OpenDDS开发者指南

3.12版

OpenDDS版本3.12

由Object Computing, Inc. (OCI)支持

http://www.opendds.org HTTP://www.objectcomputing.com

目录

前言

什么是OpenDDS？

OpenDDS是两个对象管理组（OMG）规范的开源实现。

1）用于实时系统的数据分发服务（DDS）v1.4（OMG Document formal / 2015-04-10）。 本规范详细介绍了OpenDDS实现的用于实时发布和订阅应用程序的核心功能，并在本文档中进行了描述。

2）实时发布 - 订阅有线协议DDS互操作性有线协议规范（DDSI-RTPS）v2.2（OMG Document formal / 2014-09-01）。 本规范描述了行业DDS实现之间互操作性的主要要求。 这不是规范存在的唯一协议，但它是用于DDS实现之间的互操作性测试的协议。

OpenDDS由Object Computing, Inc.（OCI）赞助，可通过 http://www.opendds.org/.

许可条款

OpenDDS是在开源软件模型下提供的。 源代码可以自由下载，并可用于检查，审查，评论和改进。 副本可以自由安装在您的所有系统和客户的系统上。 开发或运行许可证不收费。 源代码被设计为在各种硬件和操作系统体系结构中被编译和使用。 根据许可协议的条款，您可以根据自己的需要对其进行修改。 您不得使用OpenDDS软件的版权。 有关许可条款的详细信息，请参阅OpenDDS源代码分发或访问中包含的名为LICENSE的文件 http://www.opendds.org/license.html.

OpenDDS还利用其他开源软件产品，包括MPC（Make Project Creator），ACE（自适应通信环境）和TAO（ACE ORB）。
有关这些产品的更多信息可以从OCI的网站上获得
HTTP：//www.objectcomputing.COM /产品.

OpenDDS是开源的，开发团队欢迎代码，测试和想法的贡献。 用户的积极参与确保了强大的实施。 如果您有兴趣参与OpenDDS的开发，请联系OCI。 请注意，贡献并成为OpenDDS开源代码库的任何代码都受到与OpenDDS代码库其余部分相同的许可条款的约束。

关于本指南

本开发者指南对应于OpenDDS版本3.12。 本指南主要关注使用和配置OpenDDS以构建分布式发布 - 订阅应用程序的具体情况。 虽然它确实给出了OMG数据分发服务的总体概述，但本指南并不是要提供对规范的全面介绍。 本指南旨在帮助您尽快熟练使用OpenDDS。

3.12版本的亮点

修正：

- RtpsUdpDataLink :: remove_sample锁定
- 仅在需要LatencyBudget QoS或统计信息时跟踪延迟
- 样品被拒绝/失去活力损失total_count_change
- get_key_value（）更正错误的返回值
- 设置DCPSBitTransportPort没有DCPSBitTransportIPAddress没有效果
- 读者取消订阅后，作者方协会并没有被删除
- 内存泄漏
- 在删除DataWriter时发生未注册的实例
- 多个传输实例在单个传输配置中的问题
- 启用工厂时，EntityFactory QoS不启用子对象注意：
- 配置支持从DOCGroup存储库的ACE + TAO作为一个选项
- 配置使用外部ACE + TAO的改进，在Windows上交叉编译
- 覆盖修复
- 改进的Bench性能测试
- Docker Hub现在有一个OpenDDS docker镜像

TAO版本兼容性

OpenDDS 3.12与OCI TAO 2.2a的当前补丁级别以及目前的DOC Group beta / micro版本兼容。 有关详细信息，请参阅$ DDS_ROOT / README.md文件。

约定

本指南使用以下约定：

固定的音调文字	指示用户使用键盘输入的示例代码或信息。 指示已从之前修改的示例代码
粗体固定的文字	示例或文本出现在菜单或对话框中。

斜体文本	表示重点。
...	横向省略号表示该语句省略了文本。

.
.
.

垂直省略号表示示例中省略了一段代码。

编码示例

在本指南中，我们用编码示例来说明主题。 本指南中的示例仅用于说明目的，不应被视为“生产就绪”代码。 特别是，有时将错误处理保持在最低限度，以帮助读者关注示例中呈现的特定特征或技术。 所有这些示例的源代码都可作为$ DDS_ROOT / DevGuideExamples /目录中的OpenDDS源代码分发的一部分提供。 MPC文件提供了用于生成构建工具特定文件的示例，例如GNU Makefiles或Visual C ++项目和解决方案文件。 每个示例都提供了一个名为run_test.pl的Perl脚本，以便您可以轻松地运行它。

相关文件

在本指南中，我们参考对象管理组（OMG）和其他来源发布的各种规范。

OMG参考文献采取的形式为group / number，其中group代表负责制定规范的OMG工作组，或者如果规范已被正式采用，则为正式的关键字，数字代表规范在月份内的年份，月份和序列号释放。 例如，OMG DDS 1.4版规范被引用为正式/ 2015-04-10。

您可以通过预先直接从OMG网站下载任何引用的OMG规范http://www.omg.org/cgi-bin/doc？到规范的参考。 因此，规范正式/ 07-01-01成为http://www.omg.org/cgi-bin/doc?formal/07-01-01。 将此目的地提供给网络浏览器应该带您到一个可以下载参考的规范文档的网站。

有关OpenDDS的其他文档由Object Computing，Inc.生成和维护，可从OpenDDS网站获取 http://www.opendds.org.

这里有一些感兴趣的文件和它们的位置：

文件	位置
实时系统数据分发服务（DDS）v1.4（OMG文档正式版/ 2015-04-10）	http://www.omg.org/spec/DDS/1.4/PDF
实时发布 - 订阅有线协议DDS互操作性有线协议规范（DDSI-RTPS）v2.2（OMG Document formal / 2014-09-01）	http://www.omg.org/spec/DDSI-RTPS/2.2/PDF

文件	位置
OMG数据分发门户	http://portals.omg.org/dds/
OpenDDS Buid指令，体系结构和Doxygen文档	http://www.opendds.org/documentation.html
OpenDDS常见问题	http://www.opendds.org/faq.html

支持的平台

OCI定期在各种平台，操作系统和编译器上构建和测试OpenDDS。 我们不断更新OpenDDS以支持其他平台。 看到了
$ DDS_ROOT / README.md文件分发给最新的平台支持信息。

客户支持

企业正在发现，设计和构建强大且可扩展的复杂分布式应用需要大量的经验，知识和资金。 OCI可以借鉴在当今中间件技术方面拥有丰富经验并了解如何利用DDS强大功能的经验丰富的架构师的经验，帮助您成功构建和提供解决方案。

我们的服务领域包括系统架构，大规模分布式应用架构，以及面向对象的设计和开发。我们在诸如DDS（OpenDDS），CORBA（ACE + TAO，JacORB和opalORB），Java EE（JBoss），FIX
（QuickFIX）和FAST（QuickFAST）。

OpenDDS的支持服务包括：

咨询服务，帮助设计可扩展的，可扩展的，强大的发布 - 订阅解决方案，包括特定领域方法的验证，服务选择，产品定制和扩展，以及将应用程序从其他发布 - 订阅技术和产品迁移到OpenDDS 。

全天候支持，确保生产级系统的最高响应水平。

按需服务协议，用于识别和评估在开发和部署基于OpenDDS的解决方案时可能出现的小错误和问题。

我们的架构师在安全性，电信，国防，财务和其他实时分布式应用方面拥有特定和广泛的专业领域专业知识。

我们可以提供可以帮助您进行短期工作的专业人员，如建筑和设计审查，快速原型设计，

故障排除和调试。 另外，对于更大的参与，我们可以提供导师，建筑师和

程序员与您的团队一起工作，在项目的整个生命周期中提供帮助和思想领导。

请致电+1.314.579.0066或发送电子邮件给我们sales@objectcomputing.com>了解更多信息。

OCI技术培训

OCI提供了丰富的50多门精心设计的课程，旨在为开发人员提供各种技术主题的坚实基础，如面向对象的分析和设计，C ++编程，Java编程，分布式计算技术（包括DDS），模式，XML和UNIX / Linux。 我们的课程清楚地解释了主要的概念和技巧，并通过亲身实践演示了他们如何映射到现实世界的应用。

注意 *我们的培训服务不断变化，以满足客户的最新需求，并反映技术的变化。 请务必查看我们的网站http://www.objectcomputing.com/training 更新我们的教育计划。*

现场分类

我们可以在贵公司的设施提供以下课程，将其与其他员工发展计划无缝集成。 有关OCI课程中的这些课程或其他课程的更多信息，请访问我们的课程目录 http://www.objectcomputing.com/training.

CORBA简介

在这一天的课程中，您将学习分布式对象计算的好处； CORBA在开发分布式应用程序中扮演的角色；何时何地应用CORBA；以及CORBA未来的发展趋势。

用C ++编写CORBA程序

在这个为期四天的实践课程中，您将学习：CORBA在开发分布式应用程序中的角色；OMG的对象管理体系结构；如何用C ++编写CORBA客户端和服务器；如何使用CORBA服务，如命名和事件；使用CORBA异常；以及便携式对象适配器（POA）的基本和高级功能。 本课程还包括使用OMG接口定义语言（IDL）的接口规范以及OMG IDL到C ++语言映射的详细信息，并提供在C ++（使用TAO）开发CORBA客户端和服务器方面的实践。

使用TAO的高级CORBA编程

在这个密集的实践四天的课程中，您将学习到：几种先进的CORBA概念和技术，以及TAO如何支持这些概念和技术；如何配置TAO组件进行性能和空间优化；以及如何使用TAO的各种并发模型来满足您应用的端到端QoS保证。 本课程涵盖了最近增加的CORBA规范以及TAO支持实时CORBA编程（包括实时CORBA）的内容。 它还涵盖了TAO的实时事件服务，通知服务和实现存储库，并提供了以C ++开发高级TAO客户端和服务器的丰富实践经验。 本课程面向经验丰富的CORBA / C ++程序员。

使用ACE C ++框架

在这个为期4天的实践课程中，您将学习如何使用ACE（自适应通信环境）IPC服务访问点（SAP）类和Acceptor / Connector模式实现进程间通信（IPC）机制。 该课程还将向您展示如何在事件多路分解和调度中使用反应器；如何使用ACE线程封装类别来实现线程安全的应用程序；以及如何确定适合您的具体应用需求的ACE组件。

面向对象的设计模式和框架

在为期三天的课程中，您将学习与设计模式相关的关键语言和术语，了解关键设计模式，学习如何选择合适的模式应用于特定情况，并学习如何应用模式构建强大的应用程序和框架。 本课程面向希望利用高级面向对象设计技术的软件开发人员以及具有强大编程背景的管理人员，他们将参与面向对象软件系统的设计和实现。

使用C ++进行OpenDDS编程

在为期四天的课程中，您将学习如何使用OpenDDS构建应用程序，即实时系统的OMG数据分发服务（DDS）的开源实现。 您将学习如何构建通过OpenDDS共享数据的以数据为中心的系统。 您还将学习如何配置OpenDDS以满足您的应用程序的服务质量要求。 本课程适合有经验的C ++开发人员。

OpenDDS建模软件开发工具包（SDK）

在这个为期两天的课程中，开发人员和架构师将使用OpenDDS Modeling SDK来设计和构建发布/订阅应用程序

OpenDDS。 基于Eclipse的开源建模SDK使开发人员能够将应用程序的中间件组件和数据结构定义为UML模型，然后使用OpenDDS生成实现该模型的代码。 生成的代码然后可以编译并与应用程序链接，为应用程序提供无缝的中间件支持。

C ++编程使用Boost

在这个为期四天的课程中，您将学习构成Boost的最广泛使用和有用的库。 学生将通过详细的专家讲师指导培训和实践练习，学习如何轻松将这些强大的图书馆应用于自己的发展。 完成本课程后，课程参与者将准备将Boost应用到他们的项目中，使他们能够更快地生成功能强大，高效且独立于平台的应用程序。

注意

有关培训日期的信息，请致电+1.314.579.0066，通过电子邮件与我们联系 training@objectcomputing.com，*或访问我们的网站* http://www.objectcomputing.com/training *审查当前的课程时间表。*

C章节1

介绍

OpenDDS是针对实时系统规范v1.4（OMG文档正式版/ 2015-04-10）的OMG数据分发服务（DDS）的开源实现，以及实时发布订阅有线协议DDS互操作性有线协议规范（DDSI - RTPS）v2.2（OMG Document formal / 2014-09-01）。 OpenDDS是由Object Computing, Inc.（OCI）赞助的，可以从中获得http://www.opendds.org/。 本开发人员指南基于OpenDDS 3.12版本。

DDS定义了在分布式应用程序的参与者之间有效分发应用程序数据的服务。 这个服务不是特定于CORBA的。 该规范提供了平台无关模型（PIM）以及将PIM映射到CORBA IDL实现上的平台特定模型（PSM）。

有关DDS的其他详细信息，开发人员应参考DDS规范（OMG文档正式/ 2015-04-10），因为它包含所有服务功能的深入报道。

OpenDDS是Object Computing, Inc.（OCI）开发和商业支持的OMG DDS规范的开源C ++实现。 它可以从下载http://www.opendds.org/downloads.html 并与OCI TAO 2.2a版本的最新补丁级别以及最新的DOC Group版本兼容。

注意 *OpenDDS目前实现了OMG DDS 1.4版规范。 请参阅或中的合规性信息 http://www.opendds.org/ 了解更多信息。*

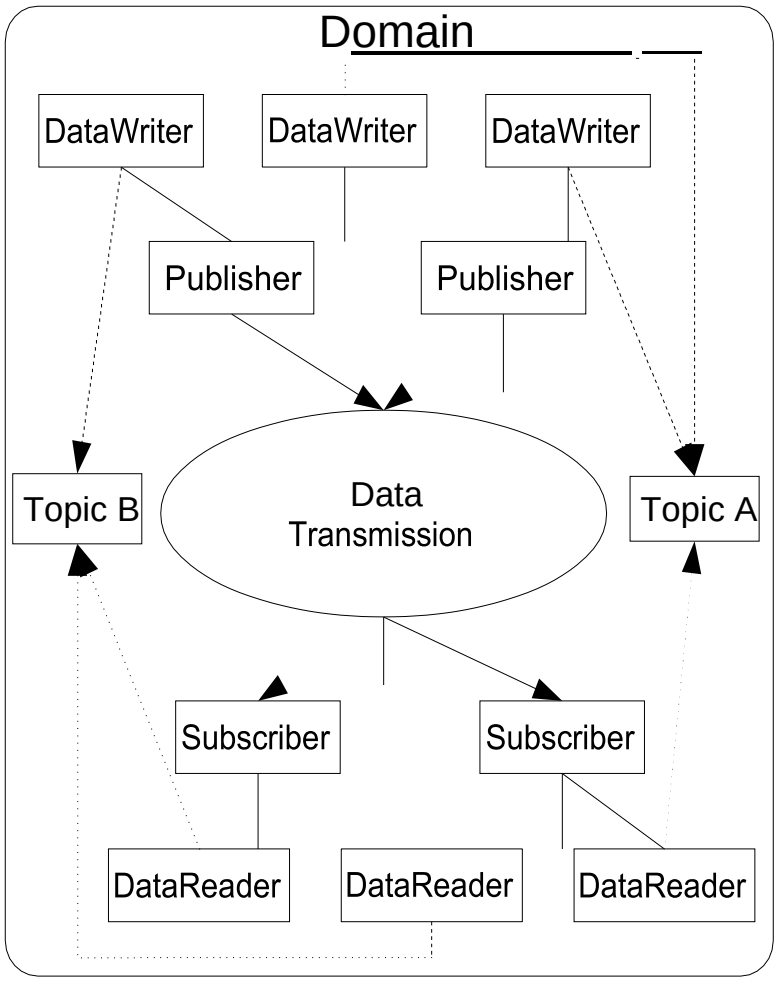
1.1. DCPS概述

在本节中，我们将介绍DCPS层的主要概念和实体，并讨论它们如何相互作用和一起工作。

1.1.1 基本概念

图1-1 显示DDS DCPS层的概述。 以下小节定义了此图中显示的概念。

图1-1 DCPS概念概述



1.1.1.1 域

该域是华府公立学校的基本划分单位。 其他每个实体都属于一个域，只能与同一域中的其他实体交互。

应用程序代码可以自由地与多个域进行交互，但必须通过属于不同域的单独实体来完成。

1.1.1.2 DomainParticipant

域参与者是应用程序在特定域内交互的入口点。 域参与者是写入或读取数据所涉及的许多对象的工厂。

1.1.1.3 话题

该主题是发布和订阅应用程序之间交互的基本手段。 每个主题在域内具有唯一的名称，并且具有发布的特定数据类型。 每个主题数据类型可以指定组成其关键字的零个或多个字段。 发布数据时，发布过程始终指定主题。 订阅者通过主题请求数据。 在DCPS术语中，您将针对主题上的不同实例发布单个数据样本。 每个实例都与密钥的唯一值相关联。 发布进程通过对每个样本使用相同的键值在同一实例上发布多个数据样本。

1.1.1.4 DataWriter

数据写入器由发布应用程序代码用来将值传递给DDS。 每个数据作者都被绑定到一个特定的主题。 应用程序使用数据写入程序的类型特定的接口来发布该主题的示例。 数据编写者负责封送数据并将其传送给发布者进行传输。

1.1.1.5 出版者

发布者负责获取已发布的数据并将其发布给域中的所有相关订阅者。 所采用的确切机制留给服务实施。

1.1.1.6 订户

订阅者从发布者那里接收数据并将其传递给与之相连的任何相关的数据读取器。

1.1.1.7 DataReader的

数据读取器从订阅者处获取数据，将其解密为适合该主题的类型，并将样本发送给应用程序。 每个数据读取器都绑定到一个特定的主题。 应用程序使用数据读取器类型特定的接口来接收样本。

1.1.2 内置主题

DDS规范定义了DDS实现中内置的一些主题。 订阅这些内置主题使应用程序开发人员能够访问正在使用的域的状态，包括注册了哪些主题，哪些数据读取器和数据写入器连接和断开，以及各种实体的QoS设置。 在订阅时，应用程序会收到指示域内实体发生变化的样本。

下表显示了DDS规范中定义的内置主题：

表1-1内置主题

主题名称	描述
DCPSParticipant	每个实例代表一个域参与者。
DCPSTopic	每个实例代表一个正常的（不是内置的）主题。
DCPSPublication	每个实例代表一个数据写入器。
DCPSSubscription	每个实例代表一个数据读取器。

1.1.3 服务质量政策

DDS规范定义了许多服务质量（QoS）策略，应用程序使用这些策略来指定服务的QoS要求。 参与者指定他们从服务中需要的行为，服务决定如何实现这些行为。 这些策略可以应用于各种DCPS实体（主题，数据编写者，数据阅读者，发布者，订阅者，域参与者），但不是所有策略对于所有类型的实体都是有效的。

订阅者和发布者使用请求与提供（RxO）模型进行匹配。 订阅者请求一组最低限度需要的策略。 发行商向潜在订户提供一组QoS策略。 然后，DDS实施尝试将所请求的策略与所提供的策略进行匹配；如果这些政策是相容的，那么协会就形成了。

第3章详细讨论了OpenDDS当前实施的QoS策略。

1.1.4 听众

DCPS层为每个实体定义了一个回调接口，允许应用程序进程“侦听”与该实体有关的某些状态变化或事件。 例如，当有数据值可供读取时，通知数据读取器监听器。

1.1.5 条件

条件和等待集允许侦听者在DDS中检测感兴趣的事件。 一般模式是

应用程序创建一个特定类型的Condition对象，如StatusCondition，并将其附加到WaitSet。

- 应用程序在WaitSet上等待，直到一个或多个条件成立。
- 应用程序调用相应的实体对象的操作来提取必要的信息。

- DataReader接口还具有采用ReadCondition参数的操作。
- QueryCondition对象是作为内容订阅配置文件实现的一部分提供的。QueryCondition接口扩展了ReadCondition接口。

1.2 OpenDDS实现

1.2.1 合规

OpenDDS符合OMG DDS和OMG DDSI-RTPS规范。 遵守细节在这里。

1.2.1.1 DDS合规性

DDS规范的第2部分为DDS实现定义了五个合规点：

1) 最小配置文件
2) 内容订阅配置文件
3) 持久性配置文件
4) 所有权配置文件
5) 对象模型简介

OpenDDS符合整个DDS规范（包括所有可选配置文件）。 这包括执行所有服务质量政策并附以下说明：

- RELIABILITY.kind =仅当使用TCP或IP组播传输（配置为可靠）或使用RTPS_UDP传输时才支持RELIABLE。
- TRANSPORT_PRIORITY没有被实现为可改变的。

1.2.1.2 DDSI-RTPS合规性

OpenDDS实现符合OMG DDSI-RTPS规范的要求。

OpenDDS RTPS实现说明

OMG DDSI-RTPS规范（正式版/ 2014-09-01）提供了实施声明，但不符合要求。 使用OpenDDS RTPS功能进行传输和/或发现时应考虑以下事项。 每个项目都提供了DDSI-RTPS规范的章节号以供进一步参考。

未在OpenDDS中实施的项目：

1）作家端内容过滤（8.7.3）
OpenDDS仍然可能丢弃任何相关读取器不需要的样本（由于内容过滤） - 这在传输层上完成

2）演示QoS的相干集（8.7.5）

3）定向写入（8.7.6）

4）产业名单（8.7.7）

5）DURABLE数据的原始编写者信息（8.7.8） - 仅用于暂时和持久性持久性，RTPS规范不支持（8.7.2.2.1）

6）密钥哈希（8.7.9）不会生成，但它们是可选的

7）nackSuppressionDuration（表8.47）和heartbeatSuppressionDuration（表8.62）。

注意 *上面的项目3和4在DDSI-RTPS规范中有描述。 但是，他们在DDS规范中没有相应的概念。*

1.2.2 DDS规范的扩展

DDS IDL模块（C ++名称空间，Java程序包）中的数据类型，接口和常量直接对应于DDS规范，只有极少数例外：

- DDS :: SampleInfo包含一个以“opendds_reserved”开头的额外字段
- 特定于类型的数据读取器（包括用于内置主题的数据读取器）具有额外的操作read_instance_w_condition（）和take_instance_w_condition（）。

其他扩展行为由OpenDDS模块/名称空间/包中的各种类和接口提供。 其中包括Recorder和Replayer等功能（见章节）12） 并且：

- OpenDDS :: DCPS :: TypeSupport添加了在DDS规范中找不到的unregister_type（）操作。
- OpenDDS :: DCPS :: ALL_STATUS_MASK，NO_STATUS_MASK和DEFAULT_STATUS_MASK对于DDS :: Entity，DDS :: StatusCondition和各种create _ *（）操作使用的DDS :: StatusMask类型是有用的常量。

1.2.3 OpenDDS体系结构

本节简要介绍了OpenDDS的实现，其功能及其一些组件。 $ DDS_ROOT环境变量应该指向基本目录

OpenDDS发行版。 OpenDDS的源代码可以在下面找到
$ DDS_ROOT / dds /目录。 DDS测试可以在$ DDS_ROOT / tests /下找到。

1.2.3.1 设计理念

OpenDDS实现和API基于对OMG IDL PSM的相当严格的解释。 在几乎所有情况下，OMG用于CORBA IDL的C ++语言映射都用于定义DDS规范中的IDL如何映射到OpenDDS公开给客户端的C ++ API。

与OMG IDL PSM的主要偏离是本地接口用于实体和各种其他接口。 这些被定义为DDS规范中的无约束（非本地）接口。 将它们定义为本地接口可以提高性能，减少内存使用量，简化客户端与这些接口的交互，并使客户端更容易构建自己的实现。

1.2.3.2 可扩展传输框架（ETF）

OpenDDS使用DDS规范定义的IDL接口初始化和控制服务使用情况。 数据传输是通过OpenDDS特定的传输框架完成的，该框架允许将服务与各种传输协议一起使用。 这被称为可插拔传输，并将OpenDDS的可扩展性作为其架构的重要组成部分。 OpenDDS当前支持TCP / IP，UDP / IP，IP多播，共享内存和RTPS_UDP传输协议图1-2。 传输通常通过配置文件来指定，并附加到发布者和订阅者进程中的各种实体上。 有关配置ETF组件的详细信息，请参阅第7.4.4节。

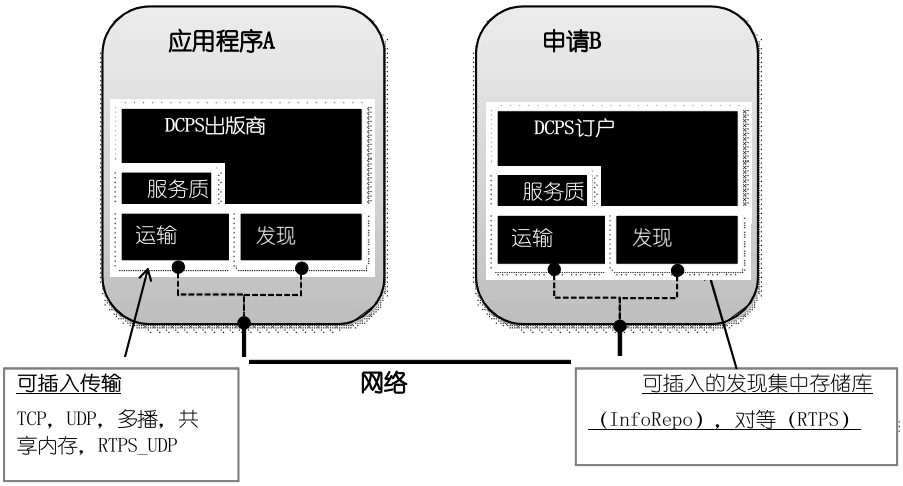


图1-2 OpenDDS可扩展传输框架

ETF使应用程序开发人员能够实现自己的定制传输。

实现一个定制的传输涉及专门化在传输框架中定义的许多类。 udp传输为开发人员创建自己的实现时提供了一个很好的基础。 有关详细信息，请参阅$ DDS_ROOT / dds / DCPS / transport / udp /目录。

1.2.3.3 DDS发现

DDS应用程序必须通过一些中央代理或通过一些分布式方案发现彼此。 OpenDDS的一个重要功能是DDS应用程序可以配置为使用DCPSInfoRepo或RTPS发现来执行发现，但是在数据写入器和数据读取器之间使用不同的传输类型进行数据传输。 OMG DDS规范（正式版/ 2015-04-10）将发现细节留给实现。 在DDS实现之间的互操作性的情况下，OMG DDSI-RTPS（formal / 2014-09-01）规范提供了点对点发现风格的要求。

OpenDDS提供了两个发现选项。

1) 信息库：集中的存储库风格，作为一个单独的进程运行，允许发布者和订阅者集中或互相发现

2) RTPS发现：利用RTPS协议来发布可用性和位置信息的点对点发现风格。

与其他DDS实现的互操作性必须使用对等方法，但在仅限OpenDDS的部署中可能很有用。

使用DCPSInfoRepo进行集中式发现

OpenDDS实现了一个称为DCPS信息库（DCPSInfoRepo）的独立服务来实现集中式发现方法。它作为一个CORBA服务器来实现。 当客户请求订阅某个主题时，DCPS信息库找到该主题，并通知任何现有的发布者该新订户的位置。
每当在非RTPS配置中使用OpenDDS时，DCPSInfoRepo进程都需要运行。 RTPS配置不使用DCPSInfoRepo。 DCPSInfoRepo不涉及数据传播，其作用范围仅限于OpenDDS应用程序发现的范围。

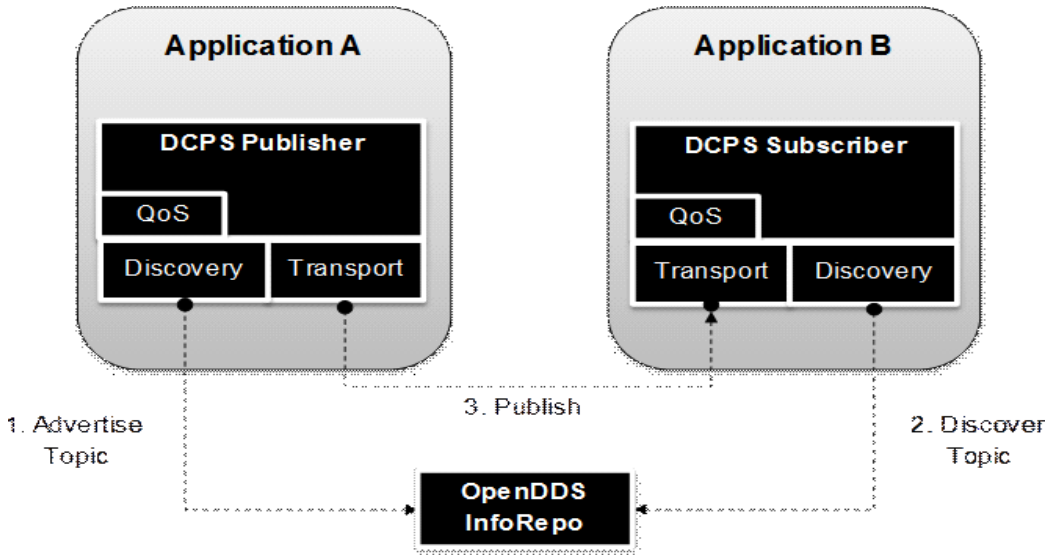


图1-3：使用OpenDDS InfoRepo进行集中式发现

应用程序开发人员可以自由运行多个信息库，每个信息库管理自己的非重叠DCPS域集。

也可以使用多个存储库来操作域，从而形成分布式虚拟存储库。 这被称为Repository Federation。 为了使各个存储库参与联合，每个存储库在启动时都必须指定自己的联合标识符值（一个32位数值）。 看到9.2 了解有关存储库联合的更多信息。

使用RTPS进行对等发现

OpenDDS功能可以满足需要对等发现模式的DDS应用程序。 这种发现方式只能通过使用当前版本的RTPS协议来完成。 这种简单的发现形式是通过在应用程序进程中运行的DDS应用程序数据读取器和数据写入器的简单配置来完成的错误：找不到参考源。 由于每个参与过程在OpenDDS中为其数据读取器和编写器激活DDSI-RTPS发现机制，因此网络端点将使用默认或配置的网络端口创建，以便DDS参与者可以开始宣传其数据读取器和数据编写者的可用性。 经过一段时间后，那些根据标准相互寻找的人将会找到彼此，并根据可扩展传输框架（ETF）中所讨论的配置的可插拔传输来建立连接。 本节将详细介绍这种灵活的配置方法7.4.1.1 和科7.4.5.5.

