***ReaderProxy*** 表示的 RTPS ***Reader*** 不再请求任何更改。

此转换在虚拟机中不执行任何逻辑操作。

2.4.9.2.14 转换场景 T14

相应的 DDS DataWriter 向 RTPS ***Writer***“the_rtps_writer”的 ***HistoryCache*** 中添加了新的 ***CacheChange***“a_change”，从而触发了此转换。更改是否与 ***ReaderProxy***“the_reader_proxy”代表的 RTPS ***Reader*** 相关，由 DDS_FILTER 决定。

转换在虚拟机中执行以下逻辑操作：

```
ADD a_change TO the_reader_proxy.changes_for_reader;
IF (DDS_FILTER(the_reader_proxy, change))
    THEN a_change.is_relevant := FALSE;
ELSE
    a_change.is_relevant := TRUE;
IF (the_rtps_writer.pushMode == true)
    THEN a_change.status := UNSENT;
ELSE
    a_change.status := UNACKNOWLEDGED;
```

2.4.9.2.15 转换场景 T15

此转换是由相应的 DDS DataWriter 从 RTPS ***Writer***“the_rtps_writer”的 ***HistoryCache*** 中删除 ***CacheChange***“a_change”触发的。例如，当 HISTORY QoS 设置为 KEEP_LAST 且 depth == 1 时，新的更改将导致 DDS DataWriter 从 ***HistoryCache*** 中删除以前的更改。

转换在虚拟机中执行以下逻辑操作：

```
a_change.is_relevant := FALSE;
```

2.4.9.2.16 转换场景 T16

RTPS ***Writer***“the_rtps_writer” 不再与 ***ReaderProxy***“the_reader_proxy”代表的 RTPS ***Reader*** 匹配的配置触发了此转换。由于打破了 DDS DataReader 与“the_rtps_writer”相关的 DDS DataWriter 的预先存在的匹配关系，因此该配置由发现协议（2.5）完成。

转换在虚拟机中执行以下逻辑操作：

```
the_rtps_writer.matched_reader_remove(the_reader_proxy);
delete the_reader_proxy;
```

2.4.9.3 ChangeForReader 说明

ChangeForReader 跟踪每个 ***CacheChange*** 相对于由相应的 ***ReaderProxy*** 标识的特定远程 RTPS ***Reader*** 的通信状态（属性状态）和相关性（属性 *is_relevant*）。

创建 ***ChangeForReader*** 时，属性 *is_relevant* 初始化为 TRUE 或 FALSE，这取决于 DDS QoS 和可能应用的过滤器。由于后续 ***CacheChange*** 的出现而导致相应的 ***CacheChange*** 与 RTPS ***Reader*** 无关时，属性 *is_relevant* 初始化为 TRUE 的 ***ChangeForReader*** 可能会将其修改为 FALSE。例如当相关 DDS DataWriter 的 DDS QoS 指定 HISTORY 类型为 KEEP_LAST，而后续的 ***CacheChange*** 修改了同一数据对象（由 ***CacheChange*** 的 instanceHandle 属性标识）的值时，就可能发生这种情况，从而使先前的 ***CacheChange*** 不相关。

图 2-20 中描述的 RTPS ***StatefulWriter*** 的行为将每个 ***ChangeForReader*** 修改为协议操作

的副作用。为了进一步定义协议，检查代表任何给定 ChangeForReader 的状态属性值的有限状态机有说明意义。如所示为一个可靠的 ***StatefulWriter***。尽力而为 ***StatefulWriter*** 仅使用状态图的子集。

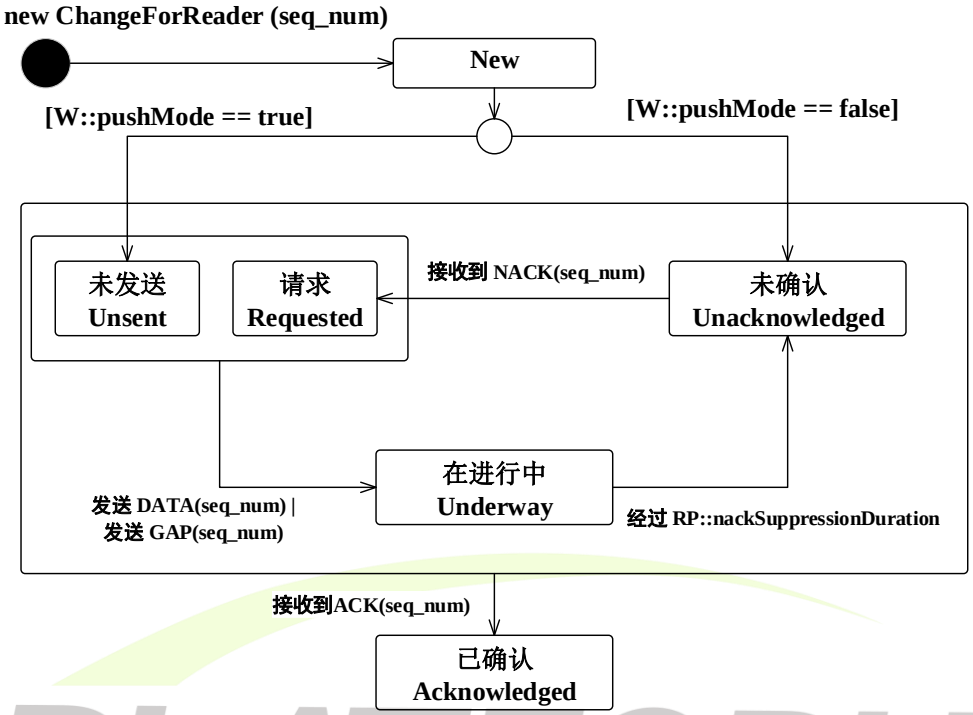


图 2-20 每个 ChangeForReader 的 status 属性值的更改

状态具有以下含义：

•<**New**>具有 ***SequenceNumber_t*** “seq_num”的 ***CacheChange*** 在 RTPS ***StatefulWriter*** 的 ***HistoryCache*** 中可用，但尚未宣布或发送给 ***ReaderProxy*** 代表的 RTPS ***Reader***。

•<**Unsent**> ***StatefulWriter*** 从未将带有此 seq_num 的 DATA 或 GAP 发送给 RTPS ***Reader***，计划在将来发送。

• <**Requested**> RTPS ***Reader*** 已通过 ACKNACK 消息请求再次发送更改。***StatefulWriter*** 计划在将来再次发送更改。

•<**Underway**>已发送 ***CacheChange***，***StatefulWriter*** 将忽略对此 ***CacheChange*** 的新请求。

•<**Unacknowledged**> ***CacheChange*** 应该由 RTPS ***Reader*** 接收，但是 RTPS ***Reader*** 尚未确认。由于消息可能已经丢失，因此 RTPS ***Reader*** 可能会请求再次发送 ***CacheChange***。

• <**Acknowledged**> RTPS ***StatefulWriter*** 知道 RTPS ***Reader*** 已收到带有 ***SequenceNumber_t*** “seq_num” 的 ***CacheChange***。

下面描述了触发状态机中转换的主要事件。请注意，此状态机仅跟踪特定 ***ChangeForReader*** 的 “*status*” 属性，并且不执行任何特定操作或发送任何消息。

•新的 ***ChangeForReader***（seq_num）：***ReaderProxy*** 创建了一个 ***ChangeForReader*** 关联类，以使用 ***SequenceNumber_t*** seq_num 跟踪 ***CacheChange*** 的状态。

•[W :: pushMode == true]：***StatefulWriter*** 的属性 W :: pushMode 的设置确定状态是更改为< **Unsent** >还是更改为< **Unacknowledged** >。尽力而为 ***Writer*** 始终使用 W ::

pushMode == true。

•收到 NACK（seq_num）：***StatefulWriter*** 已收到 ACKNACK 消息，其中 seq_num 属于 ACKNACK.readerSNState，表明 RTPS ***Reader*** 尚未收到 ***CacheChange***，并希望 ***StatefulWriter*** 再次发送它。

•已发送 DATA（seq_num）：***StatefulWriter*** 已发送包含序列号为 ***SequenceNumber_t*** seq_num 的 ***CacheChange*** 的 DATA 消息。

•已发送 GAP（seq_num）：***StatefulWriter*** 已发送一个 GAP 消息，其中 seq_num 位于 GAP 的 irrelevant_sequence_number_list 中，意味着 seq_num 与 RTPS ***Reader*** 无关。

• 收到 ACK（seq_num）：***Writer*** 已收到 ACKNACK，其中 ACKNACK.readerSNState.base > seq_num。 这意味着 RTPS ***Reader*** 已接收到序列号为 seq_num 的 ***CacheChange***。

2.4.10 RTPS Reader 参考实现

RTPS ***Reader*** 参考实现基于 2.2 中首次引入的 RTPS ***Reader*** 类的特殊化。本小节描述了 RTPS ***Reader*** 和用于建模 RTPS ***Reader*** 参考实现的所有其他类。实际行为在 2.4.11 和 2.4.12 中描述。

2.4.10.1 RTPS Reader

RTPS ***Reader*** 专门用于 RTPS ***Endpoint***，代表从一个或多个 RTPS ***Writer*** 端点接收 ***CacheChange*** 消息的参与者。参考实现 ***StatelessReader*** 和 ***StatefulReader*** 专门针对 RTPS ***Reader***，它们在关于匹配 ***Writer*** 端点的维护内容上有所不同。

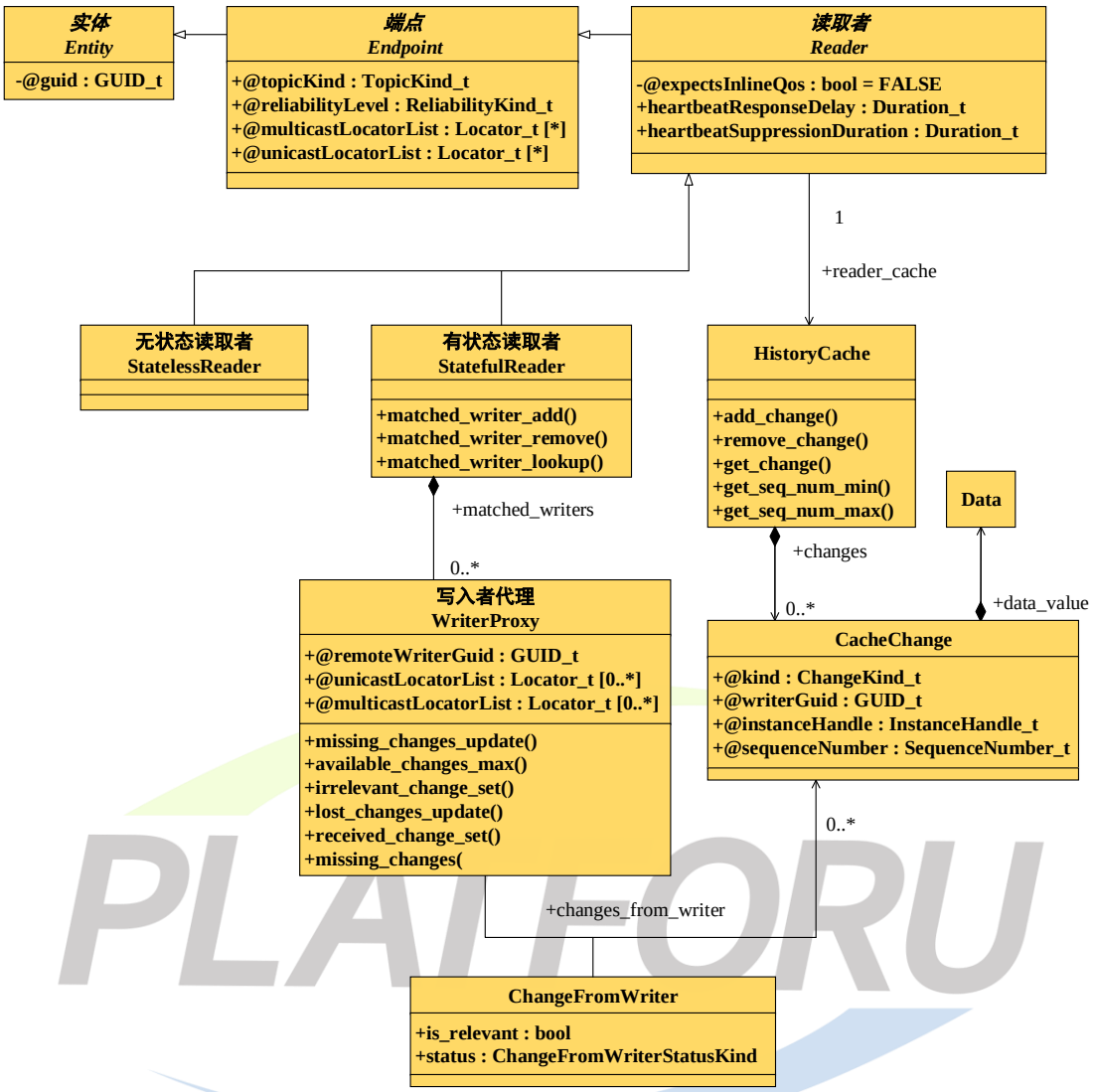


图 2-21 RTPS 读取者端点

表 2-62 列出了 RTPS ***Reader*** 的配置属性，并允许对协议行为进行微调。表 2-63 列出了 RTPS ***Reader*** 的方法。

表 2-62 RTPS Reader 配置属性

RTPS Reader : RTPS Endpoint			
属性	**类型**	**含义**	**与 DDS 关系**
heartbeatResponse Delay	Duration_t	协议调整参数，允许 RTPS ***Reader*** 延迟发送肯定或否定确认（请参阅 2.4.12.2）	N/A
heartbeatSuppression Duration	Duration_t	协议调整参数，允许 RTPS ***Reader*** 忽略收到之前的 HEARTBEAT 之后“过早”到达的 HEARTBEAT	N/A
reader_cache	History Cache	包含此 RTPS ***Reader*** 的 ***CacheChange*** 更改的历史记录。	N/A
expectsInlineQos	bool	指定 RTPS ***Reader*** 是否期望将内置 QoS 与任一数据一起发送。	

表 2-63 RTPS Reader 方法

RTPS Reader 方法		
方法名	*参数列表*	*类型*
new	<return value>	Reader
	attribute_values	Reader 和所有父类所需的属性值集。

以下子小节提供了有关方法的详细信息。

2.4.10.1.1 默认时间相关值

以下与时间相关的值用作默认值，以促进实现之间的“开箱即用”互操作性。

heartbeatResponseDelay.sec = 0;
heartbeatResponseDelay.nanosec = 500 * 1000 * 1000; // 500 毫秒
heartbeatSuppressionDuration.sec = 0;
heartbeatSuppressionDuration.nanosec = 0;

2.4.10.1.2 new

此方法将创建一个新的 RTPS ***Reader***。
新建的读取者“this”的初始化方式如下：

this.guid := <as specified in the constructor>;
this.unicastLocatorList := <as specified in the constructor>;
this.multicastLocatorList := <as specified in the constructor>;
this.reliabilityLevel := <as specified in the constructor>;
this.topicKind := <as specified in the constructor>;
this.expectsInlineQos := <as specified in the constructor>;
this.heartbeatResponseDelay := <as specified in the constructor>;
this.reader_cache := new HistoryCache;

2.4.10.2 RTPS StatelessReader

RTPS ***Reader*** 的特殊化。RTPS ***StatelessReader*** 不了解匹配的写入者 ***Writer*** 的数量，也不为每个匹配的 RTPS ***Writer*** 维护任何状态。

在当前的参考实现中，***StatelessReader*** 不会向从 ***Reader*** 父类继承的配置属性或方法添加任何配置属性或方法。因此，这两个类是相同的。虚拟机使用表 2-64 中的方法与 ***StatelessReader*** 进行交互。

表 2-64 StatelessReader 方法

StatelessReader 方法		
方法名	*参数列表*	*类型*
new	<return value>	StatelessReader
	attribute_values	StatelessReader 和所有父类所需的属性值集。

2.4.10.2.1 new

此方法将创建一个新的 RTPS ***StatelessReader***。初始化是按照 RTPS ***Reader*** 父类（2.4.10.1.2）执行的。

2.4.10.3 RTPS StatefulReader

RTPS ***Reader*** 的特殊化。RTPS ***StatefulReader*** 会为每个匹配的 RTPS ***Writer*** 记录状态。针对每个 ***Writer*** 保留的状态封装在 RTPS ***WriterProxy*** 类中。

表 2-65 RTPS StatefulReader 属性

RTPS StatefulReader : RTPS Reader			
属性	**类型**	**含义**	**与 DDS 关系**
matched_writers	WriteProxy[*]	维护与 ***Reader*** 匹配的远程 ***Writer*** 的状态。	N/A

虚拟机使用表 2-66 中的方法与 ***StatefulReader*** 进行交互。

表 2-66 StatefulReader 方法

StatefulReader 方法		
方法名	*参数列表*	*类型*
new	<return value>	StatefulReader
	attribute_values	StatefulReader 和所有父类所需的属性值集。
matched_writer_add	<return value>	void
	a_writer_proxy	WriterProxy
matched_writer_remove	<return value>	void
	a_writer_proxy	WriterProxy
matched_writer_lookup	<return value>	WriterProxy
	a_writer_guid	GUID_t

2.4.10.3.1 new

此方法将创建一个新的 RTPS ***StatefulReader***。新创建的 ***StatefulReader*** “this”的初始化方式如下：

this.attributes := <as specified in the constructor>;

this.matched_writers := <empty>;

2.4.10.3.2 matched_writer_add

此方法将 ***WriterProxy*** *a_writer_proxy* 添加到 StatefulReader :: matched_writers。

ADD a_writer_proxy TO {this.matched_writers};

2.4.10.3.3 matched_writer_remove

此方法从 StatefulReader :: matched_writers 集合中删除 ***WriterProxy*** *a_writer_proxy*。

```
REMOVE a_writer_proxy FROM {this.matched_writers};
delete a_writer_proxy;
```

2.4.10.3.4 matched_writer_lookup

此方法从 StatefulReader :: matched_writers 集合中找到具有 GUID_t *a_writer_guid* 的 ***WriterProxy***。

```
FIND proxy IN this.matched_writers SUCH-THAT (proxy.remoteWriterGuid ==
                                                a_writer_guid);
return proxy;
```

2.4.10.4 RTPS WriterProxy

RTPS ***WriterProxy*** 表示 RTPS ***StatefulReader*** 为每个匹配的 RTPS ***Writer*** 维护的信息。表 2-67 中描述了 RTPS ***WriterProxy*** 的属性。

关联是 DDS 规范定义的相应 DDS 实体匹配的结果，即 DDS DataReader 按主题匹配 DDS DataWriter，具有兼容的 QoS，属于一个公共分区，并且未被应用程序明确忽略使用 DDS。

表 2-67 RTPS WriterProxy 属性

RTPS WriterProxy			
属性	**类型**	**含义**	**与 DDS 关系**
remoteWriterGuid	GUID_t	标识匹配的 ***Writer***。	N/A。通过发现配置。
unicastLocatorList	Locator_t[*]	可用于将消息发送到匹配的 ***Writer*** 的单播（地址，端口）组合的列表。该列表可能为空。	N/A。通过发现配置。
multicastLocatorList	Locator_t[*]	可用于将消息发送到匹配的 ***Writer*** 的多播（地址，端口）组合的列表。该列表可能为空	N/A。通过发现配置。
changes_from_writer	CacheChange[*]	从匹配的 RTPS ***Writer*** 接收或期望的 ***CacheChange*** 更改列表。	N/A。用于实现 RTPS 协议的行为。

虚拟机使用表 2-68 中的方法与 WriterProxy 进行交互。

表 2-68 WriterProxy 方法

WriterProxy 方法		
方法名	*参数列表*	*类型*
new	<return value>	WriterProxy
	attribute_values	WriterProxy 所需的属性值集。
available_changes_max	<return value>	SequenceNumber_t

irrelevant_change_set	<return value>	void
	a_seq_num	SequenceNumber_t
lost_changes_update	<return value>	void
	first_available_seq_num	SequenceNumber_t
missing_changes	<return value>	SequenceNumber_t[]
missing_changes_update	<return value>	void
	last_available_seq_num	SequenceNumber_t
received_change_set	<return value>	void
	a_seq_num	SequenceNumber_t

2.4.10.4.1 new

此方法将创建一个新的 RTPS ***WriterProxy***。

新建的写入者代理“this”的初始化如下：

this.attributes := <as specified in the constructor>;

this.changes_from_writer := <all past and future samples from the writer>;

新创建的 ***WriterProxy*** 的 *changes_from_writer* 初始化为包含 ***WriterProxy*** 代表的 ***Writer*** 的所有过去和将来的样本。这只是一个概念性表示，用于描述有状态引用实现。*changes_from_writer* 中每个 ***CacheChange*** 的 ***ChangeFromWriter*** 状态被初始化为 UNKNOWN，表明 ***StatefulReader*** 最初不知道这些更改是否已经存在。如 2.4.12.3 所述，当 ***StatefulReader*** 收到实际更改或通过 HEARTBEAT 消息通知其存在时，状态将更改为 RECEIVED 或 MISSING。

2.4.10.4.2 available_changes_max

此方法返回 RTPS ***WriterProxy*** 中的 *changes_from_writer* 更改中的最大 ***SequenceNumber_t***，这些更改可供 DDS DataReader 访问。

使 DDS DataReader 可以对任何 ***CacheChange***“a_change”进行“访问”的条件是，不存在来自于 RTPS ***Writer*** 的 ***SequenceNumber_t*** 小于或等于 a_change.sequenceNumber 且状态为 MISSING 或 UNKNOWN 的更改。换句话说 available_changes_max 和所有先前的更改都为 RECEIVED 或 LOST。

虚拟机中的逻辑操作：

```
seq_num := MAX { change.sequenceNumber SUCH-THAT
                    ( change IN this.changes_from_writer
                        AND ( change.status == RECEIVED
                            OR change.status == LOST) ) };
return seq_num;
```

2.4.10.4.3 irrelevant_change_set

此方法修改 ***ChangeFromWriter*** 的状态，以指明带有 ***SequenceNumber_t***“a_seq_num”的 ***CacheChange*** 与 RTPS ***Reader***.无关。

虚拟机中的逻辑操作：

FIND change FROM this.changes_from_writer SUCH-THAT

(change.sequenceNumber == a_seq_num);
change.status := RECEIVED;
change.is_relevant := FALSE;

2.4.10.4.4 lost_changes_update

对于状态为 MISSING 或 UNKNOWN 且序列号小于“first_available_seq_num”的 ***WriterProxy*** 中的任何更改，此方法都会修改 ***ChangeFromWriter*** 中存储的状态。这些更改的状态被修改为 LOST，表示该更改不再在 RTPS ***WriterProxy*** 代表的 RTPS ***Writer*** 的 ***WriterHistoryCache*** 中可用。

虚拟机中的逻辑操作：

```
FOREACH change IN this.changes_from_writer
        SUCH-THAT ( change.status == UNKNOWN OR change.status == MISSING
            AND seq_num < first_available_seq_num ) DO {
    change.status := LOST;
}
```

2.4.10.4.5 missing_changes

此方法返回状态为“MISSING”的 ***WriterProxy*** 的更改子集。状态为“MISSING”的更改表示 RTPS ***WriterProxy*** 表示的 RTPS ***Writer*** 的 ***HistoryCache*** 中可用的一组更改，这些更改尚未被 RTPS ***Reader*** 接收。

```
return { change IN this.changes_from_writer SUCH-THAT change.status == MISSING };
```

2.4.10.4.6 missing_changes_update

对于状态为 UNKNOWN 且序列号小于或等于“last_available_seq_num”的 ***WriterProxy*** 中的任何更改，此方法都会修改 ***ChangeFromWriter*** 中存储的状态。这些更改的状态从 UNKNOWN 修改为 MISSING，表明该更改在 RTPS ***WriterProxy*** 代表的 RTPS ***Writer*** 的 ***WriterHistoryCache*** 中可用，但尚未被 RTPS ***Reader*** 接收。

虚拟机中的逻辑操作：

```
FOREACH change IN this.changes_from_writer
        SUCH-THAT ( change.status == UNKNOWN
            AND seq_num <= last_available_seq_num ) DO {
    change.status := MISSING;
}
```

2.4.10.4.7 received_change_set

此方法使用 ***SequenceNumber_t*** “a_seq_num”修改代表 ***CacheChange*** 的 ***ChangeFromWriter*** 的状态。更改的状态设置为“RECEIVED”，表明已收到更改。

虚拟机中的逻辑操作：

```
FIND change FROM this.cha;nges_from_writer
    SUCH-THAT change.sequenceNumber == a_seq_num;
change.status := RECEIVED
```

2.4.10.5 RTPS ChangeFromWriter

RTPS ***ChangeFromWriter*** 是一个关联类，它维护与 ***WriterProxy*** 表示的 RTPS ***Writer*** 有关的 RTPS ***Reader HistoryCache*** 中的 ***CacheChange*** 信息。

表 2-69 描述了 RTPS ChangeFromWriter 的属性。

表 2-69 ChangeFromWriter 属性

RTPS ChangeFromWriter			
属性	**类型**	**含义**	**与 DDS 关系**
status	ChangeFromWriterStatusKind	指明与 WriterProxy 表示的 RTPS Writer 有关的 CacheChange 的状态。	N/A。为协议所用。
is_relevant	bool	指明更改是否与 RTPS Reader 有关。	不相关更改的确定受 DDS DataReader TIME_BASED_FILTER QoS 以及 DDS ContentFilteredTopics 的使用影响。

2.4.11 RTPS StatelessReader 行为

2.4.11.1 尽力而为 StatelessReader 行为

“尽力而为带关键字”RTPS ***StatelessReader*** 的行为独立于任何写入者，并在图 2-22 中进行了描述。

“尽力而为不带关键字”RTPS ***StatelessReader*** 的行为是相同的。

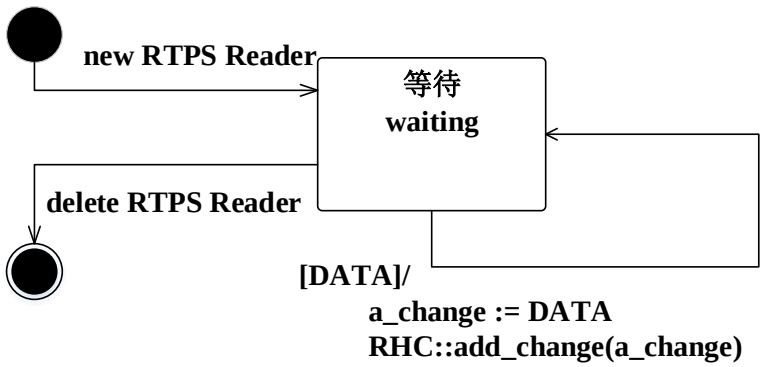


图 2-22 尽力而为带关键字 StatelessReader 的行为

表 2-70 列出了状态机转换场景。

表 2-70 尽力而为 StatelessReader 行为转换场景

转换	状态	事件	下一状态
T1	初始（initial）	RTPS Reader 被创建成功	等待（waiting）
T2	等待（waiting）	收到 DATA 消息	等待（waiting）
T3	等待（waiting）	RTPS Reader 被删除	终止（final）

2.4.11.1.1 转换场景 T1

这种转换是由创建 RTPS ***StatelessReader***“the_rtps_reader”触发的。如 2.2.9 中所述这是创建 DDS DataReader 的结果。

转换在虚拟机中不执行任何逻辑操作。

2.4.11.1.2 转换场景 T2

这种转换是由 RTPS ***Reader***“the_rtps_reader”接收到 DATA 消息触发的。DATA 消息封装了更改“a_change”。封装在 2.3.7.2 中进行了描述。

StatelessReader 的无状态性质使得它无法维护必需的信息，这些信息可以确定到目前为止从始发 RTPS ***Writer*** 所接收的最高序列号。结果是在这些情况下，相应的 DDS DataReader 可能会收到重复或无序的更改。值得注意的是如果 DDS DataReader 配置为按“源时间戳”排序数据，则在通过 DDS DataReader 访问数据时，所有可用数据仍将按顺序显示。

如 2.4.3 所述，实际的无状态实现可以尝试避免这种限制，并以非永久方式（例如使用在一定时间后使信息过期的缓存）来维护此信息，以便在可能的情况下近似有状态行为：如果维持状态，行为将得到结果。

转换在虚拟机中执行以下逻辑操作

```
a_change := new CacheChange(DATA);
the_rtps_reader.reader_cache.add_change(a_change);
```

2.4.11.1.3 转换场景 T3

转换是由 RTPS ***Reader***“the_rtps_reader”的析构触发的。这是 DDS DataReader 析构的结果，如 2.2.9 所述。

转换在虚拟机中不执行任何逻辑操作。

2.4.11.2 可靠 StatelessReader 行为

RTPS 协议不支持此组合。为了实现可靠的协议，RTPS ***Reader*** 必须为每个匹配的 RTPS ***Writer*** 记录一些状态。

2.4.12 RTPS StatefulReader 行为

2.4.12.1 尽力而为 StatefulReader 行为

图 2-23 描述了“尽力而为带关键字”RTPS ***StatefulReader*** 相对于每个匹配的 ***Writer*** 的行为。

“尽力而为不带关键字”RTPS ***StatefulReader*** 的行为是相同的。

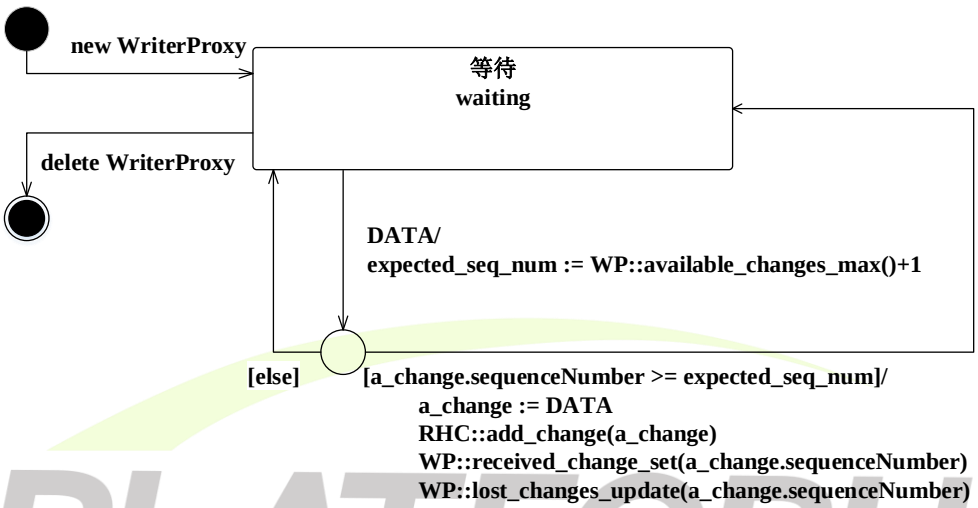


图 2-23 尽力而为带关键字 StatefulReader 相对于每个匹配的 Writer 的行为

表 2-71 列出了状态机转换场景。

表 2-71 尽力而为 StatefulReader 相对于每个匹配的 Writer 的行为的转换

转换	状态	事件	下一状态
T1	初始（initial）	RTPS Reader 配置有匹配的 RTPS Writer	等待（waiting）
T2	等待（waiting）	从匹配的 Writer 收到 DATA 消息	等待（waiting）
T3	等待（waiting）	RTPS Reader 配置为不再与 RTPS Writer 匹配	终止（final）