以下是开发人员在开发和部署使用RTPS发现的应用程序时需要考虑的其他实施限制：

1）由于UDP端口分配给域ID的方式，因此域ID应介于0和231（含）之间。 在每个OpenDDS过程中，每个域最多支持120个域参与者。

2）主题名称和类型标识符限制为256个字符。

3）由于分配了GUID，OpenDDS的本地多点传送不能用于RTPS发现（如果尝试这种情况，将发出警告）。

有关RTPS发现的更多详细信息，请参阅实时发布 - 订阅有线协议DDS互操作性有线协议规范（DDSI-RTPS）v2.2（OMG Document formal / 2014）的第8.5节-09-01）。

1.2.3.4 穿线

OpenDDS创建自己的ORB（当需要的时候）以及一个单独的线程来运行该ORB。 它也使用自己的线程来处理传入和传出的传输I / O。 创建一个单独的线程来清理意外的连接关闭时的资源。 您的应用程序可能会通过DCPS的Listener机制从这些线程回调。

当通过DDS发布样本时，OpenDDS通常会尝试使用调用线程将样本发送给任何连接的订阅者。 如果发送呼叫阻塞，则样本可能排队等待在单独的服务线程上发送。 这个行为取决于第3章中描述的QoS策略。

订阅者中的所有传入数据都由服务线程读取，并排队等候应用程序读取。 DataReader侦听器是从服务线程中调用的。

1.2.3.5 组态

OpenDDS包括一个基于文件的配置框架，用于配置调试级别，内存分配和发现等全局项目，以及发布者和订阅者的传输实现细节。 配置也可以直接在代码中实现，但是，为了方便维护并减少运行时错误，建议将配置外部化。 第7章介绍了一整套配置选项。

1.3 安装

如何构建OpenDDS的步骤可以在$ DDS_ROOT / INSTALL中找到。

为了避免编译不会使用的OpenDDS代码，有一些特性可以被排除在外。 这些功能在下面讨论。

要求占用空间小或与安全平台兼容的用户应考虑使用OpenDDS安全配置文件，该配置文件在章节13 本指南。

1.3.1 建立启用或禁用功能

配置脚本支持大多数功能。 配置脚本创建具有正确内容的配置文件，然后运行MPC。 如果您正在使用配置脚本，请使用“--help”命令行选项运行它，并查找您希望启用/禁用的功能。 如果您没有使用配置脚本，请继续阅读以下内容，以获取有关直接运行MPC的说明。

对于下面介绍的功能，MPC用于启用（默认）功能或禁用功能。 对于名为feature的功能，以下步骤用于从构建中禁用功能：

1）使用MPC的命令行“功能”参数：

```
mwc.pl -type <type> -features feature=0 DDS.mwc
```

或者，将行特征= 0添加到文件
$ ACE_ROOT / bin / MakeProjectCreator / config / default.features并使用MPC重新生成项目文件。

2）如果您使用的是gnuace MPC项目类型（如果您将使用GNU make作为您的编译系统），请将“feature = 0”行添加到文件
$ ACE_ROOT /包括/ makeinclude / platform_macros.GNU。 要显

式启用该功能，请使用上面的feature = 1。

注意 *您也可以使用$ DDS_ROOT / configure脚本启用或禁用功能。 要禁用此功能，请将--no-feature传递给脚本，以启用pass --feature。 在这种情况下，使用“ - ”代替特征名称中的“_”。 例如，要禁用功能*

content_subscription在下面讨论，通过--no-content-subscription到配置脚本。

1.3.2 禁用内置主题支持的构建

功能名称：built_in_topics

通过禁用内置主题支持，可以将核心DDS库的占用空间减少多达30%。 见章节6 有关内置主题的说明。

1.3.3 禁用合规性配置文件功能的构建

DDS规范定义了符合性配置文件，以提供一个通用术语来指示DDS实现可能支持或不支持的某些功能集。 下面给出了这些配置文件，以及用于禁用对该配置文件或该配置文件组件的支持的MPC功能的名称。

许多配置文件选项涉及QoS设置。 如果您尝试使用与禁用的配置文件不兼容的QoS值，则会发生运行时错误。 如果一个配置文件涉及到一个类，如果您尝试使用该类并且该配置文件被禁用，则会发生编译时错误。

1.3.3.1 内容订阅配置文件

功能名称：content_subscription

此配置文件添加ContentFilteredTopic，QueryCondition和MultiTopic类
在第5章讨论。

另外，可以通过使用下表中给出的特征来排除个别课程。

表1-2：内容订阅类别功能

类	特征
ContentFilteredTopic	content_filtered_topic
QueryCondition	query_condition
多主题	multi_topic

1.3.3.2 持久性配置文件

功能名称：persistence_profile

该配置文件添加QoS策略DURABILITY_SERVICE和DURABILITY QoS策略类型的设置“TRANSIENT”和“PERSISTENT”。

1.3.3.3 所有权配置文件

功能名称：ownership_profile

此配置文件添加：

- OWNERSHIP类别的设置“EXCLUSIVE”

- 支持OWNERSHIP_STRENGTH策略
- 为HISTORY QoS策略设置深度> 1。

注意 *目前，即使支持HISTORY depth> 1的OpenDDS代码仍然可用*
ownership_profile被禁用。

1.3.3.4 对象模型简介

功能名称: object_model_profile

此配置文件包含对“GROUP”的PRESENTATION access_scope设置的支持。

注意 *目前，’TOPIC’的呈现access_scope也被排除在外*
object_model_profile被禁用。

C章节2

入门

2.1 使用DCPS

本章重点介绍使用DCPS将数据从单个发布者进程分发到单个订阅者进程的示例应用程序。 它基于一个简单的信使应用程序，一个发布者发布消息，一个订阅者订阅它们。 我们使用默认的QoS属性和默认的TCP / IP传输。 这个例子的完整源代码可以在下面找到 $ DDS_ROOT / DevGuideExamples / DCPS / Messenger /目录。 其他DDS和DCPS功能将在后面的章节中讨论。

2.1.1 定义数据类型

DDS使用的每种数据类型都是使用IDL定义的。 OpenDDS使用#pragma指令来识别DDS传输和处理的数据类型。 这些数据类型由TAO IDL编译器和OpenDDS IDL编译器处理，以生成用OpenDDS传输这些类型数据所需的代码。 这是定义我们的消息数据类型的IDL文件：

```
模块Messenger {

#pragma DCPS_DATA_TYPE"Messenger :: Message"
#pragma DCPS_DATA_KEY"Messenger :: Message subject_id"

  struct Message
    {string from; 字符
    串主题; long
    subject_id; 字符串
    文本; 多少
  };
};
```

DCPS_DATA_TYPE编译指示了OpenDDS使用的数据类型。 一个完整范围的类型名称必须与该附注一起使用。 OpenDDS要求数据类型是一个结构。 该结构可能包含标量类型（短，长，浮点数等），枚举，字符串，序列，数组，结构和联合。 这个例子定义了在这个OpenDDS例子中使用的Messenger模块中的结构消息。

DCPS_DATA_KEY附注标识用作此类型的密钥的DCPS数据类型的字段。 数据类型可能有零个或多个键。 这些键用于标识主题中的不同实例。 每个键应该是数字或枚举类型，字符串或其中一种类型的typedef。[1]该编译指示将传递完全范围的类型和标识该类型的键的成员名称。 多个键用单独的DCPS_DATA_KEY编译指定。 在上例中，我们将Messenger :: Message的subject_id成员标识为一个键。 使用唯一的subject_id值发布的每个样本都将被定义为属于同一主题内的其他实例。由于我们使用的是默认QoS策略，因此具有相同subject_id值的后续样本将被视为该实例的替换值。

2.1.2 处理IDL

OpenDDS IDL首先由TAO IDL编译器处理。

1其他类型，比如结构体，序列和数组不能直接用作键，尽管数组，结构体或数组元素的单个成员可以作为键使用，当这些成员/元素是数字，枚举或字符串类型时。

```
tao_idl Messenger.idl
```

另外，我们需要使用OpenDDS IDL编译器处理IDL文件，以生成OpenDDS编组和消除消息所需的序列化和密钥支持代码，以及数据读取器和编写器的类型支持代码。 此IDL编译器位于$ DDS_ROOT / bin /中，并为每个IDL文件生成三个文件处理。 这三个文件都以原始的IDL文件名开头，如下所示：

- *〈文件名〉TypeSupport.idl*
- *〈文件名〉TypeSupportImpl.h*
- *〈文件名〉TypeSupportImpl.cpp*

例如，运行opendds_idl如下

```
opendds_idl Messenger.idl
```

生成MessengerTypeSupport.idl，MessengerTypeSupportImpl.h和MessengerTypeSupportImpl.cpp。 IDL文件包含MessageTypeSupport，MessageDataWriter和MessageDataReader接口定义。 这些是特定于类型的DDS接口，我们稍后将其用于向域注册数据类型，发布该数据类型的样本，并接收已发布的样本。 实现文件包含这些接口的实现。 生成的IDL文件本身应该用TAO IDL编译器编译生成存根和骨架。这些和实现文件应该与使用消息类型的OpenDDS应用程序链接。 OpenDDS IDL编译器有许多专门生成代码的选项。 这些选项在第8章中描述。

通常，您不会像上面那样直接调用TAO或OpenDDS IDL编译器，而是让您的构建环境为您做。 通过继承dcpsexe_with_tcp项目，使用MPC可以简化整个过程。 以下是发布者和订阅者共同的MPC文件部分

```
项目 (* idl) : dcps {
  // 这个项目确保通用组件先建成。

  TypeSupport_Files
    { Messenger.idl
  }
  custom_only = 1
}
```

dcps父项目添加了“类型支持”自定义构建规则。 上面的TypeSupport_Files部分告诉MPC使用OpenDDS IDL编译器从Messenger.idl生成消息类型支持文件。 这是发布者部分：

```
项目 (*发布者) : dcpsexe_with_tcp {exename =出
  版商
  后        += *idl

  TypeSupport_Files
    { Messenger.idl
  }

  Source_Files
    { Publisher.cpp
  }
}
```

DCPS库中的dcpsexe_with_tcp项目链接。

为了完整起见，这里是MPC文件的用户部分：

```
项目 (* Subscriber) : dcpsexe_with_tcp {exename

             =用户
  后        += *idl

  TypeSupport_Files
    { Messenger.idl
  }

  Source_Files
    { Subscriber.cpp
    DataReaderListenerImpl.cpp
  }
}
```

2.1.3 简单的消息发布者

在本节中，我们将介绍设置简单的OpenDDS发布流程所涉及的步骤。 代码被分解成逻辑部分，并在我们介绍每个部分时进行解释。 我们省略了一些不感兴趣的代码部分（如#include指令，错误处理和跨进程同步）。 此示例发布者的完整源代码位于Publisher.cpp和Writer.cpp文件中$ DDS_ROOT / DevGuideExamples / DCPS /斜挎/。

2.1.3.1 初始化参与者

main () 的第一部分将当前进程初始化为OpenDDS参与者。

```
int main (int argc, char * argv []) {try {
     DDS :: DomainParticipantFactory_var dpf =
       TheParticipantFactoryWithArgs (argc, argv) ;
     DDS :: DomainParticipant_var参与者= dpf->
       create_participant (42, //域ID
                                   PARTICIPANT_QOS_DEFAULT,
                                   0, //不需要侦听器OpenDDS :: DCPS :: DEFAULT_STATUS_MASK) ;
     如果 (! 参与者) {
       std :: cerr <<“create_participant失败。” << std :: endl; 返回1;
     }
```

TheParticipantFactoryWithArgs宏在Service_Participant.h中定义，并使用命令行参数初始化Domain Participant Factory。 这些命令行参数用于初始化OpenDDS服务使用的ORB以及服务本身。 这允许我们在命令行上传递ORB_init（）选项，以及形式-DCPS *的OpenDDS配置选项。 第7章详细介绍了可用的OpenDDS选项。

create_participant（）操作使用域参与者工厂将该过程注册为由ID 42指定的域中的参与者。参与者使用默认的QoS策略并且不使用监听器。 使用OpenDDS默认状态掩码确保中间件中的所有相关通信状态变化（例如，可用数据，失去活力）被传送到应用程序（例如，通过回听者的回调）。

用户可以使用范围（0x0～0x7FFFFFFF）中的ID来定义任意数量的域。 所有其他值保留供实施内部使用。

返回的域参与者对象引用然后用于注册我们的消息数据类型。

2.1.3.2 注册数据类型和创建主题

首先，我们创建一个MessageTypeSupportImpl对象，然后使用register_type（）操作将类型注册为类型名称。 在这个例子中，我们注册了一个没有字符串类型名称的类型，这导致MessageTypeSupport接口库标识符被用作类型名称。 也可以使用特定的类型名称，如“消息”。

```
Messenger :: MessageTypeSupport_var mts = new
  Messenger :: MessageTypeSupportImpl () ;
如果 (DDS :: RETCODE_OK! = mts-> register_type (参与者, “”) ) {std :: cerr
  <<“register_type失败。  << std :: endl;
  返回1;
}
```

接下来，我们从类型支持对象获得注册类型名称，并通过将类型名称传递给create_topic（）操作中的参与者来创建主题。

```
CORBA :: String_var type_name = mts-> get_type_name () ;

  DDS::Topic_var topic =
    参与者 - > create_topic (“电影讨论列表”,
                                type_name,
                                TOPIC_QOS_DEFAULT,
                                0,    // 不需要侦听器OpenDDS :: DCPS ::
                                DEFAULT_STATUS_MASK) ;
  if (! topic) {
    std :: cerr <<“create_topic失败。”  << std :: endl; 返回1;
  }
```

我们创建了一个名为“电影讨论列表”的主题，注册类型和默认QoS策略。

2.1.3.3 创建一个发布者

现在，我们准备使用默认的发布者QoS创建发布者。

```
DDS::Publisher_var pub =
  participant-> create_publisher (PUBLISHER_QOS_DEFAULT,
                                  0,     // 不需要侦听器OpenDDS :: DCPS ::
                                  DEFAULT_STATUS_MASK) ;
如果 (! pub) {
  std :: cerr <<“create_publisher失败。” << std :: endl; 返回1;
}
```

2.1.3.4 创建DataWriter并等待订阅者

随着出版商的到位，我们创建了数据编写器。

```
// 创建数据生成器DDS ::
DataWriter_var writer =
  pub-> create_datawriter (主题,
                           DATAWRITER_QOS_DEFAULT,
                           0,     // 不需要侦听器OpenDDS :: DCPS ::
                           DEFAULT_STATUS_MASK) ;
如果 (! 作家) {
  std :: cerr <<“create_datawriter失败。” << std :: endl; 返回1;
}
```

当我们创建数据写入器时，我们传递了主题对象引用，默认的QoS策略以及一个空的侦听器引用。 我们现在将数据写入器引用的范围缩小到MessageDataWriter对象引用，以便我们可以使用特定于类型的发布操作。

```
Messenger :: MessageDataWriter_var message_writer = Messenger :: MessageDataWriter :: _ narrow
      (writer) ;
```

示例代码使用条件和等待集，以便发布者等待订阅者连接并完全初始化。 在这样一个简单的例子中，没有等待订阅者可能会导致发布者在订阅者连接之前发布其样本。

等待用户的基本步骤是：

1) 从我们创建的数据写入器获取状态条件

2) 在条件中启用“发布匹配”状态

3) 创建一个等待集

4）将状态条件附加到等待设置

5）获取发布匹配状态

6）如果当前的匹配次数是一次或多次，则从等待设置中分离条件并继续发布

7）等待等待设置（可以限定一段时间）

8）循环回到步骤5 这里是相应

的代码：

```
// 阻止直到用户可用DDS :: StatusCondition_var条
件=
    写入器的> get_statuscondition () ; 条件>
    set_enabled_statuses (DDS ::
    PUBLICATION_MATCHED_STATUS) ;

DDS :: WaitSet_var ws = new DDS :: WaitSet;
WS-> attach_condition (条件) ;

while (true) {DDS :: PublicationMatchedStatus匹
  配;
  如果 (writer-> get_publication_matched_status (matches)
       ! = DDS :: RETCODE_OK) {
    std :: cerr <<"get_publication_matched_status失败! "
                << std::endl;
    返回1;
  }

  if (matches.current_count> = 1) {break;
  }

  DDS :: ConditionSeq条件;  DDS :: Duration_t超
  时= {60, 0};
  如果 (ws-> wait (conditions, timeout) ! = DDS :: RETCODE_OK) {std :: cerr
    <<"等待失败!   << std :: endl;
    返回1;
  }

}

WS-> detach_condition (条件) ;
```

有关状态，条件和等待集的更多细节，请参见章节4.

2.1.3.5 样本发布

该消息的发布非常简单：

```
// 撰写样本Messenger ::留言信息;
message.subject_id = 99;
message.from        ="漫画家伙"; 信息主题
```

```
                                =“评论”;
message.text                    =“最糟糕的电影永远”
message.count                   = 0;
for (int i = 0; i <10; ++ i) {
```

```
    DDS :: ReturnCode_t错误= message_writer->写 (消息,
                                                    DDS :: HANDLE_NIL) ;
    ++message.count;
    ++message.subject_id;
    if (error! = DDS :: RETCODE_OK) {
      // 记录或以其他方式处理错误情况返回1;
    }
  }
```

对于每个循环迭代，调用write () 都会将消息分发给为我们的主题注册的所有连接的订阅者。 由于subject_id是Message的关键字，每次subject_id递增并调用write () ，都会创建一个新实例（请参阅1.1.1.3）。 write () 的第二个参数指定我们发布样本的实例。应该传递一个由register_instance () 或者返回的句柄

DDS :: HANDLE_NIL。 传递DDS :: HANDLE_NIL值表示数据写入者应通过检查样本的关键字来确定实例。 参见章节2.2.1 了解在发布过程中使用实例句柄的细节。

2.1.4 设置用户

订阅者的许多代码与我们刚刚完成探索的发布者相同或类似。 我们将通过类似的部分快速进展，并参考上面的讨论细节。 此示例订户的完整源代码位于Subscriber.cpp和DataReaderListener.cpp文件中

$ DDS_ROOT / DevGuideExamples / DCPS /斜挎/。

2.1.4.1 初始化参与者

订户的起始与我们初始化服务并加入我们的域名时的发布者相同：

```
int main (int argc, char * argv [])
{
  尝试{
    DDS :: DomainParticipantFactory_var dpf =
      TheParticipantFactoryWithArgs (argc, argv) ;
    DDS :: DomainParticipant_var参与者= dpf->
      create_participant (42, //域ID
                          PARTICIPANT_QOS_DEFAULT,
                          0, //不需要侦听器OpenDDS :: DCPS :: DEFAULT_STATUS_MASK) ;
    如果 (! 参与者) {
      std :: cerr <<“create_participant失败。” << std :: endl; 返回1;
    }
```

2.1.4.2 注册数据类型和创建主题

接下来，我们初始化消息类型和主题。 请注意，如果该主题已在此域中使用相同的数据类型和兼容的QoS进行了初始化，create_topic () 调用将返回与现有主题相对应的引用。 如果类型或QoS

在我们的create_topic ()调用中指定的不匹配现有的主题，那么调用失败。 我们的订户也可以使用find_topic ()操作来简单地检索现有的主题。

```
Messenger :: MessageTypeSupport_var mts = new
  Messenger :: MessageTypeSupportImpl () ;
if (DDS :: RETCODE_OK! = mts-> register_type (participant, "") ) {
  std :: cerr <<"无法注册MessageTypeSupport。" << std :: endl; 返回1;
}

CORBA :: String_var type_name = mts-> get_type_name () ;  DDS ::

Topic_var topic =
  参与者 - > create_topic ("电影讨论列表",
                                type_name,
                                TOPIC_QOS_DEFAULT,
                                0,   // 不需要侦听器OpenDDS ::
                                DCPS :: DEFAULT_STATUS_MASK) ;
if (! topic) {
  std :: cerr <<"create_topic失败。" << std :: endl; 返回1;
}
```

2.1.4.3 创建用户

接下来，我们使用默认的QoS创建用户。

```
//创建订户DDS :: Subscriber_var
sub =
  participant-> create_subscriber (SUBSCRIBER_QOS_DEFAULT,
                                   0,   // 不需要侦听器OpenDDS ::
                                   DCPS :: DEFAULT_STATUS_MASK) ;
如果 (! sub) {
  std :: cerr <<"create_subscriber失败。" << std :: endl; 返回1;
}
```

2.1.4.4 创建一个DataReader和Listener

我们需要将侦听器对象与我们创建的数据读取器关联起来，因此我们可以使用它来检测数据何时可用。 下面的代码构造了侦听器对象。 DataReaderListenerImpl类显示在下一小节中。

```
DDS :: DataReaderListener_var侦听器 (新的DataReaderListenerImpl) ;
```

监听器分配在堆上，并分配给DataReaderListener_var对象。 此类型提供引用计数行为，以便在删除最后一个引用时自动清除侦听器。 这种用法对于OpenDDS应用程序代码中的堆分配是很典型的，并使应用程序开发人员不必主动管理分配对象的使用寿命。

现在我们可以创建数据读取器并将其与我们的主题，默认的QoS属性以及我们刚创建的侦听器对象相关联。

```
// 创建数据库
```

```
DDS::DataReader_var dr =
  子> create_datareader (主题,
                         DATAREADER_QOS_DEFAULT,
                         侦听器, OpenDDS :: DCPS ::
                         DEFAULT_STATUS_MASK) ;
if (! dr) {
  std :: cerr <<"create_datareader失败。" << std :: endl; 返回1;
}
```

此线程现在可以自由执行其他应用程序工作。 当样本可用时，我们的侦听器对象将在OpenDDS线程上调用。

2.1.5 数据读取器监听器实现

我们的监听器类实现了DDS规范定义的DDS :: DataReaderListener接口。 DataReaderListener被封装在一个DCPS :: LocalObject中，它可以解析含糊不清的成员，如_narrow和_ptr_type。 接口定义了一些我们必须执行的操作，每个操作都被调用来通知我们不同的事件。 OpenDDS :: DCPS :: DataReaderListener为OpenDDS的特殊需求（如断开连接和重新连接的事件更新）定义操作。 这里是接口定义：

```
模块DDS {
  本地接口DataReaderListener: 监听器{
    void on_requested_deadline_missed (在DataReader阅读器中,
                                       在RequestedDeadlineMissedStatus状态) ; void
    on_requested_incompatible_qos (在DataReader阅读器中,
                                   处于RequestedIncompatibleQosStatus状态) ; void
    on_sample_rejected (在DataReader阅读器中,
                        在SampleRejectedStatus状态) ; void
    on_liveliness_changed (在DataReader阅读器中,
                           在LivelinessChangedStatus状态) ; void
    on_data_available (在DataReader阅读器中) ;
    void on_subscription_matched (在DataReader阅读器中,
                                  在SubscriptionMatchedStatus状态) ; void
    on_sample_lost (在DataReader阅读器中, 处于SampleLostStatus状态) ;
  };
};
```

我们的示例监听器类使用简单的打印语句将这些监听器操作的大部分存根出来。 这个例子中唯一真正需要的操作是on_data_available ()，它是我们需要探索的这个类中唯一的成员函数。

```
void DataReaderListenerImpl :: on_data_available (DDS :: DataReader_ptr reader)
{
  ++num_reads_;

  尝试{
    Messenger :: MessageDataReader_var reader_i = Messenger :: MessageDataReader :: _ narrow (reader) ;
    if (! reader_i) {
      std :: cerr <<"阅读: _narrow失败。" << std :: endl; 返回;
    }
```

上面的代码将传入侦听器的通用数据读取器缩小到特定类型MessageDataReader接口。 以下代码将从消息中获取下一个示例

读者。 如果取得成功并返回有效数据，我们会打印出每个消息的字段。

```
Messenger ::留言信息; DDS ::
SampleInfo si;
DDS :: ReturnCode_t status = reader_i-> take_next_sample (message, si) ; if

 (status == DDS :: RETCODE_OK) {

  if (si.valid_data == 1) {

      std :: cout <<"消息: 主题              ="<< message.subject.in () << std :: endl
        << "          subject_id ="<< message.subject_id          << std::endl
        << "          从          ="<< message.from.in ()         << std::endl
        << "          计数        ="<< message.count              << std::endl
        << "          文本        ="<< message.text.in ()         << std::endl;
  }
  否则如果 (si.instance_state == DDS :: NOT_ALIVE_DISPOSED_INSTANCE_STATE)
  {
    std :: cout <<"实例被放置"<< std :: endl;
  }
  否则如果 (si.instance_state == DDS :: NOT_ALIVE_NO_WRITERS_INSTANCE_STATE)
  {
    std :: cout <<"实例未注册"<< std :: endl;
  }
  其他
  {
            std :: cerr <<"错误: 收到未知的实例状态"
           << si.instance_state << std::endl;
  }
} else if (status == DDS :: RETCODE_NO_DATA) {
    cerr <<"错误: 读者收到DDS :: RETCODE_NO_DATA! "  << std :: endl;
} else {
    cerr <<"错误: 读取消息: 错误: "<< status << std :: endl;
}
```

请注意，样本读取可能包含无效数据。 valid_data标志指示样本是否有有效的数据。 有两个样本的无效数据传递给侦听器回调用于通知目的。 一个是处理通知，在DataWriter显式调用dispose ()时收到。 另一个是未注册的通知，在DataWriter显式调用unregister ()时收到。 处理通知是在实例状态设置为NOT_ALIVE_DISPOSED_INSTANCE_STATE的情况下交付的，而注销通知是在实例状态设置为NOT_ALIVE_NO_WRITERS_INSTANCE_STATE的情况下交付的。

如果有更多样品可用，服务再次调用此功能。 但是，一次读取单个样本的值并不是处理传入数据的最有效方式。 数据读取器界面以更高效的方式提供了许多用于处理数据的不同选项。 我们在Section中讨论一些这些操作2.2.

2.1.6 清理OpenDDS客户端

在发布者和订阅者完成之后，我们可以使用下面的代码来清理OpenDDS相关的对象：

```
participant-> delete_contained_entities () ;
```

```
DPF-> delete_participant (参加者) ;
TheServiceParticipant-> shutdown () ;
```

域参与者的delete_contained_entities () 操作将删除使用该参与者创建的所有主题，订阅者和发布者。 一旦完成，我们可以使用域参与者工厂删除我们的域参与者。

由于DDS中的数据发布和订阅是分离的，因此如果在发送的所有数据已经被订阅接收之前，发布被分离（关闭），则不保证数据被传递。 如果应用程序要求接收所有发布的数据，则可以使用wait_for_acknowledgements () 操作来允许发布等待，直到收到所有写入的数据。 数据读取器必须具有RELIABILITY QoS（这是默认值）的RELIABLE设置才能使wait_for_acknowledgements () 正常工作。 此操作在各个DataWriters上调用，并包含用于限制等待时间的超时值。 下面的代码演示了如何使用wait_for_acknowledgements () 来阻塞15秒，等待订阅确认收到所有写入的数据：

```
DDS :: Duration_t shutdown_delay = {15, 0};
DDS :: ReturnCode_t结果;
result = writer-> wait_for_acknowledgments (shutdown_delay) ;  if
 (result! = DDS :: RETCODE_OK) {
  std :: cerr <<"等待确认"
              <<"数据正在通过订阅接收，一些数据"
              <<"可能没有交付"。  << std :: endl;
}
```

2.1.7 运行示例

我们现在准备好运行我们简单的例子。 在其自己的窗口中运行这些命令应该使您能够最轻松地理解输出。

首先，我们将启动一个DCPSInfoRepo服务，以便我们的发布者和订阅者可以找到另一个。

注意 *如果通过配置环境以使用RTPS发现来使用对等发现，则此步骤不是必需的。*

DCPSInfoRepo可执行文件位于$ DDS_ROOT / bin / DCPSInfoRepo中。 当我们启动DCPSInfoRepo时，我们需要确保发布者和订阅者应用程序进程也可以找到启动的DCPSInfoRepo。 这些信息可以通过以下三种方式之一提供：a）命令行上的参数b）生成并放置在共享文件中供应用程序使用；c）放置在配置文件中的参数供其他进程使用。 对于我们的简单示例，我们将通过将DCPSInfoRepo的位置属性生成到文件中来使用选项"b"，以便我们的简单发布者和订阅者可以读取它并连接到它。

从您当前的目录中输入：

视窗：

```
%DDS_ROOT%\ bin \ DCPSInfoRepo -o simple.ior
```

Unix的：

```
$DDS_ROOT/bin/DCPSInfoRepo -o simple.ior
```

-o参数指示DCPSInfoRepo生成连接信息给文件simple.ior，供发布者和订阅者使用。 在一个单独的窗口中，导航到包含simple.ior文件的相同目录，并通过输入以下命令启动订阅者应用程序：

视窗：

```
订户-DCPSInfoRepo文件：//simple.ior
```

Unix的：

```
./subscriber -DCPSInfoRepo file：//simple.ior
```

命令行参数指示应用程序使用指定的文件来定位DCPSInfoRepo。 我们的用户现在正在等待消息发送，所以我们现在将在一个单独的窗口中使用相同的参数启动发布者：

视窗：

```
发布者-DCPSInfoRepo文件：//simple.ior
```

Unix的

```
./publisher -DCPSInfoRepo file：//simple.ior
```

发布者连接到DCPSInfoRepo以查找任何订户的位置，并开始发布消息并将其写入控制台。在订户窗口中，您现在也应该看到订阅者的控制台输出，该订阅者正在阅读来自主题的消息，演示了简单的发布和订阅应用程序。

您可以在第7.3.3节和第7.4.5.5节中详细了解如何配置应用程序以实现RTPS和其他更高级的配置选项。 有关配置和使用DCPSInfoRepo的更多信息，请参阅第7.3节和第9章。有关设置和使用修改应用程序行为的QoS功能的更多信息，请参阅第3章。

2.1.8 用RTPS运行我们的例子

之前的OpenDDS示例演示了如何使用基本的OpenDDS配置和使用DCPSInfoRepo服务的集中式发现来构建和执行OpenDDS应用程序。 以下详细说明使用RTPS运行相同的示例以进行发现和可互操作的传输所需的内容。 当您的OpenDDS应用程序需要与非OpenDDS实现进行互操作时，这很重要

DDS规范，或者如果您不想在您的OpenDDS部署中使用集中式发现。

上面的Messenger例子的编码和构建在使用RTPS时不会改变，所以您不需要修改或重建发布者和订阅者服务。 这是OpenDDS架构的一个优势，即启用RTPS功能，这是一个配置练习。 第7章将介绍有关所有可用传输（包括RTPS）的配置的详细信息，但是，对于此练习，我们将使用发布者和订户将共享的配置文件启用Messenger示例的RTPS。

导航到您的发布者和订阅者所在的目录。 创建一个名为rtps.ini的新文本文件，并使用以下内容填充它：

```
[common] DCPSGlobalTransportConfig = $文件
DCPSDefaultDiscovery = DEFAULT_RTPS

[transport/the_rtps_transport]
transport_type=rtps_udp
```

有关配置文件的更多详细信息将在以后的章节中详细说明，但是需要注意的两行代码将发现方法和数据传输协议设置为RTPS。

现在让我们重新运行我们的例子，通过首先启动订阅者进程，然后发布者开始发送数据，启用RTPS。 最好在不同的窗口中启动它们，以便两个人分开工作。

使用-DCPSConfigFile命令行参数启动订阅服务器，以指向新创建的配置文件...

视窗：

```
订户-DCPSConfigFile rtps.ini
```

Unix的：

```
./subscriber -DCPSConfigFile rtps.ini
```

现在用相同的参数启动发布者... Windows：

```
发布者-DCPSConfigFile rtps.ini
```

Unix的：

```
./publisher -DCPSConfigFile rtps.ini
```

由于在RTPS规范中没有集中式发现，所以有一些规定允许等待时间来允许发现。 规范将默认设置为30秒。 当上述两个过程开始时，可能会有高达30秒的延迟

取决于他们相隔多远。 这个时候可以在后面讨论的OpenDDS配置文件中进行调整7.3.3.

由于OpenDDS的体系结构允许可插拔的发现和可插入的传输，上面的rtps.ini文件中调出的两个配置条目可以使用RTPS独立更改，另一个不使用RTPS（例如使用DCPSInfoRepo进行集中式发现）。 在我们的例子中将它们都设置为RTPS使得这个应用程序可以与其他非OpenDDS实现完全互操作。

2.2 数据处理优化

2.2.1 在发布服务器中注册和使用实例

前面的示例通过样本的数据字段隐式指定它正在发布的实例。 当调用write（）时，数据写入器查询样本的关键字段以确定实例。 发布者还可以通过调用数据写入器上的register_instance（）来显式注册实例：

```
Messenger ::留言信息;  message.subject_id = 99;
DDS :: InstanceHandle_t handle =
    message_writer-> register_instance (消息) ;
```

在填充Message结构之后，我们调用register_instance（）函数来注册实例。 该实例由subject_id值99标识（因为我们之前将该字段指定为关键字）。

我们稍后可以在发布示例时使用返回的实例句柄：

```
DDS :: ReturnCode_t ret = data_writer-> write (message, handle) ;
```

使用实例句柄发布样本可能比强制编写者查询实例稍微有效一些，并且在实例上发布第一个样本时效率更高。 如果没有显式注册，第一次写入会导致OpenDDS为该实例分配资源。

由于资源限制可能导致实例注册失败，因此许多应用程序将注册视为设置发布者的一部分，并始终在初始化数据写入器时执行。

2.2.2 读取多个样本

DDS规范提供了一些读取和写入数据样本的操作。 在上面的例子中，我们使用了take_next_sample（）操作来读取

下一个样本并从读者那里“获取”它的所有权。 消息数据读取器还具有以下操作。

- take () – 从读取器获取最多max_samples值的序列
- take_instance () – 获取指定实例的值序列
- take_next_instance () – 获取属于同一实例的样本序列，而不指定实例。

还有一些“读取”操作对应于获取相同值的这些“获取”操作中的每一个，但将样本留在读取器中，并简单地将它们标记为在SampleInfo中读取。

由于这些其他操作读取一系列的值，所以当样本快速到达时它们更有效率。 下面是一个示例调用take ()，一次最多可以读取5个样本。

```
MessageSeq消息 (5) ;  DDS :: SampleInfoSeq
sampleInfos (5) ;  DDS :: ReturnCode_t
status =
    message_dr->采取 (消息,
                          sampleInfos,
                    5, DDS :: ANY_SAMPLE_STATE,
                    DDS :: ANY_VIEW_STATE, DDS ::
                    ANY_INSTANCE_STATE) ;
```

这三个状态参数可能专门从读取器返回哪些样本。 有关其使用的详细信息，请参阅DDS规范。

2.2.3 零拷贝读取

返回一系列样本的读取操作为用户提供了获取样本副本（单个副本读取）或对样本的引用（零拷贝读取）的选项。 与大样本类型的单拷贝读取相比，零拷贝读取可以具有显着的性能改进。 测试显示，通过使用零拷贝读取，8KB或更少的样本不会获得太多收益，但在小样本上使用零拷贝几乎没有性能损失。

应用程序开发人员可以通过调用带有max_len为零的样本序列的take ()或read ()来指定使用零拷贝读取优化。 消息序列和样本信息序列构造函数都将max_len作为其第一个参数，并指定默认值零。 下面的示例代码是从中获取的
DevGuideExamples / DCPS / Messenger_ZeroCopy / DataReaderListenerImpl.cpp:

```
Messenger :: MessageSeq消息;  DDS ::
SampleInfoSeq信息;

// 获得对样本的引用 (样本的零拷贝读取)
```

```
DDS :: ReturnCode_t status = dr-> take (messages,
                                        info, DDS ::
                                        LENGTH_UNLIMITED, DDS ::
                                        ANY_SAMPLE_STATE, DDS ::
                                        ANY_VIEW_STATE, DDS ::
                                        ANY_INSTANCE_STATE) ;
```

在零拷贝读/写和单拷贝读取/读取之后，样本和信息序列的长度被设置为读取的样本的数量。 对于零拷贝读取，max_len被设置为> =长度的值。

由于应用程序代码要求对数据进行零拷贝贷款，因此在完成数据时必须返还该贷款：

```
dr-> return_loan (messages, info) ;
```

调用return_loan （）会导致将序列的max_len设置为0，并将其拥有的成员设置为false，从而允许将相同的序列用于另一个零拷贝读取。

如果数据样本序列构造函数和信息序列构造函数的第一个参数被更改为大于零的值，则返回的样本值将是副本。 在复制值时，应用程序开发人员可以选择调用return_loan （），但不需要这样做。

如果未指定序列构造函数的max_len （第一个）参数，则默认为0；因此使用零拷贝读取。 由于这个默认值，一个序列会在销毁时自动调用return_loan （）。 为了符合DDS规范，并且可以移植到DDS的其他实现中，应用程序不应该依赖此自动return_loan （）功能。

样本和信息序列的第二个参数是序列中可用的最大插槽。 如果read （）或take （）操作的max_samples参数大于此值，则由read （）或take （）返回的最大采样将受序列构造函数的此参数限制。

尽管应用程序可以通过调用长度（len）操作来更改零拷贝序列的长度，但建议您不要这样做，因为此调用会导致复制数据并创建单个拷贝序列的样本。

C章节3

服务质量

3.1 介绍

前面的示例使用各种实体的默认QoS策略。 本章讨论在OpenDDS中实施的QoS策略及其使用细节。 有关本章中讨论的策略的更多信息，请参阅DDS规范。

3.2 QoS策略

每个策略定义一个结构来指定其数据。 每个实体都支持一部分策略，并定义一个由支持的策略结构组成的QoS结构。 给定实体的一组允许策略受嵌套在其QoS结构中的策略结构的约束。 例如，发布者的QoS结构在规范的IDL中被定义如下：

```
模块DDS {
  结构PublisherQos {PresentationQosPolicy演示;
   PartitionQosPolicy分区;  GroupDataQosPolicy
   group_data;  EntityFactoryQosPolicy
   entity_factory;
 };
};
```

设置策略与获取已经设置了默认值的结构一样简单，根据需要修改个别策略结构，然后将QoS结构应用于实体（通常在创建时）。 我们举例说明如何在Section中获取各种实体类型的默认QoS策略3.2.1.

应用程序可以通过调用实体上的set_qos（）操作来更改任何实体的QoS。 如果QoS是可改变的，则现有的关联如果不再兼容则被删除，如果新的关联兼容，则添加新的关联。 DCPSInfoRepo根据QoS规范重新评估QoS兼容性和关联。 如果兼容性检查失败，则调用set_qos（）将返回一个错误。 协会重新评估可能会导致现有协会被删除或增加新的协会。

如果用户尝试更改不可变（不可更改）的QoS策略，则
set_qos（）返回DDS :: RETCODE_IMMUTABLE_POLICY。

QoS策略的一个子集是可以改变的。 一些可改变的QoS策略（如USER_DATA，TOPIC_DATA，GROUP_DATA，LIFESPAN，OWNERSHIP_STRENGTH，TIME_BASED_FILTER，ENTITY_FACTORY，WRITER_DATA_LIFECYCLE和READER_DATA_LIFECYCLE）不要求
兼容性和关联重新评估。 DEADLINE和LATENCY_BUDGET QoS策略要求兼容性重新评估，但不要关联。 PARTITION QoS策略不需要兼容性重新评估，但确实需要关联重新评估。 DDS规范将TRANSPORT_PRIORITY列为可更改的，但OpenDDS实现不支持动态修改此策略。

3.2.1 默认QoS策略值

应用程序通过实例化实体的适当类型的QoS结构并通过参考适当的方式传递它来获得实体的默认QoS策略

在适当的工厂实体上执行get_default_entity_qos（）操作。 （例如，您将使用域参与者来获取发布者或订阅者的默认QoS。）以下示例说明如何获取发布者，订阅者，主题，域参与者，数据写入者和数据读取者的默认策略。

```
// 从DomainParticipant获取默认发布者QoS: DDS :: PublisherQos pub_qos;
DDS::ReturnCode_t ret;
ret = domain_participant-> get_default_publisher_qos (pub_qos) ; 如果
 (DDS :: RETCODE_OK! = ret) {
   std :: cerr <<“无法获得默认发布者QoS”<< std :: endl;
}

// 从DomainParticipant获取默认的Subscriber QoS: DDS :: SubscriberQos sub_qos;
ret = domain_participant-> get_default_subscriber_qos (sub_qos) ; 如果
 (DDS :: RETCODE_OK! = ret) {
   std :: cerr <<“无法获得默认用户QoS”<< std :: endl;
}

// 从DomainParticipant获取默认主题QoS: DDS :: TopicQos
topic_qos;
ret = domain_participant-> get_default_topic_qos (topic_qos) ; 如果
 (DDS :: RETCODE_OK! = ret) {
   std :: cerr <<“无法获得默认主题QoS”<< std :: endl;
}

// 从DomainParticipantFactory获取默认的DomainParticipant QoS: DDS :: DomainParticipantQos dp_qos;
ret = domain_participant_factory-> get_default_participant_qos (dp_qos) ; 如果 (DDS ::
RETCODE_OK! = ret) {
   std :: cerr <<“无法获得默认参与者QoS”<< std :: endl;
}

// 从发布者获取默认的DataWriter QoS: DDS :: DataWriterQos
dw_qos;
ret = pub-> get_default_datawriter_qos (dw_qos) ; 如果
 (DDS :: RETCODE_OK! = ret) {
   std :: cerr <<“无法获取默认数据写入器QoS”<< std :: endl;
}

// 从订阅服务器获取默认的DataReader服务质量: DDS ::
DataReaderQos dr_qos;
ret = sub> get_default_datareader_qos (dr_qos) ; 如果
 (DDS :: RETCODE_OK! = ret) {
   std :: cerr <<“无法获取默认的数据读取器QoS”<< std :: endl;
}
```

下表总结了OpenDDS中可应用策略的每种实体类型的默认QoS策略。

表3-1默认的DomainParticipant QoS策略

政策	会员	默认值
用户数据	值	（空序列）
ENTITY_FACTORY	autoenable_created_entities	真正

表3-2默认主题QoS策略

政策	会员	默认值
TOPIC_DATA	值	(空序列)
耐久性	kind service_cleanup_delay.sec service_cleanup_delay.nanosec	VOLATILE_DURABILITY_QOS DURATION_ZERO_SEC DURATION_ZERO_NSEC
DURABILITY_SERVICE	service_cleanup_delay.sec service_cleanup_delay.nanosec history_kind history_depth max_samples max_instances max_samples_per_instance	DURATION_ZERO_SEC DURATION_ZERO_NSEC KEEP_LAST_HISTORY_QOS 1 LENGTH_UNLIMITED LENGTH_UNLIMITED LENGTH_UNLIMITED
截止日期	period.sec period.nanosec	DURATION_INFINITY_SEC DURATION_INFINITY_NSEC
LATENCY_BUDGET	duration.sec duration.nanosec	DURATION_ZERO_SEC DURATION_ZERO_NSEC
生动活泼	kind lease_duration.sec lease_duration.nanosec	AUTOMATIC_LIVELINESS_QOS DURATION_INFINITY_SEC DURATION_INFINITY_NSEC
可靠性	种类max_blocking_time.sec max_blocking_time.nanosec	BEST_EFFORT_RELIABILITY_QOS DURATION_INFINITY_SEC DURATION_INFINITY_NSEC
DESTINATION_ORDER	类	BY_RECEPTION_TIMESTAMP_ DESTINATIONORDER_QOS
历史	善良的 深度	KEEP_LAST_HISTORY_QOS 1
RESOURCE_LIMITS	max_samples max_instances max_samples_per_instance	LENGTH_UNLIMITED LENGTH_UNLIMITED LENGTH_UNLIMITED
的transport_priority	值	0
寿命	duration.sec duration.nanosec	DURATION_INFINITY_SEC DURATION_INFINITY_NSEC
所有权	类	SHARED_OWNERSHIP_QOS

表3-3默认发布者QoS策略

政策	会员	默认值
介绍	access_scope coherent_access ordered_access	INSTANCE_PRESENTATION_QOS 0 0
划分	名称	(空序列)
GROUP_DATA	值	(空序列)
ENTITY_FACTORY	autoenable_created_entities	真正

表3-4缺省订户QoS策略

政策	会员	默认值
介绍	access_scope coherent_access ordered_access	INSTANCE_PRESENTATION_QOS 0 0
划分	名称	(空序列)
GROUP_DATA	值	(空序列)

政策	会员	默认值
ENTITY_FACTORY	autoenable_created_entities	真正

表3-5默认的DataWriter QoS策略

政策	会员	默认值
耐久性	kind service_cleanup_delay.sec service_cleanup_delay.nanosec	VOLATILE_DURABILITY_QOS DURATION_ZERO_SEC DURATION_ZERO_NSEC
DURABILITY_SERVICE	service_cleanup_delay.sec service_cleanup_delay.nanosec history_kind history_depth max_samples max_instances max_samples_per_instance	DURATION_ZERO_SEC DURATION_ZERO_NSEC KEEP_LAST_HISTORY_QOS 1 LENGTH_UNLIMITED LENGTH_UNLIMITED LENGTH_UNLIMITED
截止日期	period.sec period.nanosec	DURATION_INFINITY_SEC DURATION_INFINITY_NSEC
LATENCY_BUDGET	duration.sec duration.nanosec	DURATION_ZERO_SEC DURATION_ZERO_NSEC
生动活泼	kind lease_duration.sec lease_duration.nanosec	AUTOMATIC_LIVELINESS_QOS DURATION_INFINITY_SEC DURATION_INFINITY_NSEC
可靠性	种类max_blocking_time.sec max_blocking_time.nanosec	RELIABLE_RELIABILITY_QOS[2] 0 100000000 (100 ms)
DESTINATION_ORDER	类	BY_RECEPTION_TIMESTAMP_ DESTINATIONORDER_QOS
历史	善良的 深度	KEEP_LAST_HISTORY_QOS 1
RESOURCE_LIMITS	max_samples max_instances max_samples_per_instance	LENGTH_UNLIMITED LENGTH_UNLIMITED LENGTH_UNLIMITED
的transport_priority	值	0
寿命	duration.sec duration.nanosec	DURATION_INFINITY_SEC DURATION_INFINITY_NSEC
用户数据	值	(空序列)
所有权	类	SHARED_OWNERSHIP_QOS
OWNERSHIP_STRENGTH	值	0
WRITER_DATA_LIFECYCLE	autodispose_unregistered_instances	1

表3-6默认的DataReader QoS策略

政策	会员	默认值
耐久性	kind service_cleanup_delay.sec service_cleanup_delay.nanosec	VOLATILE_DURABILITY_QOS DURATION_ZERO_SEC DURATION_ZERO_NSEC
截止日期	period.sec period.nanosec	DURATION_INFINITY_SEC DURATION_INFINITY_NSEC
LATENCY_BUDGET	duration.sec duration.nanosec	DURATION_ZERO_SEC DURATION_ZERO_NSEC
生动活泼	kind lease_duration.sec lease_duration.nanosec	AUTOMATIC_LIVELINESS_QOS DURATION_INFINITY_SEC DURATION_INFINITY_NSEC

2对于OpenDDS版本，最高2.0，数据编写者的默认可靠性类型是尽力而为。 对于版本2.0.1和更高版本，将其更改为可靠（符合DDS规范）。

政策	会员	默认值
可靠性	类 max_blocking_time.sec max_blocking_time.nanosec	BEST_EFFORT_RELIABILITY_QOS DURATION_INFINITY_SEC DURATION_INFINITY_NSEC
DESTINATION_ORDER	类	BY_RECEPTION_TIMESTAMP_ DESTINATIONORDER_QOS
历史	善良的 深度	KEEP_LAST_HISTORY_QOS 1
RESOURCE_LIMITS	max_samples max_instances max_samples_per_instance	LENGTH_UNLIMITED LENGTH_UNLIMITED LENGTH_UNLIMITED
用户数据	值	(空序列)
所有权	类	SHARED_OWNERSHIP_QOS
TIME_BASED_FILTER	minimum_separation.sec minimum_separation.nanosec	DURATION_ZERO_SEC DURATION_ZERO_NSEC
READER_DATA_LIFECYCLE	autopurge_nowriter_samples_delay.sec autopurge_nowriter_samples_delay.nanosec autopurge_disposed_samples_delay.sec autopurge_disposed_samples_delay.nanosec	DURATION_INFINITY_SEC DURATION_INFINITY_NSEC DURATION_INFINITY_SEC DURATION_INFINITY_NSEC

3.2.2 生动活泼

LIVELINESS策略通过其各自的QoS结构的活动成员适用于主题，数据读取器和数据写入器实体。 在一个主题上设置这个策略意味着这个主题的所有数据读者和数据编写者都是有效的。 以下是与生动性QoS策略相关的IDL：

```
enum LivelinessQPolicyKind
   {AUTOMATIC_LIVELINESS_QOS,
   MANUAL_BY_PARTICIPANT_LIVELINESS_QOS,
   MANUAL_BY_TOPIC_LIVELINESS_QOS
};

struct LivelinessQosPolicy
   { LivelinessQosPolicyKind kind;
   Duration_t lease_duration;
};
```

LIVELINESS策略控制服务何时以及如何确定参与者是否还活着，这意味着他们仍然可以访问和活跃。 种类成员设置指示活动是由服务自动声明还是由指定实体手动声明。 AUTOMATIC_LIVELINESS_QOS的设置意味着如果参与者没有为lease_duration发送任何网络流量，则服务将发送活跃性指示。 MANUAL_BY_PARTICIPANT_LIVELINESS_QOS或MANUAL_BY_TOPIC_LIVELINESS_QOS设置
意味着指定的实体（针对“按主题”设置的数据写入器或针对“按参与者”设置的域参与者）必须写入样本或在指定的心跳间隔内手动断言其活跃度。 期望的心跳间隔由lease_duration成员指定。 默认的租约期限是一个预先定义的无限值，它禁用任何生动性测试。

要在不发布样本的情况下手动确定生动性，应用程序必须在指定的心跳间隔内对数据写入器（对于“按主题”设置）或域参与者（对于“按参与者”设置）调用assert_liveliness 。

数据编写者指定（提供）他们自己的生动性标准，数据读者指定（请求）他们的作家的生动活泼性。 在租约期内没有听到的作者（通过编写样本或声明活泼性）引起LIVELINESS_CHANGED_STATUS通信状态的改变，并通知应用程序（例如，通过调用数据读取器监听器的on_liveliness_changed（）回调操作）发信号通知任何相关的等待集）

在数据编写者和数据读取者之间建立关联时考虑这一政策。 协会双方的价值必须相互协调才能建立协会。 兼容性是通过比较数据读者所要求的活泼性和数据写者的活泼性来确定的。 在确定兼容性时，考虑生动性（自动，主题手动，参与者手册）和租约期限的价值。 作者所提供的那种生动活泼必须大于或等于读者所要求的那种活泼。 活泼的价值观的排序如下：

```
MANUAL_BY_TOPIC_LIVELINESS_QOS >
MANUAL_BY_PARTICIPANT_LIVELINESS_QOS >
AUTOMATIC_LIVELINESS_QOS
```

另外，作者提供的租约期限必须小于或等于读者所要求的租约期限。 必须满足这两个条件才能使提供的和请求的生动性策略设置被认为是兼容的，并建立关联。

3.2.3 可靠性

可靠性策略通过各自QoS结构的可靠性成员应用于主题，数据读取器和数据写入器实体。以下是与可靠性QoS策略相关的IDL：

```
枚举ReliabilityQosPolicyKind
  {BEST_EFFORT_RELIABILITY_QOS,
  RELIABLE_RELIABILITY_QOS
};

struct ReliabilityQosPolicy
  { ReliabilityQosPolicyKind kind;
  Duration_t max_blocking_time;
};
```

该政策控制数据读者和作者如何处理他们处理的数据样本。 “尽力而为”的价值（BEST_EFFORT_RELIABILITY_QOS）对样品的可靠性没有任何承诺，并且在某些情况下可能会降低样品的质量。 “可靠”值（RELIABLE_RELIABILITY_QOS）表示服务最终应将所有值传递给符合条件的数据读取器。

当历史QoS策略设置为“全部保留”时，使用此策略的max_blocking_time成员，并且由于资源限制，写入器无法继续。 发生这种情况时，写入程序块超过指定的时间，写入失败，返回代码超时。 数据读取器和主题的默认值是“尽力而为”，而数据写入器的默认值是“可靠的”。

在创建数据写入者和数据读取者之间的关联期间考虑这个策略。 协会双方的价值必须兼容才能创建协会。 数据写入器的可靠性类型必须大于或等于数据读取器的值。

3.2.4 历史

HISTORY策略确定如何在特定实例的数据写入器和数据读取器中保存样本。 对于数据编写者来说，这些值一直保留到发布者检索并成功发送给所有连接的订阅者。 对于数据读取器，这些值一直保持到应用程序“采取”为止。 该策略通过其各自的QoS结构的历史成员适用于主题，数据读取器和数据写入器实体。 以下是与历史QoS策略相关的IDL：

```
枚举HistoryQosPolicyKind
  {KEEP_LAST_HISTORY_QOS,
  KEEP_ALL_HISTORY_QOS
};

struct HistoryQosPolicy
  {HistoryQosPolicyKind kind; 长深
};
```

“keep all”值（KEEP_ALL_HISTORY_QOS）指定应该保留该实例的所有可能的采样。 当指定“全部保留”并且未读取样本的数量等于max_samples_per_instance的“资源限制”字段时，则拒绝所有进入的样本。

“keep last”值（KEEP_LAST_HISTORY_QOS）指定只保留最后一个深度值。 当数据写入程序包含给定实例的深度样本时，写入该实例的新样本将排队等待发送，并丢弃最旧的未发送样本。 当数据读取器包含给定实例的深度采样时，将保留该实例的任何传入采样，并丢弃最旧的采样。

此政策默认为“保持最后”，深度为1。

3.2.5 耐久性

DURABILITY策略控制着数据写入者在发送给已知用户之后是否应该维护样本。 该策略通过其各自QoS结构的耐久性成员适用于主题，数据读取器和数据写入器实体。 以下是与耐久性QoS策略相关的IDL：

```
枚举DurabilityQosPolicyKind {VOLATILE_DURABILITY_QOS,        //
  最小耐久性TRANSIENT_LOCAL_DURABILITY_QOS,
  TRANSIENT_DURABILITY_QOS,
  PERSISTENT_DURABILITY_QOS          // 最大的耐用性
};

struct DurabilityQosPolicy
  { DurabilityQosPolicyKind kind;
};
```

默认情况下，这种类型是VOLATILE_DURABILITY_QOS。

VOLATILE_DURABILITY_QOS的耐用性意味着样本在被发送给所有已知用户之后被丢弃。 作为副作用，用户不能恢复连接之前发送的样本。

持久性类型TRANSIENT_LOCAL_DURABILITY_QOS意味着与数据写入器关联/连接的数据读取器将被发送到数据写入器历史中的所有样本。

持久性种类的TRANSIENT_DURABILITY_QOS意味着样本比数据写入器长，并且只要过程处于活动状态。 样品保存在记忆中，但没有坚持永久存储。 订阅相同的主题和同一个域内的分区的数据读取器将被发送属于同一主题/分区的所有缓存样本。

PERSISTENT_DURABILITY_QOS的持久性类型基本上提供了与暂态持久性相同的功能，除了缓存的样本被保留并且将在流程破坏之后存活。

当指定临时或持久性持久性时，DURABILITY_SERVICE QoS策略将为持久性缓存指定其他调整参数。

在创建数据写入者和数据读取者之间的关联期间考虑耐久性策略。 协会双方的价值必须兼容才能创建协会。 数据写入器的耐久性种类值必须大于或等于数据读取器的相应值。 耐久性种类值的排序如下：

```
PERSISTENT_DURABILITY_QOS >
TRANSIENT_DURABILITY_QOS >
TRANSIENT_LOCAL_DURABILITY_QOS >
VOLATILE_DURABILITY_QOS
```

3.2.6 DURABILITY_SERVICE

DURABILITY_SERVICE策略控制删除TRANSIENT或PERSISTENT持久性缓存中的示例。 此策略通过其各自的QoS结构的durability_service成员应用于主题和数据写入器实体，并提供指定样本缓存的HISTORY和RESOURCE_LIMITS的方法。 以下是与耐久性服务QoS策略相关的IDL：

```
struct DurabilityServiceQosPolicy {
  Duration_t              service_cleanup_delay;
  HistoryQosPolicyKind    history_kind;
  长                      history_depth;
  长                      max_samples;
  长                      max_instances;
  长                      max_samples_per_instance;
};
```

历史和资源限制成员与HISTORY和RESOURCE_LIMITS策略中的历史和资源限制成员类似，虽然独立。 service_cleanup_delay可以设置为所需的值。 默认情况下，它被设置为零，这意味着永远不会清理缓存的样本。

3.2.7 RESOURCE_LIMITS

RESOURCE_LIMITS策略确定服务可以使用的资源量，以满足请求的QoS。 该策略通过其各自的QoS结构的resource_limits成员适用于主题，数据读取器和数据写入器实体。 以下是与资源限制QoS策略相关的IDL。

```
struct ResourceLimitsQosPolicy
  { long max_samples;
  long max_instances;
  long max_samples_per_instance;
};
```

max_samples成员指定单个数据写入者或数据读取者可以在其所有实例中管理的最大样本数。 max_instances成员指定数据写入器或数据读取器可以管理的最大实例数。 max_samples_per_instance成员指定单个数据写入器或数据读取器中可以为单个实例管理的最大样本数。 所有这些成员的值默认为无限（DDS :: LENGTH_UNLIMITED）。

数据写入器使用资源对写入数据写入器的样本进行排队，但是由于传输的背压，还没有发送到所有数据读取器。 资源被使用

由数据读取器排队已经接收但尚未从数据读取器读取/取得的样本。

3.2.8 划分

PARTITION QoS策略允许在域内创建逻辑分区。 它只允许数据读取器和数据写入器关联，如果它们具有匹配的分区字符串。 该策略通过其各自的QoS结构的分区成员适用于发布者和订户实体。 以下是与分区QoS策略相关的IDL。

```
struct PartitionQosPolicy
  { StringSeq name;
};
```

名称成员默认为空字符串序列。 默认分区名称是一个空字符串，并使实体参与默认分区。分区名称可以包含POSIX fnmatch函数（POSIX 1003.2-1992 B.6节）定义的通配符。

数据读取器和数据写入器关联的建立取决于发布和订阅结束时匹配的分区字符串。 不匹配分区不被视为失败，不会触发任何回调或设置任何状态值。

此政策的价值可能会随时更改。 对此策略的更改可能会导致关联被删除或添加。

3.2.9 截止日期

DEADLINE QoS策略允许应用程序检测数据在指定时间内未被写入或读取的时间。 这个策略通过它们各自的QoS结构的最后期限成员适用于主题，数据写入器和数据读取器实体。 以下是与截止日期QoS策略相关的IDL。

```
struct DeadlineQosPolicy
  { Duration_t period;
};
```

周期成员的默认值是无限的，不需要任何行为。 当此策略设置为有限值时，数据写入程序会监视应用程序所做的数据更改，并通过设置相应的状态条件并触发on_offered_deadline_missed（）侦听器回调来指示未遵守策略。 检测到数据在期限到期之前没有改变的数据读取器会设置相应的状态条件并触发on_requested_deadline_missed（）侦听器回调。

在创建数据写入者和数据读取者之间的关联期间考虑这个策略。 协会双方的价值必须兼容才能创建协会。 数据读取器的最后期限必须大于或等于数据写入器的相应值。

关联的实体启用后，此策略的值可能会更改。 在数据读取器或数据写入器的策略被制定的情况下，只有当改变与读取器或写入器参与的所有关联的远程端保持一致时才成功地应用改变。 如果一个主题的策略发生了变化，那么只会影响更改后创建的数据读者和作者。 任何现有的读者或作者，以及他们之间的任何现有关联，都不会受到主题政策价值变化的影响。

3.2.10 寿命

LIFESPAN QoS策略允许应用程序指定样本何时到期。 过期的样品不会被送到订户。 该策略通过其各自的QoS结构的生命周期成员适用于主题和数据写入器实体。 以下是与寿命QoS策略相关的IDL。

```
struct LifespanQosPolicy
  { Duration_t duration;
}
```

持续时间成员的默认值是无限的，这意味着样本永远不会过期。 当使用VOLATILE以外的DURABILITY类型时，OpenDDS当前支持发布方的过期样例检测。 当前的OpenDDS实现可能不会在数据写入器和数据读取器高速缓存放入高速缓存之后过期时将其删除。

此政策的价值可能会随时更改。 此策略的更改仅影响更改后写入的数据。

3.2.11 用户数据

USER_DATA策略通过其各自的QoS结构的user_data成员应用于域参与者，数据读取器和数据写入器实体。 以下是与用户数据QoS策略相关的IDL：

```
struct UserDataQosPolicy
  { sequence<octet> value;
};
```

默认情况下，值成员没有设置。 它可以被设置为可用于将信息附加到创建的实体的任何八位字节序列。 USER_DATA策略的值是

可用于相应的内置主题数据。 远程应用程序可以通过内置的主题获取信息，并将其用于自己的目的。 例如，应用程序可以通过USER_DATA策略附加安全证书，远程应用程序可以使用该策略来验证源。

3.2.12 TOPIC_DATA

TOPIC_DATA策略通过TopicQoS结构的topic_data成员应用于主题实体。 以下是与主题数据QoS策略相关的IDL：

```
struct TopicDataQosPolicy
  { sequence<octet> value;
};
```

默认情况下，该值未设置。 可以设置为将附加信息附加到创建的主题。 TOPIC_DATA策略的值在数据写入器，数据读取器和主题内置主题数据中可用。 远程应用程序可以通过内置的主题获取信息，并以应用程序定义的方式使用它。

3.2.13 GROUP_DATA

GROUP_DATA策略通过其各自QoS结构的group_data成员应用于发布者和订阅者实体。 以下是与组数据QoS策略相关的IDL：

```
struct GroupDataQosPolicy
  { sequence<octet> value;
};
```

默认情况下，值成员没有设置。 可以设置为将附加信息附加到创建的实体。 GROUP_DATA策略的值通过内置主题传播。 数据写入器内置的主题数据包含来自发布者的GROUP_DATA，数据读取器内置的主题数据包含来自订阅者的GROUP_DATA。 可以使用GROUP_DATA策略来实现与1.1.6中描述的PARTITION策略类似的匹配机制，除了可以基于应用程序定义的策略做出决定之外。

3.2.14 的transport_priority

TRANSPORT_PRIORITY策略通过其各自QoS策略结构的transport_priority成员应用于主题和数据写入实体。 以下是与TransportPriority QoS策略相关的IDL：

```
struct TransportPriorityQosPolicy
  { long value;
};
```

transport_priority的默认值成员是零。 这个策略被认为是传输层的一个提示，指出发送消息的优先级。 较高的值表示优先级较高。 OpenDDS将优先级值直接映射到线程和DiffServ代码点值。 默认优先级为零不会修改消息中的线程或代码点。