2.4.12.1.1 转换场景 T1

这种转换是通过为 RTPS ***Reader***“the_rtps_reader”配置一个匹配的 RTPS ***Writer*** 来触发的。由于发现了与“the_rtps_reader”相关的 DDS DataReader 匹配的 DDS DataWriter，因此由发现协议（2.5）完成了此配置。

发现协议提供 ***WriterProxy*** 构造函数参数的值。

转换在虚拟机中执行以下逻辑操作：

```
a_writer_proxy := new WriterProxy(remoteWriterGuid,
                                  unicastLocatorList,
                                  multicastLocatorList);
the_rtps_reader.matched_writer_add(a_writer_proxy);
```

使用 2.4.10.4 中讨论的来自 ***Writer*** 的所有过去和将来的样本初始化 ***WriterProxy***。

2.4.12.1.2 转换场景 T2

这种转换是由 RTPS ***Reader***“the_rtps_reader”接收到 DATA 消息触发的。DATA 消息封装了更改“a_change”。封装在 2.3.7.2 中进行了描述。

尽力而为的读取者会检查与更改相关的序列号是严格大于过去从这个 RTPS ***Writer*** 收到的所有更改的最高序列号（WP:: available_changes_max（））。如果此检查失败，则 RTPS ***Reader*** 将放弃更改。这样可以确保没有重复的更改，也没有乱序的更改。

转换在虚拟机中执行以下逻辑操作：

```
a_change := new CacheChange(DATA);
writer_guid := {Receiver.SourceGuidPrefix, DATA.writerId};
writer_proxy := the_rtps_reader.matched_writer_lookup(writer_guid);
expected_seq_num := writer_proxy.available_changes_max() + 1;
if ( a_change.sequenceNumber >= expected_seq_num )
{
    the_rtps_reader.reader_cache.add_change(a_change);
    writer_proxy.received_change_set(a_change.sequenceNumber);
    if ( a_change.sequenceNumber > expected_seq_num )
    {
        writer_proxy.lost_changes_update(a_change.sequenceNumber);
    }
}
```

转换后，以下后置条件成立：

```
writer_proxy.available_changes_max() >= a_change.sequenceNumber
```


2.4.12.1.3 转换场景 T3

RTPS ***Reader***“the_rtps_reader” 不再与 ***WriterProxy***“the_writer_proxy”代表的 RTPS ***Writer*** 匹配的配置触发了此转换。此配置由发现协议（2.5）完成，因为它打破了 DDS DataWriter 与“the_rtps_reader”相关的 DDS DataReader 之间存在的匹配。

转换在虚拟机中执行以下逻辑操作：

```
the_rtps_reader.matched_writer_remove(the_writer_proxy);
delete the_writer_proxy;
```

2.4.12.2 可靠 StatefulReader 行为

图 2-24 描述了“可靠带关键字”RTPS ***StatefulReader*** 相对于每个匹配的 RTPS ***Writer*** 的行为。“可靠不带关键字”RTPS ***StatefulReader*** 的行为相同。

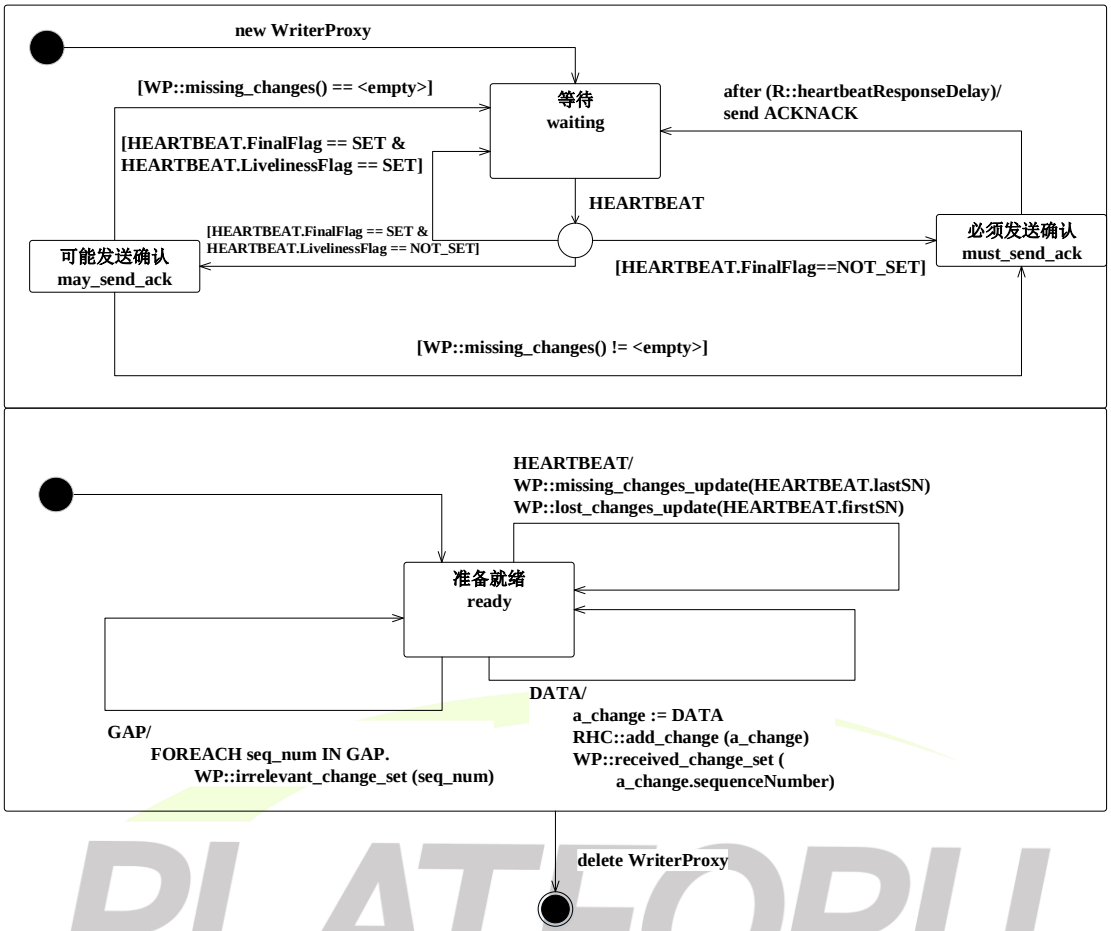


图 2-24 可靠 StatelfulReader 相对于每个匹配的 Writer 的行为

表 2-72 列出了状态机转换条件。

表 2-72 可靠 Reader 相对于每个匹配的 Writer 行为的转换

转换	状态	事件	下一状态
T1	初始 1（initial1）	RTPS Reader 配置有匹配的 RTPS Writer	等待（waiting）
T2	等待（waiting）	收到 HEARTBEAT 消息	如果(HB.FinalFlag == NOT_SET) 必须发送确认（must_send_ack） 否则如果(HB.LivelinessFlag == NOT_SET) 可能发送确认（may_send_ack） 否则 等待（waiting）
T3	可能发送确认（may_send_ack）	GuardCondition：WP::missing_changes()== <empty>	等待（waiting）
T4	可能发送确认（may_send_ack）	GuardCondition: WP::missing_changes() != <empty>	必须发送确认（must_send_ack）
T5	必须发送确认（must_send_ack）	经过 R::heartbeatResponseDelay 时间	等待（waiting）

转换	状态	事件	下一状态
T6	初始 2（initial2）	RTPS Reader 配置有匹配的 RTPS Writer	等待就绪（ready）
T7	等待就绪（ready）	收到 HEARTBEAT 消息	等待就绪（ready）
T8	等待就绪（ready）	收到 DATA 消息	等待就绪（ready）
T9	等待就绪（ready）	收到 GAP 消息	等待就绪（ready）
T10	任一状态	RTPS Reader 配置为不再与 RTPS Writer 匹配	终止（final）

2.4.12.2.1 转换场景 T1

这种转换是通过为可靠 RTPS ***StatefulReader***“the_rtps_reader”配置一个匹配的 RTPS ***Writer*** 来触发的。由于发现了与“the_rtps_reader”代表的 DDS DataReader 匹配的 DDS DataWriter，因此由发现协议（2.5）完成了此配置。

发现协议提供 ***WriterProxy*** 构造函数参数的值。

转换在虚拟机中执行以下逻辑操作：

```
a_writer_proxy := new WriterProxy(remoteWriterGuid,
                                  unicastLocatorList,
                                  multicastLocatorList);
the_rtps_reader.matched_writer_add(a_writer_proxy);
```

使用 2.4.10.4 中讨论的来自 ***Writer*** 的所有过去和将来的样本初始化 ***WriterProxy***。

2.4.12.2.2 转换场景 T2

通过接收到发往 RTPS ***StatefulReader*** “the_reader”的 HEARTBEAT 消息来触发此转换，该消息源自 ***WriterProxy***“the_writer_proxy”代表的 RTPS ***Writer***。

转换在虚拟机中不执行任何逻辑操作。请注意 HEARTBEAT 消息的接收会导致触发转换 T7（2.4.12.2.7），该转换执行逻辑操作。

2.4.12.2.3 转换场景 T3

此转换由保护条件[W :: missing_changes（）== <empty>]触发，该保护条件表示 RTPS ***Reader*** 已接收到由 ***WriterProxy*** 表示的 RTPS ***Writer*** 的 ***HistoryCache*** 中的所有更改。

转换在虚拟机中不执行任何逻辑操作。

2.4.12.2.4 转换场景 T4

此转换由保护条件[W :: missing_changes（）！= <empty>]触发，该保护条件表示 ***WriterProxy*** 表示的 RTPS ***Writer*** 的 ***HistoryCache*** 中存在某些已知更改，但 RTPS ***Reader*** 尚未接收到这些更改。

转换在虚拟机中不执行任何逻辑操作

2.4.12.2.5 转换场景 T5

此转换由计时器超时触发，超时条件为进入状态 must_send_ack 以来已过去 R :: heartbeatResponseDelay 的时间。

该转换对虚拟机中的 ***WriterProxy***“the_writer_proxy”执行以下逻辑操作：

```
missing_seq_num_set.base := the_writer_proxy.available_changes_max() + 1;
missing_seq_num_set.set := <empty>;
FOREACH change IN the_writer_proxy.missing_changes() DO
    ADD change.sequenceNumber TO missing_seq_num_set.set;
send ACKNACK(missing_seq_num_set);
```

上述逻辑动作并未表示以下事实：ACKNACK 消息的 PSM 映射在其包含序号的能力上将受到限制。在 ACKNACK 消息无法容纳丢失数据序列号的完整列表的情况下，应将其构造为包含具有较小序列号值的子集。

2.4.12.2.6 转换场景 T6

与 T1（2.4.12.2.1）相似，此转换通过为可靠 RTPS ***StatefulReader***“the_rtps_reader”配置一个匹配的 RTPS ***Writer*** 来触发。

转换在虚拟机中不执行任何逻辑操作。

2.4.12.2.7 转换场景 T7

通过接收到发往 RTPS ***StatefulReader*** “the_reader”的 HEARTBEAT 消息来触发此转换，该消息源自 ***WriterProxy***“the_writer_proxy”代表的 RTPS ***Writer***。

转换在虚拟机中执行以下逻辑操作：

```
the_writer_proxy.missing_changes_update(HEARTBEAT.lastSN);
the_writer_proxy.lost_changes_update(HEARTBEAT.firstSN);
```

2.4.12.2.8 转换场景 T8

通过接收到发往 RTPS ***StatefulReader***“the_reader”的 DATA 消息触发此转换，该消息源自 ***WriterProxy***“the_writer_proxy”代表的 RTPS ***Writer***。

转换在虚拟机中执行以下逻辑操作：

```
a_change := new CacheChange(DATA);
the_reader.reader_cache.add_change(a_change);
the_writer_proxy.received_change_set(a_change.sequenceNumber);
```

如 2.2.9 所述，使用 DDS DataReader 读取或获取方法访问数据时，任一过滤都可能执行。

2.4.12.2.9 转换场景 T9

通过接收到发往 RTPS ***StatefulReader***“the_reader”的 GAP 消息触发此转换，该消息源自 ***WriterProxy***“the_writer_proxy”代表的 RTPS ***Writer***。

转换在虚拟机中执行以下逻辑操作：

```
FOREACH seq_num IN [GAP.gapStart, GAP.gapList.base-1] DO
{
    the_writer_proxy.irrelevant_change_set(seq_num);
}
FOREACH seq_num IN GAP.gapList DO
{
    the_writer_proxy.irrelevant_change_set(seq_num);
}
```

2.4.12.2.10 转换场景 T10

RTPS ***Reader***“the_rtps_reader” 不再与 ***WriterProxy***“the_writer_proxy”代表的 RTPS ***Writer*** 匹配的配置触发了此转换。此配置由发现协议（2.5）完成，因为它打破了 DDS DataWriter 与“the_rtps_reader”相关的 DDS DataReader 之间存在的匹配。

转换在虚拟机中执行以下逻辑操作：

```
the_rtps_reader.matched_writer_remove(the_writer_proxy);
delete the_writer_proxy;
```

2.4.12.3 ChangeFromWriter 说明

ChangeFromWriter 跟踪每个 ***CacheChange*** 相对于特定远程 RTPS ***Writer*** 的通信状态（属性 *status*）和相关性（属性 *is_relevant*）。

图 2-25 中描述的 RTPS ***StatefulReader*** 的行为将每个 ***ChangeFromWriter*** 修改为协议操作的副作用。为了进一步定义协议，检查代表任何给定 ***ChangeFromWriter*** 的 *status* 属性值的状态机有说明意义。图 2-25 显示了一个可靠的 ***StatefulReader***。尽力而为 ***StatefulReader*** 仅使用状态图的子集。

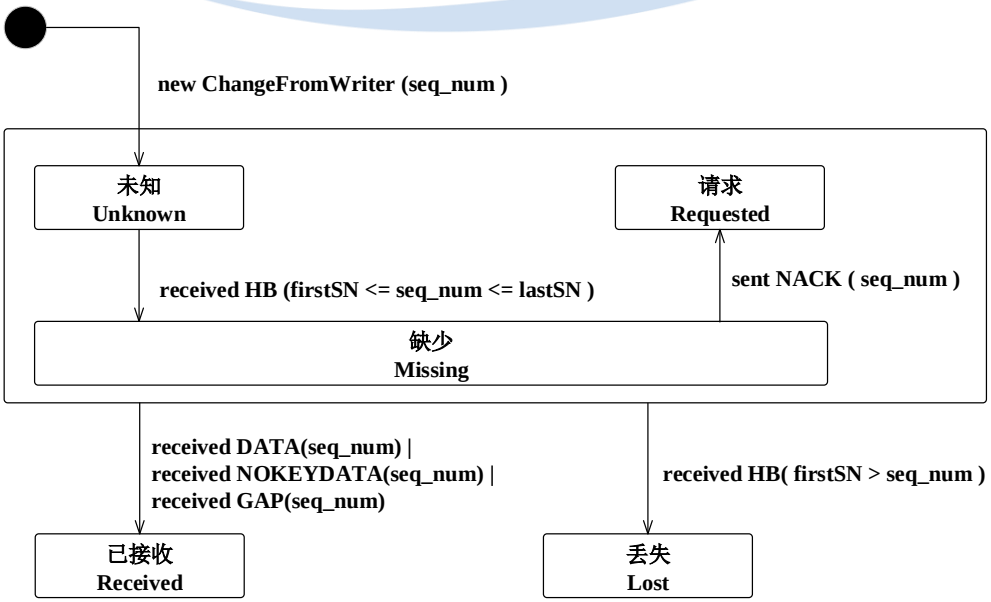


图 2-25 每个 ChangeFromWriter 的 status 属性值的变化

状态具有以下含义：

•**<Unknown>**： 具有 ***SequenceNumber_t*** seq_num 的 ***CacheChange*** 在 RTPS ***Writer*** 上可能可用或可能不可用。

•**<Missing>**： 具有 ***SequenceNumber_t*** seq_num 的 ***CacheChange*** 在 RTPS ***Writer*** 上可用，而 RTPS ***Reader*** 尚未收到。

• **<Requested>**： 从 RTPS ***Writer*** 请求具有 ***SequenceNumber_t*** seq_num 的 ***CacheChange***，响应可能处于挂起状态或正在进行中。

•**<Received>**：已接收到具有 ***SequenceNumber_t*** seq_num 的 ***CacheChange***：如果 seq_num 与 RTPS ***Reader*** 相关，则作为 DATA；如果 seq_num 不相关，则作为 GAP。

•**<Lost>**： 具有 ***SequenceNumber_t*** seq_num 的 ***CacheChange*** 在 RTPS ***Writer*** 上不再可用。它无法被接收。

下面描述了触发状态机中转换的主要事件。请注意，此状态机仅跟踪特定 ***ChangeForReader*** 的“*status*”属性，并且不执行任何特定操作或发送任何消息。

•新建 ChangeFromWriter（seq_num）：***WriterProxy*** 创建了一个 ***ChangeFromWriter*** 关联类，以跟踪具有 ***SequenceNumber_t*** seq_num 的 ***CacheChange*** 的状态。

• 收到 HB（ firstSN <= seq_num <= lastSN ）： ***Reader*** 已接收到具有 HEARTBEAT.firstSN <= seq_num <= HEARTBEAT.lastSN 的 HEARTBEAT，表示可从 RTPS ***Writer*** 获得具有该序列号的 ***CacheChange***。

•发送 NACK（seq_num）：***Reader*** 已在 ACKNACK.readerSNState 内部发送了一个包含 seq_num 的 ACKNACK 消息，指明 RTPS ***Reader*** 尚未收到 ***CacheChange*** 并正在请求再次发送。

•收到 GAP（seq_num）：***Reader*** 已收到一条 GAP 消息，其中 seq_num 位于 GAP.gapList 内，这意味着 seq_num 与 RTPS ***Reader*** 无关。

•收到 DATA（seq_num）：***Reader*** 已收到一条 DATA 消息，其 DATA.sequenceNumber == seq_num。

•收到 HB（firstSN> seq_num）：***Reader*** 已接收到具有 HEARTBEAT.firstSN> seq_num 的 HEARTBEAT，表明具有该序列号的 ***CacheChange*** 在 RTPS ***Writer*** 上不再可用。

2.4.13 Writer 存活性协议

DDS 规范要求存在存活检测机制。RTPS 通过 ***Writer*** 存活协议实现了这一要求。《***Writer*** 存活协议》定义了两个参与者之间必需的信息交换，以便维护参与者所包含的 ***Writer*** 的存活性。

所有实现都必须支持 ***Wirter*** 存活性协议，以实现互操作。

2.4.13.1 一般方法

Writer 存活性协议使用预定义的内置端点。使用内置端点意味着一旦一个参与者知道另一个参与者的存在，它就可以假设存在由远程参与者提供的内置端点，并与本地匹配的内置端点建立关联。

内置端点之间用于通信的协议与应用程序定义的端点相同。

2.4.13.2 Writer 存活性协议所需的内置端点

Writer 存活性协议所需的内置端点是 ***BuiltinParticipantMessageWriter*** 和 ***BuiltinParticipantMessageReader***。这些端点的名称反映了它们是多用途的事实。这些端点用于实现存活性协议，将来也可用于传递其他数据。

RTPS 协议为这些内置端点保留 ***EntityId_t*** 的以下值

ENTITYID_P2P_BUILTIN_PARTICIPANT_MESSAGE_WRITER

ENTITYID_P2P_BUILTIN_PARTICIPANT_MESSAGE_READER

每个 ***EntityId_t*** 实例的实际值由每个 PSM 定义。

2.4.13.3 BuiltinParticipantMessageWriter 和 BuiltinParticipantMessageReader QoS

为了实现互操作性，BuiltinParticipantMessageWriter 和 BuiltinParticipantMessageReader 都使用以下 QoS 值：

- reliability.kind = RELIABLE_RELIABILITY_QOS
- durability.kind = TRANSIENT_LOCAL_DURABILITY
- history.kind = KEEP_LAST_HISTORY_QOS
- history.depth = 1

2.4.13.4 Writer 存活性协议使用的与内置端点关联的数据类型

每个 RTPS 端点都有一个 ***HistoryCache***，用于存储与端点关联的数据对象的更改。对于 RTPS 内置端点也是如此。因此，每个 RTPS 内置端点都依赖于某个 DataType，该 DataType 表示写入其 ***HistoryCache*** 的数据的逻辑内容。

图 2-26 定义了与主题为 DCPSParticipantMessage 的 RTPS 内置端点关联的 *ParticipantMessageData* 数据类型。

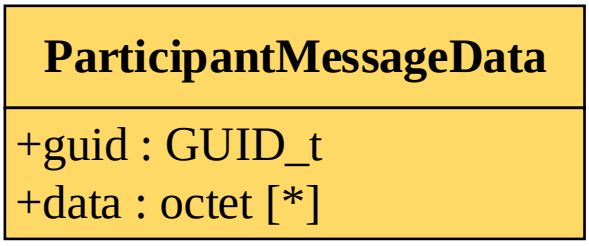


图 2-26 ParticipantMessageData

2.4.13.5 使用 BuiltinParticipantMessageWriter 和 BuiltinParticipantMessageReader 实现 Writer 存活性协议

通过将样本写入 BuiltinParticipantMessageWriter，可以断言属于参与者的一部分 ***Writer***

的存活性。如果参与者包含一个或多个存活性设置为 AUTOMATIC_LIVELINESS_QOS 的 ***Writer***，则以比共享此 QoS 的 ***Writer*** 中最小租约期限更快的速率写入一个样本。同样，如果参与者包含一个或多个存活性设置为 MANUAL_BY_PARTICIPANT_LIVELINESS_QOS 的 ***Writer***，并且以比这些 ***Writer*** 中最小的租约期限更快的速率写入一个单独的样本。这两个场景的目的是正交的，因此，如果参与者包含所描述的两种类型存活性的 ***Writer***，则必须定期写入两个独立的实例。实例使用 DDS 关键字进行区分，该 DDS 关键字由 *participantGuidPrefix* 和 *kind* 字段组成。通过此协议处理的两种类型的实时 QoS 中的每一种都有唯一的 *kind* 字段，因此在 ***HistoryCache*** 中形成两个不同的实例。

在这两种情况下，*participantGuidPrefix* 字段都包含正在写入数据的参与者的 GuidPrefix_t（并因此声明其 ***Writer*** 的存活性）。

没有使用 BuiltinParticipantMessageWriter 和 BuiltinParticipantMessageReader 来实现 DDS 存活性类型 MANUAL_BY_TOPIC_LIVELINESS_QOS。在 2.7.2.2.3 中讨论该类型。

2.4.14 可选行为

本小节描述了 RTPS 协议的可选功能。并非所有 RTPS 实现都支持可选功能。可选功能不会影响基本的互操作性，但仅在所有涉及的实现都支持它时才可用。

2.4.14.1 大数据

如 1.6 中所述，RTPS 对底层传输方式的要求很少。传输方式提供能够尽力而为发送数据包的无连接服务就足够了。

也就是说，传输方式可能会施加自己的限制。例如它可能会限制最大数据包大小（例如对于 UDP 为 64K），从而会限制最大 RTPS 子消息长度。这主要影响数据子消息，因为它限制了 *serializedData* 的最大长度或使用的数据类型的最大序列化长度。

为了解决此限制，2.3.7 引入了以下子消息以实现对大数据的分片。

- DataFrag
- HeartbeatFrag
- NackFrag

以下子小节列出了互操作性所需的相应行为。

2.4.14.1.1 如何选择分片大小

分片大小由 ***Writer*** 确定，并且必须满足以下要求：

•***Writer*** 可用的所有传输方式都必须能够容纳至少包含一个分片的 **DataFrag** 子消息。这意味着拥有最小的最大消息长度的传输方式将确定分片大小。

•对于给定的 ***Writer***，分片大小必须是固定的，并且对于所有远端 ***Reader*** 是相同的。通过固定分片大小，分片编号代表的数据不依赖于特定的远程 ***Reader***。这简化了处理来自 ***Reader*** 的否定确认（NackFrag）。

•分片大小必须满足 1KB <分片大小<64 KB。

请注意，分片大小是由 ***Writer*** 可用的所有传输方式决定的，而不仅仅是可达所有当前已知的 ***Reader*** 所需传输方式的子集。这确保了新发现的 ***Reader***（无论通过何种传输方式可达），都可以适应，而不必更改分片大小，如果改变分片大小则会违反上述要求。

2.4.14.1.2 如何发送分片

如果需要分片，则将 Data 子消息替换为一系列 DataFrag 子消息。发送 DataFrag 子消息的协议行为与发送常规 Data 子消息的协议行为匹配，但具有以下附加要求：

•DataFrag 子消息按顺序发送，其中顺序通过分片编号的升序来定义。注意这不能保证按序到达。

•仅在需要时才对数据进行分片。如果 ***Writer*** 可以使用多种传输方式，而某些传输方式不需要分段，则必须在这些传输方式上发送常规的 Data 子消息。同样，对于可变大小的数据类型，如果特定序列号不需要分段，则必须使用常规的 Data 子消息。

•对于给定的序列号，如果使用内联 QoS 参数，则必须将其包含在第一个 DataFrag 子消息中（分片号等于 1 的分片）。它们也可能包含在此序列号的后续 DataFrag 子消息中，但这不是必需的。

如果传输方式可以容纳给定分片大小的多个片段，则建议实现将尽可能多的分片封装到单个 DataFrag 消息中。

发送多个 DataFrag 消息时，可能需要进行流量控制以避免网络泛洪。可能的方法包括漏桶或令牌桶流量控制方案。这不是 RTPS 规范的一部分。

2.4.14.1.3 如何重新组装分片

DataFrag 子消息包含重新组装序列化数据所需的所有信息。收到所有分片后将应用与常规 Data 子消息相同的协议行为。

请注意，实现必须能够处理 DataFrag 子消息的无序到达。

2.4.14.1.4 可靠通信

可靠发送 DataFrag 子消息的协议行为与发送常规 Data 子消息的协议行为相匹配，但具有以下附加要求：

•Heartbeat 子消息的语义保持不变：“心跳”消息必须仅包括所有可用分片的序列号。

•AckNack 子消息的语义保持不变：当收到该序列号的所有分片时，AckNack 消息只能肯定地确认该序列号。同样，仅当缺少所有分片时，才必须否定确认该序列号。

•为了否定地确认给定序列号的分片子集，必须使用 NackFrag 子消息。数据分片后，Heartbeat 子消息可能会同时触发 AckNack 和 NackFrag 子消息。

其他注意事项：

•如上所述，一旦该序列号的所有分片均可用，Heartbeat 子消息只能包含序列号。如果 ***Writer*** 希望将给定序列号的分片的部分可用性通知 ***Reader***，则可以改用 HeartbeatFrag 子消息。分片级别的可靠性对于非常大的数据传输以及使用流控制可能会有所帮助。

•NackFrag 子消息只能在响应 Heartbeat 或 HeartbeatFrag 子消息时发送。

2.4.15 实现指南

本小节的内容不是协议正式规范的一部分。本小节的目的是为协议的高性能实现提供指

导。

2.4.15.1 ReaderProxy 和 WriterProxy 的实现

PIM 将 ***ReaderProxy*** 建模为维护与 ***Writer*** 的 ***HistoryCache*** 中的每个 ***CacheChange*** 的关联。该关联被建模为由关联类 ***ChangeForReader*** 协调。此模型的直接实现将导致为每个 ***ReaderProxy*** 维护大量信息。实际上需要的是 ***ReaderProxy*** 能够实现协议使用的方法，并且不需要使用显式关联。

例如可以通过让 ***ReaderProxy*** 维护单个序列号“*highestSeqNumSent*”来实现 **unsent_changes()** 和 **next_unsent_change()** 方法。*highestSeqNumSent* 将记录发送给 ***ReaderProxy*** 的任一 ***CacheChange*** 的序列号的最大值。使用此方法，可以通过在 ***HistoryCache*** 中查找所有更改并选择 *sequenceNumber* 大于 *highestSeqNumSent* 的更改来实现 **unsent_changes()**方法。**next_unsent_change()**的实现也将查找 ***HistoryCache*** 并返回其下一个最高序列号大于 *highestSeqNumSent* 的 ***CacheChange***。如果 ***HistoryCache*** 通过 *sequenceNumber* 维护索引，则可以有效地完成这些方法。

可以使用相同的技术来实现 **requested_changes()**， **requested_changes_set**() 和 **next_requested_change**()。在这种情况下，实现可以维护序列号的滑动窗口（可以有效地由 ***SequenceNumber_t*** *lowestRequestedChange* 和固定长度的位图表示），以存储当前是否请求特定的序列号。窗口中不适合的请求可以被忽略，因为它们对应的序列号高于窗口中的序列号，并且如果仍然缺少更改，可以依靠 ***Reader*** 稍后重新发送请求。

可以使用类似的技术来实现 **acked_changes_set**()和 **unacked_changes**()。

2.4.15.2 有效利用 Gap 和 AckNack 子消息

Gap 和 **AckNack** 子消息都经过设计，以便它们可以包含有关一组序列号的信息。为简单起见，协议描述中使用的虚拟机并不总是尝试完全使用这些子消息来存储它们将应用的所有序列号。当一种更有效的实现将这些子消息组合为一个消息时，有时会发送多个 **Gap** 或 **AckNack** 消息。所有这些实现都符合协议并且可以互操作。结合了多个 **Gap** 和 **AckNack** 子消息并利用这些子消息包含一组序列号的功能的实现将在带宽和 CPU 使用率方面更加有效。

2.4.15.3 合并多个数据子消息

RTPS 协议允许将多个子消息合并为单个 RTPS 消息。这意味着它们将共享一个 RTPS 报文头，并通过一个“网络传输事务”发送。大多数网络传输的固定开销都相对较大，而消息中额外的字节会产生额外的开销。因此将子消息组合为单个 RTPS 消息的实现通常可以更好地利用 CPU 和带宽。

一个特别常见的情况是将多个 Data 子消息合并为一个 RTPS 消息。对 AckNack 请求多个更改的响应中可能会出现此需求，或者写入者还存在尚未传输到读取者的多个更改也会导致此情况出现。在所有这些情况下，将子消息合并为更少的 RTPS 消息通常是有益的。