OpenDDS将尝试设置发送传输的线程优先级以及任何相关的接收传输。 传输优先级值从零（默认）线性映射到最大线程优先级，而不进行缩放。 如果最低线程优先级不等于零，则将其映射到传输优先级值零。 如果系统中的优先级值反转（较高的数值优先级较低），则OpenDDS会将这些值映射到从零开始的递增优先级值。 低于系统上最小（最低）线程优先级的优先级值映射到最低优先级。 大于系统上最大（最高）线程优先级的优先级值被映射到最高优先级。 在大多数系统中，线程优先级只能在进程调度程序设置为允许这些操作时才能设置。 设置进程调度器通常是一项特权操作，需要系统权限才能执行。 在基于POSIX的系统上，使用sched_get_priority_min（）和sched_get_priority_max（）的系统调用来确定线程优先级的系统范围。

如果传输实现支持，OpenDDS将尝试在用于为数据写入器发送数据的套接字上设置DiffServ代码点。 如果网络硬件符合代码点值，则更高的代码点值将导致更高优先级的样本更好（更快）的传输。 零的默认值将被映射到零的（默认）码点。 然后将从1到63的优先级值映射到相应的代码点值，并将更高的优先级值映射到最高的代码点值（63）。

OpenDDS当前不支持创建数据写入器后对transport_priority策略值的修改。 这可以通过创建新的数据编写器来解决，因为需要不同的优先级值。

3.2.15 LATENCY_BUDGET

LATENCY_BUDGET策略通过其各自的QoS策略结构的latency_budget成员应用于主题，数据读取器和数据写入器实体。 以下是与LatencyBudget QoS策略相关的IDL：

```
struct LatencyBudgetQosPolicy
  { Duration_t duration;
};
```

持续时间的默认值是零，表示延迟应该最小化。 这个政策被认为是传输层的一个暗示，表明样本的紧迫性

发送。 OpenDDS使用该值来限制延迟时间间隔，以报告从发布到订阅的样本传输的不可接受的延迟。 此政策目前仅用于监控目的。 使用TRANSPORT_PRIORITY策略来修改样本的发送。 数据写入器策略值仅用于兼容性比较，如果保留默认值0，则会导致数据读取器的所有请求持续时间值被匹配。

添加了一个额外的侦听器扩展，以允许超出策略持续时间设置的报告延迟。 OpenDDS :: DCPS :: DataReaderListener接口还有一个额外的操作，用于通知采样的测量传输延迟大于latency_budget策略持续时间。 这种方法的IDL是：

```
struct BudgetExceededStatus
  { long total_count;
  long_count_change;  DDS :: InstanceHandle_t
  last_instance_handle;
};

void on_budget_exceeded (
        在DDS :: DataReader阅读器中，
        在BudgetExceededStatus状态) ;
```

要使用扩展侦听器回调，您需要从扩展接口派生侦听器实现，如以下代码片段所示：

```
类DataReaderListenerImpl
       : 公共虚拟OpenDDS :: DCPS :: LocalObject <OpenDDS :: DCPS ::
         DataReaderListener>
```

然后，您必须为on_budget_exceeded () 操作提供一个非null实现。 请注意，您还需要为以下扩展操作提供空白实现：

```
on_subscription_disconnected ()
on_subscription_reconnected () on_subscription_lost
 () on_connection_deleted ()
```

OpenDDS还通过数据读取器的扩展接口提供汇总延迟统计信息。 该扩展接口位于OpenDDS :: DCPS模块中，IDL定义如下：

```
struct LatencyStatistics
  { GUID_t       出版物;
  unsigned long n;
  双              最大;
  双              最小;
  双              意思;
  双              方差;
};

typedef序列<LatencyStatistics> LatencyStatisticsSeq;
```

```
本地接口DataReaderEx: DDS :: DataReader {
  /// 获取一系列统计摘要。
  void get_latency_stats (inLatencyStatisticsSeq stats) ;

  /// 清除任何中间统计值。  void reset_latency_stats () ;

  /// 统计收集启用状态。 属性boolean
  statistics_enabled;
};
```

要收集统计摘要数据，您需要使用扩展接口。 您可以简单地通过动态转换OpenDDS数据读取器指针并直接调用操作来完成。 在以下示例中，我们假设通过调用DDS :: Subscriber :: create_datareader () 正确初始化reader:

```
DDS :: DataReader_var阅读器;
// ...

// 开始收集新的数据。 的dynamic_cast <OpenDDS :: DCPS ::
DataReaderImpl *> (reader.in () ) - >
  reset_latency_stats () ; 的dynamic_cast <OpenDDS :: DCPS ::
DataReaderImpl *> (reader.in () ) - >
  statistics_enabled (真) ;

// ...

// 收集数据。  OpenDDS :: DCPS ::
LatencyStatisticsSeq统计;
dynamic_cast <OpenDDS :: DCPS :: DataReaderImpl *> (reader.in () ) - > get_latency_stats (stats) ;
for (unsigned long i = 0; i <stats.length () ; ++ i)
{
  std :: cout <<“stats [”<< i <<“]: ”<< std :: endl;  std :: cout <<“n
  =“<< stats [i] .n << std :: endl;
  std :: cout <<“          max =“<< stats [i] .maximum << std :: endl; std ::
  cout <<”                 min =“<< stats [i] .minimum << std :: endl; std ::
  cout <<”                mean =“<< stats [i] .mean << std :: endl; std ::
  cout <<”variance =“<< stats [i] .variance << std :: endl;
}
```

3.2.16 ENTITY_FACTORY

ENTITY_FACTORY策略控制实体在创建时是否自动启用。 以下是与实体工厂QoS策略相关的IDL:

```
struct EntityFactoryQosPolicy
  { boolean
  autoenable_created_entities;
};
```

该策略可以应用于作为其他实体的工厂的实体，并控制这些工厂创建的实体是否在创建时自动启用。 此政策可以应用于域参与者工厂（作为域参与者的工厂），域参与者（作为

发布者，订阅者和主题的工厂），发布者（作为数据编写者的工厂）或订阅者（作为工厂为数据读者）。 该

autoenable_created_entities成员的默认值为true，表示实体在创建时自动启用。 希望在创建实体后一段时间显式启用实体的应用程序应将此策略的autoenable_created_entities成员的值设置为false，并将策略应用于相应的工厂实体。应用程序必须通过调用实体的enable（）操作来手动启用实体。

此政策的价值可能会随时更改。 此策略的更改仅影响更改后创建的实体。

3.2.17 介绍

演示QoS策略控制如何将发布者对实例的更改呈现给数据读取器。 它会影响这些更改的相对顺序以及此顺序的范围。 另外，这个政策引入了连贯变化集合的概念。

Presentation QoS的IDL如下：

```
枚举PresentationQosPolicyAccessScopeKind
  {INSTANCE_PRESENTATION_QOS,
  TOPIC_PRESENTATION_QOS, GROUP_PRESENTATION_QOS
};

struct PresentationQosPolicy
  { PresentationQosPolicyAccessScopeKind access_scope;
  boolean coherent_access;
  布尔ordered_access;
};
```

这些更改的范围（access_scope）指定应用程序可以被感知的级别：

- INSTANCE_PRESENTATION_QOS（默认值）表示独立发生更改。 对于coherent_access和ordered_access，实例访问实质上是一个no-op。 将这些值中的任何一个设置为true在订阅应用程序中都没有可观察到的影响。
- TOPIC_PRESENTATION_QOS表示接受的更改仅限于同一个数据读取器或数据写入器中的所有实例。
- GROUP_PRESENTATION_QOS表示接受的更改仅限于同一个发布者或订阅者中的所有实例。

连贯性变化（coherent_access）允许一个实例的一个或多个更改作为单个变更提供给关联的数据读取器。 如果数据阅读器没有收到发布者所做的整个连贯更改，则没有任何更改可用。 连贯性变化的语义与在许多关系数据库提供的事务中发现的类似。 默认情况下，coherent_access是错误的。

相关数据读取器也可以按照发布者发送的顺序（ordered_access）进行更改。 这与DESTINATION_ORDER QoS策略在本质上类似，但是ordered_access允许独立于实例排序来排序数据。 缺省情况下，ordered_access为false。

注意 *这个策略控制订户可用的样本的顺序和范围，但订阅者应用程序必须在读取样本时使用适当的逻辑来保证所要求的行为。 有关更多详细信息，请参阅1.4版DDS规范的第2.2.2.5.1.9节。* **注意**

3.2.18 DESTINATION_ORDER

DESTINATION_ORDER QoS策略控制给定实例内的样本可供数据读取器使用的顺序。 如果指定历史深度为1（缺省值），则该实例将反映由所有数据写入者写入该实例的最新值。 以下是目标订单Qos的IDL：

```
枚举DestinationOrderQosPolicyKind
  {BY_RECEPTION_TIMESTAMP_DESTINATIONORDER_QOS,
  BY_SOURCE_TIMESTAMP_DESTINATIONORDER_QOS
};

struct DestinationOrderQosPolicy
  { DestinationOrderQosPolicyKind
  kind;
};
```

BY_RECEPTION_TIMESTAMP_DESTINATIONORDER_QOS值（默认值）表示实例中的样本按照数据读取器接收到的顺序排序。 请注意，样本不一定是按相同数据写入器发送的顺序接收的。 为了执行这种排序，应该使用BY_SOURCE_TIMESTAMP_DESTINATIONORDER_QOS值。

BY_SOURCE_TIMESTAMP_DESTINATIONORDER_QOS值表示根据数据写入器提供的时间戳对实例内的样本进行排序。 应该注意的是，如果多个数据写入器写入相同的实例，则应注意确保时钟同步以防止数据读取器上的不正确排序。

3.2.19 WRITER_DATA_LIFECYCLE

WRITER_DATA_LIFECYCLE QoS策略控制由数据写入器管理的数据实例的生命周期。 Writer数据生命周期QoS策略的IDL如下：

```
结构WriterDataLifecycleQosPolicy {布尔
  autodispose_unregistered_instances;
};
```

当autodispose_unregistered_instances设置为true（缺省值）时，数据写入程序在未注册时处理实例。 在某些情况下，可能需要防止实例未注册时处置实例。 例如，该策略可以允许EXCLUSIVE数据写入器正常推迟到下一个数据写入器，而不会影响实例状态。 删除数据写入程序会在删除之前隐式取消注册其所有实例。

3.2.20 READER_DATA_LIFECYCLE

READER_DATA_LIFECYCLE QoS策略控制由数据读取器管理的数据实例的生命周期。 阅读器数据生命周期QoS策略的IDL如下：

```
struct ReaderDataLifecycleQosPolicy
  { Duration_t
  autopurge_nowriter_samples_delay; Duration_t
  autopurge_disposed_samples_delay;
};
```

通常情况下，数据读取器将维护所有实例的数据，直到实例没有更多的关联数据写入器，实例已经处理完毕，或者数据已被用户使用。

在某些情况下，可能需要限制这些资源的回收。 例如，这个策略可以允许加入后期的数据编写者在故障转移的情况下延长实例的生命周期。

autopurge_nowriter_samples_delay控制在实例转换到NOT_ALIVE_NO_WRITERS状态之前数据读取器在回收资源之前等待的时间。 默认情况下，autopurge_nowriter_samples_delay是无限的。

autopurge_disposed_samples_delay控制在实例转换到NOT_ALIVE_DISPOSED状态之前数据读取器在回收资源之前等待的时间。 默认情况下，autopurge_disposed_samples_delay是无限的。

3.2.21 TIME_BASED_FILTER

TIME_BASED_FILTER QoS策略控制数据读取器可能对数据实例的值更改感兴趣的频率。 以下是基于时间的过滤器QoS的IDL：

```
struct TimeBasedFilterQosPolicy
  { Duration_t minimum_separation;
};
```

数据读取器可以指定一个时间间隔（minimum_separation）。 这个间隔定义了实例值变化之间的最小延迟；这允许数据读取器在不影响关联的数据写入器的状态的情况下调整变化。 默认情况下，minimum_separation为零，表示没有数据被过滤。 这个QoS策略的确如此

不保留带宽，因为实例值更改仍然发送给订阅者进程。 它只影响通过数据读取器提供哪些样本。

3.2.22 所有权

OWNERSHIP策略控制是否有多个Data Writer能够为相同的数据对象实例编写样本。 所有权可以是EXCLUSIVE或SHARED。 以下是与所有权QoS策略相关的IDL：

```
枚举OwnershipQosPolicyKind
  {SHARED_OWNERSHIP_QOS,
  EXCLUSIVE_OWNERSHIP_QOS
};

struct OwnershipQosPolicy
  { OwnershipQosPolicyKind kind;
};
```

如果类成员设置为SHARED_OWNERSHIP_QOS，则允许多个Data Writer更新同一个数据对象实例。 如果类成员设置为EXCLUSIVE_OWNERSHIP_QOS，则只有一个数据写入器可以更新给定的数据对象实例（即数据写入器被认为是实例的所有者），关联的数据读取器将只能看到由此写入的样本数据写入器。 实例的所有者由OWNERSHIP_STRENGTH策略的值决定；具有最高强度值的数据写入者被认为是数据对象实例的所有者。 其他因素也可能影响所有权，比如最高实力数据作者是否“活着”（如“生命力”政策所定义），并且没有违反其提供的出版截止时间限制（由DEADLINE政策定义）。

3.2.23 OWNERSHIP_STRENGTH

当OWNERSHIP种类设置为EXCLUSIVE时，OWNERSHIP_STRENGTH策略与OWNERSHIP策略一起使用。以下是与所有权强度QoS策略相关的IDL：

```
struct OwnershipStrengthQosPolicy
  { long value;
};
```

value成员用于确定哪个Data Writer是数据对象实例的所有者。 默认值是零。

3.3 政策示例

以下示例代码说明了为发布者设置和应用的一些策略。

```
DDS::DataWriterQos dw_qos;
pub-> get_default_datawriter_qos (dw_qos) ;  dw_qos.history.kind

= DDS :: KEEP_ALL_HISTORY_QOS;

dw_qos.reliability.kind = DDS::RELIABLE_RELIABILITY_QOS;
dw_qos.reliability.max_blocking_time.sec = 10;
dw_qos.reliability.max_blocking_time.nanosec = 0;

dw_qos.resource_limits.max_samples_per_instance = 100;

DDS::DataWriter_var dw =
  pub-> create_datawriter (主题,
                           dw_qos,
                           0,   // 没有监听OpenDDS :: DCPS ::
                           DEFAULT_STATUS_MASK) ;
```

此代码创建具有以下特质的发布者：

- HISTORY设置为全部保留
- 可靠性设置为Reliable，最大阻断时间为10秒
- 每个实例资源限制的最大采样数设置为100

这意味着当100个样本正在等待传送时，作者可以在返回错误代码之前阻塞10秒钟。 数据读取器端的这些相同的QoS设置意味着最多有100个未读样本在被拒绝之前被框架排队。 被拒绝的样本被丢弃，SampleRejectedStatus被更新。

C章节4

条件和听众

4.1 介绍

DDS规范定义了两种单独的机制来通知DCPS通讯状态变化的应用。 大多数状态类型定义了一个结构，其中包含与状态变化有关的信息，并且可以由应用程序使用条件或侦听器来检测。 不同的状态类型在4.2.

每个实体类型（域参与者，主题，发布者，订阅者，数据读取器和数据写入器）定义其自己对应的监听器接口。 应用程序可以实现此接口，然后将其监听器实现附加到实体。 每个侦听器接口都包含可以为该实体报告的每个状态的操作。 只要发生限定状态更改，监听器就会被异步调用，并执行适当的操作。 有关不同监听器类型的详细信息，请参阅4.3.

条件与等待集一起使用，让应用程序同步等待事件。 条件的基本使用模式涉及创建条件对象，将它们附加到等待集，然后等待等待设置，直到其中一个条件被触发。 等待的结果告诉应用程序哪些条件被触发，允许应用程序采取适当的行动来获得相应的状态信息。条件在下面更详细地描述4.4.

4.2 通信状态类型

每个状态类型都与特定的实体类型相关联。 本节由实体类型组织，具有相关实体类型下小节中描述的相应状态。

下面的大部分状态都是简单的通信状态。 DATA_ON_READERS和DATA_AVAILABLE是异常状态。 简单通信状态定义了一个IDL数据结构。 他们下面的对应部分描述了这个结构及其领域。 读取状态是对应用程序的简单通知，然后根据需要读取或采取样本。

状态数据结构中的增量值报告自上次访问状态以来发生的变化。 当一个监听器被调用了这个状态或者从它的实体读取状态时，认为状态被访问。

状态数据结构中具有InstanceHandle_t类型的字段通过内置主题中用于该实体的实例句柄来标识实体（主题，数据读取器，数据写入器等）。

4.2.1 主题状态类型

4.2.1.1 不一致的主题状态

INCONSISTENT_TOPIC状态表示试图登记已经存在具有不同特征的话题。 通常，现有的主题可能有与之相关的不同类型。 与不一致主题状态关联的IDL如下所示：

```
struct InconsistentTopicStatus
  { long total_count;
  long_count_change;
};
```

total_count值是报告为不一致的主题的累计计数。 total_count_change值是自上次访问此状态以来不一致主题的增量计数。

4.2.2 订户状态类型

4.2.2.1 读者状态数据

DATA_ON_READERS状态表示在与用户相关的一些数据阅读器上有新的数据可用。 这种状态被认为是读状态，并没有定义IDL结构。 接收到此状态的应用程序可以调用订阅服务器上的get_datareaders ()来获取具有可用数据的数据读取器集合。

4.2.3 数据读取器状态类型

4.2.3.1 样本被拒绝状态

SAMPLE_REJECTED状态表示数据读取器收到的样本已被拒绝。 与样本被拒绝状态相关的IDL如下所示：

```
枚举SampleRejectedStatusKind {NOT_REJECTED,
  REJECTED_BY_INSTANCES_LIMIT,
  REJECTED_BY_SAMPLES_LIMIT,
  REJECTED_BY_SAMPLES_PER_INSTANCE_LIMIT
};

struct SampleRejectedStatus
  { long total_count;
  long_count_change;  SampleRejectedStatusKind
  last_reason;  InstanceHandle_t
  last_instance_handle;
};
```

total_count值是已被报告为被拒绝的样本的累计计数。 total_count_change值是自上次访问此状态以来拒收样本的增量计数。 last_reason值是最近被拒绝的样本被拒绝的原因。 last_instance_handle值指示上次被拒绝的示例的实例。

4.2.3.2 生动的变化状态

LIVELINESS_CHANGED状态表示对于正在发布该数据读取器的实例的一个或多个数据写入器已经存在活跃性改变。 下面列出了与生动变化状态相关的IDL：

```
struct LivelinessChangedStatus
  { long alive_count;
  long not_alive_count;
  long alive_count_change;
  long not_alive_count_change;
  InstanceHandle_t last_publication_handle;
};
```

alive_count值是当前数据读写器正在读取的主题上的数据写入器的总数。

not_alive_count值是写入数据读取器主题的数据写入者的总数，不再声称他们的活跃度。 alive_count_change值是自上次访问状态以来活动计数的更改。

not_alive_count_change值是自上次访问状态以来非活动计数的变化。

last_publication_handle是生命力发生变化的最后一位数据作者的句柄。

4.2.3.3 要求的截止时间错过状态

REQUESTED_DEADLINE_MISSED状态表示通过Deadline QoS策略请求的截止日期在特定情况下不被遵守。 与请求截止日期错过状态相关的IDL如下所示：

```
struct RequestedDeadlineMissedStatus
  { long total_count;
  long_count_change;  InstanceHandle_t
  last_instance_handle;
};
```

total_count值是已报告的错过的请求截止日期的累计计数。 total_count_change值是自上次访问此状态以来错过的请求截止日期的增量计数。 last_instance_handle值表示上次错过期限的实例。

4.2.3.4 请求的不兼容的QoS状态

REQUESTED_INCOMPATIBLE_QOS状态表示请求的一个或多个QoS策略值与所提供的不兼容。 下面列出了与请求不兼容的QoS状态相关的IDL：

```
struct QosPolicyCount
  {QosPolicyId_t policy_id; 多少
};
```

typedef序列<QosPolicyCount> QosPolicyCountSeq; struct

```
RequestedIncompatibleQosStatus {
  long total_count;
  long_count_change;  QosPolicyId_t
  last_policy_id;  QosPolicyCountSeq策略;
```

```
};
```

total_count值是报告具有不兼容QoS的数据写入者的累计次数。 total_count_change值是自上次访问此状态以来不兼容的数据写入器的增量计数。 last_policy_id值标识检测到的最后一个不兼容性中的一个不兼容的QoS策略。 策略值是一系列值，指示已为每个QoS策略检测到的不兼容性的总数。

4.2.3.5 数据可用状态

DATA_AVAILABLE状态表示样本在数据写入器上可用。 这种状态被认为是读状态，并没有定义IDL结构。 接收到此状态的应用程序可以使用数据读取器上的各种读取和读取操作来检索数据。

4.2.3.6 样本丢失状态

SAMPLE_LOST状态表示样本已经丢失，并且从未被数据读取器接收。 下面列出了与“样本丢失状态”关联的IDL：

```
struct SampleLostStatus
  { long total_count;
  long_count_change;
};
```

total_count值是报告丢失的样本的累计数量。 total_count_change值是自上次访问此状态以来丢失样本的增量计数。

4.2.3.7 订阅匹配状态

SUBSCRIPTION_MATCHED状态指示兼容的数据写入器已被匹配或先前匹配的数据写入器已不再被匹配。 与订阅匹配状态关联的IDL如下所示：

```
struct SubscriptionMatchedStatus
  { long total_count;
  long_count_change;  long
  current_count;
  long current_count_change;
  InstanceHandle_t last_publication_handle;
};
```

total_count值是与此数据读取器兼容的数据写入器的累计计数。 total_count_change值是自上次访问此状态以来总计数的增量更改。 current_count值是与此数据读取器匹配的当前数据写入器的数量。 current_count_change

值是自上次访问此状态以来当前计数的变化。 该last_publication_handle值是最后一个数据写入器匹配的句柄。

4.2.4 数据写入器状态类型

4.2.4.1 生动失落的状态

LIVELINESS_LOST状态表示数据编写者通过其Lively QoS所承诺的活跃程度尚未得到尊重。这意味着任何连接的数据读取器都将认为此数据写入器不再处于活动状态。与"活动状态丢失状态"关联的IDL如下所示：

```
struct LivelinessLostStatus
  { long total_count;
  long_count_change;
};
```

total_count值是活动数据写入器不活动的累计次数。 total_count_change值是自上次访问此状态以来总计数的增量更改。

4.2.4.2 提供的截止时间错过状态

OFFERED_DEADLINE_MISSED状态表示数据写入器提供的截止日期已经被错过了一个或多个实例。 下面列出与提供的截止时间错过状态相关的IDL：

```
struct OfferedDeadlineMissedStatus
  { long total_count;
  long_count_change;  InstanceHandle_t
  last_instance_handle;
};
```

total_count值是实例错过了最后期限的累计次数。 total_count_change值是自上次访问此状态以来总计数的增量更改。 last_instance_handle值指示错过了最后期限的最后一个实例。

4.2.4.3 提供不兼容的QoS状态

OFFERED_INCOMPATIBLE_QOS状态表示所提供的QoS与数据读取器所请求的QoS不兼容。 下面列出了与提供的不兼容QoS状态关联的IDL：

```
struct QosPolicyCount
  {QosPolicyId_t policy_id; 多少
};
```

typedef序列<QosPolicyCount> QosPolicyCountSeq;

```
struct OfferedIncompatibleQosStatus
  { long total_count;
  long_count_change;  QosPolicyId_t
  last_policy_id;  QosPolicyCountSeq
  策略;
};
```

total_count值是已经找到QoS不兼容的数据读取器的累计次数。 total_count_change值是自上次访问此状态以来总计数的增量更改。 last_policy_id值标识检测到的最后一个不兼容性中的一个不兼容的QoS策略。 策略值是一系列值，指示已为每个QoS策略检测到的不兼容性的总数。

4.2.4.4 发布匹配状态

PUBLICATION_MATCHED状态表示兼容的数据阅读器已被匹配或先前匹配的数据阅读器已不再被匹配。 与发布匹配状态关联的IDL如下所示：

```
struct PublicationMatchedStatus
  { long total_count;
  long_count_change;  long
  current_count;
  long current_count_change;
  InstanceHandle_t last_subscription_handle;
};
```

total_count值是与该数据写入器兼容的数据读取器的累计数。 total_count_change值是自上次访问此状态以来总计数的增量更改。 current_count值是与此数据写入器相匹配的当前数据读取器数量。 current_count_change值是自上次访问此状态以来当前计数的更改。 last_subscription_handle值是匹配的最后一个数据读取器的句柄。

4.3 听众

每个实体根据它可以报告的状态来定义自己的监听器接口。 任何实体的侦听器接口也都从其拥有的实体的侦听器继承，允许它也处理拥有实体的状态。 例如，订阅者监听器直接定义一个操作来处理“读取者数据”状态，并从数据读取器监听器继承。

每个状态操作都采用on_ <status_name> (<entity>, <status_struct>)，其中<status_name>是正在报告的状态的名称，<entity>是对状态被报告的实体的引用，<status_struct>是

结构与状态的细节。 读取状态省略第二个参数。 例如，以下是“采样丢失”状态的操作：

```
void on_sample_lost (在DataReader the_reader中，处于SampleLostStatus状态) ;
```

可以将监听器传递给用于创建实体的工厂函数，也可以通过在实体创建后调用set_listener () 来明确设置。 这两个函数都将状态掩码作为参数。 掩码指示在该侦听器中启用了哪些状态。 DdsDcpsInfrastructure.idl中定义了每个状态的掩码位值：

```
模块DDS {
  typedef unsigned long StatusKind;
  typedef unsigned long StatusMask;  //位掩码StatusKind

  const StatusKind INCONSISTENT_TOPIC_STATUS  = 0x0001 << 0; const
  StatusKind OFFERED_DEADLINE_MISSED_STATUS = 0x0001 << 1; const
  StatusKind REQUESTED_DEADLINE_MISSED_STATUS = 0x0001 << 2; const
  StatusKind OFFERED_INCOMPATIBLE_QOS_STATUS = 0x0001 << 5; const
  StatusKind REQUESTED_INCOMPATIBLE_QOS_STATUS= 0x0001 << 6;
  const StatusKind SAMPLE_LOST_STATUS                = 0x0001 << 7;
  const StatusKind SAMPLE_REJECTED_STATUS            = 0x0001 << 8;
  const StatusKind DATA_ON_READERS_STATUS            = 0x0001 << 9;
  const StatusKind DATA_AVAILABLE_STATUS             = 0x0001 << 10;
  const StatusKind LIVELINESS_LOST_STATUS            = 0x0001 << 11;
  const StatusKind LIVELINESS_CHANGED_STATUS         = 0x0001 << 12;
  const StatusKind PUBLICATION_MATCHED_STATUS        = 0x0001 << 13;
  const StatusKind SUBSCRIPTION_MATCHED_STATUS       = 0x0001 << 14;
};
```

只需简单地按位或“所需”状态位为听众构建一个掩码即可。 以下是将侦听器附加到数据读取器的示例（仅用于“数据可用”状态）：

```
DDS :: DataReaderListener_var侦听器 (新的DataReaderListenerImpl) ;
// 创建数据库
DDS :: DataReader_var dr = sub> create_datareader (topic,
  DATAREADER_QOS_DEFAULT,
  监听器, DDS :: DATA_AVAILABLE_STATUS) ;
```

这里是一个例子，展示了如何使用set_listener () 来更改监听器：

```
DR-> set_listener (监听器,
                   DDS :: DATA_AVAILABLE_STATUS | DDS ::
                   LIVELINESS_CHANGED_STATUS) ;
```

当明确的通信状态改变时，OpenDDS调用最具体的相关监听器操作。 这意味着，例如，数据读取器的监听器将优先于用户的监听器以查看与数据读取器相关的状态。

以下部分定义了不同的侦听器接口。 有关各个状态的更多详细信息，请参阅4.2.