请注意 Data 子消息的合并不限于源自同一 RTPS ***Writer*** 的子消息。也可以合并源自多个 RTPS ***Writer*** 实体的子消息。属于同一 DDS Publisher 的 DDS DataWriter 实体相对应的 RTPS ***Writer*** 实体是这种场景的主要候选对象。

2.4.15.4 携带 HeartBeat 子消息

RTPS 协议允许将不同类型的子消息合并为单个 RTPS 消息。一个特别有用的情况是在数据子消息之后携带 HeartBeat 子消息。这允许 RTPS ***Writer*** 显式请求对其发送的更改的确认，而无需发送单独的 HeartBeat 报文，增加额外消耗。

2.4.15.5 发送给未知的 readerId

如消息模块中所述，可以在未指定 readerId 的情况下发送 RTPS 消息（ENTITYID_UNKNOWN）。通过多播发送这些消息时这是必需的，但也允许通过单播发送单个消息，以到达同一参与者内的多个 ***Reader***。鼓励 DDS 实现使用此功能以达到最小化带宽使用。

2.4.15.6 从无反应的读取者那里回收有限的资源

一个实现可能有有限的资源可以使用。对于 ***Writer***，当所有 ***Reader*** 都已确认收到队列中的某个样本或者资源限制指示将旧样本条目用于新样本时，应该回收队列资源。

在某些情况下，活动的 ***Reader*** 将变得无响应，并且永远不会给 ***Writer*** 确认。不应阻塞没有反应的 ***Reader***，而应允许 ***Writer*** 将 ***Reader*** 视为“非活动”并继续更新其队列。***Reader*** 的状态为活动或非活动。活动的 ***Reader*** 已发送了最近收到数据的 ACKNACK。***Writer*** 应使用一种机制来确定 ***Reader*** 的不活动状态，该机制基于接收 ACKNACK 报文的速率和数量。***Writer*** 可以释放所有活动 ***Reader*** 已确认的样本，并且可以在必要时回收这些资源。 请注意当 ***Reader*** 变为非活动状态时，不能保证严格的可靠性。

2.4.15.7 设置心跳和 ACKNACK 的计数

HEARTBEAT 的 Count 元素区分逻辑 HEARTBEAT。可以忽略与先前接收到的 HEARTBEAT Count 相同的已接收到的 HEARTBEAT，以防止触发重复的修复会话。 因此实现应确保使用相同的 Count 标记相同的逻辑 HEARTBEAT。

新的 HEARTBEATS 的计数应大于所有旧的 HEARTBEATs。可以忽略接收到的计数不大于任何先前接收到的 HEARTBEAT 计数的 HEARTBEAT。

相同的逻辑适用于 ACKNACK 计数。

2.5 发现模块

RTPS 行为模块假定 RTPS 端点已正确配置，并与匹配的远程端点配对。它不对如何进行此配置进行任何假设，仅定义如何在这些端点之间交换数据。

为了能够配置端点，实现必须获取有关远程端点的存在及其属性的信息。如何获取此信息是发现模块的主题。

发现模块定义了 RTPS 发现协议。发现协议的目的是允许每个 RTPS 参与者发现其他相

关参与者及其端点。一旦发现远程端点，实现可以相应地配置本地端点以建立通信。

DDS 规范同样依赖于发现机制的使用，以在匹配的 DataWriter 和 DataReader 之间建立通信。 DDS 实现必须在加入和离开网络时自动发现远程实体的存在。通过 DDS 内置主题用户可以访问此发现信息。

本模块中定义的 RTPS 发现协议为 DDS 提供了必需的发现机制。

2.5.1 概述

RTPS 规范将发现协议分为两个独立的协议：

1.参与者（***Participant***）发现协议

2.端点（***Endpoint***）发现协议

参与者发现协议（PDP）指定参与者如何在网络中彼此发现。一旦两个参与者发现彼此，他们就会使用端点发现协议（EDP）在其包含的端点上交换信息。除了这种因果关系之外，两种协议可以视为相互独立。

实现可以选择支持多个 PDP 和 EDP，可能是实现者专属的。只要两个参与者至少具有一个共同的 PDP 和 EDP，他们就可以交换所需的发现信息。为了保证互操作性，所有 RTPS 实现都必须至少提供以下发现协议：

1.简单参与者发现协议（SPDP）

2.简单端点发现协议（SEDP）

两者都是基本的发现协议，适用于中小型网络。面向更大网络的其他 PDP 和 EDP 可能会添加到本规范的未来版本中。

最后，发现协议的作用是提供有关发现的远程端点的信息。参与者如何使用此信息来配置其本地端点取决于 RTPS 协议的实际实现，而不是发现协议规范的一部分。例如，对于 2.4.7 中引入的参考实现，获得的远程端点上的信息允许实现以下配置：

- 与每个 RTPS ***StatelessWriter*** 关联的 RTPS ***ReaderLocator*** 对象。
- 与每个 RTPS ***StatefulWriter*** 关联的 RTPS ***ReaderProxy*** 对象。
- 与每个 RTPS ***StatefulReader*** 关联的 RTPS ***WriterProxy*** 对象。

发现模块的组织方式如下：

- SPDP 和 SEDP 依赖于预定义的 RTPS 内置 Writer 和 Reader 端点来交换发现信息。

2.5.2 介绍了这些 RTPS 内置端点。

- 在 2.5.3 中讨论了 SPDP。
- 在 2.5.4 中讨论了 SEDP。

2.5.2 RTPS 内置发现端点

DDS 规范指定使用带有预定义主题和 QoS 的“内置” DDS DataReader 和 DataWriter 进行发现。

有四个预定义的内置主题：“DCPSParticipant”，“DCPSSubscription”，“DCPSPublication”和 “DCPSTopic”。与这些主题相关的数据类型也由 DDS 规范指定，主要包含实体 QoS 值。

对于每个内置主题，都有一个对应的 DDS 内置 DataWriter 和 DDS 内置 DataReader。内置 DataWriter 用于向网络其余节点通告本地 DDS 参与者和其包含的 DDS 实体（DataReaders，DataWriters 和主题）的存在和 QoS。同样，内置 DataReader 从远程参与者收集此信息，然

后 DDS 实现将其用于标识匹配的远程实体。内置 DataReader 可以充当常规 DDS DataReader，并且用户也可以通过 DDS API 对其进行访问。

RTPS 简单发现协议（SPDP 和 SEDP）采用的方法类似于内置实体概念。RTPS 将每个内置 DDS DataWriter 或 DataReader 映射到关联的内置 RTPS 端点。这些内置端点充当常规的 Writer 和 Reader 端点，并提供了使用行为模块中定义的常规 RTPS 协议在参与者之间交换所需发现信息的方法。

SPDP 与参与者如何发现彼此有关，它为“DCPSParticipant”主题映射了 DDS 内置实体。SEDP 指定了如何交换有关本地主题，DataWriters 和 DataReader 的发现信息，它为“DCPSSubscription”，“DCPSPublication”和“DCPSTopic”主题映射了 DDS 内置实体。

2.5.3 简单参与者发现协议（SPDP）

PDP 的目的是发现网络上其他参与者的存在及其属性。

参与者可以支持多个 PDP，但是出于互操作的目的，所有实现都必须至少支持简单参与者发现协议。

2.5.3.1 一般方法

RTPS 简单参与者发现协议（SPDP）使用一种简单方法来宣布和检测域中参与者的存在。

对于每个参与者，SPDP 将创建两个 RTPS 内置端点：*SPDPbuiltinParticipantWriter* 和 *SPDPbuiltinParticipantReader*。

SPDPbuiltinParticipantWriter 是一个 RTPS 尽力而为的 ***StatelessWriter***。*SPDPbuiltinParticipantWriter* 的 ***HistoryCache*** 包含一个类型为 ***SPDPdiscoveredParticipantData*** 的数据对象。该数据对象的值是从参与者的属性中设置的。如果属性发生了更改，则替换数据对象的值。

SPDPbuiltinParticipantWriter 会定期将此数据对象发送到预先配置的定位器列表，以宣布参与者在网络上的存在。这可以通过定期调用 StatelessWriter :: unsent_changes_reset 来实现，此方法将导致 ***StatelessWriter*** 将其 ***HistoryCache*** 中存在的所有更改重新发送给所有定位器。 *SPDPbuiltinParticipantWriter* 发出 ***SPDPdiscoveredParticipantData*** 的周期速率默认为 PSM 指定的值。此时间段应小于 ***SPDPdiscoveredParticipantData*** 中指定的 *leaseDuration*（参见 2.5.3.3.2）。

预先配置的定位器列表可以包括单播和多播定位器。端口号由每个 PSM 定义。这些定位器仅代表网络中可能的远程参与者，实际上不需要任何参与者在线。通过定期发送 ***SPDPdiscoveredParticipantData***，参与者可以以任何顺序加入网络。

SPDPbuiltinParticipantReader 从远程参与者接收 ***SPDPdiscoveredParticipantData*** 公告。所包含的信息包括远程参与者支持哪些端点发现协议。然后使用合适的端点发现协议与远程参与者交换端点信息。

通过响应先前未知的参与者的数据对象而发送额外的 SPDPdiscoveredParticipantData，实现可以将任一启动延迟最小化，但是此行为是可选的。实现方式还可以使用户选择是否使用来自新发现的参与者的新定位器自动扩展定位器的预配置列表。这将启用非对称定位器列表。最后两个功能是可选的，出于互操作的目的不是必需的。

2.5.3.2 SPDPdiscoveredParticipantData

SPDPdiscoveredParticipantData 定义 SPDP 部分交换的数据。

图 2-27 说明了 ***SPDPdiscoveredParticipantDat*** 的内容。如图所示，***SPDPdiscoveredParticipantData*** 专用于 ***ParticipantProxy***，包括配置发现的 ***Participant*** 所需的所有信息。***SPDPdiscoveredParticipantData*** 还专用于 DDS 定义的 DDS :: ParticipantBuiltinTopicData，提供相应的 DDS 内置 DataReader 所需的信息。

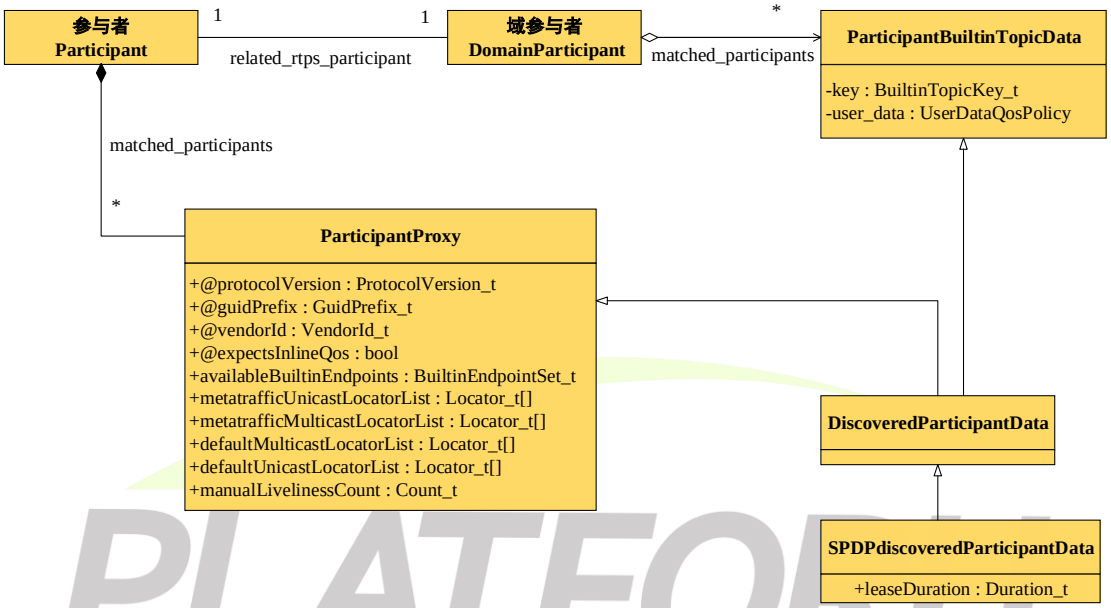


图 2-27 SPDPdiscoveredParticipantData

表 2-73 中描述了 ***SPDPdiscoveredParticipantData*** 的属性及其解释说明。

表 2-73 RTPS SPDPdiscoveredParticipantData 属性

RTPS SPDPdiscoveredParticipantData		
属性	**类型**	**含义**
protocolVersion	ProtocolVersion_t	标识参与者使用的 RTPS 协议版本。
guidPrefix	GuidPrefix_t	参与者及其包含的所有端点的通用 GuidPrefix_t。
vendorId	VendorId_t	标识包含参与者的 DDS 中间件的供应商。
expectsInlineQos	bool	声明参与者内的读取者是否期望将应用于每个数据修改的 QoS 值与数据一起封装
metatrafficUnicastLocatorList	Locator_t[*]	可用于将消息发送到参与者包含的内置端点的单播定位器列表（传输方式，地址，端口组合）
metatrafficMulticastLocatorList	Locator_t[*]	可用于将消息发送到参与者包含的内置端点的多播定位器列表（传输方式，地址，端口组合）

RTPS SPDPdiscoveredParticipantData		
属性	**类型**	**含义**
defaultUnicastLocatorList	Locator_t[1..*]	可用于将消息发送到参与所包含的用户定义的端点的单播定位器的默认列表（传输方式，地址，端口组合）。这些是单播定位器，用于在端点未指定自己的集合时使用定位器，因此必须至少存在一个定位器。
defaultMulticastLocatorList	Locator_t[*]	可用于将消息发送到参与者包含的用户定义的端点的多播定位器的默认列表（传输方式，地址，端口组合），如果端点未指定其自己的集合，将使用这些多播定位器定位器。
availableBuiltinEndpoints	BuiltinEndpointSet_t	所有参与者都必须支持 SEDP。此属性标识参与者中可用的内置 SEDP 端点的类型，使参与者可以指示它仅包含可能的内置端点的子集。另请参见 2.5.4.3 。 BuiltinEndpointSet_t 的可能值为： PUBLICATIONS_READER， PUBLICATIONS_WRITER， SUBSCRIPTIONS_READER， SUBSCRIPTIONS_WRITER， TOPIC_READER ， TOPIC_WRITER 供应商特定的扩展名可以用来表示对其他 EDP 的支持。
leaseDuration	Duration_t	每次从参与者收到公告后，该参与者被视为存活的时间。如果参与者在此时间段内未发送另一则公告，则可以认为该参与者已掉线。在这种情况下，与参与者及其端点相关的任何资源都可以被释放。
manualLivelinessCount	Count_t	用于实现 MANUAL_BY_PARTICIPANT 存活性 QoS。 断言存活时，将增加 manualLivelinessCount 并发送新的 SPDPdiscoveredParticipantData。

如 2.5.3.1 中所述，SPDPdiscoveredParticipantData 列出了参与者支持的端点发现协议。表 2-73 中显示的属性仅反映了强制性的 SEDP 所需要的内容。RTPS 规范当前没有定义其他端点发现协议。要包含其他 EDP，可以使用标准的 RTPS 扩展机制。有关其他信息，请参

阅 3.6.2。

2.5.3.3 简单参与者发现协议（SPDP）使用的内置端点

图 2-28 说明了简单参与者发现协议（SPDP）引入的内置端点。

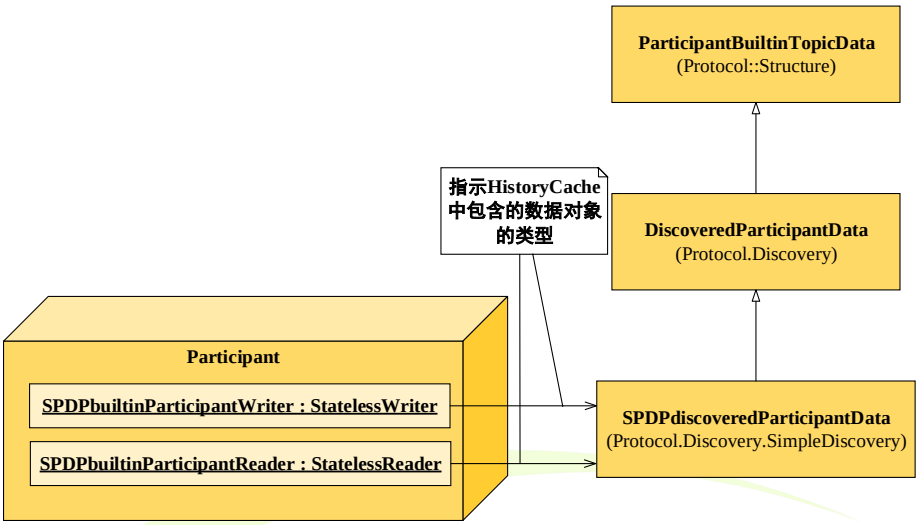


图 2-28 简单参与者发现协议使用的内置端点

协议为 SPDP 内置端点保留 ***EntityId_t*** 的以下值：

ENTITYID_SPDP_BUILTIN_PARTICIPANT_WRITER

ENTITYID_SPDP_BUILTIN_PARTICIPANT_READER

2.5.3.4 SPDPbuiltinParticipantWriter

表 2-74 中显示了用于配置 *SPDPbuiltinParticipantWriter* 的相关属性值。

表 2-74 SPDP 使用的 RTPS StatelessWriter 的属性

SPDPbuiltinParticipantWriter		
属性	**类型**	**值**
unicastLocatorList	Locator_t[*]	<自动检测> 传输种类和地址可以由应用程序自动检测或配置。端口是 SPDP 初始化的参数，或者设置为 PSM 指定的值，该值取决于 domainId。
multicastLocatorList	Locator_t[*]	<SPDP 初始化的参数> 默认为 PSM 指定的值。
reliabilityLevel	ReliabilityKind_t	BEST_EFFORT
topicKind	TopicKind_t	WITH_KEY
resendPeriod	Duration_t	<SPDP 初始化的参数> 默认为 PSM 指定的值。
readerLocators	ReaderLocator[*]	<SPDP 初始化的参数>

2.5.3.5 SPDPbuiltinParticipantReader

SPDPbuiltinParticipantReader 使用表 2-75 中显示的属性值进行配置。

表 2-75 SPDP 使用的 RTPS StatelessReader 的属性

SPDPbuiltinParticipantReader		
属性	**类型**	**值**
unicastLocatorList	Locator_t[*]	<自动检测> 传输种类和地址可以由应用程序自动检测或配置。端口是 SPDP 初始化的参数，或者设置为 PSM 指定的值，该值取决于 domainId。
multicastLocatorList	Locator_t[*]	<SPDP 初始化的参数> 默认为 PSM 指定的值。
reliabilityLevel	ReliabilityKind_t	BEST_EFFORT
topicKind	TopicKind_t	WITH_KEY

SPDPbuiltinParticipantReader 的 ***HistoryCache*** 包含有关所有存活的已发现参与者的信息。用于标识每个数据对象的关键字对应于参与者 GUID。

每次 *SPDPbuiltinParticipantReader* 收到与参与者有关的信息时，SPDP 都会检查 ***HistoryCache***，以查找与该参与者 GUID 相匹配的关键字条目。如果不存在具有匹配关键字的条目，则将添加一个新条目，该条目的关键字由参与者的 GUID 确定。

SPDP 定期检查 *SPDPbuiltinParticipantReader* ***HistoryCache***，以查找上一次收到数据距现在的时间超过其指定的 leaseDuration 的陈旧条目。过时的条目将被删除

2.5.3.6 简单参与者发现协议（SPDP）使用的逻辑端口

如上所述，每个 *SPDPbuiltinParticipantWriter* 都使用预先配置的定位器列表来宣布参与者在网络上的存在。

为了实现即插即用的互操作性，预配置的定位器列表必须使用以下众所周知的逻辑端口：

表 2-76 简单参与者发现协议使用的逻辑端口

端口	使用此端口配置的定位器
SPDP_WELL_KNOWN_UNICAST_PORT	*SPDPbuiltinParticipantReader.unicastLocatorList* 中的条目， *SPDPbuiltinParticipantWriter.readerLocator*s 中的单播条目。
SPDP_WELL_KNOWN_MULTICAST_PORT	*SPDPbuiltinParticipantReader.multicastLocatorList* 中的条目， *SPDPbuiltinParticipantWriter.readerLocators* 中的多播条目。

逻辑端口的实际值由 PSM 定义。

2.5.4 简单端点发现协议（SEDP）

端点发现协议（EDP）定义了两个参与者（***Participants***）之间必需的信息交换，以便发现彼此的 ***Writer*** 端点和 ***Reader*** 端点。

参与者（***Participants***）可以支持多个 EDP，但是出于互操作性的目的，所有实现都必须至少支持简单端点发现协议（SEDP）。

2.5.4.1 一般方法

与 SPDP 相似，简单端点发现协议（SEDP）使用预定义的内置端点。使用预定义的内置端点意味着，一旦一个参与者知道另一个参与者的存在，它就可以假设存在由远程参与者提供的内置端点，并与本地匹配的内置端点建立关联。

内置端点之间进行通信的协议与应用程序定义的端点所使用的协议相同。因此通过读取内置的 ***Reader*** 端点，协议虚拟机可以发现属于任何远程参与者的 DDS 实体的存在和其 QoS。类似地，通过内置的 ***Writer*** 端点，参与者可以将本地 DDS 实体的存在和 QoS 通知其他参与者。

因此，在 SEDP 中使用内置主题将整个发现协议的范围缩小到确定系统中存在哪些参与者以及与这些参与者内置端点相对应的 ***ReaderProxy*** 和 ***WriterProxy*** 对象的属性值。一旦知道了这些，其他一切都将通过 RTPS 协议的应用到内置 RTPS ***Reader*** 和 ***Writer*** 之间的通信产生。

2.5.4.2 简单端点发现协议（SEDP）使用的内置端点

SEDP 为“DCPSSubscription”，“DCPSPublication”和“DCPSTopic”主题映射了 DDS 内置实体。根据 DDS 规范，这些内置实体的可靠性 QoS 设置为“可靠”。因此，SEDP 将每个相应的内置 DDS DataWriter 或 DataReader 映射到相应的可靠 RTPS ***Writer*** 和 ***Reader*** 端点。

如图 2-29 所示，可以将用于“DCPSSubscription”，“DCPSPublication”和“DCPSTopic”主题的 DDS 内置 DataWriter 映射到可靠的 RTPS StatefulWriter，并将相应的 DDS 内置 DataReader 映射到可靠的 RTPS StatefulReader。实际的实现不必使用有状态的参考实现。为了实现互操作性，实现提供满足 2.4.2 中列出的一般要求的所需内置端点和可靠通信就足够了。

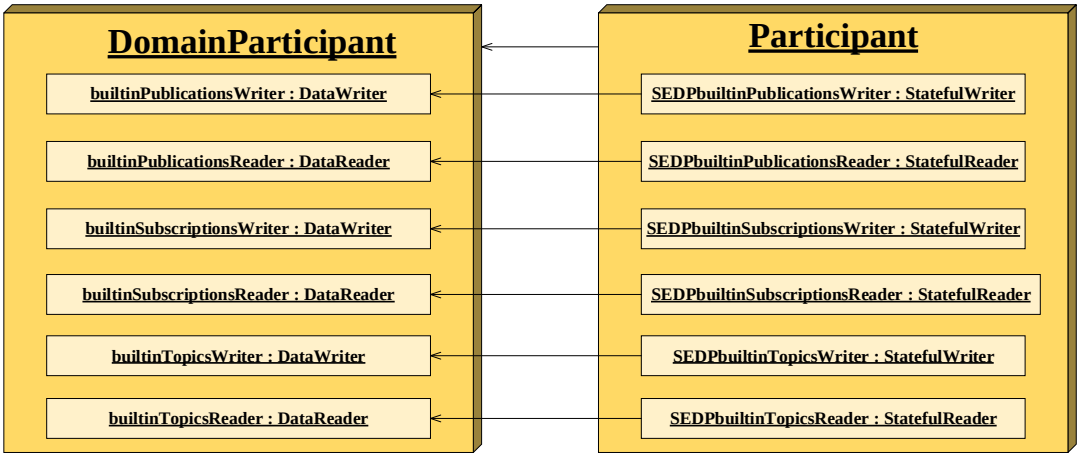


图 2-29 DDS 内置实体到相应的 RTPS 内置端点的映射示例

RTPS 协议为内置端点保留以下 EntityId_t 值：

ENTITYID_SEDP_BUILTIN_PUBLICATIONS_WRITER
ENTITYID_SEDP_BUILTIN_PUBLICATIONS_READER
ENTITYID_SEDP_BUILTIN_SUBSCRIPTIONS_WRITER
ENTITYID_SEDP_BUILTIN_SUBSCRIPTIONS_READER
ENTITYID_SEDP_BUILTIN_TOPIC_WRITER
ENTITYID_SEDP_BUILTIN_TOPIC_READER

保留的 EntityId_t 的实际值由每个 PSM 定义。

2.5.4.3 简单端点发现协议（SEDP）所需的内置端点

不需要实现提供所有内置端点。

如 DDS 规范中所述，主题传输是可选的。因此不需要实现 *SEDPbuiltinTopicsReader* 和 *SEDPbuiltinTopicsWriter* 内置端点，并且出于互操作性的目的，实现不应依赖于它们在远程参与者中的存在。

就其余内置端点而言，仅要求参与者提供匹配本地和远程端点所需的内置端点。例如，如果 DDS 参与者仅包含 DDS DataWriter，则唯一需要的 RTPS 内置端点是 *SEDPbuiltinPublicationsWriter* 和 *SEDPbuiltinSubscriptionsReader*。在这种情况下，*SEDPbuiltinPublicationsReader* 和 *SEDPbuiltinSubscriptionsWriter* 内置端点没有作用。

SPDP 指定参与者如何通知其他参与者其可用的内置端点。这在 2.5.3.2 中讨论。

2.5.4.4 简单端点发现协议（SEDP）使用的内置端点关联的数据类型

每个 RTPS 端点都有一个 ***HistoryCache***，用于存储与端点关联的数据对象的更改。这也适用于 RTPS 内置端点。因此每个 RTPS 内置端点都依赖于某个 DataType，该 DataType 表示写入其 ***HistoryCache*** 的数据的逻辑内容。

图 2-30 为“DCPSPublication”，“DCPSSubscription”和“DCPSTopic”主题定义了与 RTPS 内置端点关联的 ***DiscoveredWriterData***，***DiscoveredReaderData*** 和 ***DiscoveredTopicData*** 数据类型。与“DCPSParticipant”主题关联的数据类型在 2.5.3.2 中定义。

与每个 RTPS 内置端点关联的 DataType 包含 DDS 为相应的内置 DDS 实体指定的所有信息。因此 ***DiscoveredReaderData*** 扩展了 DDS 定义的 DDS :: SubscriptionBuiltinTopicData，***DiscoveredWriterData*** 扩展了 DDS :: PublicationBuiltinTopicData，而 ***DiscoveredTopicData*** 扩展了 DDS :: TopicBuiltinTopicData。

除了关联的内置 DDS 实体所需的数据外，“发现的”数据类型还包括协议的实现可能需要配置 RTPS 端点的所有信息。此信息包含在 RTPS ***ReaderProxy*** 和 ***WriterProxy*** 中。

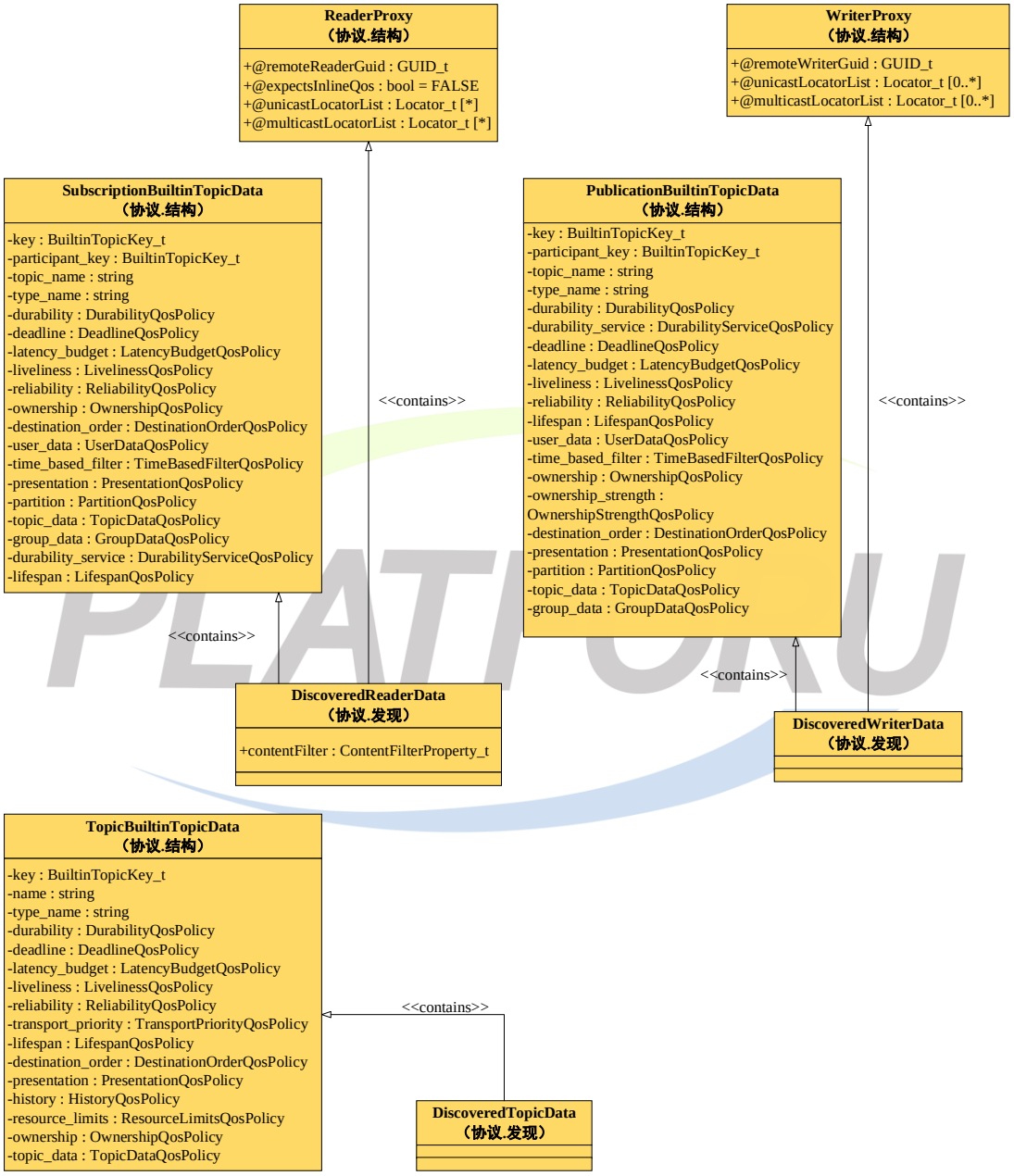


图 2-30 简单端点发现协议（SEDP）使用的内置端点关联的数据类型

协议的实现不必发送 DataTypes 中包含的所有信息。如果不存在任何可用信息，则实现可以采用 PSM 定义的默认值。PSM 还定义了如何在网络上表示发现信息。