4.3.1 主题监听器

```
接口TopicListener: 监听器{
  void on_inconsistent_topic (在Topic the_topic中,
                              InconsistentTopicStatus状态) ;
};
```

4.3.2 数据写入器监听器

```
接口DataWriterListener: 监听器{
  void on_offered_deadline_missed (在DataWriter writer中,
                                   在OfferedDeadlineMissedStatus状态) ;  void
  on_offered_incompatible_qos (在DataWriter writer中,
                               在OfferedIncompatibleQosStatus状态) ;  void
  on_liveliness_lost (在DataWriter作家,
                      在LivelinessLostStatus状态) ;  void
  on_publication_matched (在DataWriter writer中,
                          处于PublicationMatchedStatus状态) ;
};
```

4.3.3 发布者监听器

```
接口PublisherListener: DataWriterListener {
};
```

4.3.4 数据读取器监听器

```
接口DataReaderListener: 监听器{
  void on_requested_deadline_missed (在DataReader the_reader中,
                                     在RequestedDeadlineMissedStatus状态) ;  void
  on_requested_incompatible_qos (在DataReader的the_reader中,
                                 处于RequestedIncompatibleQosStatus状态) ;  void
  on_sample_rejected (在DataReader的the_reader中,
                      在SampleRejectedStatus状态) ;  void
  on_liveliness_changed (在DataReader的the_reader中,
                         在LivelinessChangedStatus状态) ;  void
  on_data_available (在DataReader the_reader中) ;
  void on_subscription_matched (在DataReader the_reader中,
                                在SubscriptionMatchedStatus状态) ;  void
  on_sample_lost (在DataReader的the_reader中,
                  在SampleLostStatus状态) ;
};
```

4.3.5 用户监听器

```
接口SubscriberListener: DataReaderListener {void on_data_on_readers
    (在订阅者the_subscriber中)；
};
```

4.3.6 域参与者监听器

```
接口DomainParticipantListener: TopicListener,
                                        PublisherListener,
                                        SubscriberListener {
};
```

4.4 条件

DDS规范定义了四种类型的条件：

- 状态条件
- 阅读条件
- 查询条件
- 警卫条件

4.4.1 状态条件

每个实体都有一个与之关联的状态条件对象，以及一个让应用程序访问状态条件的get_statuscondition () 操作。 每个条件都有一组启用的状态，可以触发该条件。 将一个或多个条件附加到等待集允许应用程序开发人员等待条件的状态设置。 一旦启用状态被触发，等待呼叫从等待集返回，并且开发者可以查询实体上的相关状态条件。 查询状态条件将重置状态。

4.4.1.1 状态条件示例

本示例在数据写入器上启用“提供的不兼容QoS”状态，等待它，然后在触发时进行查询。第一步是从数据写入器获取状态条件，启用所需的状态，并将其附加到等待设置：

```
DDS :: StatusCondition_var cond = data_writer-> get_statuscondition () ;  cond->
set_enabled_statuses (DDS :: OFFERED_INCOMPATIBLE_QOS_STATUS) ;

DDS :: WaitSet_var ws = new DDS :: WaitSet;
WS-> attach_condition (COND) ;
```

现在我们可以等待十秒钟的条件：

```
DDS :: ConditionSeq激活;  DDS ::持续时间
ten_seconds = {10, 0};
int result = ws-> wait (active, ten_seconds) ;
```

此操作的结果是活动序列中的超时或一组触发条件：

```
if (result == DDS :: RETCODE_TIMEOUT) {
```

```
cout <<“等待超时”<< std :: endl;
```


```
} else if (result == DDS :: RETCODE_OK) {DDS :: OfferedIncompatibleQosStatus
  incompatibleStatus;  data_writer-> get_offered_incompatible_qos
   (incompatibleStatus) ;
  // 根据需要访问状态字段...
}
```

开发人员可以选择将多个条件附加到一个等待集，并在每个条件下启用多个状态。

4.4.2 附加条件类型

DDS规范还定义了三种其他类型的条件：读取条件，查询条件和保护条件。 这些条件并不直接涉及状态的处理，而是允许将其他活动整合到状态和等待机制中。 这些是其他条件在这里简要介绍。 有关更多信息，请参阅DDS规范或$ DDS_ROOT / tests /中的OpenDDS测试。

4.4.2.1 阅读条件

读取条件是使用数据读取器和传递给读取和执行操作的相同掩码创建的。 当等待这个条件时，只要样本匹配指定的掩码，就会触发它。 然后可以使用读取条件作为参数的read_w_condition () 和take_w_condition () 操作来检索这些样本。

4.4.2.2 查询条件

查询条件是一种特殊形式的读取条件，使用有限形式的SQL查询来创建。 这允许应用程序过滤触发条件的数据样本，然后使用正常的读取条件机制进行读取。 参见章节5.3 有关查询条件的更多信息。

4.4.2.3 守卫条件

守护条件是一个简单的接口，允许应用程序创建自己的条件对象，并在发生应用程序事件（OpenDDS外部）时触发它。

C章节5

内容订阅配置文件

5.1 介绍

DDS的内容订阅配置文件由三个功能组成，这三个功能可以使数据读取器的行为受到所收到数据样本内容的影响。 这三个功能是：

- 内容过滤的主题
- 查询条件
- 多主题

内容过滤的主题和多主题界面从TopicDescription继承界面（而不是从主题界面，因为名称可能会提示）。

内容过滤的主题和查询条件允许使用类似于SQL的参数化查询字符串过滤（选择）数据样本。 此外，查询条件允许对从数据读取器的read（）或take（）操作返回的结果集进行排序。 多主题也具有这种选择能力以及将来自不同数据写入器的数据聚合成单个数据类型和数据读取器的能力。

如果您不打算在应用程序中使用内容订阅配置文件功能，则可以配置OpenDDS以在构建时删除对它的支持。 见页面13 有关禁用此支持的信息。

5.2 内容过滤的主题

域参与者界面包含用于创建和删除内容过滤主题的操作。 创建内容过滤的主题需要以下参数：

- 名称
  为此内容过滤的主题分配一个名称，稍后可以与该名称一起使用lookup_topicdescription（）操作。
- 相关主题
  指定此内容过滤的主题所基于的主题。 匹配数据编写者将用于发布数据样本的是相同的主题。
- 筛选表达式
  类似于SQL的表达式（请参见部分5.2.1），其定义了在内容过滤主题的数据读取器应该接收的相关主题上发布的样本的子集。
- 表达参数
  过滤器表达式可以包含参数占位符。 这个参数为这些参数提供了初始值。 表达式参数可以在创建内容过滤的主题后更改（不能更改过滤器表达式）。

一旦创建了内容过滤的主题，用户的create_datareader（）操作就会使用它来获取内容过滤数据读取器。 该数据阅读器在功能上等同于正常的数据阅读器，除了不符合过滤器表达准则的输入数据样本被丢弃。

过滤器表达式首先在出版商处进行评估，以便用户可以忽略的数据样本甚至在到达传输器之前被丢弃。 可以使用-DCPSPublisherContentFilter 0或配置文件的[common]部分中的等效设置关闭此功能。 这个策略可能会影响非默认的DEADLINE或LIVELINESS QoS策略的行为。 必须特别考虑“缺失”样本如何影响QoS行为，请参阅文档

文档/设计/ CONTENT_SUBSCRIPTION。

注意 *RTPS传输并不总是做Writer方面的过滤。 它目前不实现传输级别过滤，但可能能够在传输层上进行过滤。*

5.2.1 筛选表达式

过滤器表达式的形式语法在DDS规范的附录A中定义。 本节提供该语法的非正式总结。 查询表达式（5.3.1）和主题表达（5.4.1）也在附件A中定义。

过滤器表达式是一个或多个谓词的组合。 每个谓词是一个逻辑表达式，采取两种形式之一：

- <arg1> <RelOp> <arg2>
  - arg1和arg2是可以是字面值（整数，字符，浮点，字符串或枚举）的参数，形式为%n（其中n是参数序列中从零开始的索引）的参数占位符，或字段引用。
  - 至少有一个参数必须是一个字段引用，这是一个IDL结构体字段的名称，可以选择后跟任意数量的“。”。 和另一个字段名称来表示嵌套结构。
  - RelOp是一个来自列表的关系运算符：=，>，> =，<，<=，<>和’like’。 “like”是通配符匹配，使用%匹配任意数量的字符，_匹配单个字符。
  - 这种谓词形式的例子包括：a =’z’，b <>’str’，c <d，e =’enumerator’，f> = 3.14e3,27> g，h> i，jkl like%0
- <arg1> [NOT] BETWEEN <arg2> AND <arg3>

– 在这种形式下，参数1必须是字段引用，参数2和3必须都是文字值或参数占位符。

任何数量的谓词都可以通过使用括号和布尔运算符AND，OR和NOT来组合，以形成过滤器表达式。

5.2.2 内容过滤主题示例

下面的代码片段为消息类型创建一个内容过滤的主题。 首先，这里是消息的IDL：

```
模块Messenger {
#pragma DCPS_DATA_TYPE“Messenger :: Message”struct
  Message {
       长ID
  };
};
```

接下来，我们有创建数据读取器的代码：

```
CORBA :: String_var type_name = message_type_support-> get_type_name () ;  DDS ::
Topic_var topic = dp-> create_topic (“MyTopic”,
                                          type_name,
                                          TOPIC_QOS_DEFAULT,
                                          NULL,
                                          OpenDDS :: DCPS :: DEFAULT_STATUS_MASK) ;
DDS::ContentFilteredTopic_var cft =
  participant-> create_contentfilteredtopic ( “MyTopic过滤”,
                                                话题，“id>
                                                1”,
                                                StringSeq () ) ;
DDS::DataReader_var dr =
  订户> create_datareader (CFT,
                                  dr_qos,
                                  NULL,
                                  OpenDDS :: DCPS :: DEFAULT_STATUS_MASK) ;
```

数据读取器“dr”将仅接收具有大于1的“id”值的样本。

5.3 查询条件

查询条件接口继承自读取条件接口，因此查询条件具有读条件的所有功能以及本节中所述的附加功能。 其中一种继承的功能是查询条件可以像其他任何条件一样使用（参见章节4.4).

DataReader接口包含用于创建（create_querycondition）和删除（delete_readcondition）查询条件的操作。 创建查询条件需要以下参数：

- 示例，视图和实例状态掩码
  这些是传递给create_readcondition（），read（）或take（）的相同的状态掩码。

- 查询表达式
  类似于SQL的表达式（请参阅5.3.1）描述导致条件被触发的样本子集。 该表达式用于过滤从read_w_condition（）或take_w_condition（）操作返回的数据集。 它也可能对该数据集施加排序（ORDER BY）。
- 查询参数
  查询表达式可以包含参数占位符。 这个参数为这些参数提供了初始值。 查询条件创建后查询参数可以更改（查询表达式不能更改）。

一个特定的查询条件可以与一个等待设置（attach_condition）和一个数据读取器（read_w_condition, take_w_condition, read_next_instance_w_condition, take_next_instance_w_condition）或两者一起使用。 与等待集一起使用时，ORDER BY子句对触发等待集没有影响。 与数据读取器读取*（）或执行*（）操作一起使用时，结果数据集将仅包含与查询表达式相匹配的样本，如果存在ORDER BY子句，则它们将按ORDER BY字段排序。

5.3.1 查询表达式

查询表达式是过滤器表达式的超集（参见章节5.2.1）。 在过滤器表达式之后，查询表达式可以有选择地使用ORDER BY关键字，后跟逗号分隔的字段引用列表。 如果ORDER BY子句存在，则过滤器表达式可能为空。 以下字符串是查询表达式的示例：

- m > 100 ORDER BYn
- ORDER BY pq, r, stu
- 不是v LIKE'z%'

5.3.2 查询条件示例

下面的代码片断创建并使用一个查询条件，该类型使用struct'Message'和字段'key'（一个整型）。

```
DDS::QueryCondition_var dr_qc =
  DR-> create_querycondition (DDS :: ANY_SAMPLE_STATE,
                              DDS :: ANY_VIEW_STATE, DDS ::
                              ALIVE_INSTANCE_STATE,
                              "key> 1", DDS :: StringSeq
                               () ) ;
DDS :: WaitSet_var ws = new DDS :: WaitSet;
WS-> attach_condition (dr_qc) ;  DDS ::
ConditionSeq激活;  DDS :: Duration_t three_sec
= {3, 0};
DDS :: ReturnCode_t ret = ws-> wait (active, three_sec) ;
  // 错误处理未显示ws->
detach_condition (dr_qc) ;
```

```
MessageDataReader_var mdr = MessageDataReader :: _ narrow (dr) ;  MessageSeq数据;
DDS::SampleInfoSeq infoseq;
```

```
ret = mdr-> take_w_condition (data, infoseq, DDS :: LENGTH_UNLIMITED, dr_qc) ;
  // 错误处理未显示dr->
delete_readcondition (dr_qc) ;
```

任何以<= 1为关键字接收的样本既不会触发条件（以满足等待），也不会从take_w_condition（）中的“数据”序列中返回。

5.4 多主题

多个主题是比其他两个内容订阅功能更复杂的功能，因此描述它需要一些新的术语。

MultiTopic接口继承自TopicDescription接口，就像ContentFilteredTopic一样。 为多个主题创建的数据读取器被称为“多主题数据读取器”。多主题数据读取器接收属于任何数量的常规主题的样本。 这些主题被称为“组成主题”。多主题有一个称为“结果类型”的DCPS数据类型。多主题数据读取器实现类型特定的数据读取器接口的最终类型。 例如，如果生成的类型是消息，那么多主题数据读取器可以缩小到MessageDataReader接口。

多主题的主题表达（请参见部分5.4.1）描述了输入数据的不同字段（在组成主题上）如何映射到结果类型的字段。

域参与者界面包含用于创建和删除多个主题的操作。 创建多个主题需要以下参数：

- 名称
  为这个多主题分配一个名字，稍后可以使用这个名字
  lookup_topicdescription（）操作。
- 类型名称
  指定多主题的结果类型。 此类型必须在创建多个主题之前注册其类型支持。
- 主题表达式（也称为订阅表达式）
  类似于SQL的表达式（请参见部分5.4.1），它定义组成主题字段到结果类型字段的映射。 它也可以指定一个过滤器（WHERE子句）。
- 表达参数
  主题表达式可以包含参数占位符。 这个参数为这些参数提供了初始值。 表达式参数可以在创建多个主题后更改（主题表达式不能更改）。

一旦创建了多个主题，它就被用户的create_datareader（）
操作来获得多主题数据读取器。 这个数据读取器被应用程序使用

接收结果类型的构造样本。 这些样本的构建方式将在下面进行介绍5.4.2.2.

5.4.1 主题表达式

主题表达式使用与完整的SQL查询非常相似的语法：

```
SELECT <aggregation> FROM <selection> [WHERE <condition>]
```

- 聚合可以是“*”或逗号分隔的字段说明符列表。 每个字段说明符都有以下语法：
  - <constituent_field> [[AS] <results_field>]]
  - constituent_field是一个字段参考（见章节5.2.1）到一个组成主题的领域（未指定哪个主题）。
  - 可选的results_field是对结果类型中字段的字段引用。 如果存在，则结果字段是构造样本中的分量字段的目的地。 如果不存在，则将构成字段的数据分配给结果类型中具有相同名称的字段。 可选的“AS”不起作用。
  - 如果使用“*”作为聚合，则在结果类型中的每个字段将被分配来自其中一个组成主题类型中的同名字段的值。
- 选择列出一个或多个组成主题名称。 主题名称由“连接”关键字分隔（所有3个连接关键字相同）：
  - <topic> [{NATURAL INNER | NATURAL | INNER NATURAL} JOIN <topic>]...
  - 主题名称只能包含字母，数字和破折号（但不能以数字开头）。
  - 自然连接操作是交换和关联的，因此主题的顺序没有影响。
  - 自然连接的语义是任何具有相同名称的字段都被视为“连接键”，用于组合来自这些键出现的主题的数据。 连接操作在本章的后续章节中有更详细的描述。
- 该条件与过滤器表达式具有完全相同的语法和语义（请参见部分5.2.1）。 条件中的字段引用必须与结果类型中的字段名称匹配，而不是组成主题类型中的字段名称。

5.4.2 使用说明

5.4.2.1 加入密钥和DCPS数据密钥

DCPS数据密钥（#pragma DCPS_DATA_KEY）的概念已经在Section中讨论过了2.1.1。 为多个主题连接键是一个独特但相关的概念。

连接键是结构中出现多个组成主题的任何字段名称。 连接密钥的存在对这些主题的数据样本如何组合成一个构造的样本实施约束（请参见部分5.4.2.2）。 具体而言，该键的值必须相等，以便将来自组成主题的数据样本组合成最终类型的样本。 如果多个连接键对于相同的两个或更多个主题是共同的，则所有键的值必须相等才能组合数据。

DDS规范要求连接密钥字段具有相同的类型。 此外，OpenDDS对IDL如何定义DCPS数据密钥以处理多个主题强加两个要求：

1）每个连接密钥字段还必须是其组成主题类型的DCPS数据密钥。

2）生成的类型必须包含每个连接键，并且这些字段必须是生成类型的DCPS数据键。

部分中的示例5.4.3.1 满足这两个要求。 请注意，没有必要列出聚合中的连接键（SELECT子句）。

5.4.2.2 如何构造结果样本

尽管多主题中的许多概念都是从关系数据库领域借用的，但实时中间件如DDS不是数据库。 不是一次处理一批数据，而是从一个组成主题到达数据读取器的每个样本触发多主题特定的处理，这导致构建了结果类型和插入的零个，一个或多个样本将这些构建的样本集合到多主题数据读取器中。

具体来说，在类型为“TA”的组成主题“A”上的样本的到达导致多主题数据读取器中的以下步骤（这是对实际算法的简化）：

1）结果类型的样本被构造，并且来自存在于所得到的类型中并且在聚合中（或者是连接键）的TA的字段被从传入样本复制到构建的样本。

2) 至少有一个与A相同的连接键的主题“B”被认为是连接操作。 连接读取主题B上的READ_SAMPLE_STATE样本，其中的键值与构造样本中的键值相匹配。 连接的结果可能是零个，一个或多个样本。 如步骤1中所述，将TB中的字段复制到生成的样本中。

3) 连接主题“B”（连接到其他主题）的密钥，然后按照步骤2中的描述进行处理，并继续处理由连接密钥连接的所有其他主题。

4) 步骤2或步骤3中未访问的任何组成主题均被视为“交叉连接”（也称为跨产品连接）。 这些连接没有关键的限制。

5) 如果产生了任何构造的样本，则它们被插入到多主题数据读取器的内部数据结构中，就好像它们已经通过正常机制到达一样。 应用程序侦听器和条件被通知。

5.4.2.3 与订阅者监听器一起使用

如果应用程序注册了一个订阅者侦听器，用于读取包含多主题的同一订阅者的条件状态更改（DATA_ON_READERS_STATUS），则应用程序必须在其订阅者侦听器的on_data_on_readers（）回调方法的实现中调用notify_datareaders（）。 这个要求是必要的，因为多主题在内部使用数据阅读器监听器，当用户监听器被注册时被抢占。

5.4.3 多主题示例

本示例基于DDS规范附录A第A.3节中使用的示例主题表达式。 它演示了如何使用多主题连接操作的属性来关联来自不同主题（可能不同的发布者）的数据。

5.4.3.1 IDL和主题表达

通常我们将使用相同的字符串作为主题名称和主题类型。 在这个例子中，我们将使用不同的字符串作为类型名称和主题名称，以便说明何时使用每个字符串。

以下是组成主题数据类型的IDL：

```
#pragma DCPS_DATA_TYPE“LocationInfo”
#pragma DCPS_DATA_KEY“LocationInfo flight_id”struct
LocationInfo {
  unsigned long flight_id; 长x
  long y;
  长z
};

#pragma DCPS_DATA_TYPE“PlanInfo”
#pragma DCPS_DATA_KEY“PlanInfo flight_id”
```

```
struct PlanInfo {unsigned long
  flight_id; 字符串
  flight_name; 字符串tailno;
};
```

请注意，密钥字段的名称和类型相匹配，所以它们被设计为用作连接密钥。 结果类型（下面）也有该关键字段。

接下来，我们有用于结果数据类型的IDL：

```
#pragma DCPS_DATA_TYPE"结果"
#pragma DCPS_DATA_KEY"生成的flight_id"struct结果{
  unsigned long flight_id; 字
  符串flight_name; 长x
  long y; 身高
};
```

基于此IDL，可以使用以下主题表达式来组合来自主题的数据
使用类型LocationInfo的位置和使用类型PlanInfo的主题FlightPlan：

```
SELECT flight_name, x, y, z AS高度FROM位置NATURAL JOIN FlightPlan WHERE高度
<1000和x <23
```

综合来看，IDL和主题表达式描述了这个多重主题将如何工作。 多主题数据读取器将构建属于由flight_id键入的实例的样本。 一旦从Location和FlightPlan主题中获得相应的实例，结果类型的实例就会出现。 域中的一些其他域参与者或参与者将发布关于这些主题的数据，而且他们甚至不需要知道彼此。 由于它们每个都使用相同的flight_id来引用航班，因此多个主题可以关联来自不同来源的传入数据。

5.4.3.2 创建多主题数据读取器

为多个主题创建一个数据读取器包含几个步骤。 首先注册结果类型的类型支持，然后创建多主题本身，然后是数据读取器：

```
  ResultingTypeSupport_var ts_res = new ResultingTypeSupportImpl;  ts_res->
  register_type (dp, "") ;
  CORBA :: String_var type_name = ts_res-> get_type_name () ;  DDS ::
  MultiTopic_var mt =
    DP-> create_multitopic ( "MyMultiTopic",
                             TYPE_NAME,
                             "SELECT flight_name, x, y, z AS height""FROM Location
                               JOIN JOIN FlightPlan""WHERE height <1000 AND x
                               <23",
                             DDS :: StringSeq () ) ;  DDS ::
  DataReader_var dr =
    子> create_datareader (MT,
                            DATAREADER_QOS_DEFAULT, NULL,
```

```
                                    OpenDDS :: DCPS :: DEFAULT_STATUS_MASK) ;
```

5.4.3.3 使用多主题数据读取器读取数据

从API的角度来看，多主题数据读取器与任何其他类型的数据读取器是相同的。 这个例子使用一个等待设置和一个读取条件来阻塞，直到数据可用。

```
DDS::WaitSet_var ws = new DDS::WaitSet;
DDS::ReadCondition_var rc =
  DR-> create_readcondition (DDS :: ANY_SAMPLE_STATE,
                             DDS :: ANY_VIEW_STATE, DDS :: ANY_INSTANCE_STATE) ;
WS-> attach_condition (RC) ;
DDS :: Duration_t infinite = {DDS :: DURATION_INFINITE_SEC,
                              DDS::DURATION_INFINITE_NSEC};
DDS :: ConditionSeq激活;
ws-> wait (active, infinite) ;  //错误处理未显示ws->
detach_condition (rc) ;
ResultingDataReader_var res_dr = ResultingDataReader :: _ narrow (dr) ; 结果序列数据;
DDS :: SampleInfoSeq信息;
res_dr-> take_w_condition (data, info, DDS :: LENGTH_UNLIMITED, rc) ;
```

C章节6

内置主题

6.1 介绍

在OpenDDS中，默认情况下会创建并发布内置主题，以交换有关在部署中运行的DDS参与者的信息。 当使用DCPSInfoRepo服务的集中式发现方法中使用OpenDDS时，此服务将发布内置主题。 对于DDSI-RTPS部署，在进程中实例化的内部OpenDDS实现将创建并使用Built-In-Topics，以实现与其他DDS参与者类似的发现信息交换。 参见章节7.3.3 了解RTPS发现配置的描述。

6.2 内置的DCPSInfoRepo主题组态

使用DCPSInfoRepo时，可以使用-NOBITS的命令行选项来禁止发布内置主题。

为每个域定义了四个单独的主题。 每个专用于特定的实体（域参与者，主题，数据写入器，数据读取器），并发布描述域中每个实体的状态的实例。

内置主题的订阅会自动为每个域参与者创建。 参与者对内置主题的支持可以通过DCPSBit配置选项进行切换（请参见章节7.2）（注意：这个选项不能用于RTPS发现）。 要查看内置的主题数据，只需获取内置的订阅者，然后使用它访问感兴趣的内置主题的数据读取器。 数据读取器可以像任何其他数据读取器一样使用。

第6.3 通过6.6 提供关于为每个内置主题发布的数据的详细信息。 显示如何从内置主题读取的示例遵循这些部分。

如果您不打算在应用程序中使用内置主题，则可以在构建时配置OpenDDS以删除内置主题支持。 这样做可以将核心DDS库的占用空间减少多达30%。 参见章节1.3.2 有关禁用内置主题支持的信息。

6.3 DCPSParticipant主题

DCPSParticipant主题发布有关域的域参与者的信息。 以下是定义为此主题发布的结构的IDL：

```
struct ParticipantBuiltinTopicData {
  BuiltinTopicKey_t key;  //包含3个long数组的结构UserDataQosPolicy user_data;
};
```

每个域参与者都由一个唯一的键定义，并且是本主题中的自己的实例。

6.4 DCPSTopic主题

注意 *配置为进行RTPS发现时，OpenDDS不支持此内置主题。*

DCPSTopic主题发布有关该主题的信息。 以下是定义为此主题发布的结构的IDL：

```
struct TopicBuiltinTopicData {
```

```
  BuiltinTopicKey_t key; 字
  符串名称;
  字符串type_name;  DurabilityQosPolicy耐
  久性;  QosPolicy截止日期;
  LatencyBudgetQosPolicy latency_budget;
  LivelinessQosPolicy活泼;  ReliabilityQosPolicy可靠性;
  TransportPriorityQosPolicy transport_priority; 寿命QOS政
  策寿命;  DestinationOrderQosPolicy destination_order;
  HistoryQosPolicy历史;  ResourceLimitsQosPolicy
  resource_limits;  OwnershipQosPolicy所有权;
  TopicDataQosPolicy topic_data;
};
```

每个主题都由一个唯一的键标识，并且是该内置主题中的自己的实例。 上面的成员标识该主题的名称，主题类型的名称以及该主题的一组QoS策略。

6.5 DCPSPublication主题

DCPSPublication主题发布有关数据写入器的信息。 以下是定义为此主题发布的结构的IDL：

```
struct PublicationBuiltinTopicData
  {BuiltinTopicKey_t key;  BuiltinTopicKey_t
  participant_key; 字符串topic_name;
  字符串type_name;  DurabilityQosPolicy耐
  久性;  DeadlineQosPolicy截止日期;
  LatencyBudgetQosPolicy latency_budget;
  LivelinessQosPolicy活泼;  ReliabilityQosPolicy
  可靠性; 寿命QOS政策寿命;  UserDataQosPolicy
  user_data;
  所有权强度QosPolicy ownership_strength;
  PresentationQosPolicy演示文稿;  PartitionQosPolicy分区;
  TopicDataQosPolicy topic_data;  GroupDataQosPolicy
  group_data;
};
```

每个数据写入器在创建时都被分配一个唯一的密钥，并在该主题中定义自己的实例。 上面的字段标识Data Writer所属的域参与者（通过其密钥），主题名称和类型以及应用于Data Writer的各种QoS策略。

6.6 DCPSSubscription主题

DCPSSubscription主题在域中发布有关数据读取器的信息。 以下是定义为此主题发布的结构的IDL：

```
struct SubscriptionBuiltinTopicData
  {BuiltinTopicKey_t key;  BuiltinTopicKey_t
  participant_key; 字符串topic_name;
  字符串type_name;  DurabilityQosPolicy耐
  久性;  DeadlineQosPolicy截止日期;
  LatencyBudgetQosPolicy latency_budget;
  LivelinessQosPolicy活泼;  ReliabilityQosPolicy可靠性;
  DestinationOrderQosPolicy destination_order;
  UserDataQosPolicy user_data;  TimeBasedFilterQosPolicy
  time_based_filter;  PresentationQosPolicy演示文稿;
  PartitionQosPolicy分区;  TopicDataQosPolicy topic_data;
  GroupDataQosPolicy group_data;
};
```

每个数据读取器在创建时都被分配一个唯一的键，并在该主题中定义自己的实例。 以上字段标识数据读取器所属的域参与者（通过其密钥），主题名称和类型以及应用于数据读取器的各种QoS策略。

6.7 内置的主题订阅示例

以下代码使用域参与者来获取内置订户。 然后使用订阅者获取DCPSParticipant主题的数据读取器，然后读取该读者的样本。

```
Subscriber_var bit_subscriber = participant-> get_builtin_subscriber () ;  DDS :: DataReader_var dr =
  bit_subscriber-> lookup_datareader (BUILT_IN_PARTICIPANT_TOPIC) ;  DDS :: ParticipantBuiltinTopicDataDataReader_var
part_dr =
  DDS :: ParticipantBuiltinTopicDataDataReader :: _窄 (DR) ;

DDS :: ParticipantBuiltinTopicDataSeq part_data;  DDS ::
SampleInfoSeq信息;
DDS :: ReturnCode_t ret = part_dr-> read (part_data, infos, 20,
                                          DDS :: ANY_SAMPLE_STATE, DDS ::
                                          ANY_VIEW_STATE, DDS ::
                                          ANY_INSTANCE_STATE) ;

// 检查退货状态并阅读参与者数据
```

其他内置主题的代码是相似的。

C章节7

运行时配置

7.1 配置方法

OpenDDS包含一个基于文件的配置框架，用于配置与特定发布者和订阅者相关的全局选项和选项，例如发现和传输配置。 OpenDDS还允许通过命令行配置有限数量的选项并通过配置API进行配置。 本章总结了OpenDDS支持的配置选项。

OpenDDS配置涉及三个主要方面：

1）通用配置选项 – 在全局级别配置DCPS实体的行为。 这允许在计算环境中单独部署的进程共享指定行为的通用设置（例如，所有读者和作者应该使用RTPS发现）。

2）发现配置选项 – 配置发现机制的行为。 OpenDDS支持多种方法来发现和关联作家和读者，详见本节7.3.

3）传输配置选项 – 配置从OpenDDS的DCPS层中抽象传输层的可扩展传输框架（ETF）。每个可插拔传输器都可以单独配置。

OpenDDS的配置文件是一个可读的ini样式的文本文件。表7-1显示可用配置节类型的列表，因为它们与它们配置的OpenDDS区域相关。

表7-1配置文件部分

重点地区	文件部分标题
全局设置	[共同]
发现	[域名] [资料库] [rtps_discovery]
静态发现	[端点] [主题] [datawriterqos] [datareaderqos] [publisherqos] [subscriberqos]
运输	[配置] [运输]

对于除[common]之外的每个节类型，节标题的语法都采用[section type / instance]的形式。 例如，一个[repository]节类型将总是在配置文件中使用，如下所示：

[repository / repo_1]其中repository是部分类型，repo_1是存储库配置的实例名称。 部分将进一步解释如何使用实例来配置发现和传输7.3 和7.4.