SEDP 使用的 RTPS 内置端点及其关联的数据类型如图 2-31 所示。

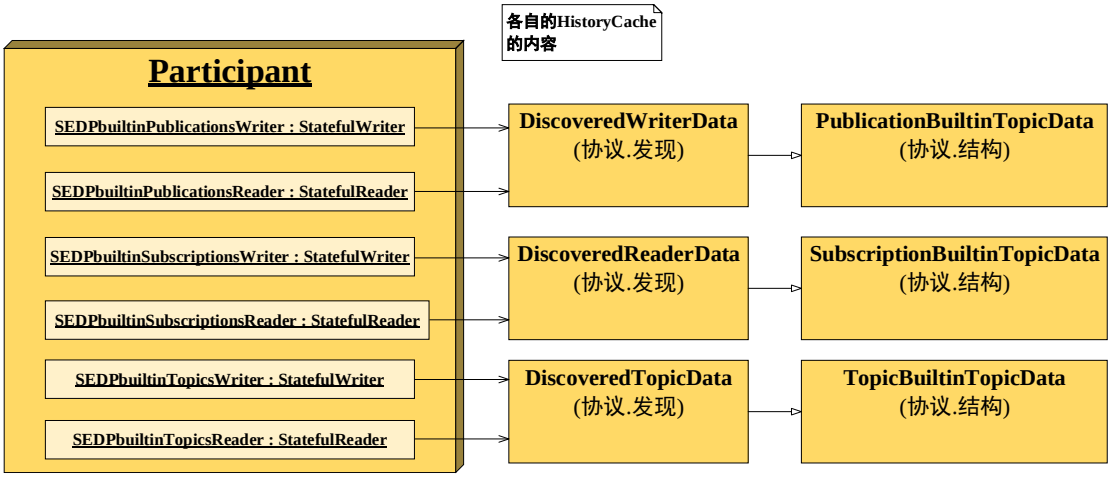


图 2-31 内置端点和与其各自 HistoryCache 关联的 DataType

每个内置端点的 ***HistoryCache*** 的内容可以从以下几个方面进行描述：数据类型，基数，数据对象插入，数据对象修改和数据对象删除。

•数据类型。缓存中存储的数据类型。这部分由 DDS 规范定义。

•基数。可以潜在地存储在缓存中的不同数据对象（每个具有不同的关键字）的数量。

•数据对象插入。将新数据对象插入缓存的条件。

•数据对象修改。修改现有数据对象的值的条件。

•数据对象删除。从缓存中删除现有数据对象的条件。

描述每个内置端点的 ***HistoryCache*** 是说明性的。

2.5.4.4.1 SEDPbuiltinPublicationsWrite 和 SEDPbuiltinPublicationsReader

表 2-77 描述了 *SEDPbuiltinPublicationsWriter* 和 *SEDPbuiltinPublicationsReader* 的 ***HistoryCache***。

表 2-77 SEDPbuiltinPublicationsWriter 和 SEDPbuiltinPublicationsReader 的 HistoryCache 的内容

方面	描述
数据类型	DiscoveredWriterData
基数	DomainParticipant 包含的 DataWriter 的数量。参与者中的每个 DataWriter 与一个描述 SEDPbuiltinPublicationsWriter 的 WriterHistoryCache 中存储的 DataWriter 的数据对象之间存在一一对应的关系
数据对象插入	每次在 DomainParticipant 中创建 DataWriter 时。
数据对象修改	每次修改现有 DataWriter 的 QoS 时。
数据对象删除	每次删除属于 DomainParticipant 的现有 DataWriter 时。

2.5.4.4.2 SEDPbuiltinSubscriptionsWriter 和 SEDPbuiltinSubscriptionsReader

表 2-78 描述了 *SEDPbuiltinSubscriptionsWriter* 和 *SEDPbuiltinSubscriptionsReader* 的 ***HistoryCache***。

表 2-78 SEDPbuiltinSubscriptionsWriter 和 SEDPbuiltinSubscriptionsReader 的 HistoryCache 的内容

方面	描述
数据类型	DiscoveredReaderData
基数	DomainParticipant 包含的 DataReader 的数量。 参与者中的每个 DataReader 与一个描述 SEDPbuiltinSubscriptionsWriter 的 WriterHistoryCache 中存储的 DataReader 的数据对象之间存在一一对应的关系
数据对象插入	每次在 DomainParticipant 中创建 DataReader 时
数据对象修改	每次修改现有 DataReader 的 QoS 时
数据对象删除	每次删除属于 DomainParticipant 的现有 DataReader 时

2.5.4.4.3 SEDPbuiltinTopicsWriter 和 SEDPbuiltinTopicsReader

表 2-79 描述了 *SEDPbuiltinTopicsWriter* 和 *BuiltinTopicsReader* 的 ***HistoryCache***。

表 2-79 SEDPbuiltinTopicsWriter 和 SEDPbuiltinTopicsReader 的 HistoryCache 的内容

方面	描述
数据类型	DiscoveredTopicData
基数	DomainParticipant 创建的 Topic 数。 DomainParticipant 创建的每个 Topic 与一个数据对象之间存在一一对应的关系，该数据对象描述了存储在 BuiltinTopicsWriter 的 WriterHistoryCache 中的 Topic。
数据对象插入	每次在 DomainParticipant 中创建 Topic 时
数据对象修改	每次修改现有 Topic 的 QoS 时
数据对象删除	每次删除属于 DomainParticipant 的现有 Topic 时

2.5.5 与 RTPS 虚拟机的交互

为了进一步说明 SPDP 和 SEDP，此小节描述了如何将 SPDP 提供的信息用于在 RTPS 虚拟机中配置 SEDP 内置端点。

2.5.5.1 发现新的远程参与者

通过使用 ***SPDPbuiltinParticipantReader***，本地参与者“*local_participant*”发现了 ***DiscoveredParticipantData*** *participant_data* 所描述的另一个参与者的存在。被发现的参与者使用 SEDP。

下面的伪代码将 *local_participant* 中的本地 SEDP 内置端点配置为与发现的参与者中的相应 SEDP 内置端点进行通信。

请注意，如何配置端点取决于协议的实现。对于有状态参考实现，此操作执行以下逻辑步骤：

```
IF ( PUBLICATIONS_READER IS_IN participant_data.availableEndpoints ) THEN
    guid=<participant_data.guidPrefix,
        ENTITYID_SEDP_BUILTIN_PUBLICATIONS_READER>;
    writer = local_participant.SEDPbuiltinPublicationsWriter;
    proxy = new ReaderProxy( guid,
                            participant_data.metatrafficUnicastLocatorList,
                            participant_data.metatrafficMulticastLocatorList);
    writer.matched_reader_add(proxy);
ENDIF

IF ( PUBLICATIONS_WRITER IS_IN participant_data.availableEndpoints ) THEN
    guid=<participant_data.guidPrefix,
        ENTITYID_SEDP_BUILTIN_PUBLICATIONS_WRITER>;
    reader = local_participant.SEDPbuiltinPublicationsReader;
    proxy = new WriterProxy( guid,
                            participant_data.metatrafficUnicastLocatorList,
                            participant_data.metatrafficMulticastLocatorList);
    reader.matched_writer_add(proxy);
ENDIF

IF ( SUBSCRIPTIONS_READER IS_IN participant_data.availableEndpoints ) THEN
    guid=<participant_data.guidPrefix,
        ENTITYID_SEDP_BUILTIN_SUBSCRIPTIONS_READER>;
    writer = local_participant.SEDPbuiltinSubscriptionsWriter;
    proxy = new ReaderProxy( guid,
                            participant_data.metatrafficUnicastLocatorList,
                            participant_data.metatrafficMulticastLocatorList);
    writer.matched_reader_add(proxy);
ENDIF

IF ( SUBSCRIPTIONS_WRITER IS_IN participant_data.availableEndpoints ) THEN
    guid=<participant_data.guidPrefix,
        ENTITYID_SEDP_BUILTIN_SUBSCRIPTIONS_WRITER>;
    reader = local_participant.SEDPbuiltinSubscriptionsReader;
```

```
        proxy = new WriterProxy( guid,
                                 participant_data.metatrafficUnicastLocatorList,
                                 participant_data.metatrafficMulticastLocatorList);
        reader.matched_writer_add(proxy);
    ENDIF

    IF ( TOPICS_READER IS_IN participant_data.availableEndpoints ) THEN
        guid=<participant_data.guidPrefix,
             ENTITYID_SEDP_BUILTIN_TOPICS_READER>;
        writer = local_participant.SEDPbuiltinTopicsWriter;
        proxy = new ReaderProxy( guid,
                                 participant_data.metatrafficUnicastLocatorList,
                                 participant_data.metatrafficMulticastLocatorList);
        writer.matched_reader_add(proxy);
    ENDIF

    IF ( TOPICS_WRITER IS_IN participant_data.availableEndpoints ) THEN
        guid=<participant_data.guidPrefix,
             ENTITYID_SEDP_BUILTIN_TOPICS_WRITER>;
        reader = local_participant.SEDPbuiltinTopicsReader;
        proxy = new WriterProxy( guid,
                                 participant_data.metatrafficUnicastLocatorList,
                                 participant_data.metatrafficMulticastLocatorList);
        reader.matched_writer_add(proxy);
    ENDIF
```

2.5.5.2 删除先前发现的参与者

根据远程参与者的 *leaseDuration*，本地参与者“*local_participant*”获悉先前发现的 GUID_t 为 *participant_guid* 的参与者不再存在。参与者“*local_participant*”必须重新配置与 GUID_t 为 *participant_guid* 的参与者中的端点进行通信的所有本地端点。

对于有状态参考实现，此操作执行以下逻辑步骤：

```
    guid=<participant_guid.guidPrefix,
         ENTITYID_SEDP_BUILTIN_PUBLICATIONS_READER>;
    writer = local_participant.SEDPbuiltinPublicationsWriter;
    proxy = writer.matched_reader_lookup(guid);
    writer.matched_reader_remove(proxy);

    guid=<participant_guid.guidPrefix,
         ENTITYID_SEDP_BUILTIN_PUBLICATIONS_WRITER>;
    reader = local_participant.SEDPbuiltinPublicationsReader;
    proxy = reader.matched_writer_lookup(guid);
    reader.matched_writer_remove(proxy);
```

```
guid=<participant_guid.guidPrefix,
      ENTITYID_SEDP_BUILTIN_SUBSCRIPTIONS_READER>;
writer = local_participant.SEDPbuiltinSubscriptionsWriter;
proxy = writer.matched_reader_lookup(guid);
writer.matched_reader_remove(proxy);

guid=<participant_guid.guidPrefix,
      ENTITYID_SEDP_BUILTIN_SUBSCRIPTIONS_WRITER>;
reader = local_participant.SEDPbuiltinSubscriptionsReader;
proxy = reader.matched_writer_lookup(guid);
reader.matched_writer_remove(proxy);

guid=<participant_guid.guidPrefix, ENTITYID_SEDP_BUILTIN_TOPICS_READER>;
writer = local_participant.SEDPbuiltinTopicsWriter;
proxy = writer.matched_reader_lookup(guid);
writer.matched_reader_remove(proxy);

guid=<participant_guid.guidPrefix, ENTITYID_SEDP_BUILTIN_TOPICS_WRITER>;
reader = local_participant.SEDPbuiltinTopicsReader;
proxy = reader.matched_writer_lookup(guid);
reader.matched_writer_remove(proxy);
```


2.5.5.3 支持替代发现协议

参与者和端点发现协议的要求可能会因部署场景而异。例如为速度和简便性而优化的协议（例如将要部署在 LAN 上的嵌入式设备中的协议）可能无法很好地扩展到 WAN 环境中的大型系统。

因此，RTPS 规范允许实现支持多个 PDP 和 EDP。有多种实现发现协议的可能方法，包括使用静态发现，基于文件的发现，中央查找服务等。RTPS 出于互操作性的目的而提出的唯一要求是，所有 RTPS 实现都至少支持以下内容： SPDP 和 SEDP。预计随着时间的流逝，将开发出一系列可互操作的发现协议，以满足特定的部署需求。

如果实现支持多个 PDP，则每个 PDP 可能会进行不同的初始化，并发现一组不同的远程参与者。必须至少使用 SPDP 与使用其他供应商的 RTPS 实现的远程参与者联系，以确保互操作性。当远程参与者使用相同的 RTPS 实现时，则没有这样的要求。

即使所有参与者都使用 SPDP，远程参与者仍可能使用不同的 EDP。SPDP 交换的信息中包含参与者支持的 EDP。所有参与者都必须至少支持 SEDP，因此他们总是至少有一个共同的 EDP。但是，如果两个参与者都支持另一个 EDP，则可以改用此替代协议。在这种情况下，无需创建 SEDP 内置端点，或者如果它们已经存在，则无需配置它们以匹配新的远程参与者。这种方法使供应商可以根据需要自定义 EDP，而不会损害互操作性。

2.6 版本和可扩展性

此版本的 RTPS 协议的实现不仅应能够处理具有相同主版本的 RTPS 消息，而且还可能

处理具有更高的次版本的 RTPS 消息。

2.6.1 主版本中允许的扩展

在此主版本中，该协议的未来次版本可以通过以下方式增强该协议：

- 可以在 RTPS 消息中的任何位置引入和使用带有其他 *submessageId* 的附加子消息。实现应使用 SubmessageHeader 中的 *submessageLength* 字段跳过未知的子消息。
- 可以将其他字段添加到当前次版本中已定义的子消息的末尾。实现应使用 SubmessageHeader 中的 *submessageLength* 字段跳过其他字段。
- 可以添加具有新 ID 的其他内置端点。实现应忽略任何未知的内置端点。
- 可以添加带有新 *parameterIds* 的附加参数。实现应忽略任何未知参数。

所有这些更改都需要增加次版本号。

2.6.2 主版本中无法更改的内容

在同一主版本中不能更改以下各项：

- 无法删除子消息。
- 除非如 2.6.1 中所述，否则无法修改子消息。
- *submessageIds* 的含义无法修改。

所有这些更改都需要增加主版本号。

2.7 使用 RTPS 实现 DDS QoS 和高级 DDS 功能

RTPS 协议及其扩展机制提供了实现 DDS 所需的核心功能。本小节定义了如何使用 RTPS 来实现 DDS QoS 参数。

此外，本小节定义了实现以下高级 DDS 功能所需的 RTPS 协议扩展：

- 基于内容过滤的主题，请参考 2.7.3
- 相干集，请参考 8.7.5

所有扩展都基于 RTPS 提供的标准扩展机制。

为了保证互操作性，本小节构成了规范的标准部分。

2.7.1 向数据子消息中添加内联参数

Data 和 **DataFrag** 子消息可选地包含 **ParameterList** *SubmessageElement*，用于存储内联 QoS 参数和其他信息。

如果 ***Reader*** 没有保留匹配的远程 ***Writer*** 或其配置的 QoS 参数的列表（即是无状态 ***Reader***），则带有内联 QoS 参数的 **Data** 子消息包含使 ***Reader*** 能够应用所有特定于 ***Writer*** 的 QoS 参数所需的所有信息。

无状态 ***Reader*** 需要接收内联 QoS 来获取远程 ***Writer*** 上的信息，这是要求 ***Writer*** 在 ***Reader*** 请求时发送内联 QoS 的原因（2.4.2.2.2）。

对于不可变的 QoS，所有 RxO QoS 都以内联的方式发送，以使无状态 ***Reader*** 在 QoS 不兼容的情况下拒收数据样本。与 ***Reader*** 相关的可变 QoS 是通过内联的方式发送的，因此无

论 ***Reader*** 上保持的状态量如何，它们都可以立即生效。请注意有状态 ***Reader*** 可以选择依靠其远程 ***Writer*** 的缓存信息，而不是所接收的内联 QoS。

无状态 ***Reader*** 使用发现协议向远程 ***Writer*** 宣布它希望接收内联 QoS 参数，如发现模块（2.5）中所述。如果期望内联 QoS 参数，则实现还必须包括主题名称作为内联参数。这样可以确保在接收方可以将子消息传递给该主题的所有 ***Reader***，包括无状态 ***Reader***。

与 ***Reader*** 是否期望内联 QoS 参数无关，**Data** 子消息可能还包含与相干集和基于内容过滤主题相关的内联参数。在后面的小节中对此进行了更详细的描述。

为了提高性能，有状态的实现可能会忽略内联 QoS，而仅依赖于通过发现获得的缓存值。请注意不解析内联 QoS 可能会延迟新 QoS 生效的时间点，因为新 QoS 必须首先通过发现协议传输。

2.7.2 DDS QoS 参数

表 2-80 概述了哪些 QoS 参数会影响 RTPS 有线协议，哪些可能会显示为内联 QoS。在下面的小节中将更详细影响有线协议的参数。

表 2-80 使用 RTPS 有线协议实现 DDS QoS 参数

QoS	对 RTPS 协议的影响	是否可能显示为内联 QoS
USER_DATA	无	否
TOPIC_DATA	无	否
GROUP_DATA	无	否
DURABILITY	参看 2.7.2.2.1	是
DURABILITY_SERVICE	无	否
PRESENTATION	参看 2.7.2.2.2	是
DEADLINE	无	是
LATENCY_BUDGET	无	是
OWNERSHIP	无	是
OWNERSHIP_STRENGTH	无	是
LIVELINESS	参看 2.7.2.2.3	是
TIME_BASED_FILTER	参看 2.7.2.2.4	否
PARTITION	无	是
RELIABILITY	参看 2.7.2.2.5	是
TRANSPORT_PRIORITY	无	是
LIFESPAN	无	是
DESTINATION_ORDER	参看 2.7.2.2.6	是
HISTORY	无	否
RESOURCE_LIMITS	无	否
ENTITY_FACTORY	无	否
WRITER_DATA_LIFECYCLE	参看 2.7.2.2.7	否
READER_DATA_LIFECYCLE	无	否

2.7.2.1 内联 DDS QoS 参数

表 2-80 列出了可能通过内联方式显示的标准 DDS QoS 参数。

如果 ***Reader*** 希望接收内联 QoS 参数，并且缺少这些 QoS 参数任意一个，它将采用该 QoS 参数的默认值，默认值由 DDS 定义。

内联参数将添加到 **Data** 子消息中，以使其自描述。为了获得自描述消息，不仅必须将表 2-80 中定义的参数与子消息一起发送，而且还必须发送参数 TOPIC_NAME。 此参数包含子消息所属的主题的名称。

2.7.2.2 影响有线协议的 DDS QoS 参数

2.7.2.2.1 DURABILITY

尽管 *volatile* 和 *transient-local* 持久性 QoS 不影响 RTPS 协议，但 *transient* 和 *persistent* 的持久性 QoS 支持可能会影响。当前版本的规范未涵盖此内容。

2.7.2.2.2 PRESENTATION

第 2.7.5 节定义了如何实现 PRESENTATION QoS 的一致性访问策略。

此 QoS 的其他方面不影响 RTPS 协议。

2.7.2.2.3 LIVELINESS

实现必须遵循以下方法：

•DDS_AUTOMATIC_LIVELINESS_QOS：通过 ***BuiltinParticipantMessageWriter*** 保持存活性。对于给定的参与者，为了在 LIVELINESS QoS 设置为 AUTOMATIC 的情况下保持其 ***Writer*** 实体的活动性，实现必须刷新参与者的存活性（即发送 ParticipantMessageData，请参阅 2.4.13.5），该数据发送速度要比租约持续时间最小的作家快。

• DDS_MANUAL_BY_PARTICIPANT_LIVELINESS_QOS：存活性是通过 ***BuiltinParticipantMessageWriter*** 保持的。如果参与者有任何 MANUAL_BY_PARTICIPANT 的 ***Writer***，则实现必须定期检查是否其中任意一个调用了 **write()**，**assert_liveliness()**，**dispose()**或 **unregister_instance()**方法。此检查的周期等于 ***Writer*** 中最小的租约期限。如果调用了上述任何一个方法，则实现必须刷新参与者的存活度（即发送 ParticipantMessageData，请参阅 2.4.13.5）。

•DDS_MANUAL_BY_TOPIC_LIVELINESS_QOS：通过发送数据或设置了存活标记的显式心跳消息来保持存活性。通过在 ***Writer*** 实体上调用 **write()**，**dispose()**或 **unregister_instance()**方法生成的标准 RTPS 消息足以断言将 LIVELINESS QoS 设置为 MANUAL_BY_TOPIC 的 ***Writer*** 的活泼性。调用 **assert_liveliness()**方法时，***Writer*** 必须发送设置了 final 标志和 liveliness 标志的心跳消息。

2.7.2.2.4 TIME_BASED_FILTER

通过在 ***Writer*** 端应用基于时间的过滤器，实现可以优化带宽使用。这样将不会发送将在 ***Reader*** 端丢弃的数据。

当在 ***Writer*** 端过滤掉一个或多个数据更新时，实现必须改为发送 **Gap** 子消息，指明哪些样本已被过滤掉。此子消息必须在下一次更新之前发送，并通知 ***Reader*** 缺少的更新已被过滤，而不仅仅是丢失。

2.7.2.2.5 RELIABILITY

实现必须满足 2.4.2 中定义的可靠的 RTPS 协议对互操作性的要求。

2.7.2.2.6 DESTINATION_ORDER

为了实现 DDS_BY_SOURCE_TIMESTAMP_DESTINATIONORDER_QOS 策略，实现必须包括 **InfoTimestamp** 子消息，并具有来自 ***Writer*** 的每次更新。

2.7.2.2.7 WRITER_DATA_LIFECYCLE

如果启用了 *autodispose_unregistered_instances*，则注销实例的数据消息也必须对其进行处置。这限制了 DisposedFlag 和 UnregisteredFlag 标志的可用值。

2.7.3 内容过滤的主题

内容过滤的主题使 DDS DataReader 可以请求中间件根据内容过滤掉数据样本。

仅在 ***Reader*** 端进行过滤时，中间件会简单地丢弃未通过过滤器的样本。这种情况下无需进一步扩展 RTPS。

在许多情况下，除了在 ***Reader*** 端进行过滤之外，实现还将受益于 ***Writer*** 端的过滤。通过在 ***Writer*** 端进行过滤，中间件不会发送无法通过 ***Reader*** 端过滤器的样本。这样可以节省带宽。当发送样本时，***Writer*** 可以包含通过了哪些过滤器的信息。这样就可以在 ***Writer*** 一侧仅应用一次过滤器，而不是每个 ***Reader*** 一次。***Reader*** 仅检查是否在 ***Writer*** 端过滤了样本。如果是这样，则不需要应用过滤器。

为了支持 ***Writer*** 端过滤，标准的 RTPS 扩展机制用于：

- 在端点发现阶段包括 ***Reader*** 过滤器信息。
- 在每个数据样本中包括过滤结果。

2.7.3.1 使用内置端点交换过滤器信息

内容过滤的主题在 ***Reader*** 端定义。为了实现 ***Writer*** 端过滤，必须将有关 ***Reader*** 使用的过滤器的信息传播到匹配的远程 ***Writer***。这需要扩展与 RTPS 内置端点关联的数据类型。

如图 2-31 所示，与 RTPS 内置端点关联的数据类型扩展了 DDS 内置主题数据类型，其中包括所有相关的 QoS。由于 DDS 并未将内容过滤的主题定义为 ***Reader*** QoS 策略（而是 DDS 定义了单独的内容过滤的主题），因此 RTPS 向 DiscoveredReaderData 添加了一个额外

的 ***ContentFilterProperty_t*** 字段，如表 2-81 所定义。

表 2-81 内容过滤器属性

ContentFilterProperty_t		
属性	**类型**	**值**
contentFilteredTopicName	string	与 ***Reader*** 关联的内容过滤主题的名称。 长度必须为非零。
relatedTopicName	string	与内容过滤主题相关的主题名称。 长度必须为非零。
filterClassName	string	标识此过滤器所属的过滤器类。RTPS 可以支持多种过滤器类（SQL，正则表达式，自定义过滤器等）。 长度必须为非零。 RTPS 预定义以下值：“DDSSQL”。如果未指定，则为默认过滤器类名称。 匹配 DDS 指定的 SQL 过滤器，该过滤器在所有实现中都必须可用。
filterExpression	string	实际的过滤器表达式。 对于使用 *filterClassName* 指定的过滤器类，必须为有效表达式。 长度必须为非零。
expressionParameters	stringSequence	定义过滤器表达式中每个参数的值。 如果过滤器表达式不包含任何参数，则长度可以为零。

ContentFilterProperty_t 字段提供所有必需的信息，以在 ***Writer*** 端启用内容过滤。例如，对于默认的 DDSSQL 过滤器类，包含成员 a，b 和 c 的数据类型的有效过滤器表达式可以是“（a <5）AND（b ==%0）AND（c> =%1）”表达式参数为“5”和“3”。为了使 Writer 应用过滤器，必须将其配置为处理指定过滤器类的过滤器。否则 ***Writer*** 将忽略过滤器信息，而不过滤任何数据样本。

DDS 允许用户在运行时修改过滤器表达式参数。每次修改参数时，将使用端点发现协议交换更新的信息。这等同于更新可变的 QoS 值。

2.7.3.2 在每个数据样本中包括内联过滤结果

通常在 ***Writer*** 端应用过滤时，如果样本未通过远程 ***Reader*** 的过滤器，则不会发送该样本。在这种情况下，**Data** 子消息将被替换为 **Gap** 子消息。这样可以确保样本在 ***Reader*** 端不被视为“丢失”。这种方法与在 ***Writer*** 端应用基于时间的过滤器相匹配。本小节剩余讨论仅涉及 **Data** 子消息，但对 **DataFrag** 子消息采用相同的方法。

在某些情况下，例如当使用多播发送数据时，***Reader*** 仍可能会接收未通过其过滤器的样本。另一个场景是属于同一参与者的多个 ***Reader***。在那种情况下，***Writer*** 只需要发送一条发往 ENTITYID_UNKNOWN 的 RTPS 消息（请参阅 2.4.15.5）。每个 ***Reader*** 可能使用不同的过滤器，在这种情况下 ***Writer*** 需要在发送样本之前应用多个过滤器。

在这两种用例中，都有两个选择：

1.该样本未通过任何远程 ***Reader*** 的任意过滤器。在这种情况下，**Data** 子消息再次被 **Gap** 子消息代替。

2.样本通过部分或全部过滤器。在这种情况下，仍必须发送样本，并且 ***Writer*** 必须在 **Data** 子消息中包含应用了哪些过滤器以及相应结果的信息。

Data 子消息的 *inlineQos* 元素用于包括必要的过滤器信息。更具体地说，添加了一个新参数，其中包含表 2-82 中所示的信息。

表 2-82 与数据样本关联的内容过滤器信息

ContentFilterInfo_t		
属性	**类型**	**值**
filterResult	FilterResult_t	对于每个过滤器签名，结果表明样本是否通过过滤器。
filterSignatures	FilterSignature_t[]	应用于样本的过滤器列表

过滤器签名 ***FilterSignature_t*** 唯一标识过滤器并基于表 2-81 中列出的过滤器属性。PSM 定义了如何表示和计算过滤器签名。样本是否通过了 ***Writer*** 端应用的过滤器，由 *filterResult_t* 属性（由 PSM 定义）编码。

请注意当过滤器的表达式参数更改时，过滤器签名也会更改。在接收到更新的参数值之前，***Writer*** 端过滤器可能会使用过时的表达式参数，在这种情况下内联过滤器签名将与 ***Reader*** 期望的签名不匹配。***Reader*** 将忽略过滤器结果，而应用其本地过滤器。

2.7.3.3 互操作性要求

Writer 端过滤是一种优化，它是可选的，因此互操作性不是必需的。

如果出现以下情况，样本将始终在 ***Reader*** 端进行过滤：

- ***Writer*** 端未应用任何过滤。
- ***Writer*** 端未应用 ***Reader*** 期望的过滤器。如前所述，如果尚未通知 ***Writer*** 有关更新的过滤器参数，则可能会发生这种情况。
- ***Reader*** 端不支持 ***Writer*** 端过滤（因此将忽略内联过滤器信息）。

同样，***Writer*** 可能不会过滤样本，因为

- 实现不支持内容过滤的主题（在这种情况下，将忽略 ***Reader*** 的过滤器属性）。
- ***Reader*** 的过滤器信息被拒绝（例如无法识别的过滤器类别）。如果实现支持内容过滤主题，则它必须至少识别 DDS 规范要求的“DDSSQL”过滤器类。对于所有其他过滤器类，这两种实现都必须允许用户注册相同的自定义过滤器类。
- 其他特定于实现的限制，例如每个 ***Writer*** 能够存储其过滤器信息的远程 ***Reader*** 数量的资源限制。

2.7.4 实例生命周期状态的变化

除了写入数据之外，DDS DataWriter 可以注册数据对象实例（**register_instance** 方法，更新其值（**write** 方法），处置数据对象实例（**dispose** 方法）和注销数据对象实例（**unregister_instance** 方法），这些方法中的每一个都可能引起通知，该通知会发送到匹配的

DDS DataReader 上。DDS DataReader 可以通过检查 DDS DataReader ***read*** 或 ***take*** 方法返回的 ***SampleInfo*** 中的 ***LifecycleState instance_state*** 字段来确定更改的性质。

RTPS 使用常规 **Data** 子消息和内联 QoS 参数扩展机制来传达生命周期更改。内联 QoS 中的序列化信息包含新的 ***LifecycleState***，即实例是否已注册，未注册或已处置。实际详细信息取决于 PSM（例如，请参见 3.6.3.4）。

RTPS 的实现必须使用 **Data** 子消息来传达生命周期更改。这样做时允许 RTPS 的实现在 SerializedPayload 子消息元素内包含数据对象的关键字（请参见 2.3.7.2）。这是因为关键字 Key 足以唯一标识 ***LifecycleState*** 更改适用的数据对象实例。

在 DDS DataWriter 写入该数据对象实例的第一个值之前，不需要 RTPS 的实现来传输注册更改。

2.7.5 一致性集合

DDS 规范提供了将一组样本更新定义为一致性集合的功能。仅在收到一致性集合中的所有更新后，才通知 DataReader 有新的更新。

使用容器 Publisher 表示一致集合的开始和结束，通过这种方法在 DataWriter 端定义了构成一致性集合的内容。一个一致的集合可以仅跨越指定 DataWriter 写入的实例（访问范围 TOPIC），也可以跨越属于同一发布者的多个 DataWriter（访问范围 GROUP）。获取详细信息，请参考 DDS 规范。