-DCPSConfigFile命令行参数可用于将配置文件的位置传递给OpenDDS。 例如：

视窗：

```
发布者-DCPSConfigFile pub.ini
```

Unix的：

```
./publisher -DCPSConfigFile pub.ini
```

当初始化域参与者工厂时，命令行参数被传递给服务参与者单例。 这是通过使用TheParticipantFactoryWithArgs宏来完成的：

```
#include <dds / DCPS / Service_Participant.h> int

main (int argc, char * argv [])
{

  DDS :: DomainParticipantFactory_var dpf =
    TheParticipantFactoryWithArgs (argc, argv) ;
```

Service_Participant类还提供允许应用程序配置DDS服务的方法。 看到头文件*$ DDS_ROOT / dds / DCPS / Service_Participant.h获取详细信息。*

以下小节详细介绍了每个配置文件部分以及与这些部分相关的可用选项。

7.2 常用配置选项

OpenDDS配置文件的[common]部分包含诸如调试输出级别，DCPSInfoRepo进程的位置以及内存预分配设置等选项。 下面是一个常见的例子：

```
[common]
DCPSDebugLevel=0
DCPSInfoRepo = localhost: 12345
DCPSLivelinessFactor = 80 DCPSChunks = 20
DCPSChunksAssociationMultiplier = 10
DCPSBitLookupDurationMsec = 2000
DCPSPendingTimeout = 30
```

没有必要指定每个选项。

名称以“DCPS”开头的[common]部分中的选项值可以被命令行参数覆盖。 命令行参数与配置选项名称相同，前缀为“ - ”。 例如：

订户-DCPSInfoRepo localhost：12345

下表总结了[通用]配置选项：

表7-2常用配置选项

选项	描述	默认
DCPSBit=[1|0]	切换内置的主题支持。	1
DCPSBitLookupDurationMsec =毫秒	给定实例句柄时，框架在检索BIT数据时将等待潜在的内置主题信息的最大持续时间（以毫秒为单位）。 参与者代码可以在框架接收并处理相关的BIT信息之前获得远程实体的实例句柄。 框架等待着 给定的时间量在操作失败之前。	2000
DCPSBitTransportIPAddress =地址	IP地址，用于标识内置主题的tcp传输使用的本地接口。 注意：此属性仅适用于DCPSInfoRepo 组态。	INADDR_ANY
DCPSBitTransportPort =端口	tcp传输用于内置主题的端口。如果使用默认值“0”，操作系统将选择一个端口使用。 注意：此属性仅适用于DCPSInfoRepo 组态。	0
DCPSChunks =正	当RESOURCE_LIMITS QoS值无限时，数据写入者和读者的缓存分配器将预先分配的可配置数量的块。 当所有的预分配块正在使用，OpenDDS分配从 堆。	20
DCPSChunkAssociationMultiplier =正	用于DCPSChunks的乘数或resource_limits.max_samples值，以确定预分配的浅拷贝块的总数。 将其设置为大于连接数的值，以便预分配的块句柄不会耗尽。 写入多个数据读取器的样本将不会被多次复制，但是该样本的浅拷贝句柄用于管理向每个数据读取器的递送。 手柄的大小很小，所以不需要设置这个值 接近连接的数量。	10
DCPSDebugLevel =正	整数值，用于控制调试信息的数量 DCPS图层打印。 有效值是0到10。	0

选项	描述	默认
DCPSDefaultAddress	包含local_address的传输实例的local_address的主机部分的缺省值。 仅在DCPSDefaultAddress设置为非空时应用值并且在传输中没有指定local_address。	
DCPSDefaultDiscovery=[DEFAULT_REPO| DEFAULT_RTPS| DEFAULT_STATIC| 用户定义的配置实例名称]	指定用于未明确配置的任何域的发现配置。DEFAULT_REPO转换为使用DCPSInfoRepo。 DEFAULT_RTPS指定使用RTPS进行发现。 DEFAULT_STATIC指定使用静态发现。 参见章节7.3 有关配置的详细信息发现。	DEFAULT_REPO
DCPSGlobalTransportConfig =名称	指定应该用作全局配置的传输配置的名称。 所有不指定传输配置的实体都使用此配置。 $ file的特殊值使用包含所有传输的传输配置在配置文件中定义的实例。	默认配置按照中所述使用7.4.1
DCPSInfoRepo =对象引用	查找DCPS信息库的对象参考。 这可以是完整的CORBA IOR或简单的主机：端口字符串。	文件：//repo.ior
DCPSLivelinessFactor =正	活跃性租约期限之后的活跃性消息被发送的百分比。值为80意味着从上次检测到的心跳的20％的延迟缓冲信息。	80
DCPSPendingTimeout =秒	数据写入器将阻塞的最大持续时间（以秒为单位）允许未发送的样本耗尽删除。 默认情况下，此选项将无限期地阻止。	0
DCPSPersistentDataDir =路径	文件系统上持久数据的路径存储。 如果目录不存在，它将自动创建。	OpenDDS耐用-data-dir来
DCPSPublisherContentFilter=[1|0]	控制内容过滤主题的过滤器表达式评估策略。 启用后（1），出版商可以放下任何样品，然后将其交给运输工具这些样本将被用户忽略。	1
DCPSRTISerialization=[0|1]	控制是否使用非标准序列化填充当与使用RTPS传输的RTI实体集成时。	0
DCPSTransportDebugLevel =正	用于控制传输日志记录的整数值粒度。 法定值从0到5。	0
pool_size=n_bytes	安全配置文件内存池的大小，以字节为单位。	41943040 (40 MB)
pool_granularity=n_bytes	安全配置文件内存池的粒度（以字节为单位）。 一定是8的倍数。	8

选项	描述	默认
调度程序= [SCHED_RR | SCHED_FIFO | SCHED_OTHER]	选择要使用的线程调度程序。 将调度程序设置为默认值以外的值需要大多数系统上的权限。 可以设置SCHED_RR，SCHED_FIFO或SCHED_OTHER的值。 SCHED_OTHER是大多数系统上的默认调度程序； SCHED_RR是一种循环调度算法；和SCHED_FIFO允许每个线程运行，直到它在切换到另一个线程之前阻塞或完成。	SCHED_OTHER
scheduler_slice =微秒	某些操作系统（如SunOS）在选择除默认值之外的其他调度程序时，需要设置时间片值。 对于那些系统，这个选项可以用来设置以微秒为单位的值。	没有
DCPSBidirGIOP=[0|1]	使用TAO的BiDirectional GIOP功能与DCPSInfoRepo进行交互。 启用BiDir后，需要更少的套接字，因为两个客户端可以使用相同的套接字和服务器角色。	1

DCPSInfoRepo选项的值传递给CORBA :: ORB :: string_to_object（），可以是TAO（file，IOR，corbaloc，corbaname）可以理解的任何对象URL类型。 一个简单的<host>：<port>形式的端点描述也被接受。 它相当于corbaloc :: <host>：<port> / DCPSInfoRepo。

DCPSChunks选项允许应用程序开发人员在RESOURCE_LIMITS设置为无限时调整预先分配的内存量。 一旦分配的内存耗尽，额外的块将从堆中分配/释放。 当预分配的内存耗尽时，从堆中分配的这种特性提供了灵活性，但是当预分配的内存耗尽时，性能会降低。

7.3 发现配置

在DDS实现中，参与者在应用程序进程中被实例化，并且必须发现彼此以进行通信。 DDS实现使用域的功能来为同一域中的DDS参与者之间交换的数据提供上下文。 当编写DDS应用程序时，参与者被分配到一个域，并且需要确保他们的配置允许每个参与者发现同一个域中的其他参与者。

OpenDDS提供了一个集中式发现机制，一个对等发现机制和一个静态发现机制。 集中式机制使用运行DCPSInfoRepo进程的单独服务。 RTPS对等机制使用DDSI-RTPS发现协议标准来实现非集中式发现。 静态发现机制使用配置文件来确定哪些作者和读者应该关联，并使用底层传输来确定哪些作者和读者存在。

存在许多配置选项来满足DDS应用程序的部署需求。 除静态发现外，如果没有通过命令行或配置文件提供配置，则每个机制都使用默认值。

以下各节介绍如何配置高级发现功能。 例如，某些部署可能需要使用多个DCPSInfoRepo服务或DDSI-RTPS发现来满足互操作性要求。

7.3.1 域配置

OpenDDS配置文件使用[domain]部分类型来配置一个或多个发现域，每个域指向同一个文件或默认发现配置中的发现配置。 OpenDDS应用程序可以使用DCPSInfoRepo服务的集中式发现方法或使用RTPS发现协议标准的对等发现方法，或者在同一部署中使用这两种方法的组合。 DCPSInfoRepo方法的节类型是[repository]，RTPS发现配置的节类型是[rtps_discovery]。 静态发现机制没有专门的部分。 相反，用户需要引用DEFAULT_STATIC实例。 一个域只能引用一种类型的发现部分。

见部分7.3.2 为了配置[repository]部分，7.3.3 进行配置[rtps_discovery]和7.3.4 用于配置静态发现。

最终一个域被分配一个整数值，一个配置文件可以通过两种方式来支持这个。 首先是简单地将实例值分配给域的整数值，如下所示：

```
[域/ 1]
DiscoveryConfig = DiscoveryConfig1 (更多
    属性...)
```

我们的示例配置由domain关键字标识的单个域，后跟实例值/ 1。 在这种情况下斜杠后的实例值是分配给域的整数值。 对于相同的内容的另一种语法是使用一个更可识别的（友好）的名称，而不是一个数字的域名，然后将DomainId属性添加到该部分以提供整数值。这里是一个例子：

```
[域/书籍]
DomainId=1
DiscoveryConfig=DiscoveryConfig1
```

该域名有一个友好的书名。 DomainId属性分配读取配置的DDS应用程序所需的整数值1。多个域实例可以用这种格式在单个配置文件中标识。

一旦建立了一个或多个域实例，就必须为该域标识发现属性。 DiscoveryConfig属性必须指向保存发现配置的另一部分，或指定用于发现的内部默认值之一（例如DEFAULT_REPO，DEFAULT_RTPS或DEFAULT_STATIC）。 我们例子中的实例名称是DiscoveryConfig1。 此实例名称必须与[repository]或[rtps_discovery]的节类型相关联。

这是我们例子的扩展：

```
[domain/1]
DiscoveryConfig=DiscoveryConfig1

[repository / DiscoveryConfig1] RepositoryIor
= host1.mydomain.com: 12345
```

在这种情况下，我们的域指向一个用于OpenDDS的[repository]部分DCPSInfoRepo服务。 参见章节7.3.2 更多细节。

在配置文件中没有标识特定域的情况将会出现。 例如，如果OpenDDS应用程序为其参与者分配域ID 3，并且上述示例未提供域ID为3的配置，则可以使用以下内容：

```
[common] DCPSInfoRepo = host3.mydomain.com:
12345 DCPSDefaultDiscovery = DEFAULT_REPO

[domain/1]
DiscoveryConfig=DiscoveryConfig1

[repository / DiscoveryConfig1] RepositoryIor
= host1.mydomain.com: 12345
```

DCPSDefaultDiscovery属性告诉应用程序分配没有在配置文件中找到的域id的任何参与者，以使用发现类型为DEFAULT_REPO，意思是“使用DCPSInfoRepo服务”，并且可以在host3.mydomain上找到DCPSInfoRepo服务。 COM：12345。

如图所示表7-2DCPSDefaultDiscovery属性有三个可以使用的其他值。 DEFAULT_RTPS常量值通知没有域配置的参与者使用RTPS发现来查找其他参与者。 同样，DEFAULT_STATIC常量值通知没有域配置的参与者使用静态发现来查找其他参与者。

DCPSDefaultDiscovery属性的最后一个选项是告诉应用程序使用其中一个定义的发现配置作为文件中未被调用的任何参与者域的默认配置。 这里是一个例子：

```
[common]
DCPSDefaultDiscovery=DiscoveryConfig2
```


```
[domain/1]
DiscoveryConfig=DiscoveryConfig1

[repository / DiscoveryConfig1] RepositoryIor =
host1.mydomain.com: 12345

[domain/2]
DiscoveryConfig=DiscoveryConfig2

[repository / DiscoveryConfig2] RepositoryIor =
host2.mydomain.com: 12345
```

通过将DCPSDefaultDiscovery属性添加到[common]部分，任何尚未分配给域ID为1或2的参与者将使用DiscoveryConfig2的配置。 有关用于RTPS发现的类似配置的更多解释，请参见部分7.3.3.

以下是[域]部分的可用属性。

表7-3域部分配置属性

选项	描述
域ID =正	代表域的整数值 与存储库相关联。
DomainRepoKey=k	映射存储库的关键值 （已弃用。提供向后兼容性）。
DiscoveryConfig =配置实例名称	一个用户定义的字符串，引用同一配置文件中的[资源库]或[rtps_discovery]节的实例名称，或者其中一个内部默认值（DEFAULT_REPO，DEFAULT_RTPS或DEFAULT_STATIC）。 （另见DCPSDefaultDiscovery属性中表7-2)
DefaultTransportConfig =配置	引用实例的用户定义的字符串 [config]部分的名称。 参见章节7.4.

7.3.2

为DCPSInfoRepo配置应用程序OpenDDS DCPSInfoRepo是用于参与者发现的本地或远程节点上的服务。 配置参与者应该如何找到DCPSInfoRepo是其目的部分。 假设例如DCPSInfoRepo服务在myhost.mydomain.com:12345的主机和端口上启动。应用程序可以使他们的OpenDDS参与者意识到如何通过命令行选项或通过读取配置文件来找到该服务。

在2.1.7的“Getting Started”示例中，“运行示例”为可执行文件提供了一个命令行参数来查找DCPSInfoRepo服务，如下所示：

```
发布者-DCPSInfoRepo文件: //repo.ior
```

这假定DCPSInfoRepo已经以下面的语法启动了：

视窗：

```
%DDS_ROOT%\ bin \ DCPSInfoRepo -o repo.ior
```

Unix的：

```
$DDS_ROOT/bin/DCPSInfoRepo -o repo.ior
```

DCPSInfoRepo服务在该文件中生成位置对象信息，参与者需要读取该文件以最终连接。但是，在大多数生产环境中，使用基于文件的IOR来查找发现服务是不现实的，因此应用程序可以使用类似以下的命令行选项来简单指向DCPSInfoRepo所在的主机和端口。

```
发行商-DCPSInfoRepo myhost.mydomain.com:12345
```

以上假定DCPSInfoRepo已经在主机（myhost.mydomain.com）上启动，如下所示：

视窗：

```
%DDS_ROOT%\ bin \ DCPSInfoRepo -ORBListenEndpoints iiop: //: 12345
```

Unix的：

```
$ DDS_ROOT / bin / DCPSInfoRepo -ORBListenEndpoints iiop: //: 12345
```

如果应用程序需要使用配置文件进行其他设置，将发现内容放在文件中会更方便，并减少命令行的复杂性和混乱。 配置文件的使用也引入了多个应用程序进程共享公共OpenDDS配置的机会。 上面的例子可以很容易地移动到配置文件的[common]部分（假设pub.ini文件）：

```
[common] DCPSInfoRepo = myhost.mydomain.com:
12345
```

启动我们的可执行文件的命令行现在将变成以下内容：

```
发布者-DCSPConfigFile pub.ini
```

从部分讨论域来强化我们的例子，配置文件可以指定具有分配给这些域的发现配置的域。 在这种情况下，RepositoryIor属性用于获取将在命令行上提供的指向正在运行的DCPSInfoRepo服务的相同信息。 这里配置了两个域：

```
[domain/1]
DiscoveryConfig=DiscoveryConfig1

[repository / DiscoveryConfig1] RepositoryIor = myhost.mydomain.com:
12345
```


```
[domain/2]
DiscoveryConfig=DiscoveryConfig2

[repository / DiscoveryConfig2] RepositoryIor =
host2.mydomain.com: 12345
```

[domain / 1]下的DiscoveryConfig属性指示域1中的所有参与者使用名为DiscoveryConfig1的实例中定义的配置。 在上面，这被映射到一个[repository]节，给出myhost.mydomain.com:12345的RepositoryIor值。

最后，在配置DCPSInfoRepo时，域实例条目下的DiscoveryConfig属性还可以包含DEFAULT_REPO的值，该值指示参与者使用此实例在提供的位置使用属性DCPSInfoRepo的定义。 以下面的配置文件为例：

```
[common] DCPSInfoRepo = localhost:
12345

[domain/1]
DiscoveryConfig=DiscoveryConfig1

[repository / DiscoveryConfig1] RepositoryIor =
myhost.mydomain.com: 12345

[domain/2]
DiscoveryConfig=DEFAULT_REPO
```

在这种情况下，域2中的任何参与者将被指示参考在我们的示例的[common]部分中定义的DCPSInfoRepo的发现属性。 如果DCPSInfoRepo值在[common]部分中未提供，则可以将其作为参数提供给命令行，如下所示：

```
发布者-DCPSInfoRepo localhost: 12345 -DCPSConfigFile pub.ini
```

这将设置DCPSInfoRepo的值，以便读取配置文件pub.ini的参与者遇到DEFAULT_REPO时，会有一个值。 如果在配置文件或命令行中未定义DCPSInfoRepo，则DCPSInfoRepo的OpenDDS默认值为file: //repo.ior。 如前所述，这在生产环境中不太可能是最有用的，并且应该导致通过本节中描述的方法之一来设置DCPSInfoRepo的值。

7.3.2.1

配置多个DCPSInfoRepo实例单个OpenDDS进程中的DDS实体可以与多个DCPS信息存储库（DCPSInfoRepo）相关联。

存储库信息和域关联可以使用配置文件或通过应用程序API进行配置。 内部默认值，命令行参数和配置

文件选项将按原样应用于不想使用多个文件的现有应用程序
DCPSInfoRepo关联。

参考图7-1 作为使用多个DCPSInfoRepo存储库的过程的示例。 进程A和B是典型的应用程序进程，它们被配置为相互通信并在InfoRepo_1中互相发现。 这是一个简单的基本发现的使用。 但是，使用指定的域（域1）已经应用了额外的上下文层。 DDS实体（数据读取器/数据写入器）仅限于与同一个域内的其他实体进行通信。 这提供了一种在应用程序需要时分离流量的有用方法。 进程C和D的配置方式相同，但在域2中运行并使用InfoRepo_2。 如果您的应用程序需要使用多个域并拥有独立的发现服务，那么就会遇到挑战。 在我们的例子中，这是过程E. 它包含两个订户，一个订阅InfoRepo_1的出版物，另一个订阅InfoRepo_2的出版物。
什么允许配置工作可以在configE.ini文件中找到。

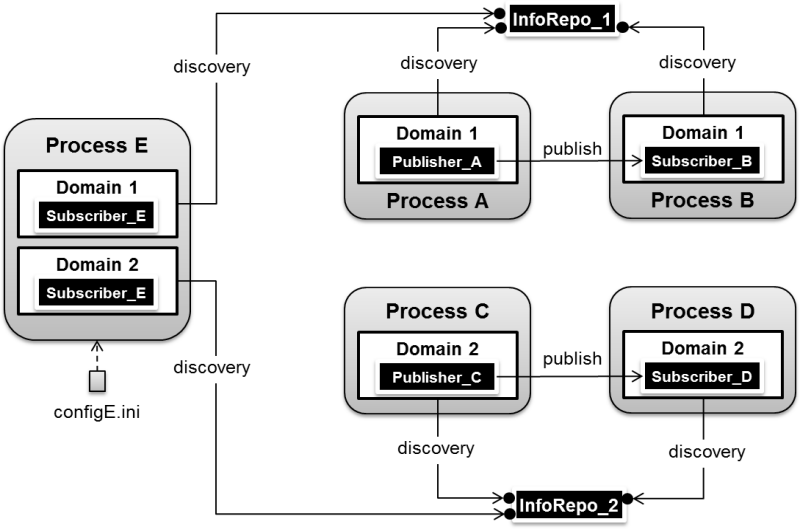


图7-1多个DCPSInfoRepo配置

现在我们将看看配置文件（称为configE.ini）来演示Process E如何与两个域进行通信并分离DCPSInfoRepo服务。 对于这个例子，我们将只显示配置的发现方面，不显示传输内容。

configE.ini

```
[domain/1]
DiscoveryConfig=DiscoveryConfig1

[repository / DiscoveryConfig1] RepositoryIor
= host1.mydomain.com: 12345

[域/ 2]
```

```
DiscoveryConfig=DiscoveryConfig2

[repository / DiscoveryConfig2] RepositoryIor =
host2.mydomain.com: 12345
```

E进程E时图7-1在上面的配置中读取它发现多个域部分的发生。 如Section所述，每个域都有一个实例整数和一个DiscoveryConfig属性。

对于第一个域（[domain / 1]），将为DiscoveryConfig属性提供用户定义的名称DiscoveryConfig1值。 该属性使OpenDDS实现找到存储库或rtps_discovery的部分标题以及DiscoveryConfig1的实例名称。 在我们的示例中，找到[repository / DiscoveryConfig1]部分标题，这成为域实例[domain / 1]（整数值1）的发现配置。 现在找到的部分告诉我们，该域应该使用的DCPSInfoRepo的地址可以通过使用RepositoryIor属性值来找到。 特别是它是host1.mydomain.com和端口12345. RepositoryIor的值可以是完整的CORBA IOR或简单的主机:端口字符串。

在此配置文件中找到第二个域标题[domain / 2]，以及相应的存储库部分[repository / DiscoveryConfig2]，它表示第二个感兴趣的域和InfoRepo_2存储库的配置。 单个配置文件中可能有任何数量的存储库或域部分。

注意 *未明确配置的域将自动与默认发现配置关联。*

单独的DCPSInfoRepos可以与多个域关联，但不能在多个DCPSInfoRepos之间共享域。

注意 以下是[repository]部分的有效属性。

表7-4多个存储库配置部分

选项	描述
RepositoryIor=ior	存储库IOR或主机：端口。
RepositoryKey =键	存储库的唯一键值。 （不推荐使用。为了向后兼容而提供）

7.3.3 配置DDSI-RTPS发现

OMG DDSI-RTPS规范给出了以下简单描述，形成了OpenDDS使用的发现方法的基础，以及用于完成发现操作的两种不同协议。 OMG DDSI-RTPS规范第8.5.1节摘录如下：

"RTPS规范将发现协议分成两个独立的协议：

1. 参与者发现协议
2. 端点发现协议

参与者发现协议（PDP）指定参与者在网络中发现彼此的方式。 一旦两个参与者相互发现，他们就使用端点发现协议（EDP）交换他们包含的端点信息。 除了这种因果关系之外，两种协议都可以被认为是独立的。"

本节中讨论的配置选项允许用户指定属性值来更改简单参与者发现协议（SPDP）和/或简单端点发现协议（SEDP）默认设置的行为。

DDSI-RTPS可以像在Section中那样为单个域或多个域配置7.3.2.1.

通过在我们的示例配置文件的[common]部分指定一个属性来实现一个简单的配置。

configE.ini (用于RTPS)

```
[common]
DCPSDefaultDiscovery=DEFAULT_RTPS
```

DDSI-RTPS发现的所有默认值都采用这种形式。 这个相同的基本配置的变体是指定一个部分来保存RTPS发现的更多特定参数。 以下示例使用[common]节来指向[rtps_discovery]节的实例，后面跟着由用户提供的TheRTPSConfig的实例名称。

```
[共同]
DCPSDefaultDiscovery=TheRTPSConfig

[rtps_discovery/TheRTPSConfig]
ResendPeriod=5
```

实例[rtps_discovery / TheRTPSConfig]现在是指定了改变默认DDSI-RTPS设置的属性的位置。在我们的示例中，ResendPeriod = 5条目设置了可用数据读取器/数据写入器的定期通告与检测网络上其他数据读取器/数据写入器的存在之间的秒数。 这将覆盖30秒的默认值。

如果您的OpenDDS部署使用多个域，则以下配置方法将[域]部分标题的使用与[rtps_discovery]相结合，以允许用户按域指定特定设置。 它可能看起来像这样：

configE.ini

```
[共同]
```

```
DCPSDebugLevel=0

[domain/1]
DiscoveryConfig=DiscoveryConfig1

[rtps_discovery/DiscoveryConfig1]
ResendPeriod=5

[domain/2]
DiscoveryConfig=DiscoveryConfig2

[rtps_discovery/DiscoveryConfig2]
ResendPeriod=5
SedpMulticast=0
```

OpenDDS中有关DDSI-RTPS发现的一些重要实现说明如下：

1) 由于UDP端口分配给域ID的方式，因此域ID应介于0和231（含）之间。 在每个OpenDDS过程中，每个域最多支持120个域参与者。

2) 由于分配了GUID，OpenDDS的本地多点传送不能用于RTPS发现（如果尝试这种情况，将发出警告）。

OMG DDSI-RTPS规范详细介绍了可以从影响DDSI-RTPS发现行为的默认值调整的几个属性。 找到他们表7-5.

表7-5 RTPS发现配置选项

选项	描述	默认
ResendPeriod =秒	参与者发布之间进程等待的秒数（参见OMG DDSI-RTPS规范中的第8.5.3节 详情）。	30
PB =端口	港口基地号码。 此编号设置了派生用于简单端点发现协议（SEDP）的端口号的起点。 该属性与DG，PG，DO（或DX）和D1一起使用来构建RTPS发现通信所需的端点。 （请参阅OMG DDSI-RTPS规范中的第9.6.1.1节 端点被构造）	7400
DG=n	表示域增益的整数值。 这是一个有助于制定组播的乘数 或用于RTPS的单播端口。	250

选项	描述	默认
PG=n	协助配置SPDP单播端口并用作参与者的偏移乘数的整数使用以下公式分配地址： PB + DG * domainId + d1 + PG * participantId （请参阅OMG DDSI-RTPS规范中的9.6.1.1部分了解这些端点的方式 建）	2
D0=n	一个整数值，用于帮助计算SPDP多播配置中可分配端口的偏移量。 使用的公式是： PB + DG * domainId + d0 （请参阅OMG DDSI-RTPS规范中的9.6.1.1部分了解这些端点的方式 建）	0
D1=n	在SPDP单播配置中帮助提供计算可分配端口的偏移量的整数值。 使用的公式是： PB + DG * domainId + d1 + PG * participantId （请参阅OMG DDSI-RTPS规范中的9.6.1.1部分了解这些端点的方式 建）	10
SedpMulticast=[0|1]	一个布尔值（0或1），确定组播是否用于SEDP流量。 当设置为1时，使用多播。 当设置为零（0）单播时 SEDP被使用。	1
SedpLocalAddress =地址：端口	配置由SEDP创建和使用的传输实例绑定到指定的本地地址和端口。 为了不指定端口，可以从设置中删除，但是尾随：必须 出席。	系统默认 地址
SpdpLocalAddress =地址	本地接口（无端口）的地址，SPDP将使用该接口来绑定到特定的接口。	DCPSDefaultAd 着装或IPADDR_ANY
DX=n	一个整数值，用于帮助计算SEDP组播配置中可分配端口的偏移量。 使用的公式是： PB + DG * domainId + dx 这只有当SedpMulticast = 1时才有效。这是一个OpenDDS扩展，而不是OMG DDSI-RTPS规范的一部分。	2

选项	描述	默认
SpdpSendAddrs = [host: port], [host: port] ...	用作SPDP内容目标的主机：端口对的列表（以逗号或空格分隔）。 这可以是单播和多播的组合地址。	
InteropMulticastOverride= group_address	指定要用于SPDP发现的多播组的网络地址。 这覆盖了规范的互操作性组。 例如，可以使用它来指定路由组的使用地址提供更大的发现范围。	239.255.0.1
TTL =正	作为发现一部分发送的多播数据报的生存时间（TTL）字段的值。 此值指定数据报在被网络丢弃之前将经过的跳数。 默认值1表示所有数据都是仅限于本地网络子网。	1
MulticastInterface=iface	指定此发现实例要使用的网络接口。 这使用标识网络接口的平台特定格式。 上Linux系统，这将是类似eth0。	系统默认界面是用过的
GuidInterface=iface	指定确定哪个本地MAC地址应该使用的网络接口出现在由此节点生成的GUID中。	系统/ ACE库默认使用

注意 *如果设置了环境变量OPENDDS_RTPS_DEFAULT_D0，则将其值用作D0默认值。*

7.3.4 配置静态发现

当DDS域具有固定数目的进程和数据读取器/写入器时，可以使用静态发现。 数据读者和作者统称为端点。 静态发现机制仅使用配置文件必须能够确定每个端点的网络地址和QoS设置。 静态发现机制使用这些信息来确定读者和作者之间所有潜在的关联。 域参与者通过底层传输提供的提示来了解端点的存在。

目前，静态发现只能用于使用RTPS UDP传输的端点。

静态发现引入了以下配置文件部分：[topic / *]，[datawriterqos / *]，[datareaderqos / *]，[publisherqos / *]，[subscriberqos / *]和[endpoint / *]。 话题/*] （表7-6）部分用于介绍一个主题。 [datawriterqos / *]（表7-7），[datareaderqos / *]（表7-8），[publisherqos / *]（表

7-9）和[subscriberqos / *]（表7-10）部分用于描述关联类型的QoS。 [endpoint / *]（表7-11）部分描述了数据读取器或写入器。

数据读取器和写入器对象必须由用户标识，以便静态发现机制可以将它们与配置文件中正确的[endpoint / *]部分相关联。 这是通过将DomainParticipantQos的user_data设置为长度为6的八位字节序列完成的。该八位字节序列的表示形式作为每个八位字节具有两个十六进制数字的字符串出现在[endpoint / *]节的参与者值中。
类似地，DataReaderQos或DataWriterQos的user_data必须被设置为对应于[endpoint / *]节中的实体值的长度为3的八位字节序列。 例如，假设配置文件包含以下内容：

```
[topic/MyTopic]
type_name=TestMsg::TestMsg

[endpoint/MyReader]
type=reader
topic=MyTopic
config=MyConfig
domain=34
participant=0123456789ab
entity=cdef01

[config / MyConfig]传输= MyTransport

[transport / MyTransport] transport_type =
rtps_udp use_multicast = 0 local_address =
1.2.3.4: 30000
```

配置DomainParticipantQos的相应代码是：

```
DDS::DomainParticipantQos dp_qos;
domainParticipantFactory-> get_default_participant_qos (dp_qos) ;  dp_qos.user_data.value.length (6) ;
dp_qos.user_data.value[0] = 0x01;
dp_qos.user_data.value[1] = 0x23;
dp_qos.user_data.value[2] = 0x45;
dp_qos.user_data.value[3] = 0x67;
dp_qos.user_data.value[4] = 0x89;
dp_qos.user_data.value[5] = 0xab;
```

配置DataReaderQos的代码是相似的：

```
DDS::DataReaderQos qos;
订户> get_default_datareader_qos (QOS) ;
qos.user_data.value.length (3) ;  qos.user_data.value
[0] = 0xcd;  qos.user_data.value [1] = 0xef;
qos.user_data.value [2] = 0x01;
```

在这个例子中，域ID是34，应该被传递给调用

create_participant。

在这个例子中，MyReader的端点配置引用了MyConfig，MyConfig又引用了MyTransport。 传输配置在Section中描述7.4。 该

静态发现的重要细节是，至少有一个传输包含一个已知的网络地址（1.2.3.4:30000）。如果无法为端点确定地址，将会发出错误。 静态发现实现还检查数据读取器或数据写入器对象的QoS是否与配置文件中指定的QoS匹配。

表7-6 [topic / *]配置选项

选项	描述	默认
名称=字符串	主题的名称。	实例名称 部分
TYPE_NAME =字符串	唯一定义样本类型的标识符。 这通常是一个CORBA 接口存储库类型 名称。	需要

表7-7 [datawriterqos / *]配置选项

选项	描述	默认
durability.kind=[挥发性| TRANSIENT_LOCAL]	参见章节3.2.5.	看到表3-5.
deadline.period.sec=[数字| DURATION_INFINITE_SEC]	参见章节3.2.9.	看到表3-5.
deadline.period.nanosec=[数字| DURATION_INFINITE_NANOSEC]	参见章节3.2.9.	看到表3-5.
latency_budget.duration.sec=[数字| DURATION_INFINITE_SEC]	参见章节3.2.15.	看到表3-5.
latency_budget.duration.nanosec=[数字| DURATION_INFINITE_NANOSEC]	参见章节3.2.15.	看到表3-5.
liveliness.kind=[AUTOMA TIC| MANUAL_BY_TOPIC| MANUAL_BY_PARTICIPANT]	参见章节3.2.2.	看到表3-5.
liveliness.lease_duration.sec=[数字| DURATION_INFINITE_SEC]	参见章节3.2.2.	看到表3-5.
liveliness.lease_duration.nanosec=[数字| DURATION_INFINITE_NANOSEC]	参见章节3.2.2.	看到表3-5.
reliability.kind=[BEST_EFFORT|RELIABILE]	参见章节3.2.3.	看到表3-5.
reliability.max_blocking_time.sec=[数字| DURATION_INFINITE_SEC]	参见章节3.2.3.	看到表3-5.
reliability.max_blocking_time.nanosec=[数字| DURATION_INFINITE_NANOSEC]	参见章节3.2.3.	看到表3-5.
destination_order.kind=[BY_SOURCE_TIMESTAMP| BY_RECEPTION_TIMESTAMP]	参见章节3.2.18.	看到表3-5.
history.kind=[KEEP_LAST|KEEP_ALL]	参见章节3.2.4.	看到表3-5.
history.depth =数字	参见章节3.2.4.	看到表3-5.

选项	描述	默认
resource_limits.max_samples =数字	参见章节3.2.7.	看到表3-5.
resource_limits.max_instances =数字	参见章节3.2.7.	看到表3-5.
resource_limits.max_samples_per_instance= 数字	参见章节3.2.7.	看到表3-5.
transport_priority.value =数字	参见章节3.2.14.	看到表3-5.
lifespan.duration.sec=[数字| DURATION_INFINITE_SEC]	参见章节3.2.10.	看到表3-5.
lifespan.duration.nanosec=[数字| DURATION_INFINITE_NANOSEC]	参见章节3.2.10.	看到表3-5.
ownership.kind=[SHARED|EXCLUSIVE]	参见章节3.2.22.	看到表3-5.
ownership_strength.value =数字	参见章节3.2.23.	看到表3-5.

表7-8 [datareaderqos / *]配置选项

选项	描述	默认
durability.kind=[挥发性| TRANSIENT_LOCAL]	参见章节3.2.5.	看到表3-6.
deadline.period.sec=[数字| DURATION_INFINITE_SEC]	参见章节3.2.9.	看到表3-6.
deadline.period.nanosec=[数字| DURATION_INFINITE_NANOSEC]	参见章节3.2.9.	看到表3-6.
latency_budget.duration.sec=[数字| DURATION_INFINITE_SEC]	参见章节3.2.15.	看到表3-6.
latency_budget.duration.nanosec=[数字| DURATION_INFINITE_NANOSEC]	参见章节3.2.15.	看到表3-6.
liveliness.kind=[AUTOMA TIC| MANUAL_BY_TOPIC| MANUAL_BY_PARTICIPANT]	参见章节3.2.2.	看到表3-6.
liveliness.lease_duration.sec=[数字| DURATION_INFINITE_SEC]	参见章节3.2.2.	看到表3-6.
liveliness.lease_duration.nanosec=[数字| DURATION_INFINITE_NANOSEC]	参见章节3.2.2.	看到表3-6.
reliability.kind=[BEST_EFFORT|RELIABILE]	参见章节3.2.3.	看到表3-6.
reliability.max_blocking_time.sec=[数字| DURATION_INFINITE_SEC]	参见章节3.2.3.	看到表3-6.
reliability.max_blocking_time.nanosec=[数字| DURATION_INFINITE_NANOSEC]	参见章节3.2.3.	看到表3-6.
destination_order.kind=[BY_SOURCE_TIMESTAMP| BY_RECEPTION_TIMESTAMP]	参见章节3.2.18.	看到表3-6.
history.kind=[KEEP_LAST|KEEP_ALL]	参见章节3.2.4.	看到表3-6.
history.depth =数字	参见章节3.2.4.	看到表3-6.

选项	描述	默认
resource_limits.max_samples =数字	参见章节3.2.7.	看到表3-6.
resource_limits.max_instances =数字	参见章节3.2.7.	看到表3-6.
resource_limits.max_samples_per_instance= 数字	参见章节3.2.7.	看到表3-6.
time_based_filter.minimum_separation.sec=[数字| DURATION_INFINITE_SEC]	参见章节3.2.21.	看到表3-6.
time_based_filter.minimum_separation.nanosec=[数字| DURATION_INFINITE_NANOSEC]	参见章节3.2.21.	看到表3-6.
reader_data_lifecycle。 autopurge_nowriter_samples_delay.sec = [数字| DURATION_INFINITE_SEC]	参见章节3.2.20.	看到表3-6.
reader_data_lifecycle。 autopurge_nowriter_samples_delay.nanosec = [数字| DURATION_INFINITE_NANOSEC]	参见章节3.2.20.	看到表3-6.
reader_data_lifecycle。 autopurge_dispose_samples_delay.sec = [数字| DURATION_INFINITE_SEC]	参见章节3.2.20.	看到表3-6.
reader_data_lifecycle。 autopurge_dispose_samples_delay.nanosec = [数字| DURATION_INFINITE_NANOSEC]	参见章节3.2.20.	看到表3-6.

表7-9 [publisherqos / *]配置选项

选项	描述	默认
presentation.access_scope=[INSTANCE|TOPIC|GROUP]	参见章节3.2.17.	看到表3-3.
presentation.coherent_access=[true|false]	参见章节3.2.17.	看到表3-3.
presentation.ordered_access=[true|false]	参见章节3.2.17.	看到表3-3.
partition.name = NAME0, NAME1, ...	参见章节3.2.8.	看到表3-3.

表7-10 [subscriberqos / *]配置选项

选项	描述	默认
presentation.access_scope=[INSTANCE|TOPIC|GROUP]	参见章节3.2.17.	看到表3-4.
presentation.coherent_access=[true|false]	参见章节3.2.17.	看到表3-4.
presentation.ordered_access=[true|false]	参见章节3.2.17.	看到表3-4.
partition.name = NAME0, NAME1, ...	参见章节3.2.8.	看到表3-4.

表7-11 [端点/ *]配置选项

选项	描述	默认
域=数字	范围0-231中端点的域标识。 用于 形成端点的GUID。	需要

选项	描述	默认
参与者=十六进制串	12个十六进制数字的字符串。 用于形成端点的GUID。 具有相同域/参与者组合的所有端点应该位于同样的过程。	需要
实体=十六进制串	6个十六进制数字的字符串。 用于形成端点的GUID。 的组合域/参与者/实体应该是唯一的。	需要
类型=［读者|作家］	确定实体是数据读取器还是数据作家。	需要
话题=名称	指的是[话题/ *]部分。	需要
datawriterqos =名称	指[datawriterqos / *]部分。	看到表3-5.
datareaderqos =名称	指[datareaderqos / *]部分。	看到表3-6.
publisherqos =名称	指[publisherqos / *]部分。	看到表3-3.
subscriberqos =名称	指[subscriberqos / *]部分。	看到表3-4.
配置	引用[config / *]中的传输配置部分。 这用于确定端点的网络地址。	

7.4 传输配置

从OpenDDS 3.0开始，新的传输配置设计已经实现。 这个设计的基本目标是：

- 允许简单部署忽略传输配置并使用智能默认值进行部署（发布者或订阅者不需要传输代码）。
- 只使用配置文件和命令行选项来启用应用程序的灵活部署。
- 允许在各个数据编写者和编写者中混合传输的部署。 发布者和订阅者根据传输配置，QoS设置和网络可达性的细节来协商使用适当的传输实现。
- 支持复杂网络中更广泛的应用程序部署。
- 支持优化的传输开发（例如并置和共享内存传输 - 注意这些目前没有实现）。
- 将对可靠性QoS策略的支持与基础传输集成在一起。
- 尽可能避免依赖ACE Service Configurator及其配置文件。

不幸的是，实现这些新功能涉及打破与OpenDDS传输配置代码和以前版本文件的向后兼容性。 有关如何将现有应用程序转换为使用新的传输配置设计的信息，请参阅$ DDS_ROOT / docs / OpenDDS_3.0_Transition.txt。

7.4.1 概观

7.4.1.1 运输概念

本节概述了传输配置中涉及的概念以及它们如何交互。

每个数据读取器和写入器都使用由一组有序传输实例组成的传输配置。 每个传输实例都指定一个传输实现（即tcp，udp，multicast，shmem或rtps_udp），并可以自定义由该传输定义的配置参数。 传输配置和传输实例由传输注册管理，可以通过配置文件或编程API创建。

可以为域参与者，发布者，订阅者，数据写入者和数据读取者指定传输配置。 当数据读取器或写入器启用时，它使用它可以找到的最具体的配置，直接绑定到它或通过其父实体可访问。 例如，如果数据写入器指定传输配置，则始终使用它。 如果数据写入器没有指定配置，则会尝试使用其发布者或域参与者的顺序。 如果这些实体都没有指定传输配置，则将从传输注册表中获取全局传输配置。 全局传输配置可由用户通过配置文件，命令行选项或传输注册表上的成员函数调用指定。 如果用户未定义，则使用默认的传输配置，其中包含所有可用的传输实现及其默认配置参数。 如果您没有专门加载或链接其他传输实现，则OpenDDS使用tcp传输进行所有通信。

7.4.1.2 OpenDDS如何选择一个传输

目前，OpenDDS的行为是数据写入者主动连接到被动等待这些连接的数据读取器。 数据读取器“监听”在其传输配置中定义的每个传输实例上的连接。 数据编写者使用他们的传输实例来“连接”到数据读取器的传输实例。 由于这里讨论的逻辑连接不对应于传输的物理连接，所以OpenDDS经常将它们称为数据链接。

当数据写入器尝试连接到数据读取器时，首先会尝试查看是否存在可用于与该数据读取器通信的现有数据链接。 Data Writer通过每个传输实例迭代（按定义顺序），并查找读取器定义的传输实例的现有数据链接。 如果找到现有的数据链接，它将用于Data Writer和Reader之间的所有后续通信。

如果找不到现有的数据链接，则数据写入器将按照传输配置中定义的顺序尝试使用不同的传输实例进行连接。 任何不与另一方"匹配"的传输实例都被跳过。 例如，如果编写者指定udp和tcp传输实例，而读者仅指定tcp，则忽略udp传输实例。 匹配算法也可能受到QoS参数，实例配置以及传输实现的其他细节的影响。 成功"连接"的第一对传输实例将导致用于所有后续数据示例发布的数据链接。

7.4.2 配置文件示例

下面的例子解释了通过文件传输配置的基本功能，并描述了一些常见的用例。 接下来是这些功能的完整参考文档。

7.4.2.1 单传输配置

为您的应用程序提供传输配置的最简单方法是使用OpenDDS配置文件。 下面是一个示例配置文件，可能由运行在具有两个网络接口的计算机上的应用程序使用，这些网络接口只想使用其中的一个进行通信：

```
[通用] DCPSGlobalTransportConfig = myconfig

[config/myconfig]
transports=mytcp

[transport/mytcp]
transport_type=tcp
local_address=myhost
```

这个文件做了以下（从底部开始）：

1) 定义一个名为mytcp的传输实例，传输类型为tcp，本地地址指定为myhost，它是我们要使用的网络接口对应的主机名。

2) 定义使用传输实例的名为myconfig的传输配置mytcp作为其唯一的传输。

3） 使名为myconfig的传输配置成为此进程中所有实体的全局传输配置。

使用这个配置文件的过程利用我们为它创建的所有数据读取器和写入器的自定义传输配置（除非我们专门绑定代码中的其他配置，如7.4.2.3).

7.4.2.2 使用混合传输

本示例将应用程序配置为主要使用多播，并在无法使用多播时“回退”到tcp。 这里是配置文件：

```
[通用] DCPSGlobalTransportConfig = myconfig

[config / myconfig] transports = mymulticast, mytcp

[transport/mymulticast]
transport_type=multicast

[transport/mytcp]
transport_type=tcp
```

名为myconfig的传输配置现在包含两个传输实例mymulticast和mytcp。 这些传输实例都不指定除transport_type之外的任何参数，因此它们使用这些传输实现的默认配置。 用户可以自由使用以下参考部分中列出的任何特定于传输的配置参数。

假设所有参与进程使用这个配置文件，应用程序试图使用多播来启动数据写入者和读者之间的通信。 如果初始多播通信因任何原因失败（可能是因为中间路由器未通过多播通信），则使用tcp来发起连接。

7.4.2.3 使用多个配置

对于许多应用程序，一个配置不同等适用于给定过程中的所有通信。 这些应用程序必须创建多个传输配置，然后将其分配给进程的不同实体。

对于这个例子，考虑托管在具有两个网络接口的计算机上的应用程序，这些网络接口需要通过一个接口传送一些数据，剩下的则通过另一接口传送。 这是我们的配置文件：

```
[common]
DCPSGlobalTransportConfig=config_a

[配置/ config_a]
```

```
输送= tcp_a

[config/config_b]
transports=tcp_b

[transport/tcp_a]
transport_type=tcp
local_address=hosta

[transport/tcp_b]
transport_type=tcp
local_address=hostb
```

假设hosta和hostb是分配给两个网络接口的主机名，我们现在有单独的配置，可以在各自的网络上使用tcp。 上面的文件将“A”端配置设置为默认值，这意味着我们必须手动将我们想要使用另一端的任何实体绑定到“B”端配置。

OpenDDS提供了两种分配配置给实体的机制：

- 通过将配置附加到实体（读者，作者，发布者，订阅者或域参与者）
- 通过将配置与域相关联的配置文件以下是源代码机制（使用域参

与者）：

```
DDS::DomainParticipant_var dp =
        DPF-> create_participant (MY_DOMAIN,
                                  PARTICIPANT_QOS_DEFAULT,
                                  DDS :: DomainParticipantListener :: _ nil () ,
                                  OpenDDS :: DCPS :: DEFAULT_STATUS_MASK) ;

OpenDDS :: DCPS :: TransportRegistry :: instance ()  - > bind_config (“config_b”, dp) ;
```

此域参与者拥有的任何数据写入者或读者现在应使用“B”端配置。

注意 *将配置直接绑定到数据写入器或读取器时，必须在启用读写器之前进行bind_config调用。 将配置绑定到域参与者，发布者或订阅者时，这不是问题。 参见章节3.2.16 有关如何创建未启用的实体的详细信息。*

7.4.3 传输注册表示例

OpenDDS允许开发人员通过C ++ API定义传输配置和实例。 OpenDDS :: DCPS :: TransportRegistry类用于构造OpenDDS :: DCPS :: TransportConfig和OpenDDS :: DCPS :: TransportInst对象。

TransportConfig和TransportInst类包含与以下定义的选项相对应的公共数据成员。 本节包含的代码相当于

简单的传输配置文件中描述。 首先，我们需要包含正确的头文件：

```
#include <dds/DCPS/transport/framework/TransportRegistry.h>
#include <dds/DCPS/transport/framework/TransportConfig.h>
#include <dds/DCPS/transport/framework/TransportInst.h>
#include <dds/DCPS/transport/tcp/TcpInst.h>

using namespace OpenDDS::DCPS;
```

接下来我们创建传输配置，创建传输实例，配置
传输实例，然后将实例添加到配置的实例集合中：

```
TransportConfig_rch cfg = TheTransportRegistry-> create_config (“myconfig”) ;
TransportInst_rch inst = TheTransportRegistry-> create_inst (“mytcp”, // name
                                                                “TCP”) ;  //类型

// 必须强制转换为TcpInst才能访问特定于传输的选项TcpInst_rch tcp_inst =
dynamic_rchandle_cast <TcpInst> (inst) ;  tcp_inst-> local_address_str_
=“myhost”;

// 将inst添加到config cfg->
instances_.push_back (inst) ;
```

最后，我们可以使我们新定义的传输配置成为全局传输配置：

```
TheTransportRegistry-> global_config ( “的myconfig”) ;
```

此代码应在任何数据读取器或写入器启用之前执行。

请参阅上面包含的头文件以获取可以使用的公共数据成员和成员函数的完整列表。 请参阅以下各节中的选项说明，以全面了解这些设置的语义。

回过头来，将这段代码与原来的配置文件进行比较，配置文件比相应的C ++代码简单得多，并具有在运行时可修改的优点。 很容易看出为什么我们建议几乎所有的应用程序都应该使用配置文件机制来进行传输配置。

7.4.4 传输配置选项

传输配置在OpenDDS配置文件中通过格式为[config / <name>]的部分指定，其中<name>是该进程中该配置的唯一名称。 下表总结了指定传输配置时的选项：

表7-12传输配置选项

选项	描述	默认
输送= INST1 [, INST2] [, ...]	传输实例名称的有序列表 这个配置将被利用。 每个传输配置都需要该字段。	没有
swap_bytes=*[0|1]*	值为0将导致DDS序列化源计算机的本地字节序中的数据；值为1会导致DDS以相反的字节序列化数据。 接收方将调整数据的字节顺序，所以不需要在机器之间匹配这个选项。 这个选项的目的是让开发者决定哪个 如有必要，方将进行最后的调整。	0
passive_connect_duration =毫秒	初始被动连接建立超时（毫秒）。 默认情况下，这个选项等待十秒。 值为零将等待 无限期（不推荐）。	10000

passive_connect_duration选项通常设置为非零的正整数。 如果没有合适的连接超时，订户端点可能会在等待远程端发起连接时进入死锁状态。 因为发布者和订阅者都可以有多个传输实例，所以需要将此选项设置为足够高的值，以便发布者遍历这些组合，直到成功为止。

除了用户定义的配置之外，OpenDDS还可以隐式定义两个传输配置。 第一个是默认配置，包括链接到进程中的所有传输实现。 如果没有找到，则只使用tcp。 每个这些传输实例都使用该传输实现的默认配置。 这是用户没有定义的全局传输配置。

每当使用OpenDDS配置文件时，将定义第二个隐式传输配置。 它被赋予与正被读取的文件相同的名称，并且包括在该文件中定义的所有传输实例，按照其名称的字母顺序。 用户可以通过在同一文件中指定DCPSGlobalTransportConfiguration = $ file选项来最容易地使用此配置。 $文件值总是绑定到当前文件的隐式文件配置。

7.4.5 传输实例选项

传输实例在OpenDDS配置文件中通过[transport / <name>]格式的部分指定，其中<name>是该实例中的唯一名称

处理。 每个传输实例都必须使用有效的传输实现类型指定transport_type选项。 以下部分列出了可以指定的其他选项，从所有传输类型通用的选项开始，然后是针对每种传输类型的选项。

使用动态库时，只要在配置文件中定义了该类型的实例，就会动态加载OpenDDS传输库。 当使用自定义传输实现或静态链接时，应用程序开发人员负责确保传输实现代码与其可执行文件链接。

7.4.5.1 所有传输通用的配置选项

下表总结了所有传输通用的传输配置选项：

表7-13公共传输配置选项

选项	描述	默认
transport_type =运输	运输的类型；可用传输列表可以通过传输框架以编程方式扩展。 tcp，udp，多播，shmem和rtps_udp都包含在OpenDDS中。	没有
queue_messages_per_pool =正	当检测到背压时，要发送的消息排队。 当消息队列必须成长，这个数字增长。	10
queue_initial_pools =正	背压队列的初始池数量。 两个背压队列值的默认设置预分配空间50条消息（5条10条消息池）。	5
MAX_PACKET_SIZE =正	传输包的最大尺寸，包括其传输头，样本头和样本数据。	2147481599
max_samples_per_packet =正	运输中的最大样本数量包。	10
optimum_packet_size =正	即使仍然有排队的样本要发送，大于这个大小的传输包也将通过网络发送。 此值可能会影响性能，具体取决于您的网络配置和应用性质。	4096
thread_per_connection= *[0|1]*	启用或禁用每个连接发送的线程战略。 默认情况下，此选项被禁用。	0

选项	描述	默认
datalink_release_delay =秒	datalink_release_delay是没有关联后数据链路释放的延迟（以秒为单位）。 读取器/写入器关联时增加此值可以减少重新建立的开销经常被添加和移除。	10

只要线程上下文切换的开销没有超过并行写入的好处，启用thread_per_connection选项将提高向不同进程写入多个数据读取器时的性能。 网络性能与上下文切换开销的这种平衡最好通过实验来确定。 如果一台机器有多个网卡，则可以通过为每个网卡创建一个传输来提高性能。

7.4.5.2 TCP / IP传输配置选项

有许多可配置的tcp传输选项。 正确配置的运输为下层的堆叠干扰提供了额外的弹性。几乎所有可用于定制连接和重新连接策略的选项都有合理的默认值，但最终应根据对特定DDS应用程序和目标环境中的网络质量和所需QoS的仔细研究来选择这些值。

local_address选项被对等体用来建立连接。 默认情况下，TCP传输在已经解析FQDN（完全限定的域名）的NIC上选择临时端口号。 因此，如果您有多个NIC或者您希望指定端口号，您可能希望显式设置地址。 当您配置主机间通信时，local_address不能是本地主机，应该配置一个外部可见的地址（即192.168.0.2），或者您可以不指定，在这种情况下将使用FQDN和临时端口。

FQDN分辨率取决于系统配置。 在没有FQDN（例如example.objectcomputing.com）的情况下，OpenDDS将使用任何发现的短名称（例如）。 如果失败，它将使用从回送地址（例如localhost）解析的名称。

注意 *OpenDDS IPv6支持要求基础ACE / TAO组件在启用IPv6支持的情况下构建。local_address必须是IPv6十进制地址或带端口号的FQDN。 FQDN必须可解析为IPv6地址。*

tcp传输作为一个独立的库存在，需要被链接才能使用它。 使用动态链接构建时，OpenDDS会在配置文件中引用时自动加载传输库，或者在没有其他传输库
指定运输。

当静态构建tcp库时，应用程序必须直接链接库。 为此，您的应用程序必须首先包含用于服务初始化的正确标头：<dds / DCPS / transport / tcp / Tcp.h>。

您还可以编程方式配置发布者和订阅者传输实现，如下所述。 配置用户和发布者应该是相同的，但是应该为每个传输实例分配不同的地址/端口。

下表总结了对于特定的传输配置选项
tcp传输：

表7-14 TCP / IP配置选项

选项	描述	默认
conn_retry_attempts =正	放弃和调用on_publication_lost () 和之前的重新连接次数 on_subscription_lost () 回调。	3
conn_retry_initial_delay =毫秒	初始延迟（毫秒）重新连接尝试。 一旦检测到丢失的连接，就尝试重新连接。 如果这个重新连接失败，再次尝试 在这个特定的延迟之后被做。	500
conn_retry_backoff_multiplier =正	重新连接尝试的退避乘数。 在如上所述的初始延迟之后，随后的延迟由该乘法器和前一延迟的乘积确定。 例如，在conn_retry_initial_delay为500，conn_retry_backoff_multiplier为1.5的情况下，第二次重新连接尝试将在第一次重试连接失败后0.5秒；第二次重试连接失败后第三次尝试将是0.75秒；第四次尝试将在第三次之后1.125秒 重试连接失败。	2.0
enable_nagle_algorithm=*[0|1]*	启用或禁用Nagle的算法。 默认情况下，它被禁用。 启用Nagle的算法可能会增加 吞吐量以增加延迟为代价。	0
local_address =主机：端口	连接接受者的主机名和端口。 默认值是FQDN和端口0，这意味着操作系统将选择端口。 如果仅指定了主机并且省略了端口号，则“：”是 仍然需要在主机说明符上。	FQDN：0

选项	描述	默认
max_output_pause_period =毫秒	不能发送排队消息的最大周期（毫秒）。 如果排队的样本没有超过这个时间，那么连接将被关闭，并调用_ * _ lost （）回调函数。 默认值为零意味着这个检查不是做的。	0
passive_reconnect_duration =毫秒	被动连接方等待连接的时间段（毫秒）重新连接。 如果在这段时间内没有重新连接那么on _ _ _ lost （）回调将被调用。	2000
pub_address =主机：端口	用配置的字符串覆盖发送给对等体的地址。 这可以用于防火墙遍历和其他高级网络配置。	

TCP / IP重新连接选项

当TCP / IP连接关闭时，OpenDDS尝试重新连接。 重新连接过程是（一个成功的重新连接结束这个序列）：

- 一旦检测到丢失的连接立即尝试重新连接。
- 如果失败，则等待conn_retry_initial_delay毫秒并尝试重新连接。
- 虽然我们还没有试过conn_retry_attempts，但是等待（以前的等待时间* conn_retry_backoff_multiplier）毫秒并尝试重新连接。

7.4.5.3 UDP / IP传输配置选项

udp传输是仅支持尽力而为传输的裸骨传输。 像tcp，local_address一样，它同时支持IPv4和IPv6地址。

udp作为独立的库存在，因此需要像其他传输库一样进行链接和配置。 当使用动态库构建时，OpenDDS会在配置文件中引用它时自动加载库。 当udp库静态构建时，应用程序必须直接链接到库。 另外，您的应用程序还必须包含用于服务初始化的正确标题：
<DDS / DCPS /运输/ UDP / Udp.h>。

下表总结了对于特定的传输配置选项
udp运输：

表7-15 UDP / IP配置选项

选项	描述	默认
local_address =主机：端口	监听套接字的主机名和端口。 默认为底层操作系统选择的值。 端口可以省略，在这种情况下值应该以“：”结尾。	FQDN：0
send_buffer_size =正	以字节为单位的总发送缓冲区大小UDP有效负载。	平台的价值ACE_DEFAULT_MAX_SOCKET_BUFSIZ
rcv_buffer_size =正	总接收缓冲区大小（字节）为UDP有效载荷。	平台的价值ACE_DEFAULT_MAX_SOCKET_BUFSIZ

7.4.5.4 IP多播传输配置选项

组播传输为基于传输配置参数的尽力而又可靠的传输提供统一的支持。

数据交换在同行之间尽力而为地交付施加最少的开销，但是它不提供交付的任何保证。数据可能因无响应或无法访问的对等体或重复收到而丢失。

可靠的交付提供了保证的数据交付到相关联的同行，没有重复的代价是额外的处理和带宽。 可靠交付是通过两个主要机制实现的：双向同行握手和对缺失数据的否定确认。 这些机制中的每一个都是有界的，以确保确定性的行为，并且是可配置的，以确保用户环境可能的最广泛的适用性。

多播支持多种配置选项：

default_to_ipv6和port_offset选项影响如何选择默认多播组地址。 如果default_to_ipv6设置为“1”（启用），则将使用默认的IPv6地址（[FF01 :: 80]）。port_offset选项确定组地址未使用时使用的默认端口，默认值为49152。

group_address选项可以用来手动定义一个多播组加入以交换数据。 支持IPv4和IPv6地址。 与tcp一样，OpenDDS IPv6支持要求底层ACE / TAO组件在启用IPv6支持的情况下构建。

在具有多个网络接口的主机上，可能需要指定应该在特定接口上加入多播组。local_address选项可以设置为接收组播流量的本地接口的IP地址。

如果需要可靠的交付，可以指定可靠的选项（默认）。 配置选项的其余部分影响多播传输使用的可靠性机制：

syn_backoff，syn_interval和syn_timeout配置选项会影响握手机制。 syn_backoff是在计算重试之间的退避延迟时使用的指数基数。 syn_interval选项定义重试握手之前等待的最小毫秒数。 syn_timeout定义在放弃握手之前要等待的最大毫秒数。

给定syn_backoff和syn_interval的值，可以计算握手尝试之间的延迟（以syn_timeout为界）：

```
delay = syn_interval * syn_backoff ^ number_of_retries
```

例如，如果假设默认配置选项，则握手尝试之间的延迟将分别为0, 250, 1000, 2000, 4000和8000毫秒。

nak_depth，nak_interval和nak_timeout配置选项会影响否定确认机制。 nak_depth确定传输为保留传入修复请求而保留的数据报的最大数量。 nak_interval配置选项定义了修复请求之间等待的最小毫秒数。 这个间隔是随机的，以防止类似的关联对等体之间的潜在冲突。 修复请求之间的最大延迟限制为最小值的两倍。

nak_timeout配置选项定义在放弃之前等待修复请求的最长时间。

nak_delay_intervals配置选项定义在初始nak之后nak之间的间隔数。

nak_max配置选项限制缺少样本的最大次数。 使用此选项，以便在nak_timeout之前，naks不会被重复发送给不可恢复的数据包。

目前，在使用这种运输方式时，有一些要求超出了ETF已经规定的要求：

- 每个多播组最多可以使用一个DDS域；
- 给定的参与者可能只有每个多播组附加一个单播组播传输；如果您希望在同一个进程中在同一个多播组上发送和接收样本，则必须使用独立的参与者。

组播作为独立的库存在，因此需要像其他传输库一样进行链接和配置。 当使用动态库构建时，OpenDDS会在配置文件中引用它时自动加载库。 当静态构建多播库时，应用程序必须直接链接到库。 另外，您的应用程序还必须包含用于服务初始化的正确标题：*<DDS / DCPS /传输/多播/ Multicast.h>*。

下表总结了对于特定的传输配置选项
组播传输：

表7-16 IP组播配置选项

选项	描述	默认
default_to_ipv6=*[0|1]*	启用IPv6默认组地址选择。 默认情况下，此选项被禁用。	0
group_address =主机：端口	多播组加入发送/接收数据。	224.0.0.128：<端口>， [FF01 :: 80]：<端口>
local_address =地址	如果非空，则用于加入组播的本地网络接 口的地址 组。	
nak_delay_intervals =正	之后naks之间的间隔数目 最初的nak。	4
nak_depth =正	要保留的数据报的数量 服务修理请求（仅可靠）。	32
nak_interval =毫秒	等待的最小毫秒数 维修请求之间（可靠）。	500
nak_max =正	缺少的最大次数 样品将停止。	3
nak_timeout =毫秒	放弃修复响应之前等待的最大毫秒数（可靠 只要）。 默认值是30秒。	30000
port_offset =正	用于在未指定组地址时设置端口号。 当指定组地址时，使用其中的端口号。 如果没有指定组地址，则使用端口偏移作为端口号。 这个值不应小于49152。	49152
rcv_buffer_size =正	套接字接收缓冲区的大小（以字节为单位）。 一个 零值表示使用系统默认值。	0
可靠=[0 | 1]	实现可靠的通信。	1
syn_backoff =正	握手期间使用的指数基数 重试；值越小，尝试之间的延迟越短。	2.0

选项	描述	默认
syn_interval =毫秒	等待的最小毫秒数 在关联期间的握手尝试之间。	250
syn_timeout =毫秒	等待的最大毫秒数 在放弃协会期间的握手回应之前。 默认值 是30秒。	30000
TTL =正	任何的生存时间（ttl）字段的值 数据报发送。 默认值1意味着所有的数据都被 限制在本地网络上。	1
async_send=*[0|1]*	使用异步I / O（在平台上）发送数据报 有效地支持它）。	

7.4.5.5 RTPS_UDP传输配置选项

OMG DDSI-RTPS（正式/ 2014-09-01）规范的OpenDDS实现包括满足规范要求所需的传输协议以及需要与其他DDS实现互操作的传输协议。 rtps_udp传输是可用于开发人员的可插入传输之一，并且是实现之间可互操作通信所必需的。 本节将讨论开发人员可用于配置OpenDDS以使用此传输的选项。

为了提供来自Section的单个配置示例的RTPS变体，下面的配置文件简单地将transport_type属性修改为值rtps_udp。 所有其他项目保持不变。

```
[通用] DCPSGlobalTransportConfig = myconfig

[config/myconfig]
transports=myrtps

[transport/myrtps]
transport_type=rtps_udp
local_address=myhost
```

为了将我们的例子扩展到如Section所示的混合传输配置，下面显示了使用与tcp传输混合的rtps_udp传输。 这个允许的有趣模式是一个已部署的OpenDDS应用程序，例如，可以使用tcp与其他OpenDDS参与者进行通信，同时使用rtps_udp与非OpenDDS参与者进行互操作性配置。

```
[通用] DCPSGlobalTransportConfig = myconfig

[配置/的myconfig]
输送= mytcp，myrtps

[传输/ myrtps]
```

```
transport_type=rtps_udp

[transport/mytcp]
transport_type=tcp
```

与使用rtps_udp传输协议相关的一些实现说明如下：

1) WRITER_DATA_LIFECYCLE（8.7.2.2.7）注意到相同的数据子消息应该处置和取消注册一个实例。 OpenDDS可能使用两个数据子消息。

2) RTPS传输实例不能由不同的域参与者共享。

3) 传输自动选择（协商）部分支持RTPS，
只有在可靠模式下，rtps_udp传输才会通过握手阶段。

表7-17 RTPS_UDP配置选项

选项	描述	默认
use_multicast=*[0|1]*	rtps_udp传输可以使用单播或多播。 当设置为0（false）时，传输使用单播，否则将使用值1（真） 多播。	1
multicast_group_address *=网络地址*	当传输设置为多播时，这是应该使用的多播网络地址。 如果没有为网络地址，端口指定端口7401将被使用。	239.255.0.2:7401
multicast_interface=*iface*	指定此传输实例使用的网络接口。 这使用标识网络接口的平台特定格式。 上 Linux系统，这将是类似eth0。	使用系统默认界面
local_address =地址：端口	将套接字绑定到给定的地址和端口。 端口可以省略，但尾随“：”是必需的。	系统默认
nak_depth =正	要保留的数据报的数量 服务修理请求（仅可靠）。	32
nak_response_delay =毫秒	协议调整参数，允许RTPS编写器将响应（以毫秒为单位）延迟到来自否定确认的数据请求。 （见OMG DDSI-RTPS表8.47） 规范）	200
heartbeat_period =毫秒	协议调整参数，以毫秒为单位指定RTPS Writer宣布数据可用性的频率。 （见OMG DDSI-RTPS表8.47） 规范）	1000

选项	描述	默认
heartbeat_response_delay= *毫秒*	协议调整参数（以毫秒为单位），允许RTPS阅读器延迟发送肯定或否定确认。 该参数用于减少网络风暴的发生。 （参见OMG DDSI-RTPS的第8.4.12.2节 规格（正式/ 2014-09-01））	500
handshake_timeout =毫秒	在放弃握手响应之前等待的最大毫秒数 协会。 默认值是30秒。	30000
TTL =正	任何发送的多播数据报的生存时间（ttl）字段的值。 此值指定数据报在被网络丢弃之前将经过的跳数。 默认值1意味着所有的数据都被限制在本地 网络子网。	1