为了支持一致性集合，RTPS 使用内联 QoS 参数扩展机制在每个 **Data** 子消息中内联附加信息。附加信息表示特定一致性集合的成员资格。本节剩余的讨论仅涉及 **Data** 子消息，但对 **DataFrag** 子消息采用相同的方法。

对于访问范围 TOPIC，属于同一一致性集合的所有 **Data** 子消息都具有严格单调递增的序列号（因为它们源自同一 ***Writer***）。因此，通过属于该一致性集合的第一个样本更新的序列号来唯一地识别一致性集合。属于同一一致性集合的所有样本更新都包含有相同序列号的内联 QoS 参数。这种方法还使 ***Reader*** 可以轻松确定一致性集合何时开始。

Writer 的一致性集合的结尾由以下任意一种数据的到达来定义：

- 来自此 ***Writer*** 的 **Data** 子消息，它属于新的一致性集合。
- 来自此 ***Writer*** 的 **Data** 子消息，其中不包含一致性集合的内联 QoS 参数，或者包含值 SEQUENCENUMBER_UNKNOWN 的一致性集合的内联 QoS 参数。两种方法是等效的。

请注意，**Data** 子消息不一定需要包含 *serializedPayload*。这样就可以在 ***Writer*** 写入下一个数据之前通知 ***Reader*** 有关一致集合的结尾。

最后，访问范围 GROUP 所需的扩展机制尚未定义。

2.7.6 定向写入

通过在 DDS 发布-订阅框架中标记拥有预期接收者句柄的样本，可以启用直接点对点通信。

RTPS 通过使用内联 QoS 参数扩展机制支持定向写入。序列化信息表示目标 ***Reader*** 的 GUID。

当 ***Writer*** 发送定向样本时，只有具有匹配 GUID 的接收者才接受该样本；所有其他接

收者都确认收到样本但直接吸收不提交应用，就像是 GAP 消息一样。

2.7.7 属性列表

属性列表是应用于DDS实体的用户可定义属性的列表。列表中的条目是通用名称/值对。用户将一对名称/值定义为 DDS 参与者，DataWriter 或 DataReader 的属性。此可扩展列表允许应用非 DDS 指定的属性。

RTPS 协议支持将属性列表作为内联参数。可以在发现阶段或作为串联 QoS 传输属性。

2.7.8 原始写入者信息

支持 DDS 持久性 QoS 的 TransientLocal，Transient 或 Persistent 级别的服务需要发送代表永久写入者接收和存储的数据。

转发消息的服务需要指明转发的消息属于另一个写入者的消息流，这样如果读取者从另一个来源（例如另一个转发服务或原始写入者）接收到相同的消息，则可以按照重复消息处理。

RTPS 协议通过包含原始写入者的信息来支持消息的这种转发。

当 RTPS ***Reader*** 接收到此信息时，会将其视为普通的 ***CacheChange***，但是一旦 ***CacheChange*** 准备好提交给 DDS DataReader，它不会提交。相反它将转交给与 ORIGINAL_WRITER_INFO 内联 QoS 中表明的 RTPS ***Writer*** 进行通信的 RTPS ***Reader*** 的 ***HistoryCache***，并将其视为具有此处的序列号，而且 ParameterList 也包含在 ORIGINAL_WRITER_INFO 中。

表 2-83 原始写入者信息

ContentFilterInfo_t		
属性	**类型**	**值**
originalWriterGUID	GUID_t	第一个生成消息的 RTPS ***Writer*** 的 GUID。
originalWriterSN	SequenceNumber_t	从原始 ***Writer*** 发送的 ***CacheChange*** 的序列号。
originalWriterQoS	ParameterList	内联参数列表应应用于第一个生成样本的 RTPS ***Writer*** 发送的 ***CacheChange***。

2.7.9 关键字哈希

关键字哈希为关键字提供了提示，该关键字唯一地标识了 DDS DataWriter 已注册的对象集中正在更改的数据对象。

名义上关键字是 **Data** 子消息序列化数据的一部分。通过使用关键字哈希，可以提供比从接收的数据对象反序列化出完整关键字更快的替代方法，从而为实现带来好处。

当DataReader未收到关键字哈希时，应从数据本身计算得出。如果子消息中没有数据，则 DataReader 应该使用默认的零值关键字哈希。

如果存在 KeyHash，则应按照 3.6.3.3 中的描述进行计算。

3 平台特定模型（PSM）：UDP / IP

3.1 引言

本节定义了平台特定模型（PSM），该模型将协议 PIM 映射到 UDP/IP。此 PSM 的目标是直接在 UDP/IP 之上以最小的开销提供映射。

UDP/IP 作为 DDS 应用程序的传输方式的适用性源于以下几个因素：

•普遍可用性。作为 IP 栈的核心部分，UDP/IP 几乎在所有操作系统上都可用。

•轻量级。UDP/IP 是一个非常简单的协议，它在 IP 之上添加了最少的服务。使用它可以以最小的开销使用基于 IP 的网络。

•尽力而为。UDP/IP 提供了一种尽力而为的服务，可以很好地映射许多实时数据流的服务质量需求。在需要的情况下，RTPS 协议提供了一种机制，以在 UDP 提供的尽力而为服务之上获得可靠传输。

•无连接。UDP/IP 提供无连接服务；这允许多个 RTPS 端点共享一个操作系统 UDP 资源（套接字/端口），同时允许消息交错传输，从而为每个单独的数据流有效地提供了带外机制。

•可预测的行为。与 TCP 不同，UDP 不引入计时器，该计时器将导致操作在不定的时间量内阻塞。这样，可以更简单地模拟使用 UDP 对实时应用程序的影响。

•可扩展性和多播支持。UDP/I 原生支持多播，从而可以将单个消息有效地分发给大量接收者。

3.2 符号约定

3.2.1 命名空间

本文档中的所有定义都是“RTPS”命名空间的一部分。为了便于阅读和理解，在本文档的定义和类中省略了命名空间前缀。

3.2.2 结构的 IDL 表示法和 CDR 有线表示法

以下部分通常定义结构，例如：

```
typedef octet OctetArray3[3];

struct EntityId_t
{
    OctetArray3 entityKey;
    octet entityKind;
} ;
```

这些定义使用 OMG IDL（接口定义语言）。当这些结构通过网络发送时，它们将使用相应的 CDR 表示进行编码。

3.2.3 位和字节的表示

本文档通常使用以下表示法来表示一个八位位组或字节：

```
+-+-+-+-+-+-+-+-+
| 7 |6| 5| 4| 3 |2|1| 0|
+-+-+-+-+-+-+-+-+
```

在这种表示法中，最左边的位（位 7）是最高有效位（“MSB”），最右边的位（位 0）是最低有效位（“LSB”）。

字节流按每行 4 个字节的顺序排序，如下所示：

```
0.....2.....8.....16.....23.....32
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  第一字节    |                |                |  第四个字节  |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
                                ----------流方向----------->>>>
```

在此表示形式中，流中最先出现的字节在左侧。最左边的位是第一个字节的 MSB。最右边的位是第四个字节的 LSB。

3.3 RTPS 类型的映射

3.3.1 全局唯一标识符（GUID）

GUID 是所有 RTPS 实体都存在的属性，用于在 DDS 域中唯一标识它们（请参见 2.2.4.1）。PIM 将 GUID 定义为由能够容纳 12 个字节的 ***GuidPrefix_t*** *prefix* 和能够容纳 4 个字节的 ***EntityId_t*** *entityId* 组成。本节定义了 PSM 如何映射这些结构。

3.3.1.1 GuidPrefix_t 的映射

PSM 将 ***GuidPrefix_t*** 映射到以下结构：

```
typedef octet GuidPrefix_t[12];
```

PIM 定义的保留常量 GUIDPREFIX_UNKNOWN 映射为：

```
#define GUIDPREFIX_UNKNOWN{0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00}
```

3.3.1.2 EntityId_t 的映射

2.2.4.3 声明 ***EntityId_t*** 是参与者内部端点的唯一标识。

PSM 将 ***EntityId_t*** 映射到以下结构：

```
typedef octet OctetArray3[3];
struct
{
```

```
    OctetArray3 entityKey;
    octet entityKind;
};
```

PIM 定义的保留常量 ENTITYID_UNKNOWN 映射为：

```
#define ENTITYID_UNKNOWN {{0x00, 0x00, 0x00}, 0x00}
```

EntityId_t 中的 *entityKind* 字段编码实体的类型（***Participant***, ***Reader***, ***Writer***）以及该实体是否是内置实体（由协议完全预定义，自动实例化），用户定义的实体（由协议定义 ，但只能由用户根据应用程序的需要实例化）或供应商专用的实体（由协议的供应商专用扩展定义，因此可以被另一供应商的实现忽略）。

如果未预定义（请参见下文），则中间件实现可以随意选择 ***EntityId_t*** 中的 *entityKey* 字段，只要最终的 ***EntityId_t*** 在参与者中是唯一的即可。

有关对象是内置实体，供应商专用的实体还是用户定义的实体的信息均编码在 *entityKind* 的两个最高有效位中。 这两位设置为：

- 用户定义的实体为“00”。
- 内置实体为“11”。
- 供应商专用的实体为“01”。

有关实体类型的信息被编码在 entityKind 字段的后六位中。表 3-1 提供了本协议 2.2 版本中支持的 *entityKind* 可能值的完整列表。这些已在本协议的主要版本（2）中固定。可以在协议的更高次要版本中添加新实体种类，以便使用新的实体来扩展模型。

表 3-1 EntityId_t 的 entityKind 八位位组

实体种类	用户定义实体	内置实体
未知	0x00	0xc0
参与者	N/A	0xc1
写入者（带关键字）	0x02	0xc2
写入者（不带关键字）	0x03	0xc3
读取者（不带关键字）	0x04	0xc4
读取者（带关键字）	0x07	0xc7

3.3.1.3 预定义的 EntityId

如上所述，内置实体的实体 ID 由 RTPS 协议完全预定义。

PIM 指定参与者的 ***EntityId_t*** 具有预定义的值 ENTITYID_PARTICIPANT（2.2.4.2）。 表 3-2 中显示了所有预定义实体 ID 的对应 PSM 映射。在本协议的主要版本（2）中，这些实体 ID 的含义无法更改，但是以后的次要版本可能会添加额外的保留实体 ID。

表 3-2 由 RTPS 协议完全预定义的 EntityId_t 值

实体	EntityId_t 的对应值（名称=值）
参与者（participant）	ENTITYID_PARTICIPANT = {{00,00,01},c1}
SEDPbuiltinTopicWriter	ENTITYID_SEDP_BUILTIN_TOPIC_WRITER = {{00,00,02},c2}
SEDPbuiltinTopicReader	ENTITYID_SEDP_BUILTIN_TOPIC_READER = {{00,00,02},c7}
SEDPbuiltinPublicationsWriter	ENTITYID_SEDP_BUILTIN_PUBLICATIONS_WRITER = {{00,00,03},c2}

实体	EntityId_t 的对应值（名称=值）
SEDPbuiltinPublicationsReader	ENTITYID_SEDP_BUILTIN_PUBLICATIONS_READER = {{00,00,03},c7}
SEDPbuiltinSubscriptionsWriter	ENTITYID_SEDP_BUILTIN_SUBSCRIPTIONS_WRITER = {{00,00,04},c2}
SEDPbuiltinSubscriptionsReader	ENTITYID_SEDP_BUILTIN_SUBSCRIPTIONS_READER = {{00,00,04},c7}
SPDPbuiltinParticipantWriter	ENTITYID_SPDP_BUILTIN_PARTICIPANT_WRITER = {{00,01,00},c2}
SPDPbuiltinSdpParticipantReader	ENTITYID_SPDP_BUILTIN_PARTICIPANT_READER = {{00,01,00},c7}
BuiltinParticipantMessageWriter	NTITYID_P2P_BUILTIN_PARTICIPANT_MESSAGE_WRITER= {{00,02,00},c2}
BuiltinParticipantMessageReader	ENTITYID_P2P_BUILTIN_PARTICIPANT_MESSAGE_READE= {{00,02,00},c7}

3.3.1.4 协议 2.2 版本中不推荐使用的 EntityId

协议 2.2 版本中使用的发现协议不建议使用表 3-3 中所示的 EntityId。除非协议 2.2 之前的版本具有相同的含义，否则本协议的将来版本不应使用这些 EntityId。希望发现早期版本的实现应利用这些 EntityId。

表 3-3 协议 2.2 版本中不推荐使用的 EntityId

实体	对应的 entityId
客户端（Client）	0x05
服务器（Server）	0x06
writerApplications	{{00,00,01},c2}
readerApplications	{{00,00,01},c7}
writerClients	{{00,00,05},c2}
readerClients	{{00,00,05},c7}
writerServices	{{00,00,06},c2}
readerServices	{{00,00,06},c7}
writerManagers	{{00,00,07},c2}
readerManagers	{{00,00,07},c7}
writerApplicationsSelf	{{00,00,08},c2}

3.3.1.5 GUID_t 的映射

PSM 将 GUID_t 映射到以下结构：

```
struct GUID_t
{
    GuidPrefix_t guidPrefix;
    EntityId_t entityId;
```

};

3.2.4 规定，所有在同一 DomainParticipant 内的 RTPS 实体共享相同的 *guidPrefix*。 此外，2.2.4.2 声明只要 DDS 域内的每个 DomainParticipant 具有唯一的 *guidPrefix*，实现者就可以自由选择 *guidPrefix*。PIM 限制了这种自由。

为了符合此规范，RTPS 协议的实现应将 *guidPrefix* 的前两个字节设置为与其分配的 *vendorId* 相匹配（请参见 2.3.3.1.3）。这样可以确保 *guidPrefix* 在 DDS 域中保持唯一，即使使用多种协议实现也是如此。换句话说，RTPS 协议的实现可以自由使用它们认为适当的任何技术来为 *guidPrefix* 生成唯一值，只要它们满足以下约束：

guidPrefix[0] = vendorId[0]

guidPrefix[1] = vendorId[1]

RTPS 2.x 协议的未来版本也应遵循此规则来生成 *guidPrefix*。

如前所述，设置这前两个字节的值的唯一目的是保证跨多种实现生成唯一的 *guidPrefix*。此值不应用于其他目的。这样可以确保所做的更改不会破坏与本协议先前版本的互操作性。

在 3.4.4 中进一步描述了保留 *vendorId* 的使用。

PIM 定义的保留常量 GUID_UNKNOWN 映射为：

#define GUID_UNKNOWN{ GUIDPREFIX_UNKNOWN, ENTITYID_UNKNOWN }

3.3.2 子消息或内置主题数据中出现的类型的映射

表 3-4 指定了由 PIM 引入的那些类型的 PSM 映射，这些类型出现在协议发送的消息中。无需映射虚拟机专用的类型，但这些类型不会出现在消息中。

表 3-4 线上出现的值类型的 PSM 映射

类型	PSM 映射

Time_t	***类型映射***
	struct Time_t
	{
	long seconds; // time in seconds
	unsigned long fraction; // time in sec/2^32
	};
	时间的表示是由 IETF 网络时间协议（NTP）标准（IETF RFC 1305）定义的。在此表示形式中，时间使用以下公式通过秒和秒的分数表示：
	time = seconds + (fraction / 2^(32))
	时间原点由保留值 TIME_ZERO 表示，并且对应于 1970 年 1 月 1 日的 Unix 首要纪元 0h。
	
	保留值的映射：
	#define TIME_ZERO {0, 0}
	#define TIME_INVALID {-1, 0xffffffff}
	#define TIME_INFINITE {0x7fffffff, 0xffffffff}

VendorId_t	***类型映射*** typedef octet OctetArray2[2];

类型	PSM 映射

	struct VendorId_t
	{
	OctetArray2 vendorId;
	};
	
	保留值的映射：
	#define VENDORID_UNKNOWN {0,0}

SequenceNumber_t	***类型映射***
	struct SequenceNumber_t
	{
	long high;
	unsigned long low;
	};
	使用此结构，64 位序列号为：
	seq_num = *high* * 2^32 + *low*
	
	保留值的映射：
	#define SEQUENCENUMBER_UNKNOWN {-1,0}

FragmentNumber_t	***类型映射***
	struct FragmentNumber_t
	{
	unsigned long value;
	};

Locator_t	***类型映射***
	typedef octet OctetArray16[16];
	struct Locator_t
	{
	long kind;
	unsigned long port;
	OctetArray16 address;
	};
	如果 Locator_t *kind* 为 LOCATOR_KIND_UDPv4，则该地址包含 IPv4 地址。在这种情况下，地址的前 12 个八位位组必须为零。后四个八位位组用于存储 IPv4 地址。IPv4 地址的点记法“a.b.c.d”与其在 Locator_t 的 *address* 字段中的表示之间映射为：
	address = {0,0,0,0,0,0,0,0,0,0,0,0,a,b,c,d}
	如果 Locator_t *kind* 为 LOCATOR_KIND_UDPv6，则该地址包含 IPv6 地址。IPv6 地址通常使用速记十六进制表示法，该十六进制表示法将 *address* 字段中的 16 个八位字节一对一映射。例如，IPv6 地址“FF00:4501:0:0:0:0:0:32”的表示为：
	address = {0xff,0,0x45,0x01,0,0,0,0,0,0,0,0,0,0,0,0x32}

类型	PSM 映射
	保留值的映射： #define LOCATOR_INVALID \ {LOCATOR_KIND_INVALID, LOCATOR_PORT_INVALID, LOCATOR_ADDRESS_INVALID} #define LOCATOR_KIND_INVALID -1 #define LOCATOR_ADDRESS_INVALID {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0} #define LOCATOR_PORT_INVALID 0 #define LOCATOR_KIND_RESERVED 0 #define LOCATOR_KIND_UDPv4 1 #define LOCATOR_KIND_UDPv6 2

TopicKind_t	*类型映射*
	struct TopicKind_t
	{
	long value;
	};
	保留值的映射：
	#define NO_KEY 1
	#define WITH_KEY 2

ReliabilityKind_t	*类型映射*
	struct ReliabilityKind_t
	{
	long value;
	};
	保留值的映射：
	#define BEST_EFFORT 1
	#define RELIABLE 3

Count_t	*类型映射*
	struct Count_t
	{
	long value;
	};

ProtocolVersion_t	*类型映射* struct ProtocolVersion_t { octet major;

类型	PSM 映射
	octet minor; }; *保留值的映射：* #define PROTOCOLVERSION_1_0 {1,0} #define PROTOCOLVERSION_1_1 {1,1} #define PROTOCOLVERSION_2_0 {2,0} #define PROTOCOLVERSION_2_1 {2,1} #define PROTOCOLVERSION_2_2 {2,2} #define PROTOCOLVERSION PROTOCOLVERSION_2_2 *遵循本文档版本的实现实现协议版本 2.2（主要版本= 2，次要版本= 2）。*

KeyHash	*类型映射*
	typedef octet OctetArray16[16];
	struct KeyHash_t
	{
	OctetArray16 value;
	};

StatusInfo_t	*类型映射*
	typedef octet OctetArray4[4];
	struct StatusInfo_t
	{
	OctetArray4 value;
	};

ParameterId_t	*类型映射*
	struct ParameterId_t
	{
	short value;
	};

ContentFilterProperty_t	*类型映射*
	typedef string<256> String256;
	typedef sequence<string> StringSequence;
	struct ContentFilterProperty_t
	{
	String256 contentFilteredTopicName;
	String256 relatedTopicName;
	String256 filterClassName;
	string filterExpression;
	StringSequence expressionParameters;
	};

ContentFilterInfo_t	*类型映射* typedef sequence<long> FilterResult_t;

类型	PSM 映射

	typedef long FilterSignature_t[4];
	typedef sequence<FilterSignature_t> FilterSignatureSequence;
	
	struct ContentFilterInfo_t
	{
	FilterResult_t filterResult;
	FilterSignatureSequence filterSignatures;
	};

Property_t	***类型映射***
	struct Property_t
	{
	string name;
	string value;
	};

EntityName_t	***类型映射***
	struct EntityName_t
	{
	string name;
	};

OriginalWriterInfo_t	***类型映射***
	struct OriginalWriterInfo_t
	{
	GUID_t originalWriterGUID;
	SequenceNumber_t originalWriterSN;
	ParameterList originalWriterQos;
	};

BuiltinEndpointSet_t	***类型映射*** typedef unsigned long BuiltinEndpointSet_t; *而且有* #define DISC_BUILTIN_ENDPOINT_PARTICIPANT_ANNOUNCER 0x00000001 << 0; #define DISC_BUILTIN_ENDPOINT_PARTICIPANT_DETECTOR 0x00000001 << 1; #define DISC_BUILTIN_ENDPOINT_PUBLICATION_ANNOUNCER 0x00000001 << 2; #define DISC_BUILTIN_ENDPOINT_PUBLICATION_DETECTOR 0x00000001 << 3; #define DISC_BUILTIN_ENDPOINT_SUBSCRIPTION_ANNOUNCER 0x00000001 << 4; #define DISC_BUILTIN_ENDPOINT_SUBSCRIPTION_DETECTOR 0x00000001 << 5;

类型	PSM 映射
	#define DISC_BUILTIN_ENDPOINT_PARTICIPANT_PROXY_ANNOUNCER 0x00000001 << 6; #define DISC_BUILTIN_ENDPOINT_PARTICIPANT_PROXY_DETECTOR 0x00000001 << 7; #define DISC_BUILTIN_ENDPOINT_PARTICIPANT_STATE_ANNOUNCER 0x00000001 << 8; #define DISC_BUILTIN_ENDPOINT_PARTICIPANT_STATE_DETECTOR 0x00000001 << 9; #define BUILTIN_ENDPOINT_PARTICIPANT_MESSAGE_DATA_WRITER 0x00000001 << 10; #define BUILTIN_ENDPOINT_PARTICIPANT_MESSAGE_DATA_READER 0x00000001 << 11;

除了 PIM 引入的类型之外，UDP PSM 还引入了表 3-5 列出的其他类型。

表 3-5 UDP PSM 引入的其他类型

类型	描述和 PSM 映射

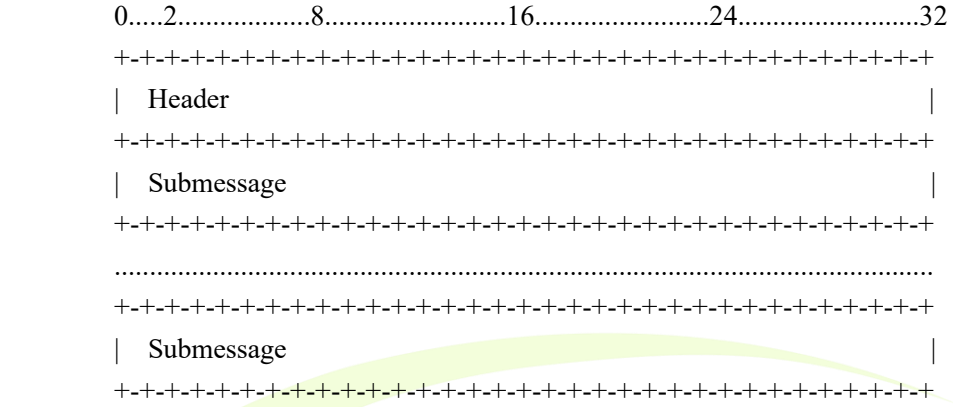
LocatorUDPv4_t	***描述***
	Locator_t 的特殊化，用于使用更紧凑的表示形式来保存 UDP IPv4 定位器。
	等效于 locator_t，种类设置为 LOCATOR_KIND_UDPv4。
	只需要能够容纳一个 IPv4 地址和一个端口号。
	协议保留以下值：
	LOCATORUDPv4_INVALID
	类型映射
	struct LocatorUDPv4_t
	{
	unsigned long address;
	unsigned long port;
	};
	
	IPv4 地址的点记法“a.b.c.d”与其表示为无符号长整数之间的映射如下：
	address = (((a*256 + b)*256) + c)*256 + d
	
	保留值的映射：
	#define LOCATORUDPv4_INVALID {0, 0}

3.4 RTPS 消息的映射

3.4.1 整体结构

PIM 中的子小节 2.3.3 将消息的总体结构定义为由**报文头（Header）**和可变数量的**子消息（Submessages）**组成。

PSM 将每个子消息根据消息的开头在 32 位边界上对齐。



消息的长度众所周知。该长度不是由 RTPS 协议显式发送的，而是用来发送消息的底层传输方式的一部分。对于 UDP/IP，消息的长度就是 UDP 有效负载的长度。

3.4.2 PIM 子消息元素的映射

每个 RTPS 子消息都是根据一组预定义的原子构件（称为“子消息元素”）构建的，如 2.3.5 中所定义。本小节描述了 PIM 定义的每个 **SubmessageElement** 的 PSM 映射。

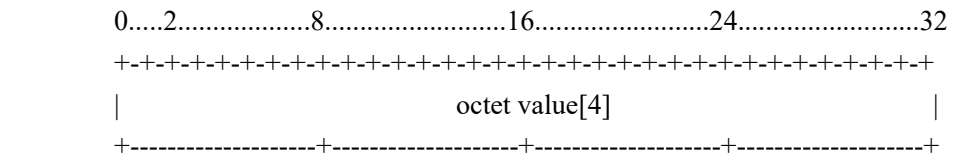
3.4.2.1 EntityId

以下 IDL 定义给出了 2.3.5.1 中定义的 **EntityId** SubmessageElement 的 PSM 映射：

```
typedef EntityId_t EntityId;
```

遵循 CDR 编码规范，**EntityId** SubmessageElement 的线上内容表示为：

EntityId：



3.4.2.2 GuidPrefix

以下 IDL 定义给出了 2.3.5.1 中定义的 **GuidPrefix** SubmessageElement 的 PSM 映射：

typedef GuidPrefix_t GuidPrefix;

遵循 CDR 编码规范，**GuidPrefix** SubmessageElement 的线上内容表示为：

GuidPrefix：

```
0.....2.....8.....16.....24.....32
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
+                                                               +
|                          octet value[12]                      |
+                                                               +
|                                                               |
+---------------+---------------+---------------+---------------+
```

3.4.2.3 VendorId

以下 IDL 定义给出了 2.3.5.2 中定义的 **VendorId** SubmessageElement 的 PSM 映射：

typedef VendorId_t VendorId;

遵循 CDR 编码规范，**VendorId** SubmessageElement 的线上内容表示为：

VendorId：

```
0.....2.....8.....16
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|      octet vendorId [2]       |
+---------------+---------------+
```


3.4.2.4 ProtocolVersion

以下 IDL 定义给出了 2.3.5.3 中定义的 **ProtocolVersion** SubmessageElement 的 PSM 映射：

typedef ProtocolVersion_t ProtocolVersion;

遵循 CDR 编码规范，**ProtocolVersion** SubmessageElement 的线上内容表示为：

ProtocolVersion：

```
0.....2.....8.....16
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|   octet major   |   octet minor   |
+---------------+---------------+
```

3.4.2.5 SequenceNumber

以下 IDL 定义给出了 2.3.5.4 中定义的 **SequenceNumber** SubmessageElement 的 PSM 映射：

typedef SequenceNumber_t SequenceNumber;

遵循 CDR 编码规范，**SequenceNumber** SubmessageElement 的线上内容表示为：

SequenceNumber：

```
0.....2.....8.....16.....24.....32
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                long                high                |
+-------------------+-------------------+-------------------+-------------------+
|                unsigned long       low                 |
+-------------------+-------------------+-------------------+-------------------+
```

3.4.2.6 SequenceNumberSet

PSM 将 2.3.5.5 中定义的 **SequenceNumberSet** SubmessageElement 映射到以下结构:

```
typedef sequence<long, 8> LongSeq8;
struct SequenceNumberSet
{
    SequenceNumber_t bitmapBase;
    LongSeq8 bitmap;
};
```

上面的结构提供了一种紧凑的表示形式，可对多达 256 个序列号的集合进行编码。***SequenceNumberSet*** 的表示形式包括集合中的第一个序列号（*bitmapBase*）和最多 256 位的 *bitmap*。*bitmap* 中的位数由 numBits 表示。*bitmap* 中每个位的值表明将位的偏移量与 *bitmapBase* 相加获得的 SequenceNumber 在 ***SequenceNumberSet*** 中是否包含（包含则 bit = 1，不包含则 bit = 0）。

更准确地说，当且仅当满足以下两个条件时，***SequenceNumber***“*seqNum*”才属于 ***SequenceNumberSet***“*seqNumSet*”：

seqNumSet.bitmapBase<=seqNum<seqNumSet.bitmapBase+

seqNumSet.numBits(bitmap[deltaN/32] & (1 << (31 - deltaN%32))) == (1 << (31 -deltaN%32))

其中

deltaN = seqNum - seqNumSet.bitmapBase

有效的 ***SequenceNumberSet*** 必须满足以下条件：

- bitmapBase> = 1
- 0 <numBits <= 256
- 包含相关位的 M =（numBits + 31）/ 32 个 long.

本文档对特定的 *bitmap* 使用以下表示法：

bitmapBase/numBits:bitmap

在 *bitmap* 中，与序列号 *bitmapBase* 对应的位在左侧。末尾的“0”位可以表示为一个“0”。

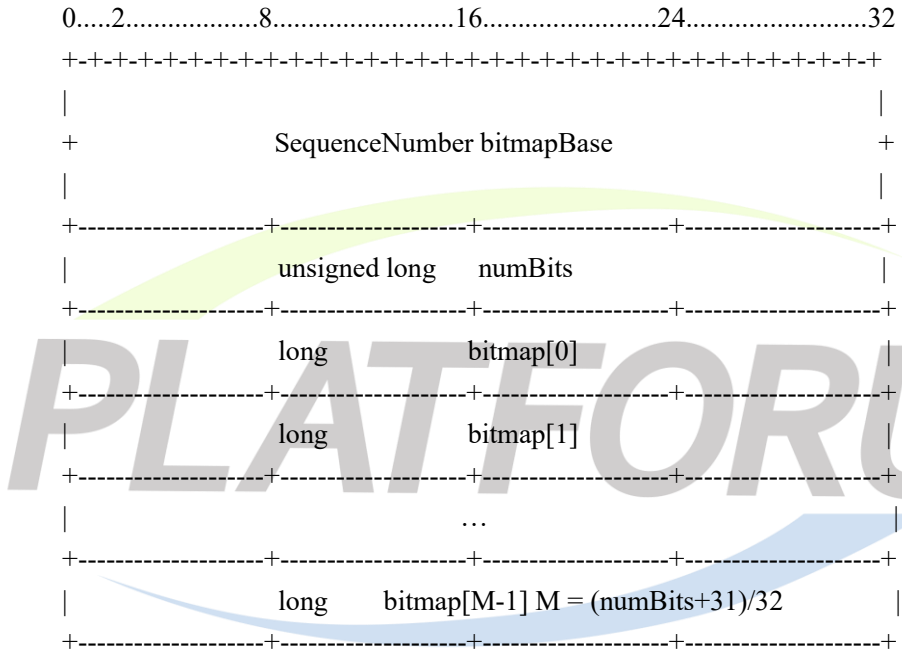
例如，在 *bitmap*“1234/12:00110”中，bitmapBase = 1234 和 numBits = 12。这些位适用于序列号：

表 3-6 bitmap 示例：“1234/12:00110”的含义

类型	描述和 PSM 映射
1234	0
1235	0
1236	1
1237	1
1238-1245	0

SequenceNumberSet SubmessageElement 的线上内容表示为：

SequenceNumberSet：



numBits 字段对有效位的数量和位图元素的数量进行编码。由于此优化，此 SubmessageElement 不遵循 CDR 编码。

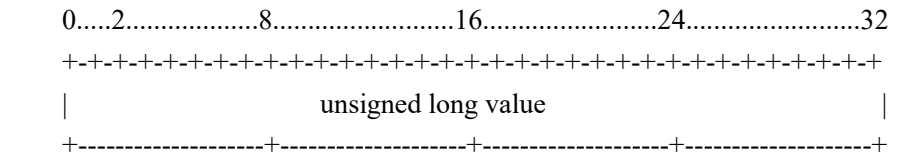
3.4.2.7 FragmentNumber

以下 IDL 定义给出了 2.3.5.6 中定义的 **FragmentNumber** SubmessageElement 的 PSM 映射：

```
typedef FragmentNumber_t FragmentNumber;
```

遵循 CDR 编码规范，**FragmentNumber** SubmessageElement 的线上内容表示为：

FragmentNumber：



3.4.2.8 FragmentNumberSet

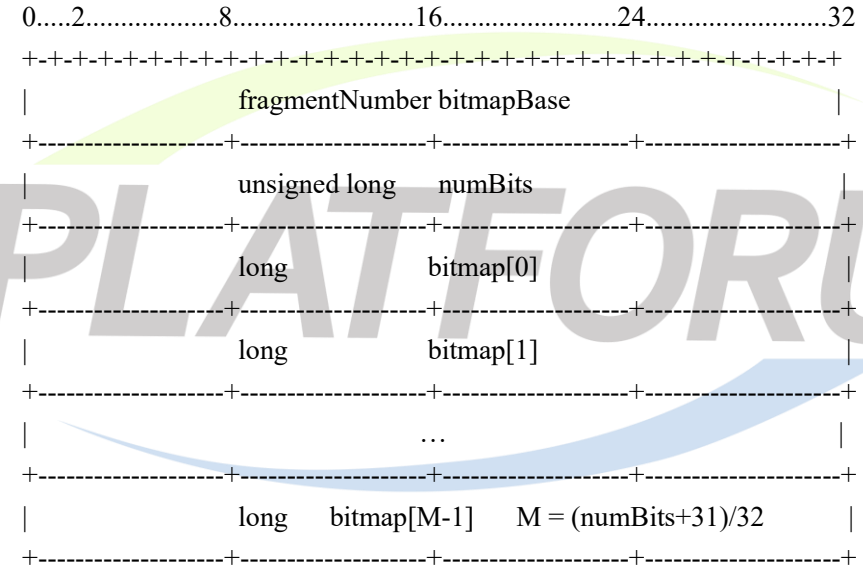
PSM 将 2.3.5.7 中定义的 **FragmentNumberSet** SubmessageElement 映射到以下结构：

```
typedef sequence<long, 8> LongSeq8;
struct FragmentNumberSet
{
    FragmentNumber_t bitmapBase;
    LongSeq8 bitmap;
};
```

上面的结构提供了一个紧凑的表示形式，可对一组最多 256 个分片号进行编码。**FragmentNumberSet** 的表示形式包括集合中的第一个分片编号（*bitmapBase*）和最多 256 位的 *bitmap*。解释与 **SequenceNumberSet** 的解释一致。

FragmentNumberSet SubmessageElement 的线上内容表示为：

FragmentNumberSet：



numBits 字段对有效位的数量和 *bitmap* 元素的数量进行编码。由于进行了优化，因此此 SubmessageElement 不遵循 CDR 编码。

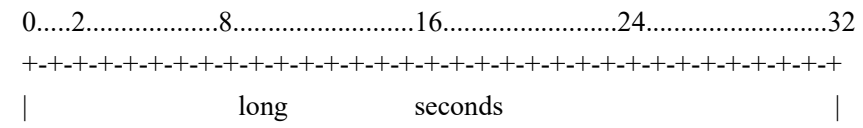
3.4.2.9 Timestamp

以下 IDL 定义给出了 2.3.5.8 中定义的 **Timestamp** SubmessageElement 的 PSM 映射：

```
typedef Time_t Timestamp;
```

遵循 CDR 编码规范，**Timestamp** SubmessageElement 的线上内容表示为：

Timestamp：



```
+-------------------+-------------------+-------------------+-------------------+
|                   unsigned long   fraction                                    |
+-------------------+-------------------+-------------------+-------------------+
```

3.4.2.10 LocatorList

以下 IDL 定义给出了 2.3.5.11 中定义的 **LocatorList** SubmessageElement 的 PSM 映射：

```
typedef sequence<Locator_t, 8> LocatorList;
```

遵循 CDR 编码规范，**LocatorList** SubmessageElement 的线上内容表示为：

LocatorList：

```
0.....2.....8.....16.....24.....32
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                   unsigned long         numLocators                 |
+-------------------+-------------------+-------------------+-------------------+
|                   Locator_t             locator_1                   |
~                                 ...                                 ~
|                   Locator_t             locator_numLocators         |
+-------------------+-------------------+-------------------+-------------------+
```

其中每个 Locator_t 具有以下线上内容表示：

```
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                   long                  kind                        |
+-------------------+-------------------+-------------------+-------------------+
|                   unsigned long         port                        |
+-------------------+-------------------+-------------------+-------------------+
|                                                                       |
+                                                                       +
|                                                                       |
+                   octet                 address[16]                 +
|                                                                       |
+                                                                       +
|                                                                       |
+-------------------+-------------------+-------------------+-------------------+
```

3.4.2.11 ParameterList

ParameterList 包含带有结尾标记的参数列表。**ParameterList** 中的每个参数都相对于 **ParameterList** 的开始以 4 字节对齐。

每个参数的 IDL 表示为：

```
typedef short ParameterId_t
struct Parameter
{
    ParameterId_t parameterId;
```

```
    short length;
    octet value[length]; // Pseudo-IDL: array of non-const length
};
```

parameterId 标识参数的类型。

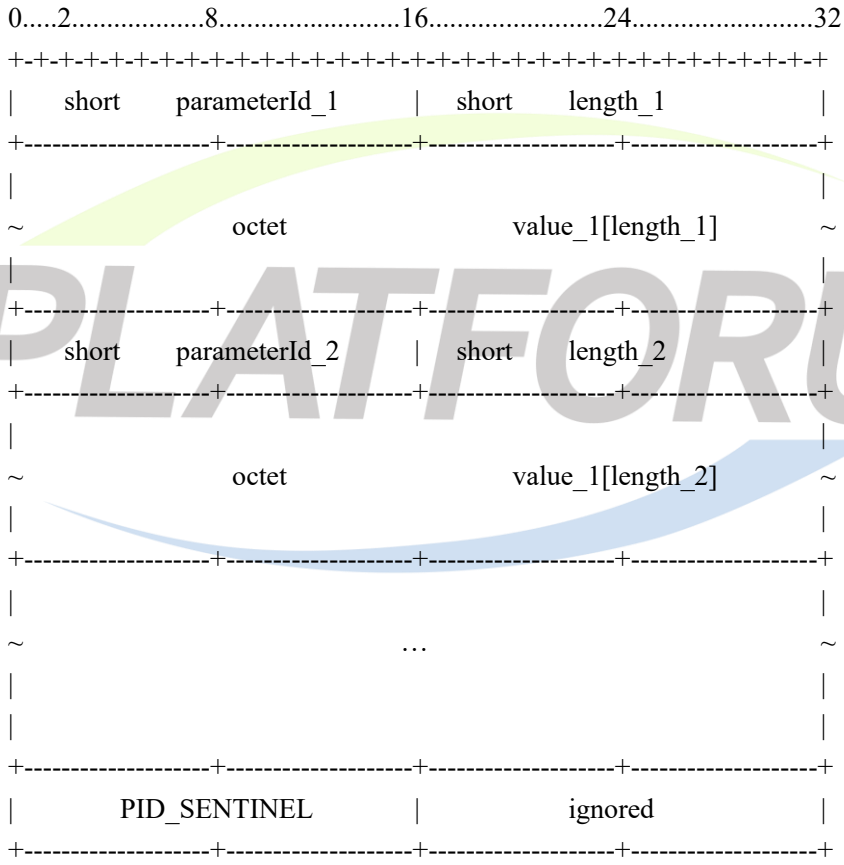
length 编码跟随 *length* 的八位字节数，以达到下一个参数的 ID（或标记的 ID）。因为每个 *parameterId* 都以 4 字节为边界开始，所以 *length* 始终是 4 的倍数。

value 包含与指定的 *parameterId* 对应的参数类型的 CDR 封装。为了实现对齐，为每个参数在逻辑上重置 CDR 流（即不需要初始填充）。

ParameterList 可以包含多个 *parameterId* 值相同的参数。这用于提供此类参数的值的集合。

对 **ParameterList** 进行封装使扩展协议和引入新参数成为可能，并且仍然能够与协议的早期版本保持互操作性。

ParameterList 的线上内容表示为：



有两个用于封装的 *parameterId* 的预定义值：

```
#define PID_PAD (0)

#define PID_SENTINEL (1)
```

PID_SENTINEL 用于终止参数列表，并且忽略其长度。PID_PAD 用于强制后续参数的对齐，并且其长度可以是任意值（只要是 4 的倍数）。

本协议 2.2 版本中 *parameterId* 可能值的完整集合显示在 3.6.3 中。

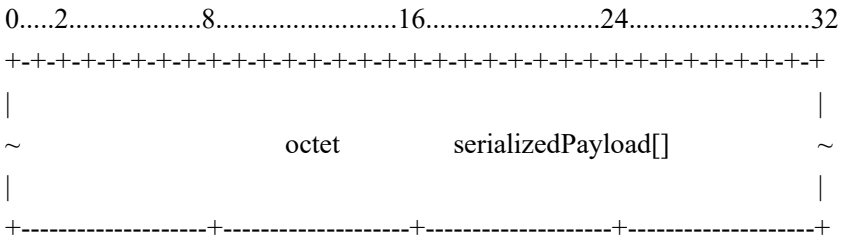
3.4.2.12 SerializedPayload

SerializedPayload SubmessageElement 包含应用程序定义的数据对象的值或唯一标识该数据对象的关键字值的序列化表示。

用于将应用程序级数据类型封装到序列化字节流中的过程的规范严格来说并不是 RTPS 协议的一部分。但是出于互操作性的目的，所有实现都必须使用一致的表示形式（请参见第 4 章“数据封装”）。

SerializedPayload 的线上内容表示为：

SerializedPayload：



请注意使用 CDR 时，原始类型必须与其长度对齐。例如，长整数必须从 4 字节边界开始。从 CDR 流的开头开始计算边界。

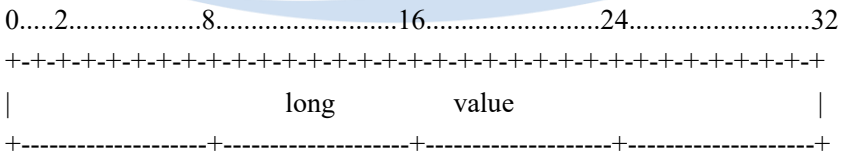
3.4.2.13 Count

PSM 将 2.3.5.10 中定义的 **Count** SubmessageElement 映射到以下结构：

typedef Count_t Count;

遵循 CDR 编码规范，**Count** SubmessageElement 的线上内容表示为：

Count：



3.4.3 其他 SubmessageElement

除了 PIM 引入的 SubmessageElements，UDP PSM 还引入了以下附加的 SubmessageElements。

3.4.3.1 long valueLocatorUDPv4

LocatorUDPv4 SubmessageElement 与包含单个 LOCATOR_KIND_UDPv4 类型的定位器的 **LocatorList** SubmessageElement 相同。引入 LocatorUDPv4 可以在 IPv4 上使用 UDP 时提供更紧凑的表示形式。

表 3-7 LocatorUDPv4 SubmessageElement 的结构

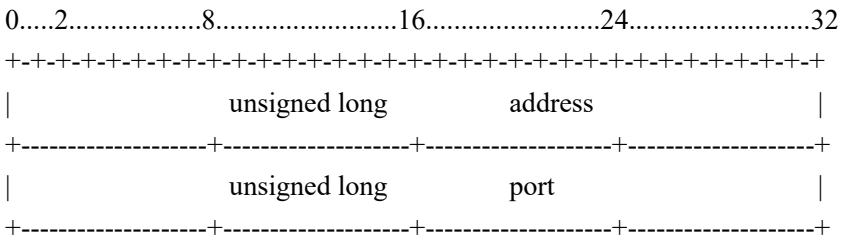
字段	类型	含义
value	LocatorUDPv4_t	单个 IPv4 地址和端口。

PSM 将 **LocatorUDPv4** SubmessageElement 映射到以下结构：

typedef LocatorUDPv4_t LocatorUDPv4;

遵循 CDR 编码规范，**LocatorUDPv4** SubmessageElement 的线上内容表示为：

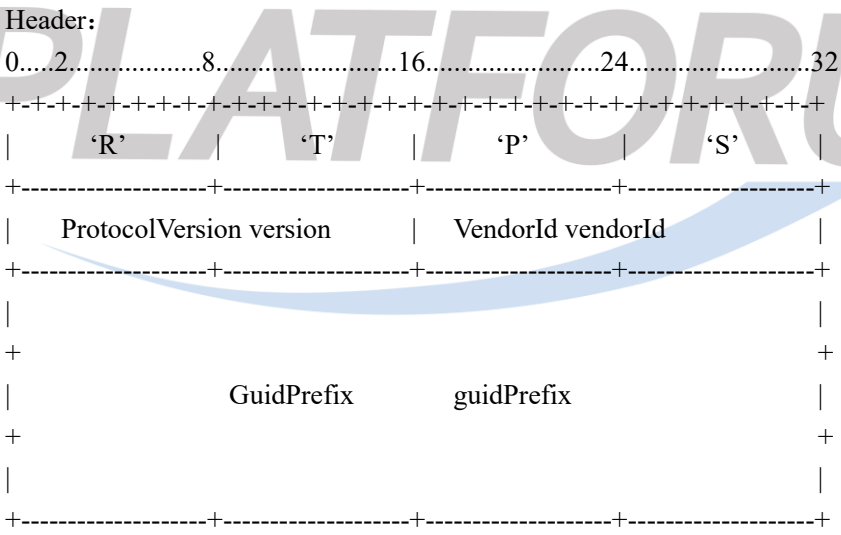
LocatorUDPv4：



3.4.4 RTPS Header 的映射

PIM 中的子节 2.3.7 规定所有消息都应包含一个 RTPS Header。 RTPS Header 的 PSM 映射如下所示：

Header：



报文头（Header）的结构在本协议的主要版本（2）中无法更改。

RTPS 标头包含 *vendorId* 字段（请参见 2.3.5.2）。为了符合 DDS 互操作性规范，供应商必须具有保留的供应商 ID 并使用它。有关在哪里可以找到供应商 ID 的当前列表以及如何请求分配新 ID 的详细信息，请参见 2.3.3.1.3。

3.4.5 RTPS 子消息的映射

3.4.5.1 子消息头（Submessage Header）

PIM 中的子节 2.3.3.2 将所有子消息的结构定义为由前导的 **SubmessageHeader** 和紧随其后的可变数量的 **SubmessageElements** 组成。

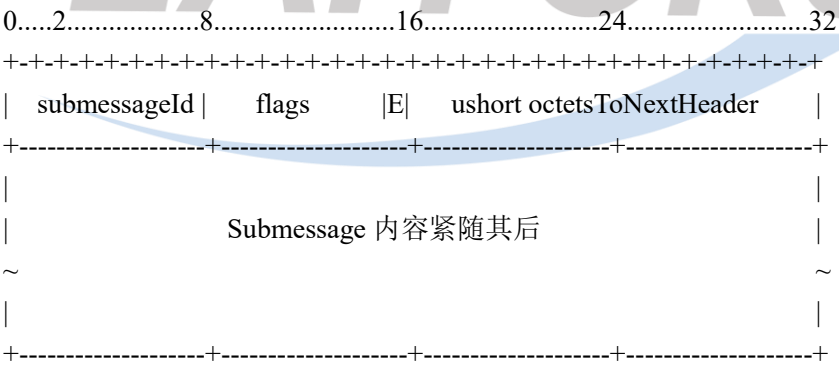
PSM 将 **SubmessageHeader** 映射为以下结构：

```
struct SubmessageHeader
{
    octet submessageId;
    octet flags;
    unsigned short submessageLength; /* octetsToNextHeader */
};
```

使用 3.2.3 中定义的字节流表示形式，*submessageLength* 定义为从 Submessage 内容的开始到下一个 Submessage 头的开始的八位字节数。以下如此定义，UDP PSM 的剩余部分会将 *submessageLength* 称为 *octetsToNextHeader*。参见 3.4.5.1.3。

遵循 CDR 编码规范，**SubmessageHeader** 的线上内容表示为：

SubmessageHeader：



在协议的主要版本（2）中，此常规结构无法更改。以下子节更详细地讨论 **SubmessageHeader** 的每个成员。

3.4.5.1.1 SubmessageId

该八位位组标识子消息的类型。ID 为 0x00 至 0x7f（包括 0x00）的子消息是特定于协议的。它们被定义为 RTPS 协议的一部分。2.2 版定义了以下子消息：

```
enum SubmessageKind
{
    PAD                 = 0x01, /* Pad */
    ACKNACK             = 0x06, /* AckNack */
    HEARTBEAT           = 0x07, /* Heartbeat */
    GAP                 = 0x08, /* Gap */
    INFO_TS             = 0x09, /* InfoTimestamp */
    INFO_SRC            = 0x0c, /* InfoSource */
    INFO_REPLY_IP4      = 0x0d, /* InfoReplyIp4 */
    INFO_DST            = 0x0e, /* InfoDestination */
    INFO_REPLY          = 0x0f, /* InfoReply */
    NACK_FRAG           = 0x12, /* NackFrag */
    HEARTBEAT_FRAG      = 0x13, /* HeartbeatFrag */
    DATA                = 0x15, /* Data */
    DATA_FRAG           = 0x16, /* DataFrag */
};
```

在此主要版本（2）中，无法修改子消息 ID 的含义。可以在更高的次要版本中添加其他子消息。ID 为 0x80 到 0xff（包括 0）的子消息是特定于供应商的；它们不会在协议的将来版本中定义。它们的解释取决于遇到 Submessage 时当前的 *vendorId*。

3.4.5.1.2 flags

PIM 中的小节 2.3.3.2 将 *EndiannessFlag* 定义为存在于所有 Submessages 中的标志，该标志指明用于编码 Submessage 的字节序。PSM 将 *EndiannessFlag* 标志映射到标志的最低有效位（LSB）。因此这一位始终存在于所有子消息中，并表示用于对子消息中的信息进行编码的字节序。*EndiannessFlag* 用文字“E”表示。E = 0 表示大端，E = 1 表示小端。

EndiannessFlag 的值可以从以下表达式获得：

E = SubmessageHeader.flags & 0x01

标志中的其他位的解释取决于 Submessage 的类型。

在以下对子消息的描述中，字符“X”用于指示在协议的 2.2 版本中未使用的标志。RTPS 2.2 版的实现应在发送时将其设置为零，而在接收时将其忽略。协议的较高次要版本可以使用这些标志。

3.4.5.1.3 octetsToNextHeader

此字段的表示形式是 CDR 无符号短整型数据（ushort）。

如果 *octetsToNextHeader*> 0，则它是从 Submessage 内容的第一个八位位组到下一个 Submessage 报文头的第一个八位位组之间的八位组数（如果 Submessage 不是消息中的最后一个 Submessage），或者消息中剩余的八位字节数（如果 Submessage 是消息中的最后一个 Submessage）。消息的解释器可以知道这两种情况，因为它知道消息的总长度。

如果 *octetsToNextHeader* == 0 且 Submessage 的类型不是 PAD 或 INFO_TS，则 Submessage 是消息中的最后一个 Submessage，并一直到消息的末尾。如果它们是消息中的最后一个子消息，则可以发送大于 64k（可以存储在 *octetsToNextHeader* 字段中的大小）的子消息。

如果 *octetsToNextHeader* == 0 且 Submessage 的类型为 PAD 或 INFO_TS，则下一个 Submessage 报文头在当前 Submessage 报文头之后立即开始，或者 PAD 或 INFO_TS 是消息中的最后一个 Submessage。

3.4.5.2 AckNack 子消息

PIM 中的子条款 2.3.7.1 定义了 AckNack 子消息的逻辑内容。PSM 将 AckNack 子消息映射到以下线上内容表示形式：

AckNack：

```
0.....2.....8.....16.....24.....32
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|   ACKNACK   | X|X|X|X|X|F|E|          octetsToNextHeader          |
+-------------------+-------------------+-------------------+-------------------+
|                EntityId                   readerId                           |
+-------------------+-------------------+-------------------+-------------------+
|                EntityId                   writerId                           |
+-------------------+-------------------+-------------------+-------------------+
|                                                                               |
~                SequenceNumberSet      readerSNState                          ~
|                                                                               |
+-------------------+-------------------+-------------------+-------------------+
|                Count                      count                              |
+-------------------+-------------------+-------------------+-------------------+
```

3.4.5.2.1子消息报文头中的标志

除 *EndiannessFlag* 外，**AckNack** 子消息还引入了 *FinalFlag*（38 页上的“内容”）。 PSM 将 *FinalFlag* 标志映射到标志的第二个最低有效位（LSB）。

FinalFlag 用文字“F”表示。F = 1 表示读取者不需要写入者的回应。F = 0 表示写入者必须响应 AckNack 消息。

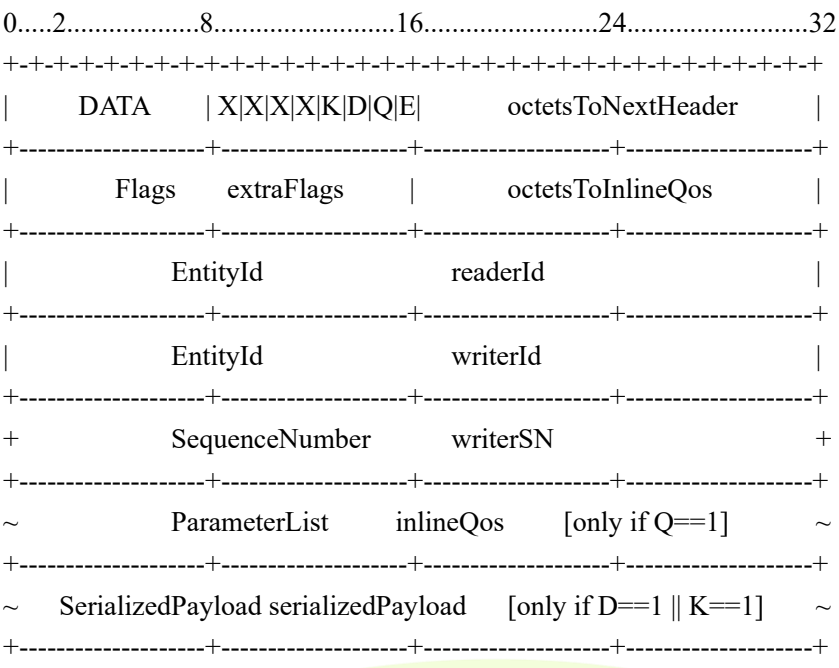
可以从以下表达式获得 *FinalFlag* 的值：

F = SubmessageHeader.flags & 0x02

3.4.5.3 Data 子消息

PIM 中的子节 2.3.7.2 定义了 **Data** 子消息的逻辑内容。PSM 将 **Data** 子消息映射为以下线上内容表示形式：

Data：



3.4.5.3.1 子消息报文头中的标志

除了 *EndiannessFlag* 外，**Data** 子消息还引入了 *InlineQosFlag*，*DataFlag* 和 *Key*（请参阅第 39 页的“内容”）。 PSM 将这些标志映射如下：

*InlineQosFlag*用文字“Q”表示。Q=1 表示 **Data** 子消息包含 **inlineQos** SubmessageElement。可以从以下表达式获得 InlineQosFlag 的值：

Q = SubmessageHeader.flags & 0x02

DataFlag 用文字“D”表示。可以从以下表达式中获取 *DataFlag* 的值。

D = SubmessageHeader.flags & 0x04

KeyFlag 用文字“K”表示。*KeyFlag* 的值可以从以下表达式中获取。

K = SubmessageHeader.flags & 0x08

DataFlag 与 *KeyFlag* 结合解释如下：

- D = 0 和 K = 0 表示没有 **serializedPayload** SubmessageElement。
- D = 1 和 K = 0 表示 **serializedPayload** SubmessageElement 包含序列化的数据。
- D = 0 和 K = 1 表示 **serializedPayload** SubmessageElement 包含序列化的 Key。
- 在本协议版本中，D = 1 和 K = 1 是无效的组合。

3.4.5.3.2 extraFlags

extraFlags 字段为子消息报文头中提供了额外的 16 位标志空间，不仅仅是原来的 8 位标志。这些额外的位将支持协议的扩展而不会损害向后兼容性。

此版本的协议应将 *extraFlags* 中的所有位设置为零。

3.4.5.3.3 octetsToInlineQos

此字段表示的是 CDR 无符号短整型数据（ushort）。

octetsToInlineQos 字段包含从紧随该字段之后的第一个八位字节开始到 **inlineQos** SubmessageElement 的第一个八位字节开始的八位字节数。如果 **inlineQos** SubmessageElement 不存在（即未设置 InlineQosFlag），则 *octetsToInlineQos* 包含到 *inlineQos* 之后的下一个字段的偏移量。

处理接收到子消息的协议的实现应始终使用 *octetsToInlineQos* 跳过它不希望或不理解的任何子消息头元素，并继续处理 **inlineQos** SubmessageElement（如果 *inlineQos* 不存在，则继续处理 *inlineQos* 之后的第一个子消息元素）。该规则是必需的，以便接收方能够与使用协议将来版本的发送方进行互操作，该发送方可能在 *inlineQos* 之前包含其他子消息头。

3.4.5.4 DataFrag 子消息

PIM 中的子节 2.3.7.3 定义了 **DataFrag** 子消息的逻辑内容。PSM 将 DataFrag 子消息映射为以下线上内容表示形式：

DataFrag：

```
0.....2.....8.....16.....24.....32
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|   DATA_FRAG | X|X|X|X|X|K|Q|E|          octetsToNextHeader          |
+---------------+-----------------+-----------------+-----------------+
|          Flags      extraFlags      |       octetsToInlineQos       |
+---------------+-----------------+-----------------+-----------------+
|                   EntityId                    readerId                |
+-------------------+-------------------+-------------------+-------------------+
|                   EntityId                    writerId                |
+-------------------+-------------------+-------------------+-------------------+
+                   SequenceNumber          writerSN                    +
+-------------------+-------------------+-------------------+-------------------+
|                   FragmentNumber          fragmentStartingNum         |
+-------------------+-------------------+-------------------+-------------------+
|   ushort fragmentsInSubmessage  |          ushort fragmentSize          |
+-------------------+-------------------+-------------------+-------------------+
|                   unsigned long               sampleSize                |
+-------------------+-------------------+-------------------+-------------------+
~                   ParameterList         inlineQos       [only if Q==1]      ~
+-------------------+-------------------+-------------------+-------------------+
~                       SerializedPayload serializedPayload                   ~
+-------------------+-------------------+-------------------+-------------------+
```

3.4.5.4.1 子消息报文头中的标志

除了 *EndiannessFlag* 外，**DataFrag** 子消息还引入了 *KeyFlag* 和 *InlineQosFlag*（请参阅

第 41 页的“内容”)。 PSM 将这些标志映射如下：

InlineQosFlag 用文字“Q”表示。Q = 1 表示 **DataFrag** 子消息包含 **inlineQos** SubmessageElement。

可以从以下表达式获得 *InlineQosFlag* 的值：

Q = SubmessageHeader.flags & 0x02

KeyFlag 用文字“K”表示。

可以从以下表达式获得 *KeyFlag* 的值：

K = SubmessageHeader.flags & 0x04

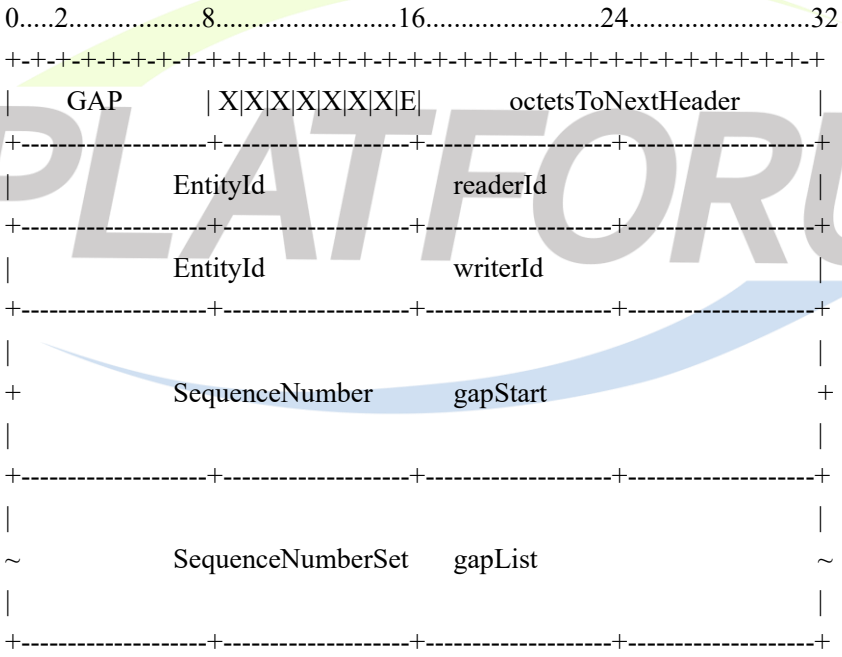
K = 0 表示 **serializedPayload** SubmessageElement 包含序列化的 Data。

K = 1 表示 **serializedPayload** SubmessageElement 包含序列化的 Key。

3.4.5.5 Gap 子消息

PIM 中的子节 2.3.7.4 定义了 **Gap** 子消息的逻辑内容。PSM 将 **Gap** 子消息映射为以下线上内容表示形式：

Gap：



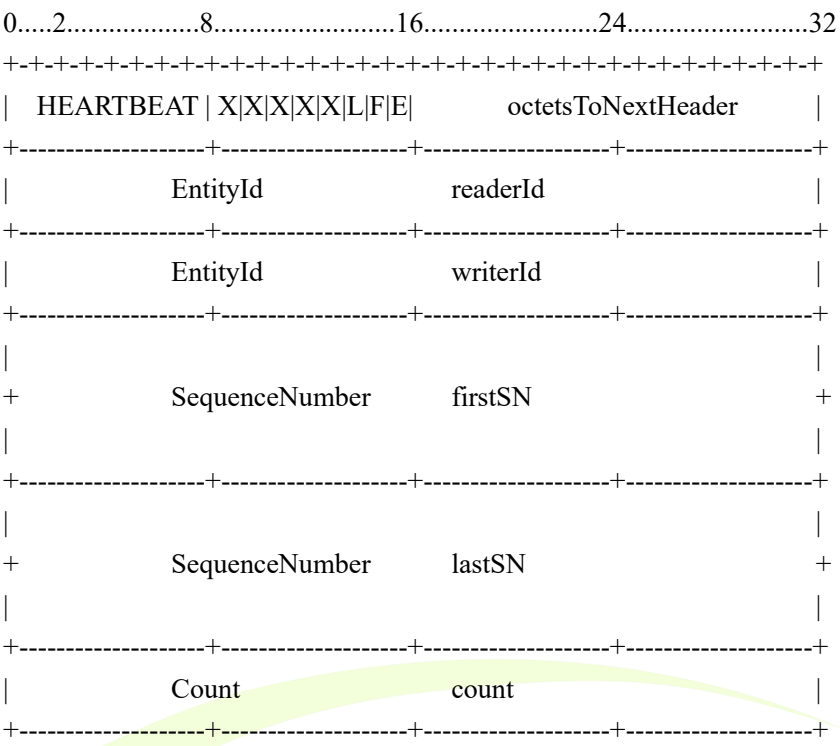
3.4.5.5.1子消息报文头中的标志

除了 *EndiannessFlag* 外，此子消息没有其他标志。

3.4.5.6 HeartBeat 子消息

PIM 中的子节 2.3.7.5 定义了 **HeartBeat** 子消息的逻辑内容。PSM 将 **HeartBeat** 子消息映射为以下线上内容表示形式：

HeartBeat：



3.4.5.6.1子消息报文头中的标志

除 *EndiannessFlag* 外，**HeartBeat** 子消息还引入了 *FinalFlag* 和 *LivelinessFlag*（第 44 页上的“内容”）。 PSM 将 *FinalFlag* 标志映射到标志的第二个最低有效位（LSB），将 *LivelinessFlag* 映射到标志的第三个最低有效位（LSB）。

FinalFlag 用文字“F”表示。 F = 1 表示 ***Writer*** 不需要 ***Reader*** 的响应。F = 0 表示 ***Reader*** 必须响应 **HeartBeat** 消息。

可以从以下表达式获得 *FinalFlag* 的值：

F = SubmessageHeader.flags & 0x02

LivelinessFlag 用文字“L”表示。L = 1 表示与 RTPS ***Reader*** 关联的 DDS DataReader 应该“手动”刷新与消息的 RTPS ***Writer*** 关联的 DDS DataWriter 的存活性。

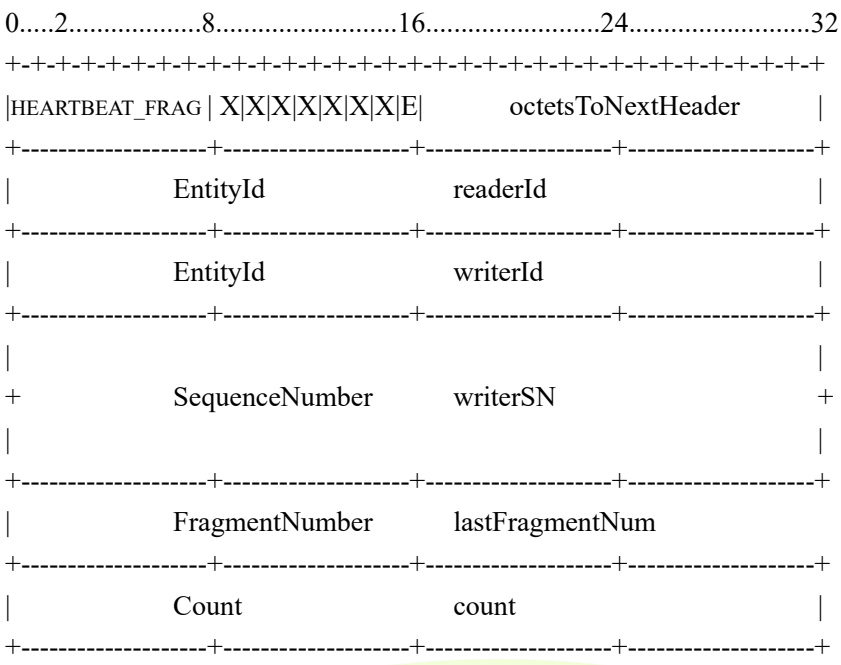
LivelinessFlag 的值可以从以下表达式获得：

L = SubmessageHeader.flags & 0x04

3.4.5.7 HeartBeatFrag 子消息

PIM 中的子节 2.3.7.6 定义 **HeartBeatFrag** 子消息的逻辑内容。PSM 将 **HeartBeatFrag** 子消息映射到以下线上内容表示形式：

HeartBeatFrag：



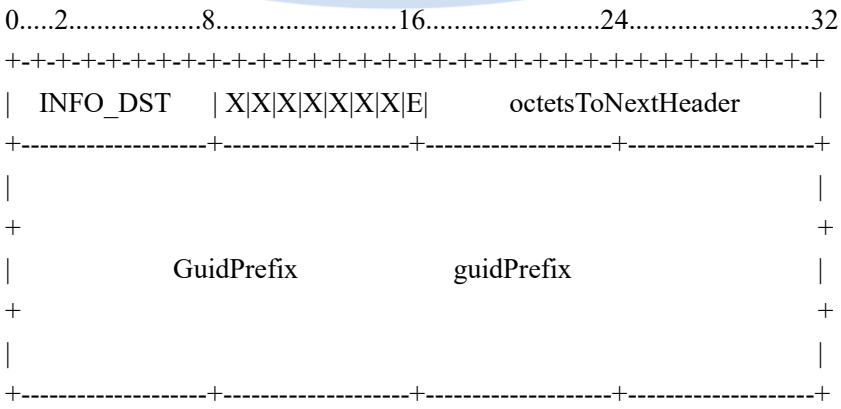
3.4.5.7.1 子消息报文头中的标志

除了 *EndiannessFlag* 外，此子消息没有其他标志。

3.4.5.8 InfoDestination 子消息

PIM 中的子节 2.3.7.7 定义了 **InfoDestination** 子消息的逻辑内容。PSM 将 **InfoDestination** 子消息映射为以下线上内容表示形式：

InfoDestination：



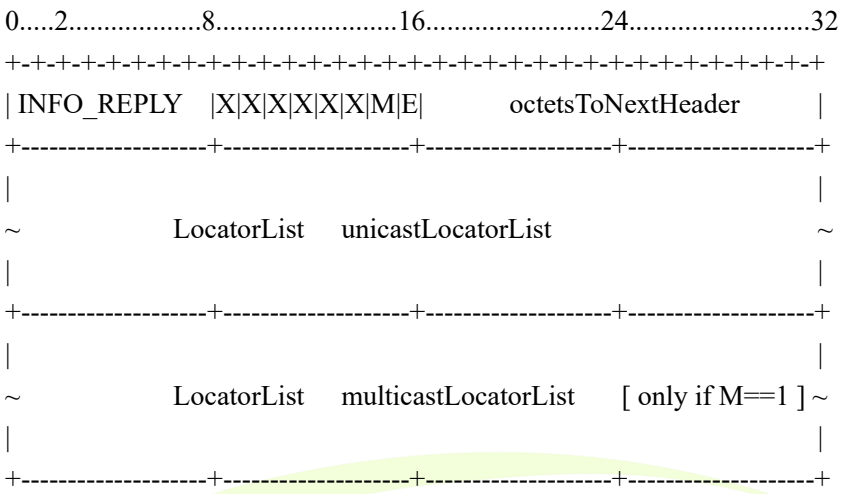
3.4.5.8.1 子消息报文头中的标志

除了 *EndiannessFlag* 外，此子消息没有其他标志。

3.4.5.9 InfoReply 子消息

PIM 中的子节 2.3.7.8 定义了 **InfoReply** 子消息的逻辑内容。PSM 将 **InfoReply** 子消息映射为以下线上内容表示形式：

InfoReply：



3.4.5.9.1 子消息报文头中的标志

除 *EndiannessFlag* 外，**InfoReply** 子消息还引入了 *MulticastFlag*（第 48 页的“内容”）。PSM 将 *MulticastFlag* 标志映射到标志字段的第二个最低有效位（LSB）。

MulticastFlag 用文字“M”表示。M = 1 表示 **InfoReply** 还包含一个 *multicastLocatorList*。

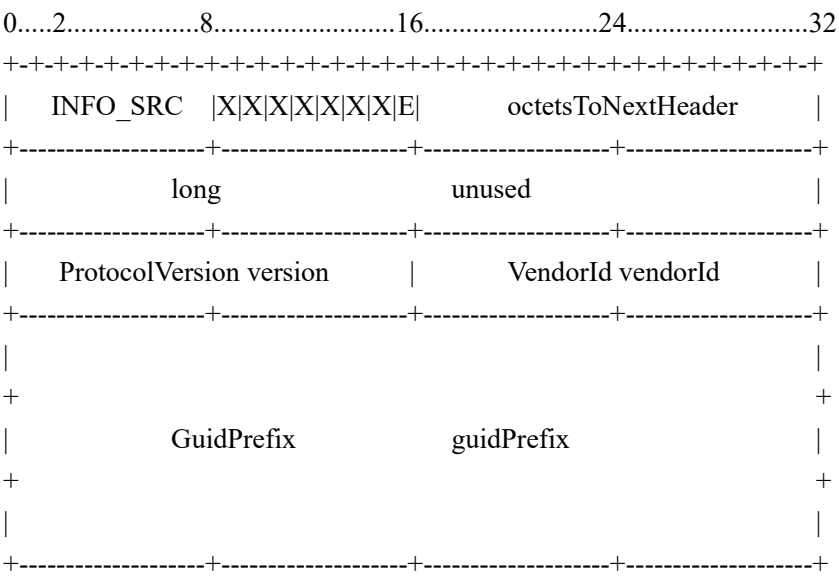
可以从以下表达式获得 MulticastFlag 的值：

M = SubmessageHeader.flags & 0x02

3.4.5.10 InfoSource 子消息

PIM 中的子节 2.3.7.9 定义了 **InfoSource** 子消息的逻辑内容。PSM 将 **InfoSource** 子消息映射为以下线上内容表示形式：

InfoSource：



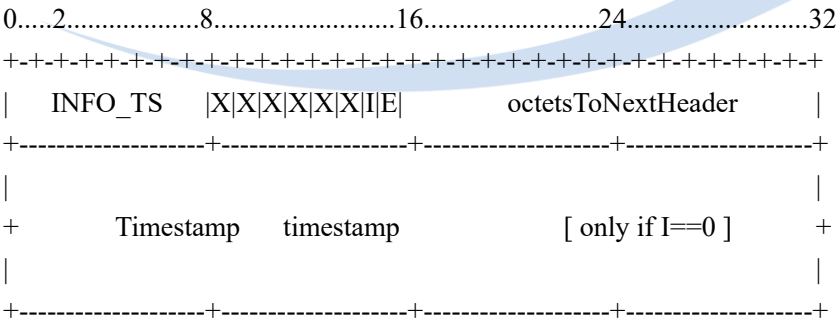
3.4.5.10.1 子消息报文头中的标志

除了 *EndiannessFlag* 外，此子消息没有其他标志。

3.4.5.11 InfoTimestamp 子消息

PIM 中的子节 2.3.7.10 定义了 **InfoTimestamp** 子消息的逻辑内容。PSM 将 **InfoTimestamp** 子消息映射为以下线上内容表示形式：

InfoTimestamp：



3.4.5.11.1 子消息报文头中的标志

除了 *EndiannessFlag* 外，**InfoTimestamp** 子消息还引入了 *InvalidateFlag*（第 50 页上的“内容”）。PSM 将 *InvalidateFlag* 标志映射到标志字段的第二个最低有效位（LSB）。

InvalidateFlag 用文字“I”表示。I = 0 表示 **InfoTimestamp** 还包含一个时间戳。 I = 1 表示后续子消息不应被认为具有有效的时间戳。

可以从以下表达式获得 *InvalidateFlag* 的值：

I = SubmessageHeader.flags & 0x02

3.4.5.12 Pad 子消息

PIM 中的子节 2.3.7.12 定义了 **Pad** 子消息的逻辑内容。PSM 将 **Pad** 子消息映射到以下线上内容表示形式：

Pad：

```
0.....2.....8.....16.....24.....32
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|          PAD          |X|X|X|X|X|X|E|          octetsToNextHeader          |
+-----------------------+-----------------------+-----------------------+-----------------------+
```

3.4.5.12.1 子消息报文头中的标志

除了 *EndiannessFlag* 外，此子消息没有其他标志。

3.4.5.13 NackFrag 子消息

PIM 中的子节 2.3.7.11 定义了 **NackFrag** 子消息的逻辑内容。PSM 将 **NackFrag** 子消息映射为以下线上内容表示形式：

NackFrag：

```
0.....2.....8.....16.....24.....32
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|   NACK_FRAG | X|X|X|X|X|X|E|          octetsToNextHeader          |
+-------------------+-------------------+-------------------+-------------------+
|                   EntityId                    readerId                        |
+-------------------+-------------------+-------------------+-------------------+
|                   EntityId                    writerId                        |
+-------------------+-------------------+-------------------+-------------------+
|                                                                               |
+               SequenceNumber              writerSN                            +
|                                                                               |
+-------------------+-------------------+-------------------+-------------------+
|                                                                               |
~               FragmentNumberSet           fragmentNumberState                 ~
|                                                                               |
+-------------------+-------------------+-------------------+-------------------+
|                   Count                       count                           |
+-------------------+-------------------+-------------------+-------------------+
```

3.4.5.13.1 子消息报文头中的标志

除了 *EndiannessFlag* 外，此子消息没有其他标志。

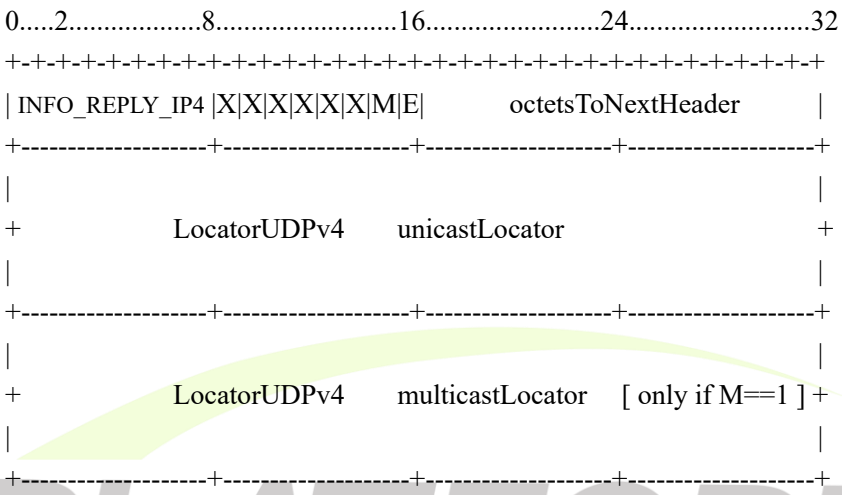
3.4.5.14 InfoReplyIp4 子消息（特定于 PSM）

InfoReplyIp4 子消息是 UDP PSM 引入的另一个子消息。

它的使用和解释与包含单个 LOCATOR_KIND_UDPv4 类型的单个单播和可能一个单独的多播定位符的 **InfoReply** 子消息相同。出于效率考虑而提供它，可以代替 **InfoReply** 子消息来使用它以提供更紧凑的表示形式。

PSM 将 **InfoReplyIp4** 子消息映射为以下线上内容表示形式：

InfoReplyIp4：



3.4.5.14.1 子消息报文头中的标志

除了 *EndiannessFlag* 外，**InfoReplyIp4** 子消息还引入了 *MulticastFlag*。PSM 将 *MulticastFlag* 标志映射到标志字段的第二个最低有效位（LSB）。

MulticastFlag 用文字“M”表示。M = 1 表示 **InfoReplyIp4** 还包含一个 *multicastRLocator*。

可以从以下表达式获得 *MulticastFlag* 的值：

M = SubmessageHeader.flags & 0x02

3.5 RTPS 消息封装

在 UDP/IP 上使用 RTPS 时，一条消息就是一个 UDP/IP 数据报的内容（有效载荷）。

3.6 RTPS 协议的映射

3.6.1 默认定位器

3.6.1.1 发现流量

发现流量是参与者和端点发现协议生成的流量。对于简单发现协议（SPDP 和 SEDP），

发现流量是内置端点之间交换的流量。

使用众所周知的端口（请参阅 2.5.3.4）配置 SPDP 内置端点。UDP PSM 将这些众所周知端口映射到表 3-8 中列出的端口号表达式。

表 3-8 内置端点使用的端口

发现流量类型	SPDP 已知端口	默认端口号表达式
多播	SPDP_WELL_KNOWN_MULTICAST_PORT	PB + DG * *domainId* + d0
单播	SPDP_WELL_KNOWN_UNICAST_PORT	PB + DG * *domainId* + d1 + PG * *participantId*

其中

domainId = DDS 域标识符

PartitionId = 参与者标识符

PB，DG，d0，d1 = 可调参数（定义如下）

domainId 和 *participantId* 标识符用于避免同一节点上的参与者之间的端口冲突。同一节点上和同一域中的每个参与者都必须使用唯一的 *participantId*。在多播的情况下，同一域中的所有参与者共享相同的端口号，因此在端口号表达式中不使用 *participantId* 标识符。

为了简化 SPDP 的配置，理想情况下 *participantId* 值从 0 开始，并且对于同一节点和同一域中的每个其他参与者递增。这样，对于给定的域，参与者可以通过宣布该节点上与 *participantId* 从 0 到 N-1 相对应的端口号，来向给定节点上的 N 个远程参与者宣布他们的存在。

SEDP 内置端点使用的默认端口与 SPDP 使用的默认端口匹配。如果节点选择不对 SEDP 使用默认端口，则它可以将新端口号作为 SPDP 期间交换信息的一部分。

3.6.1.2 用户流量

用户流量是用户定义的端点（即非内置端点）之间交换的流量。它适用于与发现无关的所有流量。默认情况下用户定义的端点使用表 3-9 中列出的端口号表达式。

表 3-9 用户定义的端点使用的端口

用户流量类型	默认端口号表达式
多播	PB + DG * *domainId* + d2
单播	PB + DG * *domainId* + d3 + PG * *participantId*

用户定义的端点可以选择不使用默认端口。在这种情况下，远程端点会获取端口号，作为简单端点发现协议（SEDP）期间交换的信息的一部分。

3.6.1.3 默认端口号

端口号表达式使用以下参数：

DG = DomainId 增加

PG = ParticipantId 增加

PB = 端口基本号

d0，d1，d2，d3 = 附加偏移

实现必须公开这些参数，以便用户可以自定义它们。

为了达到开箱即用的互操作性，必须使用以下默认值：

PB = 7400

DG = 250

PG = 2

d0 = 0

d1 = 10

d2 = 1

d3 = 11

考虑 UDP 端口号限制为 64K，上述默认值允许使用大约 230 个域，每个域每个节点最多 120 个参与者。

3.6.1.4 简单参与者发现协议（SPDP）的默认设置

使用 SPDP 时，每个参与者都将宣告发送到预先配置的定位器列表。上面讨论了在配置这些定位器时使用哪些端口。本小节描述了实现即插即用互操作性所需的所有其余设置。

3.6.1.4.1默认多播地址

为了实现即插即用的互操作性，默认的预先配置定位器列表必须包含以下多播定位器（假设为 UDPv4）：

DefaultMulticastLocator =

{LOCATOR_KIND_UDPv4, “239.255.0.1”, PB + DG * domainId + d0}

所有参与者必须宣告并监听此多播地址。

SPDPbuiltinParticipantWriter.readerLocators CONTAINS DefaultMulticastLocator

SPDPbuiltinParticipantReader.multicastLocatorList CONTAINS DefaultMulticastLocator

3.6.1.4.2默认宣告速率

发送 SPDP 定期通知的默认速率等于 30 秒。

SPDPbuiltinParticipantWriter.resendPeriod = {30, 0};

3.6.2 内置端点的数据表示

3.6.2.1 ParticipantMessageData 内置端点的数据表示

PIM（2.4）中的行为模块定义了数据类型 *ParticipantMessageData*。此类型是 ***BuiltinParticipantMessageWriter*** 和 ***BuiltinParticipantMessageReader*** 内置端点的逻辑内容。

PSM 将 *ParticipantMessageData* 类型映射到以下 IDL：

```
typedef octet OctetArray4[4];
typedef sequence<octet> OctetSeq;
struct ParticipantMessageData
{
    GuidPrefix_t participantGuidPrefix;
    OctetArray4 kind;
    OctetSeq data;
};
```

RTPS 保留 *kind* 字段的以下值：

```
#define PARTICIPANT_MESSAGE_DATA_KIND_UNKNOWN
                                        {0x00, 0x00, 0x00, 0x00}
#define PARTICIPANT_MESSAGE_DATA_KIND_AUTOMATIC_LIVELINESS_UPDATE
                                        {0x00, 0x00, 0x00, 0x01}
#define PARTICIPANT_MESSAGE_DATA_KIND_MANUAL_LIVELINESS_UPDATE
                                        {0x00, 0x00, 0x00, 0x02}
```

RTPS 还保留未设置最高有效位的 *kind* 字段的所有值，以供将来使用。因此：

```
kind.value [0]＆0x80 == 0    //由 RTPS 保留

kind.value [0]＆0x80 == 1    //供应商特定 kind
```

实现可以决定数据字段的最大长度，但必须能够支持至少 128 个字节。

遵循 CDR 编码规范，*ParticipantMessageData* 结构的线上内容表示为：

ParticipantMessageData：

```
0.....2.....8.....16.....24.....32
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                   unsigned long               data.length     |
+-------------------+-------------------+-------------------+-------------------+
|                                                               |
~                   octet[]                     data.value      ~
|                                                               |
+-------------------+-------------------+-------------------+-------------------+
```

3.6.2.2 简单发现协议内置端点

PIM（2.5）中的发现模块定义了数据类型 ***SPDPdiscoveredParticipantData***，***DiscoveredWriterData***，***DiscoveredReaderData*** 和 ***DiscoveredTopicData***。这些类型定义了 RTPS 内置端点之间发送的数据的逻辑内容。

PSM 将这些类型映射到以下 IDL：

```
struct SPDPdiscoveredParticipantData
{
    DDS::ParticipantBuiltinTopicData ddsParticipantData;
    participantProxy participantProxy;
    Duration_t leaseDuration;
};
```

```
struct DiscoveredWriterData
{
    DDS::PublicationBuiltinTopicData ddsPublicationData;
    WriterProxy mWriterProxy;
};

struct DiscoveredReaderData
{
    DDS::SubscriptionBuiltinTopicData ddsSubscriptionData;
    ReaderProxy mReaderProxy;
    ContentFilterProperty_t contentFilterProperty;
};

struct DiscoveredTopicData
{
    DDS::TopicBuiltinTopicData ddsTopicData;
};
```

每个 DDS 内置主题数据类型均由 DDS 规范定义。

使用标准 **Data** 子消息发送发现数据。为了在保留协议版本之间互操作的同时允许 QoS 扩展，**Data** 子消息内 *SerializedData* 的线上内容表示使用 **ParameterList** SubmessageElement 的格式。也就是说，*SerializedData* 将每个 QoS 和其他信息封装在由 *ParameterId* 标识的单独参数中。在每个参数内，使用 CDR 编码封装参数值。

例如为了在 ***SPDPdiscoveredParticipantData*** 中添加供应商专用的端点发现协议（EDP），供应商可以定义供应商专用的 *parameterId* 并将其用于将新参数添加到 ***SPDPdiscoveredParticipantData*** 中包含的 **ParameterList** 中。此 *parameterId* 的存在将表示支持相应的 EDP。由于这是特定于供应商的 *parameterId*，因此其他供应商的实现将忽略该参数及其包含的信息。参数本身将包含使用 CDR 封装的供应商专用的 EDP 所需的任何其他数据。

为了优化，协议的实现可以选择在 **Data** 子消息中不包含参数，前提是协议包含与该 **Data** 子消息中已存在的其他参数冗余的信息。这种优化的结果是实现可以省略表 3-10 中列出的参数的序列化。

表 3-10 ParameterID 子空间

内置端点	可省略的参数	可以找到省略参数的信息的参数
SPDPdiscoveredParticipantData	articipantProxy::guidPrefix	ParticipantBuiltinTopicData::key
DiscoveredReaderData	ReaderProxy::remoteReaderGuid	SubscriptionBuiltinTopicData::key
DiscoveredWriterData	ReaderProxy::remoteWriterGuid	PublicationBuiltinTopicData::key

例如发送包含 ***SPDPdiscoveredParticipantData*** 的 DATA 消息的协议的实现可以省略包含 *guidPrefix* 的参数。如果 *guidPrefix* 不存在于 DATA 消息中，则在接收方的协议实现必须从 DATA 消息中始终存在的“key”参数中计算出该值。

3.6.2.2.1 ParameterId 空间

如 3.4.2.11 中所述，ParameterId 空间为 16 位宽。为了适应特定于供应商的选项和将来对协议的扩展，将 ParameterId 空间划分为多个子空间。表 3-11 列出了 ParameterId 子空间。

表 3-11 ParameterId 子空间

位	值	含义
ParameterId & 8000 (MSB)	0	保留的 ParameterId。
	1	供应商特定的 ParameterId。 其他供应商的实现将无法识别。
ParameterId & 4000	0	如果无法识别 ParameterId，请跳过并忽略该参数。
	1	如果无法识别 ParameterId，则将该参数视为不兼容的 QoS。 在这种情况下，两个实体之间将不会建立任何通信。

第一子空间划分启用特定于供应商的 ParameterId。将来的次要版本的 RTPS 协议可以添加新参数，最大 ParameterId 为 0x7fff。范围 0x8000 到 0xffff 保留给特定于供应商的选择使用，并且本协议任何将来的版本都不会使用。

为了向后兼容，将两个子空间再次细分。如果需要 ParameterId，但不存在 ParameterId，则协议将采用默认值。同样如果存在 ParameterId 但未被识别，则协议将跳过或忽略该参数，或将该参数视为不兼容的 QoS。实际行为取决于 ParameterId 值，请参阅表 3-11。

3.6.2.2.2 ParameterID 值

表 3-12 总结了用于封装内置实体的数据的 ParameterId 列表。表 3-13 列出了每个 parameterID 应用于的实体及其默认值。

表 3-12 ParameterId 值

名称	ID	类型
PID_PAD	***0x0000***	***N/A***
PID_SENTINEL	***0x0001***	***N/A***
PID_USER_DATA	***0x002c***	***UserDataQosPolicy***
PID_TOPIC_NAME	***0x0005***	***string<256>***
PID_TYPE_NAME	***0x0007***	***string<256>***
PID_GROUP_DATA	***0x002d***	***GroupDataQosPolicy***
PID_TOPIC_DATA	***0x002e***	***TopicDataQosPolicy***
PID_DURABILITY	***0x001d***	***DurabilityQosPolicy***
PID_DURABILITY_SERVICE	***0x001e***	***DurabilityServiceQosPolicy***
PID_DEADLINE	***0x0023***	***DeadlineQosPolicy***
PID_LATENCY_BUDGET	***0x0027***	***LatencyBudgetQosPolicy***
PID_LIVELINESS	***0x001b***	***LivelinessQosPolicy***
PID_RELIABILITY	***0x001a***	***ReliabilityQosPolicy***
PID_LIFESPAN	***0x002b***	***LifespanQosPolicy***

名称	ID	类型
PID_DESTINATION_ORDER	*0x0025*	*DestinationOrderQosPolicy*
PID_HISTORY	*0x0040*	*HistoryQosPolicy*
PID_RESOURCE_LIMITS	*0x0041*	*ResourceLimitsQosPolicy*
PID_OWNERSHIP	*0x001f*	*OwnershipQosPolicy*
PID_OWNERSHIP_STRENGTH	*0x0006*	*OwnershipStrengthQosPolicy*
PID_PRESENTATION	*0x0021*	*PresentationQosPolicy*
PID_PARTITION	*0x0029*	*PartitionQosPolicy*
PID_TIME_BASED_FILTER	*0x0004*	*TimeBasedFilterQosPolicy*
PID_TRANSPORT_PRIORITY	*0x0049*	*TransportPriorityQoSPolicy*
PID_PROTOCOL_VERSION	*0x0015*	*ProtocolVersion_t*
PID_VENDORID	*0x0016*	*VendorId_t*
PID_UNICAST_LOCATOR	*0x002f*	*Locator_t*
PID_MULTICAST_LOCATOR	*0x0030*	*Locator_t*
PID_MULTICAST_IPADDRESS	*0x0011*	*IPv4Address_t*
PID_DEFAULT_UNICAST_LOCATOR	*0x0031*	*Locator_t*
PID_DEFAULT_MULTICAST_LOCATOR	*0x0048*	*Locator_t*
PID_METATRAFFIC_UNICAST_LOCATOR	*0x0032*	*Locator_t*
PID_METATRAFFIC_MULTICAST_LOCATOR	*0x0033*	*Locator_t*
PID_DEFAULT_UNICAST_IPADDRESS	*0x000c*	*IPv4Address_t*
PID_DEFAULT_UNICAST_PORT	*0x000e*	*Port_t*
PID_METATRAFFIC_UNICAST_IPADDRESS	*0x0045*	*IPv4Address_t*
PID_METATRAFFIC_UNICAST_PORT	*0x000d*	*Port_t*
PID_METATRAFFIC_MULTICAST_IPADDRESS	*0x000b*	*IPv4Address_t*
PID_METATRAFFIC_MULTICAST_PORT	*0x0046*	*Port_t*
PID_EXPECTS_INLINE_QOS	*0x0043*	*boolean*
PID_PARTICIPANT_MANUAL_LIVELINESS_COUNT	*0x0034*	*Count_t*
PID_PARTICIPANT_BUILTIN_ENDPOINTS	*0x0044*	*unsigned long*
PID_PARTICIPANT_LEASE_DURATION	*0x0002*	*Duration_t*
PID_CONTENT_FILTER_PROPERTY	*0x0035*	*ContentFilterProperty_t*
PID_PARTICIPANT_GUID	*0x0050*	*GUID_t*
PID_PARTICIPANT_ENTITYID	*0x0051*	*EntityId_t*
PID_GROUP_GUID	*0x0052*	*GUID_t*
PID_GROUP_ENTITYID	*0x0053*	*EntityId_t*
PID_BUILTIN_ENDPOINT_SET	*0x0058*	*BuiltinEndpointSet_t*
PID_PROPERTY_LIST	*0x0059*	*sequence<Property_t>*
PID_TYPE_MAX_SIZE_SERIALIZED	*0x0060*	*long*
PID_ENTITY_NAME	*0x0062*	*EntityName_t*
PID_KEY_HASH	*0x0070*	*KeyHash_t*
PID_STATUS_INFO	*0x0071*	*StatusInfo_t*

表 3-13 ParameterId 映射和默认值

名称	字段使用	默认值
PID_PAD	-	*N/A*
PID_SENTINEL	-	*N/A*
PID_USER_DATA	*ParticipantBuiltinTopicData:user_data* *PublicationBuiltinTopicData::user_data* *SubscriptionBuiltinTopicData::user_data*	*参阅 DDS 规范*
PID_TOPIC_NAME	*TopicBuiltinTopicData::name* *PublicationBuiltinTopicData::topic_name* *SubscriptionBuiltinTopicData::topic_name*	*N/A*
PID_TYPE_NAME	*TopicBuiltinTopicData::type_name* *PublicationBuiltinTopicData::type_name* *SubscriptionBuiltinTopicData::type_name*	*N/A*
PID_GROUP_DATA	*PublicationBuiltinTopicData::group_data* *SubscriptionBuiltinTopicData::group_data*	*参阅 DDS 规范*
PID_TOPIC_DATA	*PublicationBuiltinTopicData::topic_data* *SubscriptionBuiltinTopicData::topic_data*	*参阅 DDS 规范*
PID_DURABILITY	*TopicBuiltinTopicData::durability* *PublicationBuiltinTopicData::durability*	*参阅 DDS 规范*
PID_DURABILITY_SERVICE	*TopicBuiltinTopicData::durability_service* *PublicationBuiltinTopicData::durability_service*	*参阅 DDS 规范*
PID_DEADLINE	*TopicBuiltinTopicData::deadline* *PublicationBuiltinTopicData::deadline* *SubscriptionBuiltinTopicData::deadline*	*参阅 DDS 规范*
PID_LATENCY_BUDGET	*TopicBuiltinTopicData::latency_budget* *PublicationBuiltinTopicData::latency_budget* *SubscriptionBuiltinTopicData::latency_budget*	*参阅 DDS 规范*
PID_LIVELINESS	*TopicBuiltinTopicData::liveliness* *PublicationBuiltinTopicData::liveliness* *SubscriptionBuiltinTopicData::liveliness*	*参阅 DDS 规范*
PID_RELIABILITY	*TopicBuiltinTopicData::reliability* *PublicationBuiltinTopicData::reliability* *SubscriptionBuiltinTopicData::reliability*	*参阅 DDS 规范*
PID_LIFESPAN	*TopicBuiltinTopicData::lifespan* *PublicationBuiltinTopicData::lifespan* *SubscriptionBuiltinTopicData::lifespan*	*参阅 DDS 规范*
PID_DESTINATION_ORDER	*TopicBuiltinTopicData::destination_order* *PublicationBuiltinTopicData::destination_order* *SubscriptionBuiltinTopicData::destination_order*	*参阅 DDS 规范*
PID_HISTORY	*TopicBuiltinTopicData::history*	*参阅 DDS 规范*
PID_RESOURCE_LIMITS	*TopicBuiltinTopicData::resource_limits*	*参阅 DDS 规范*
PID_OWNERSHIP	*TopicBuiltinTopicData::ownership*	*参阅 DDS 规范*
PID_OWNERSHIP_STRENGTH	*PublicationBuiltinTopicData::ownership_strength*	*参阅 DDS 规范*
PID_PRESENTATION	*PublicationBuiltinTopicData::presentation*	*参阅 DDS 规范*

名称	字段使用	默认值
PID_PARTITION	*PublicationBuiltinTopicData::partition* *SubscriptionBuiltinTopicData::partition*	*参阅 DDS 规范*
PID_TIME_BASED_FILTER	*SubscriptionBuiltinTopicData::time_based_filter*	*参阅 DDS 规范*
PID_PROTOCOL_VERSION	*ParticipantProxy::protocolVersion*	*N/A*
PID_VENDORID	*ParticipantProxy::vendorId*	*N/A*
PID_UNICAST_LOCATOR	*ReaderProxy::unicastLocatorList* *WriterProxy::unicastLocatorList*	*N/A*
PID_MULTICAST_LOCATOR	*ReaderProxy::multicastLocatorList* *WriterProxy::multicastLocatorList*	*N/A*
PID_MULTICAST_IPADDRESS	*ReaderProxy::multicastLocatorList.address* *WriterProxy::multicastLocatorList.address*	*N/A*
PID_DEFAULT_ UNICAST_LOCATOR	*ParticipantProxy::defaultUnicastLocatorList*	*N/A*
PID_DEFAULT_ MULTICAST_LOCATOR	*ParticipantProxy::defaultMulticastLocatorList*	*N/A*
PID_METATRAFFIC_ UNICAST_LOCATOR	*ParticipantProxy::metatrafficUnicastLocatorList*	*N/A*
PID_METATRAFFIC_ MULTICAST_LOCATOR	*ParticipantProxy::metatrafficMulticastLocatorList*	*N/A*
PID_DEFAULT_ UNICAST_IPADDRESS	*ParticipantProxy::defaultUnicastLocatorList.address*	*N/A*
PID_DEFAULT_ UNICAST_PORT	*ParticipantProxy::defaultUnicastLocatorList.port*	*N/A*
PID_METATRAFFIC_ UNICAST_IPADDRESS	*ParticipantProxy::metatrafficUnicastLocatorList.address*	*N/A*
PID_METATRAFFIC_ UNICAST_PORT	*ParticipantProxy::metatrafficMulticastLocatorList.port*	*N/A*
PID_METATRAFFIC_ MULTICAST_IPADDRESS	*ParticipantProxy::metatrafficMulticastLocatorList.address*	*N/A*
PID_METATRAFFIC_ MULTICAST_PORT	*ParticipantProxy::metatrafficMulticastLocatorList.port*	*N/A*
PID_EXPECTS_INLINE_QOS	*ParticipantProxy::expectsInlineQos*	*FALSE*
PID_PARTICIPANT_MANUAL_ LIVELINESS_COUNT	*ParticipantProxy::manualLivelinessCount*	*N/A*
PID_PARTICIPANT_BUILTIN_ ENDPOINTS	*ParticipantProxy::availableBuiltinEndpoints*	
PID_PARTICIPANT_LEASE_ DURATION	*SPDPdiscoveredParticipantData::leaseDuration*	*{100, 0}*
PID_PARTICIPANT_GUID	*ParticipantBuiltinTopicData::key* *PublicationBuiltinTopicData::participant_key* *SubscriptionBuiltinTopicData::participant_key*	*N/A*
PID_PARTICIPANT_ENTITYID	*保留供协议将来使用*	

名称	字段使用	默认值
PID_GROUP_GUID	*保留供协议将来使用*	
PID_GROUP_ENTITYID	*保留供协议将来使用*	

3.6.3 用于表示内联 QoS 的 ParameterId 定义

PIM（2.3）中的消息模块为 Data（2.3.7.2）和 DataFrag（8.3.7.3）子消息提供了与子消息一致的 QoS 策略的方法。使用 **ParameterList** 封装 QoS 策略。

子小节 2.7.2.1 定义了可以出现在 inlineQos SubmessageElement 中的完整参数集。表 3-14 中列出了相应的 parameterIds 集。

表 3-14 内联 QoS 参数

名称	ID	IDL 内容描述
PID_PAD	*参考表 3-12*	*N/A*
PID_SENTINEL		*N/A*
PID_TOPIC_NAME		*string<256>*
PID_DURABILITY		*DurabilityQosPolicy*
PID_PRESENTATION		*PresentationQosPolicy*
PID_DEADLINE		*DeadlineQosPolicy*
PID_LATENCY_BUDGET		*LatencyBudgetQosPolicy*
PID_OWNERSHIP		*OwnershipQosPolicy*
PID_OWNERSHIP_STRENGTH		*OwnershipStrengthQosPolicy*
PID_LIVELINESS		*LivelinessQosPolicy*
PID_PARTITION		*PartitionQosPolicy*
PID_RELIABILITY		*ReliabilityQosPolicy*
PID_TRANSPORT_PRIORITY		*TransportPriorityQoSPolicy*
PID_LIFESPAN		*LifespanQosPolicy*
PID_DESTINATION_ORDER		*DestinationOrderQosPolicy*
PID_CONTENT_FILTER_INFO	*0x0055*	*ContentFilterInfo_t*
PID_COHERENT_SET	*0x0056*	*SequenceNumber_t*
PID_DIRECTED_WRITE	*0x0057*	*sequence<GUID_t>*
PID_ORIGINAL_WRITER_INFO	*0x0061*	*OriginalWriterInfo_t*
PID_KEY_HASH	*0x0070*	*KeyHash_t*
PID_STATUS_INFO	*0x0071*	*StatusInfo_t*

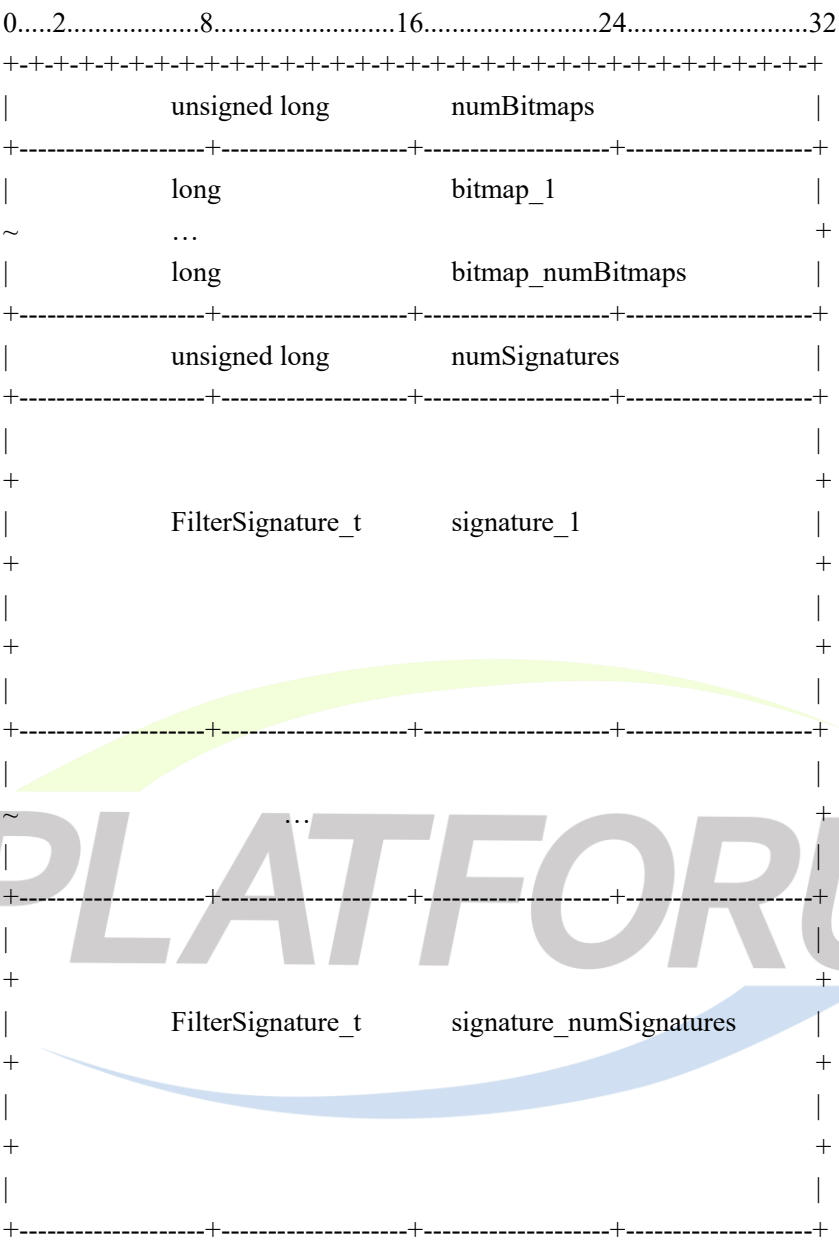
可以内联显示的策略包括 DataWriter QoS 策略的子集（在 3.6.2 中定义的 ParameterId）和一些其他 QoS（为其定义了新的 ParameterId）。

以下小节更详细地描述了这些附加的 QoS。

3.6.3.1 内容过滤器信息（PID_CONTENT_FILTER_INFO）

遵循 CDR 编码规范，***ContentFilterInfo_t***（参见表 3-4）内联 QoS 的线上内容表示为：

ContentFilterInfo_t：



filterResult 成员被编码为位图。位 0（MSB）对应于第一个过滤器签名，位 1 对应于第二个过滤器签名，依此类推。内容过滤器信息内联 QoS 无效，除非出现以下情况：

numBitmaps == ([numSignatures/32] + (numSignatures%32 ? 1 : 0))

位图的解释如下：

表 3-15 内容过滤器信息内联 QoS 中 filterResult 成员的解释

位值	解释
0	样本用相应的过滤器过滤，没有通过。
1	样本通过相应的过滤器过滤并通过。

过滤器的签名被计算为过滤器 ***ContentFilterProperty_t*** 中所有字符串的 128 位 MD5 校验和。更准确地说，所有字符串都组合到以下字符数组中：

[contentFilteredTopicName relatedTopicName filterClassName filterExpression expressionParameters[0] expressionParameters[1] ... expressionParameters[numParams - 1]]

其中每个单独的字符串都包含其 NULL 终止字符。过滤器签名是通过获取上述字符序列的 MD5 校验和来计算的。

3.6.3.2 一致性集（PID_COHERENT_SET）

一致性设置的内联 QoS 参数对 ***SequenceNumber_t*** 使用 CDR 编码。

按照 2.7.5 的定义，属于同一一致性集合的所有 **Data** 和 **DataFrag** 子消息必须包含一致性集合的内联 QoS 参数，其值等于该集合中第一个样本的序列号。

例如假设一个一致的集合包含来自给定 ***Writer*** 的序列号分别为 3、4、5 和 6 的样本更新。通过将一致集性内联 QoS 参数包含值 3 来标识此一致集中的样本。表 3-16 中列出了 ***Writer*** 可用来表示该一致集结尾的一些示例 **Data** 子消息。

表 3-16 表示子集结束的示例数据子消息

数据子消息元素（子集）	示例 1（新的一致集）	示例 2（无一致集）	示例 3（无一致集）
DataFlag	1	0	0
InlineQosFlag	1	1	0
KeyHashSuffix	识别对象	忽略	忽略
writerSN	7	7	7
InlineQos (PID_COHERENT_SET)	7	SEQUENCENUMBER_UNKNOWN	N/A
SerializedData	有效数据	N/A	N/A

3.6.3.3 KeyHash (PID_KEY_HASH)

关键字哈希内联参数包含 *KeyHash_t* 的 CDR 编码。*KeyHash_t* 被定义为一个 16 字节的八位字节数组（请参见表 3-4），因此关键字哈希内联参数仅能应对这 16 字节。

使用下面两种算法中的一种从后续的数据计算得到 *KeyHash_t*，具体取决于数据类型是否保证所有关键字段的顺序 CDR 封装的最大大小小于 128 位（*KeyHash_t* 的大小）。

- 如果保证所有关键字段的顺序 CDR 封装的最大大小小于 128 位，则应将 *KeyHash_t* 计算为序列中所有关键字段的 CDR 大端的封装。在所有关键字段都被封装后，*KeyHash_t* 中任何未填充的位应设置为零。
- 否则，*KeyHash_t* 将被计算为一个 128 位 MD5 摘要（IETF RFC 1321），该序列应用于序列中所有关键字段的 CDR 大端封装。

请注意，要使用的算法取决于数据类型，而不取决于任何特定的数据值。

示例 1.假定以下 IDL 描述的类型：

```
struct TypeWithShortKey
{
    long id;            /*假设定义为关键字段*/
    string name<6>;     /*假设定义为关键字段*/
                        /*其他非关键字段*/
};
```

然后我们知道关键字段的 CDR 封装的最大大小为 15 字节(“id”字段为 4，字符串“name”的长度为 4，字符串的最大长度为 7 字节（包括额外用于终止的字节 NUL）。

在这个例子中，*KeyHash_t* 应该被计算为：

[CDR(id), CDR(name), <零填充到 16 个字节>]

其中 CDR（x）代表该字段的大端 CDR 封装。

这种类型的具体数据值，例如{32，“hello”，...}将被封装为：

```
0……..8…….....16…….....24…….....32
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|   0x00   |   0x00   |   0x00   |   0x20   |
|   0x00   |   0x00   |   0x00   |   0x06   |
|    ‘h’   |    ‘e’   |    ‘l’   |    ‘l’   |
|    ‘o’   |   0x00   |   0x00   |   0x00   |
+----------+----------+----------+----------+
```

请注意为清楚起见，请使用一种符号，其中每个字节都可以表示为十六进制数字（例如 0x20）或字符（例如“h”）；

示例 2.假定以下 IDL 描述的类型：

```
struct TypeWithShortKey
{
    long id;            /*假设定义为关键字段*/
    string name<8>;     /*假设定义为关键字段*/
                        /*其他非关键字段*/
};
```

我们知道，关键字段的 CDR 封装的最大大小为 17 个字节（“id”字段为 4 个字节，字符串“name”的长度为 4 个字节，加上字符串的最大 9 个字节（包括额外用于终止的字节 NUL）。

在这个例子中，*KeyHash_t* 应该被计算为：

MD5 ([CDR(id), CDR (name)])

3.6.3.4 StatusInfo_t（PID_STATUS_INFO）

状态信息参数包含 *StatusInfo_t* 的 CDR 编码。*StatusInfo_t* 定义为 4 字节的八位字节数组（请参见表 3-4），因此状态信息内联参数仅复制这 4 字节。

状态信息参数可能会出现在 **Data** 或 **DataFrag** 子消息中。

StatusInfo_t 应解释为具有如下所示布局的 32 位标志：

```
0.....2.....8.....16.....24.....32
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| X|X|X|X|X|X|X|X|X|X|X|X|X|X|X|X|X|X|X|X|X|X|X|X|X|X|X|X|U|D|
+-------------------+-------------------+-------------------+-------------------+
```

用文字“X”表示的标志在此协议版本中未使用，***Writer*** 应将其设置为零，而 ***Reader*** 则不对其进行解释，以便可以在协议的未来版本中使用它们而不会破坏互操作性。

状态信息中的标志提供有关子消息所引用的数据对象的状态的信息。具体而言，状态信息用于将更改传达给数据对象实例的 ***LifecycleState***。

本协议的当前版本定义了 *DisposeFlag* 和 *UnregisterFlag*。

DisposeFlag 用文字“D”表示。

D = 1 表示 DDS DataWriter 已处理关键字出现在子消息中的数据对象的实例。

UnregisterFlag 用文字“U”表示。

U = 1 表示 DDS DataWriter 已注销关键字出现在子消息中的数据对象的实例。

如果 *DisposeFlag* 和 *UnregisterFlag* 均为零，则关键字显示在子消息中的数据对象在 DDS DataWriter 中具有存活的生命周期状态。

请注意，该协议不需要 DDS DataWriter 传播“注册”方法。因此，DDS DataWriter 可以将“注册”实现为本地方法。由于 DDS DataWriter 注册方法不提供数据值传播，因此 DataReader 对该注册方法的使用将受到限制。

3.6.4 协议 2.2 版本弃用的 ParameterId

本协议的 2.2 版弃用了表 3-17 中所示的 ParameterId。除非协议 2.2 之前的版本具有相同的含义，否则协议的未来版本不应使用这些参数。希望与早期版本进行互操作的实现应发送并处理表 3-17 中的参数。

表 3-17 弃用的 ParameterId 值

名称	ID	历史
PID_PERSISTENCE	***0x0003***	
PID_TYPE_CHECKSUM	***0x0008***	
PID_TYPE2_NAME	***0x0009***	
PID_TYPE2_CHECKSUM	***0x000a***	
PID_EXPECTS_ACK	***0x0010***	
PID_MANAGER_KEY	***0x0012***	
PID_SEND_QUEUE_SIZE	***0x0013***	
PID_RELIABILITY_ENABLED	***0x0014***	
PID_VARGAPPS_SEQUENCE_NUMBER_LAST	***0x0017***	
PID_RECV_QUEUE_SIZE	***0x0018***	
PID_RELIABILITY_OFFERED	***0x0019***	

4 数据封装

4.1 引言

数据封装严格来说不是 RTPS 协议的一部分。如 2.3.5.12 中所述，RTPS 协议与如何封装 **SerializedData** SubmessageElement 中的数据无关。而是由 DDS 类型插件负责数据封装，该插件将对数据进行序列化和反序列化。

但是出于互操作性的目的，来自不同 DDS 实现的类型插件以相同的方式封装数据非常重要。第 4 章定义了所有 DDS 类型插件都将使用的通用数据封装方案。

4.2 数据封装

数据封装的常用方法是 OMG CDR。根据特定的数据类型，实现可能希望使用替代的封装方法。例如 RTPS 内置端点使用 **ParameterList** 封装来交换发现信息。通过 **ParameterList** 封装，可以轻松扩展数据类型，同时保持向后兼容性。添加新的 QoS 值时，此功能很重要。

为了支持多种数据封装方案，需要一些描述封装方案的附加信息。也就是说，**SerializedData** 必须同时包含数据封装方案标识符和实际数据本身。DDS 类型插件在反序列化其余数据之前会解析数据封装方案标识符。

为了实现互操作性，DDS 实现必须至少支持针对应用程序定义的数据类型的 CDR 封装。与内置主题关联的数据的封装必须使用 **ParameterList**，如 3.6.2 中所述。

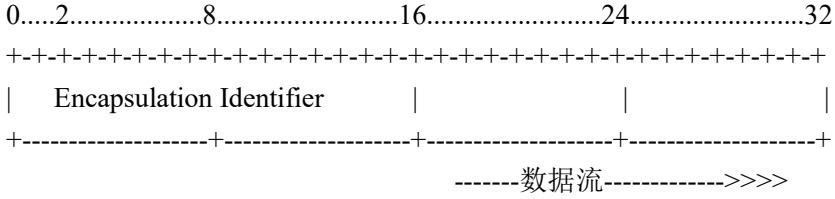
4.2.1 标准数据封装方案

4.2.1.1 通用方法

所有数据封装方案都必须以封装方案标识符开头。

```
typedef octet Identifier[2];
```

标识符占据序列化数据流的前两个八位位组，如下所示：



序列化数据流的其余部分包含实际数据或其他封装特定信息。

表 4-1 列出了当前的预定义数据封装方案。

表 4-1 预定义的数据封装方案

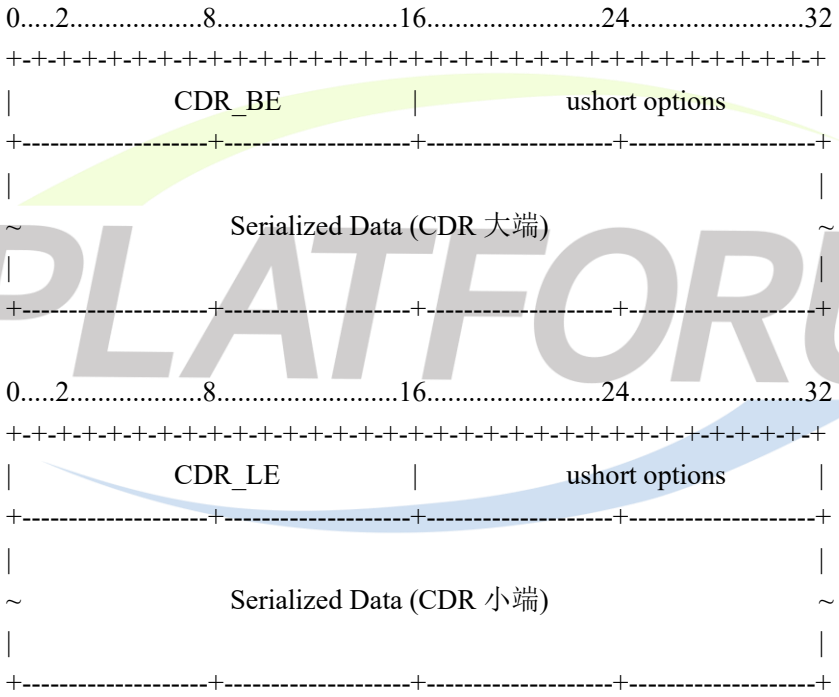
封装方案标识符	值	描述
CDR_BE	{0x00, 0x00}	OMG CDR 大端（参见 4.2.1.2）

CDR_LE	{0x00, 0x01}	OMG CDR 小端（参见 4.2.1.2）
PL_CDR_BE	{0x00, 0x02}	ParameterList（3.4.2.11）。 参数列表及其参数均使用 OMG CDR 大端封装。 见 4.2.1.3。
PL_CDR_LE	{0x00, 0x03}	ParameterList（3.4.2.11）。 参数列表及其参数均使用 OMG CDR 小端封装。 见 4.2.1.3。

可以在规范的未来版本中添加其他数据封装方案，例如 XML。

4.2.1.2 OMG CDR

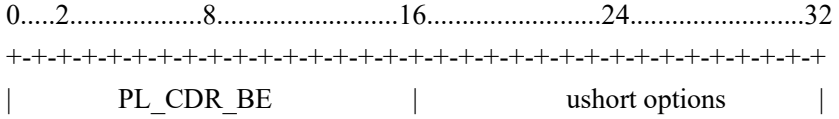
除了封装方案标识符之外，OMG CDR 封装还指定一个 16 位选项字段，后跟使用 CDR 编码的数据。选项字段留作将来的扩展。规范的当前版本在读取时不应解释它。

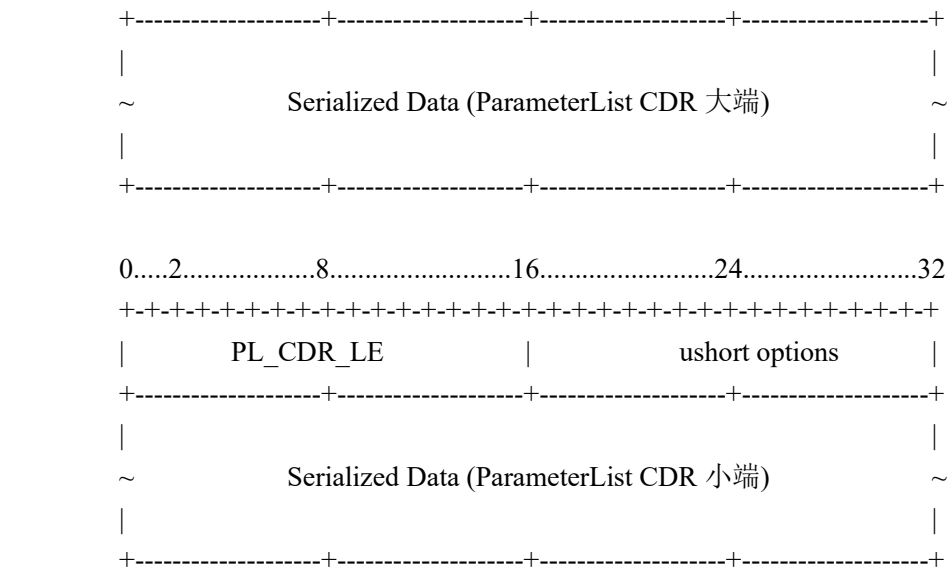


在封装大块序列化数据之后完成分片，因此 SerializedDataFragment 可能包含其不透明且分片的 SerializedData 示例的封装头。

4.2.1.3 ParameterList

除了封装方案标识符之外，**ParameterList** 封装还指定一个 16 位选项字段，后跟使用 **ParameterList** 编码的数据。选项字段留作将来的扩展。规范的当前版本在写入时应将其设置为零，而在读取时不对其进行解释。





在封装大块序列化数据之后完成分片，因此 SerializedDataFragment 可能包含其不透明且分片的 SerializedData 示例的封装头。

4.2.2 示例

4.2.2.1 OMG CDR

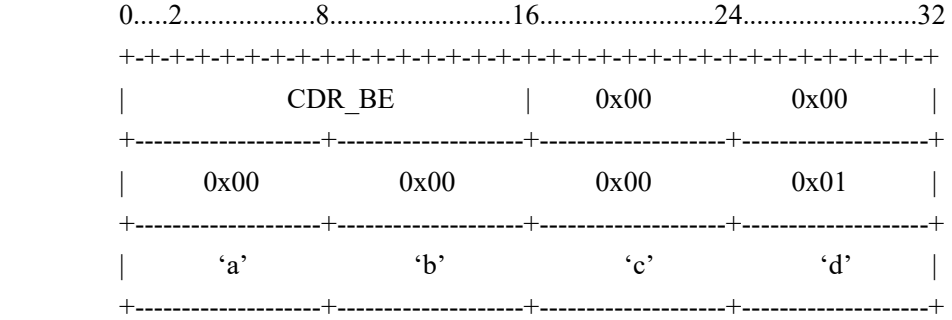
考虑以下以 IDL 表示的数据类型：

```
struct example
{
    long a;
    char b[4];
};
```

就本示例而言，我们假设以下值：

a = 1;
b[0] = ‘a’, b[1] = ‘b’, b[2] = ‘c’, b[3] = ‘d’;

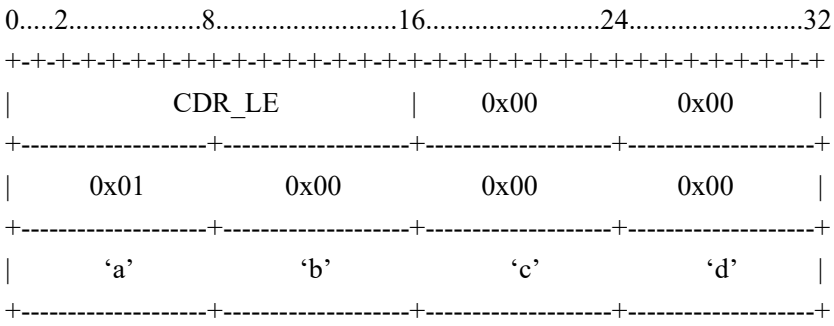
下图显示了使用大端格式的 CDR 时生成的封装：



其中

CDR_BE = {0x00, 0x00}

使用 CDR 以小端格式编码相同数据实例将产生下面这种封装：



其中

CDR_BE = {0x00, 0x01}