7.4.5.6 共享内存传输配置选项

下表总结了shmem传输特有的传输配置选项。 这种传输类型支持带有POSIX / XSI共享内存的Unix类平台和Windows平台。 共享内存传输类型只能在同一主机上的传输实例之间提供通信。 作为运输谈判的一部分（见7.4.2.2），如果有多个传输实例可用于主机之间的通信，则将跳过共享内存传输实例，以便可以使用其他类型。

表7-18共享内存传输配置选项

选项	描述	默认
pool_size =字节	单个共享内存池的大小 分配。	16 MB
datalink_control_size =字节	为每个控制区分配的大小 数据链接。 这个分配来自由pool_size定义的共享内存池。	4 KB

7.5 记录

默认情况下，OpenDDS框架只会在有一个没有被返回码指示的严重错误时记录。 OpenDDS用户可以通过DCPS和传输层上的控件增加日志记录的数量。

7.5.1 DCPS层记录

登录OpenDDS的DCPS层由DCPSDebugLevel配置选项和命令行选项控制。 它也可以在应用程序代码中使用以下代码进行设置：

```
OpenDDS :: DCPS :: set_DCPS_debug_level (水平)
```

级别默认值为0，其值为0到10，如下定义：

- 0 - 表示严重错误的日志，不是由返回码（几乎没有）表示的。
- 1 - 每个进程应该发生一次的日志或警告
- 2 - 每个DDS实体应该发生一次的日志
- 4 - 与管理界面相关的日志
- 6 - 每N个样本写入/读取应该发生的日志
- 8 - 每个样本写入/读取应该发生一次的日志
- 10 - 每个样本写入/读取可能发生多次的日志

7.5.2 传输层记录

OpenDDS传输层日志记录通过DCPSTransportDebugLevel配置选项进行控制。 例如，要将传输层日志记录添加到任何OpenDDS应用程序，请将以下选项添加到命令行中：

```
-DCPSTransportDebugLevel级别
```

传输层日志记录级别还可以通过适当地设置变量以编程方式进行配置：

```
OpenDDS::DCPS::Transport_debug_level = level;
```

有效的运输日志记录级别从0到5，随着输出的详细程度的提高。

注意 *传输日志记录级别6可用于生成系统跟踪日志。 建议不要使用这个级别，因为生成的数据量可能会很大，而且大部分只对OpenDDS开发者感兴趣。 将日志记录级别设置为6需要将DDS_BLD_DEBUG_LEVEL宏定义为6并重建OpenDDS。*

C章节8

opendds_idl

选项

8.1 opendds_idl命令行选项

OpenDDS IDL编译器使用位于中的opendds_idl可执行文件调用

$ DDS_ROOT / bin /中。 它解析一个IDL文件，并生成序列化和关键支持代码，OpenDDS需要对IDL文件中描述的类型进行编组和解组，以及数据读者和作者的类型支持代码。 对

于每个处理的IDL文件，如xyz.idl，它会生成三个文件：xyzTypeSupport.idl，

xyzTypeSupportImpl.h和xyzTypeSupportImpl.cpp。 在典型的用法中，opendds_idl被传递了一些选项和IDL文件名作为参数。 例如，

```
$DDS_ROOT/bin/opendds_idl Foo.idl
```

下表总结了opendds_idl支持的选项。

表8-1 opendds_idl命令行选项

选项	描述	默认
-v	启用详细执行	安静的执行
-h	打印帮助（用法）消息并退出	N/A
-V	打印TAO和 OpenDDS	N/A
-Wb，export_macro =宏	导出用于生成C ++的宏 实现代码。	没有使用导出宏
-Wb，export_include =文件	生成的#include中的额外头文件 代码 - 这个头#定义了导出宏	没有额外的包括
-Wb，pch_include =文件	预编译的头文件包含在 生成的C ++文件	没有预编译头文件 包括
-dname [=值]	定义一个预处理宏	N/A
-Idir	将dir添加到预处理器包含路径	N/A
-o outputdir	输出目录opendds_idl应该 放置生成的文件。	当前目录
-wb，JAVA	启用OpenDDS Java绑定以生成 TypeSupport实现类	没有Java支持
-Gws	生成Wireshark解剖器配置 文件的形式<file> _ws.ini。	没有Wireshark解剖器 配置生成
-Gitl	生成中间型语言 Wireshark解析器使用的数据类型的描述。	没有生成
-GfaceTS	产生FACE（未来机载能力 环境）运输服务API	没有生成
-Lface	为FACE生成IDL-to-C ++映射	没有生成
-Lspcpp	为安全生成IDL-to-C ++映射 轮廓	没有生成
-Wb，tao_include_prefix = S	将字符串前缀为#include指令 意味着包含由tao_idl生成的头文件	N/A
-wb，V8	生成转换数据的类型支持 样本到v8 JavaScript对象	没有生成
-IC包括	将#include指令添加到生成的 .cpp文件。	没有添加
-St	禁止生成IDL TypeCodes -L选项中的任何一个都存在。	IDL TypeCodes 产生

代码生成选项允许应用程序开发人员在各种环境中使用生成的代码。 由于IDL可能包含预处理指令（#include，
#define等），C ++预处理器由opendds_idl调用。 -I和-D选项允许自定义预处理步骤。 -Wb，export_macro选项（将导出作为别名接受）允许您将导出宏添加到类定义中。 如果生成的代码要驻留在共享库中，并且编译器（例如Visual C ++或GCC）使用导出宏（Visual C ++上的dllexport /覆盖GCC上的隐藏可见性），则这是必需的。 如果在使用预编译头的组件中使用生成的实现代码，则需要-Wb，pch_include选项。

C章节9

DCPS

信息库

9.1 DCPS信息库选项

下表显示了DCPSInfoRepo服务器的命令行选项：

表9-1 DCPS信息库选项

选项	描述	默认
-o文件	将DCPSInfo对象的IOR写入指定的文件	*repo.ior*
-NOBITS	禁用发布内置主题	内置的主题是发表
- 一个地址	监听内置主题的地址（内置时）主题发表）。	随机端口
-z	打开详细的传输日志记录	最小的运输日志记录。
-r	从永久文件中恢复	1（真）
-FederationId <id>	任何一个存储库的唯一标识符联邦。 这是以32位十进制数值的形式提供的。	N/A
-FederateWith <ref>	加入联合的存储库联合参考。 这是以字符串形式提供的一个有效的CORBA对象引用：stringified IOR，file：或corbaloc：参考字符串。	N/A
-?	显示命令行用法并退出	N/A

OpenDDS客户端通常使用DCPSInfoRepo输出的IOR文件来定位服务。 该-o选项允许您将IOR文件放入特定于应用程序的目录或文件名。 客户端可以使用file：//IOR前缀来使用该文件。

不使用内置主题的应用程序可能需要使用-NOBITS来禁用它们，以减少服务器上的负载。 如果您正在发布内置主题，则使用-a选项可以选择用于这些主题的tcp传输的监听地址。

使用-z选项会导致调用许多传输级别的调试消息。 此选项仅在使用定义的DCPS_TRANS_VERBOSE_DEBUG环境变量构建DCPS库时才有效。

-FederationId和-FederateWith选项用于控制将多个DCPSInfoRepo服务器联合成一个逻辑存储库。 看到9.2 了解联邦功能的说明以及如何使用这些选项。

文件持久性作为一个ACE服务对象来实现，并通过服务配置指令来控制。 目前可用的配置选项有：

表9-2 InfoRepo持久性指令

选项	描述	默认
-文件	持久文件的名称	*InforepoPersist*
-重启	擦除旧的持久性数据。	0（假）

以下指令：

静态PersistenceUpdater_Static_Service"-file info.pr -reset 1"

将持续DCPSInfoRepo更新到本地文件info.pr. 如果这个名字的文件已经存在，它的内容将被删除。 与命令行选项-r一起使用，DCPSInfoRepo可以转化为先前的状态。 使用持久性时，使用TCP固定端口号，使用以下命令行选项启动DCPSInfoRepo进程。 这允许现有客户端重新连接到重新启动的InfoRepo。

-ORBListenEndpoints iiop: //: <port>

9.2 存储库联合

注意 *存储库联合应该被认为是一个实验性的功能。*

存储库联合允许多个DCPS信息存储库服务器相互协作为一个联合服务。 这允许应用程序从一个存储库获取服务元数据和事件，如果原始存储库不再可用，则从另一个存储库获取它们。

虽然创建此功能的动机是能够提供对DDS服务元数据的容错度量，但其他用例也可以从此功能中受益。 这包括初始分离系统成为联合的能力，并获得在原本不可达的应用程序之间传递数据的能力。 这方面的一个例子包括两个独立建立内部DDS服务的平台在应用程序之间传递数据；在操作期间的某个时刻，系统彼此之间可达，并且联合存储库允许数据在不同平台上的应用程序之间传递。

OpenDDS当前的联合功能只能在启动应用程序和存储库时静态指定存储库联合。 计划在未来的OpenDDS发行版中动态发现和加入联盟。

OpenDDS通过在内置主题上使用LIVELINESS服务质量策略来自动检测存储库的丢失情况。当使用联合时，LIVELINESS QoS策略被修改为非无限值。 当LIVELINESS对于内置主题丢失时，应用程序将启动故障转移序列，使其与另一个存储库服务器相关联。
由于联合实现当前使用内置主题ParticipantDataDataReaderListener实体，应用程序不应该为此主题安装自己的侦听器。 这样做会影响联合实施检测存储库故障的能力。

联合实现使用保留的DDS域在联合中分发存储库数据。 用于联合的默认域由常量Federator :: DEFAULT_FEDERATIONDOMAIN定义。

目前，只有联合拓扑的静态规范是可用的。 这意味着每个DCPS信息库以及使用联合DDS服务的每个应用程序都需要包含联合配置作为其配置数据的一部分。 这是通过在每个参与进程中指定联合中的每个可用存储库并将每个存储库分配给配置文件中的不同键值来完成的，如Section7.3.2.1.

每个应用程序和存储库必须在其配置信息中包含相同的一组存储库。 故障转移排序将尝试以数字顺序到达存储库键值的下一个存储库（从最后一个换行到第一个）。 这个序列对于每个配置的应用程序是唯一的，并且应该是不同的，以避免重载任何单独的存储库

一旦指定了拓扑信息，则需要使用两个额外的命令行参数来启动存储库。 这些显示在表9-1。 一，
-FederationId <值>，指定联合中的存储库的唯一标识符。 这是一个32位的数字值，对于所有可能的联邦拓扑来说都是唯一的。

所需的第二个命令行参数是-FederateWith <ref>。 这会导致存储库在初始化之后并接受来自应用程序的连接之前，在<ref>对象引用中加入联合。

只有以联合身份识别号码开始的储存库才能参与联合。 第一个存储库启动时不应该使用-FederateWith命令行指令。 所有其他人都必须有此指令才能建立初始联合。 有一个命令行工具（联合）提供，可用于建立联邦关联，如果这不是在启动时完成的。 参见章节9.2.1 为了说明。 在当前的纯静态实现中，有可能在完全建立联合拓扑之前资源库的故障可能导致部分不可用的服务。 由于当前的局限性，强烈建议在启动应用程序之前始终建立存储库的联合拓扑。

9.2.1 联邦管理

已经提供了一个新的命令行工具来允许对联邦存储库进行一些最小的运行时管理。 此工具允许在没有-FederateWith选项的情况下启动存储库，以便参与联合。 由于联合存储库的操作和故障转移顺序取决于连接拓扑的存在，

建议在启动将使用联合存储库集的应用程序之前使用此工具。

该命令名为repoctl，位于$ DDS_ROOT / bin /目录中。 它有一个命令格式的语法：

```
repoctl <cmd> <arguments>
```

每个单独的命令都有自己的格式，如图所示表9-3。 一些选项包含端点信息。 这个信息由一个可选的主机规范组成，与一个冒号所需的端口规范分开。 此端点信息用于使用corbaloc：语法创建CORBA对象引用，以便找到存储库服务器的“Federator”对象。

表9-3 repoctl存储库管理命令

命令	句法	描述
加入	repoctl join <target> <peer> [<federation domain>]	调用<peer>将<target>加入联合。 如果存在，则传递<federation domain>或传递默认的“联合身份验证域”值。
离开	回去离开<target>	使<target>正常离开联邦，删除使用<target>的应用程序之间的所有托管关联作为存储库，而不是使用的应用程序 作为存储库的<target>。
关掉	repoctl shutdown <target>	导致关闭<target>而不删除任何托管关联。 这是一样的效果 作为在操作中已经崩溃的存储库。
杀	repoctl kill <target>	杀死<target>存储库而不管它 联邦地位。
帮帮我	repoctl帮助	打印使用信息并退出。

连接命令指定两个存储服务器（按端点），并要求第二个加入联合中的第一个：

```
repoctl join 2112 otherhost:1812
```

这将生成一个corbaloc :: otherhost：1812 / Federator的CORBA对象引用，联邦者连接并调用联合操作。 连接操作调用通过解析对象引用corbaloc :: localhost：2112 / Federator来传递默认的Federation Domain值（因为我们没有指定一个）和连接的存储库的位置。

命令参数的完整说明显示在表9-4.

表9-4联合管理命令参数

选项	描述
<目标>	这是可用于查找用于管理联合身份验证行为的存储库的Federator :: Manager CORBA接口的端点信息。 这用于命令离开和关闭联邦操作，并确定连接的连接存储库 命令。
<对等>	这是可用于查找用于管理联合身份验证行为的存储库的Federator :: Manager CORBA接口的端点信息。 这用于命令联合联邦 操作。
<联合域>	这是联盟参与者用于在联合存储库之间分发服务元数据的域规范。 这只需要在同一组存储库中存在多个联合时指定，目前不支持。 默认域是 足够单个联合会。

9.2.2 联合示例

为了说明联邦的设置和使用，本节将介绍一个简单的示例，该示例将建立一个联合和使用它的工作服务。

这个例子是基于一个两个版本库联合，其中简单的消息发布者和订阅者来自2.1 配置为使用联合存储库。

9.2.2.1 配置联合示例

有两个配置文件为这个例子创建一个消息发布者和订阅者。

此示例的消息发布者配置pub.ini如下所示：

```
[common]
DCPSDebugLevel=0

[domain/information]
DomainId=42
DomainRepoKey=1

[repository/primary]
RepositoryKey=1
RepositoryIor = corbaloc ::本地主机: 2112 / InfoRepo

[repository / secondary] RepositoryKey = 2
RepositoryIor = file: //repo.ior
```

请注意，DCPSInfo属性/值对已从[common]部分中省略。 这已被7.5中描述的[domain / user]部分所取代。 用户域

是42，所以域配置为使用服务元数据和事件的主要存储库。

[repository / primary]和[repository / secondary]部分定义了用于该应用程序的联合（两个存储库）中使用的主要和次要存储库。 RepositoryKey属性是一个内部键值，用于唯一标识存储库（并允许域与之关联，如前所述
[域名/信息]部分）。 RepositoryIor属性包含到达指定存储库的可解析对象引用的字符串值。主要存储库在本地主机的端口2112处引用，预期可通过TAO IORTable以对象名称/ InfoRepo的形式提供。 预计二级存储库
通过本地目录中名为repo.ior的文件提供一个IOR值。 用户进程使用

sub.ini文件进行配置，如下所示：

```
[common]
DCPSDebugLevel=0

[domain/information]
DomainId=42
DomainRepoKey=1

[repository / primary] RepositoryKey = 1
RepositoryIor = file: //repo.ior

[repository/secondary]
RepositoryKey=2
RepositoryIor = corbaloc ::本地主机: 2112 / InfoRepo
```

请注意，这与pub.ini文件相同，除了订户已指定位于本地主机的端口2112上的存储库是辅助存储库，并且由repo.ior文件定位的存储库是主存储库。 这与发布者的分配相反。这意味着当订阅者开始使用位于文件中包含的IOR的存储库时，发布者开始使用端口2112处的存储库来获取元数据和事件。 在每种情况下，如果存储库被检测为不可用，则应用程序将尝试使用其他存储库（如果可以到达）。

这个存储库不需要任何特殊的配置规范来参与联合，所以在这个例子中不需要文件。

9.2.2.2 运行联合示例

该示例是通过首先启动存储库并联合它们来执行的，然后启动应用程序发布者和订阅者处理的过程与Section2.1.7.

启动第一个存储库为：

```
$DDS/bin/DCPSInfoRepo -ORBSvcConf tcp.conf -o repo.ior -FederationId 1024
```

-o repo.ior选项确保存储库IOR将按照配置文件的预期放置到文件中。 -FederationId 1024选项将值1024分配给此存储库作为联合中的唯一标识。 -ORBSvcConf tcp.conf选项与上例中的相同。

启动第二个存储库：

```
$DDS/bin/DCPSInfoRepo -ORBSvcConf tcp.conf \
  -ORBListenEndpoints iiop: // localhost: 2112 \
  -FederationId 2048 -FederateWith file: //repo.ior
```

请注意，这是所有打算在一个命令行上。 -ORBSvcConf tcp.conf选项与上例中的相同。-ORBListenEndpoints iiop: // localhost: 2112选项可确保存储库将监听以前的配置文件所期望的端口。 -FederationId 2048选项将值2048分配为联合中的存储库唯一标识。-FederateWith file: //repo.ior选项启动与位于指定文件中的IOR中的存储库的联合，该文件由以前启动的存储库写入。

一旦存储库已经启动并建立了联合（这将在第二个存储库初始化后自动完成），那么应用程序发布者和订阅者进程就可以启动，并且应该像在前面的例子中那样执行2.1.7.

C章节10

Java绑定

10.1 介绍

OpenDDS提供Java JNI绑定。 Java应用程序可以像C ++应用程序一样使用完整的OpenDDS中间件。

有关入门信息，请参阅$ DDS_ROOT / java / INSTALL文件，其中包括先决条件和依赖关系。

有关使用Java绑定开发应用程序时遇到的常见问题，请参阅$ DDS_ROOT / java / FAQ文件。

10.2 IDL和代码生成

OpenDDS Java绑定不仅仅是一个存在于一个或两个.jar文件中的库。 DDS规范定义了DDS应用程序和DDS中间件之间的交互。 特别是，DDS应用程序发送和接收强类型的消息，这些类型是由应用程序开发人员在IDL中定义的。

为了让应用程序根据这些用户定义类型与中间件进行交互，必须在编译时根据此IDL生成代码。 C ++，Java，甚至还有一些额外的IDL代码被生成。 在大多数情况下，应用程序开发人员不需要关心所有生成的文件的细节。 OpenDDS附带的脚本会自动执行此过程，以便最终结果是一个本地库（.so或.dll）和一个Java库（.jar或只是一个类目录），它们一起包含所有生成的代码。

下面是生成的文件和哪些工具生成它们的描述。 在这个例子中，Foo.idl包含模块Baz中包含的单个结构体Bar（IDL模块类似于C ++命名空间和Java包）。 在每个文件名的右边是产生它的工具的名字，后面跟着它的用途。

表10-1生成的文件说明

文件	生成工具
Foo.idl	开发人员对DDS样本类型的描述
FOO {C, S}。{H, INL, CPP}	tao_idl：IDL的C ++表示
FooTypeSupport.idl	opendds_idl：DDS类型特定的接口
FooTypeSupport {C, S}。{H, INL, CPP}	tao_idl
巴兹/ BarSeq {帮助器，架构.java}	idl2jni
*巴兹/ BarData {读者，作家} *。java的*	idl2jni
*巴兹/ BarTypeSupport *的.java*	idl2jni（TypeSupportImpl除外，见下文）
FooTypeSupportJC。{H, CPP}	idl2jni：JNI本地方法实现
FooTypeSupportImpl。{H, CPP}	opendds_idl：DDS类型特定的C ++ impl。
巴兹/ BarTypeSupportImpl.java	opendds_idl：DDS类型特定的Java impl。
*巴兹/酒吧*的.java*	idl2jni：IDL结构的Java表示
FooJC。{H, CPP}	idl2jni：JNI本地方法实现

Foo.idl：

```
module Baz {
#pragma DCPS_DATA_TYPE"Baz :: Bar"struct Bar
  {
    长x
  };
};
```

10.3 建立一个OpenDDS Java项目

这些说明假定您已经完成了安装步骤
$ DDS_ROOT / java / INSTALL文件，包括定义各种环境变量。

1) 从一个空的目录开始，这个目录将被用于你的IDL和从它生成的代码。
$ DDS_ROOT / java / tests / messenger / messenger_idl /就是这样设置的。

2) 创建一个IDL文件，描述您将与OpenDDS一起使用的数据结构。 有关示例，请参阅Messenger.idl。 该文件将至少包含一行以“#pragma DCPS_DATA_TYPE”开头的行。 为了这些说明，我们将调用文件Foo.idl。

3) C ++生成的类将被打包在一个共享库中，以便在运行时由JVM加载。 这要求将打包的类导出为外部可见性。 ACE提供了用于生成正确的导出宏的实用程序脚本。脚本用法如下所示：

Unix的：

```
$ACE_ROOT/bin/generate_export_file.pl Foo > Foo_Export.h
```

视窗：

```
%ACE_ROOT%\ bin \ generate_export_file.pl Foo> Foo_Export.h
```

4) 从这个模板创建一个MPC文件Foo.mpc：

```
    项目：dcps_java {

  idlflags       + = -Wb, stub_export_include = Foo_Export.h \
                     -Wb, stub_export_macro = Foo_Export
  dcps_ts_flags + = -Wb, export_macro = Foo_Export idl2jniflags + =
  -Wb, stub_export_include = Foo_Export.h \
                     -Wb, stub_export_macro = Foo_Export
  dynamicflags + = FOO_BUILD_DLL

     具体{
    的JARname       = DDS_Foo_types
  }

  TypeSupport_Files
    { Foo.idl
    }
 }
```

如果你不需要创建一个jar文件，你可以省略具体的{...}块。 在这种情况下，您可以直接使用将在当前目录的classes子目录下生成的Java .class文件。

5) 运行MPC以生成特定于平台的构建文件。

Unix的：

```
$ACE_ROOT/bin/mwc.pl -type gnuace
```

视窗：

```
%ACE_ROOT%\ bin \ mwc.pl -type [CompilerType]
```

CompilerType可以是vc71，vc8，vc9，vc10和nmake

确定这是运行的ActiveState Perl。

6）编译生成的C ++和Java代码Unix：

```
make (GNU make，所以这可能是Solaris系统上的"gmake")
```

视窗：

使用您的首选方法构建生成的.sln（解决方案）文件。 这可以是Visual Studio IDE或其中一种命令行工具。 如果使用IDE，请使用devenv或vcexpress（Express Edition）从命令提示符启动它，以便继承环境变量。 用于构建的命令行工具包括vcbuild和使用适当的参数调用IDE（devenv或vcexpress）。

成功完成后，您将拥有一个本机库和一个Java .jar文件。 本地库名称如下所示：

Unix：

```
libFoo.so
```

Windows：

```
Foo.dll (Release) 或者Food.dll (调试)
```

您可以通过向Foo.mpc文件添加以下行来更改这些库（包括.jar文件）的位置：

```
libout = $ (PROJECT_ROOT) / lib
```

其中PROJECT_ROOT可以是在构建时定义的任何环境变量。

7）您现在拥有编译和运行Java OpenDDS应用程序所需的所有Java和C ++代码。 生成的.jar文件需要添加到你的类路径中。 生成的C ++库需要在运行时加载：

Unix的：

将包含libFoo.so的目录添加到LD_LIBRARY_PATH。 视窗：

将包含Foo.dll（或Food.dll）的目录添加到PATH。 如果您正在使用调试版本（Food.dll），则需要通知OpenDDS中间件不应该查找Foo.dll。 为此，请在Java VM参数中添加-Dopendds.native.debug = 1。

请参阅$ DDS_ROOT / java / tests / messenger /中的发布者和订阅者目录，了解使用OpenDDS Java绑定发布和订阅应用程序的示例。

8）如果您对Foo.idl进行了后续更改，请重新运行MPC（上述步骤#5）。 这是必需的，因为对Foo.idl的某些更改将会影响生成哪些文件并需要进行编译。

10.4 简单的消息发布者

本节介绍一个简单的OpenDDS Java发布过程。 完整的代码可以在$ DDS_ROOT / java / tests / messenger / publisher / TestPublisher.java中找到。
无用的细分，如导入和错误处理在这里被省略了。 代码已经被分解并在逻辑小节中解释。

10.4.1 初始化参与者

DDS应用程序通过获取参与者工厂的初始引用来启动。 对静态方法TheParticipantFactory.WithArgs（）的调用返回一个Factory引用。 这也透明地初始化C ++参与者工厂。 然后，我们可以为特定的域创建参与者。

```
public static void main (String [] args)

    {DomainParticipantFactory dpf =
        TheParticipantFactory.WithArgs (new StringSeqHolder (args) ) ;  if
     (dpf == null) {
       System.err.println ("找不到域参与者工厂") ; 返回;
    }
    final int DOMAIN_ID = 42;
    DomainParticipant dp = dpf.create_participant (DOMAIN_ID,
       PARTICIPANT_QOS_DEFAULT.get (), null, DEFAULT_STATUS_MASK.value) ;
    if (dp == null) {
       System.err.println ("域参与者创建失败") ; 返回;
    }
```

对象创建失败由空返回表示。 create_participant () 的第三个参数需要一个Participant事件侦听器。 如果其中一个不可用，则可以像我们的例子中那样传递null。

10.4.2 注册数据类型和创建主题

接下来，我们使用register_type () 操作向DomainParticipant注册数据类型。 我们可以指定一个类型名称或传递一个空字符串。 传递一个空字符串表示中间件应该简单地使用由IDL编译器为该类型生成的标识符。

```
MessageTypeSupportImpl servant = new MessageTypeSupportImpl () ;  if
 (servant.register_type (dp, "") ! = RETCODE_OK.value) {
   System.err.println ("register_type failed") ; 返回;
}
```

接下来，我们使用类型支持服务器的注册名称创建一个主题。

```
主题top = dp.create_topic ("电影讨论列表",
                                servant.get_type_name () ,
                                TOPIC_QOS_DEFAULT.get () , null,
                                DEFAULT_STATUS_MASK.value) ;
```

现在我们有一个名为"电影讨论列表"的主题，注册数据类型和默认QoS策略。

10.4.3 创建一个发布者

接下来，我们创建一个发布者：

```
Publisher pub = dp.create_publisher
    (PUBLISHER_QOS_DEFAULT.get () ,
   null,
   DEFAULT_STATUS_MASK.value) ;
```

10.4.4 创建一个DataWriter并注册一个实例

通过发布者，我们现在可以创建一个DataWriter：

```
DataWriter dw = pub.create_datawriter (
   顶部, DATAWRITER_QOS_DEFAULT.get () , null, DEFAULT_STATUS_MASK.value) ;
```

DataWriter是针对特定主题的。 对于我们的例子，我们使用默认的DataWriter QoS策略和一个空的DataWriterListener。

接下来，我们将泛型DataWriter缩小到特定于类型的DataWriter，并注册我们希望发布的实例。 在我们的数据定义IDL中，我们指定了subject_id字段作为键，所以它需要填充实例id（在我们的例子中是99）：

```
MessageDataWriter mdw = MessageDataWriterHelper.narrow (dw) ; 消息msg =
新消息 () ;
msg.subject_id = 99;
int handle = mdw.register (msg) ;
```

我们的例子等待任何同龄人被初始化和连接。 然后发布一些消息，这些消息分发给同一个域中的这个主题的任何用户。

```
msg.from =“OpenDDS-Java”;
msg.subject =“评论”;
msg.text =“最糟糕的电影永远” msg.count
= 0;
int ret = mdw.write (msg, handle) ;
```

10.5 设置订阅者

用户的大部分初始化代码与发布者完全相同。 用户需要在同一个域中创建一个参与者，注册一个相同的数据类型，并创建相同的命名主题。

```
public static void main (String [] args)

    {DomainParticipantFactory dpf =
        TheParticipantFactory.WithArgs (new StringSeqHolder (args) ) ;  if
     (dpf == null) {
       System.err.println (“找不到域参与者工厂”) ; 返回;
     }
     DomainParticipant dp = dpf.create_participant (42, PARTICIPANT_QOS_DEFAULT.get
        () , null, DEFAULT_STATUS_MASK.value) ;
     if (dp == null) {
       System.err.println (“域参与者创建失败”) ; 返回;
     }

     MessageTypeSupportImpl servant = new MessageTypeSupportImpl () ;  if
      (servant.register_type (dp, “”) ! = RETCODE_OK.value) {
       System.err.println (“register_type failed”) ; 返回;
     }
     主题top = dp.create_topic (“电影讨论列表”,
                                    servant.get_type_name () ,
                                    TOPIC_QOS_DEFAULT.get () , null,
                                    DEFAULT_STATUS_MASK.value) ;
```

10.5.1 创建一个订户

与出版商一样，我们创建订阅者：

```
Subscriber sub = dp.create_subscriber (SUBSCRIBER_QOS_DEFAULT.get () , null,
  DEFAULT_STATUS_MASK.value) ;
```

10.5.2 创建一个DataReader和监听器

向中间件提供DataReaderListener是通知接收数据和访问数据的最简单方法。 因此，我们创建一个DataReaderListenerImpl的实例，并将其作为DataReader创建参数传递给它：

```
DataReaderListenerImpl listener = new DataReaderListenerImpl () ;  DataReader dr
 = sub.create_datareader (
    顶部, DATAREADER_QOS_DEFAULT.get () , 侦听器,
    DEFAULT_STATUS_MASK.value) ;
```

任何传入的消息将由中间件线程中的监听器接收。 应用程序线程此时可以自由执行其他任务。

10.6 DataReader监听器履行

应用程序定义的DataReaderListenerImpl需要实现规范的DDS.DataReaderListener接口。OpenDDS提供了一个抽象类DDS._DataReaderListenerLocalBase。 应用程序的侦听器类扩展了这个抽象类，并实现了抽象方法来添加特定于应用程序的功能。

我们的DataReaderListener示例删除了大部分Listener方法。 唯一实现的方法是从中间件可用的消息回调：

```
公共类DataReaderListenerImpl扩展DDS._DataReaderListenerLocalBase {

    private int num_reads_;

    public synchronized void on_data_available (DDS.DataReader reader) {
        ++num_reads_;
        MessageDataReader mdr = MessageDataReaderHelper.narrow (reader) ;  if (mdr
        == null) {
          System.err.println (“read: narrow failed。”) ; 返回;
        }
```

Listener回调被传递给一个通用的DataReader的引用。 应用程序将其缩小为特定于类型的DataReader：

```
MessageHolder mh = new MessageHolder (new Message () ) ;  SampleInfoHolder sih =
new SampleInfoHolder (new SampleInfo (0, 0, 0,
     新的DDS.Time_t () , 0, 0, 0, 0, 0, false) ) ;  int
status = mdr.take_next_sample (mh, sih) ;
```

然后它为实际的消息和关联的SampleInfo创建持有者对象，并从DataReader中获取下一个样本。 一旦拿出，该样本将从DataReader的可用样本池中删除。

```
        if (status == RETCODE_OK.value) {

          System.out.println ("SampleInfo.sample_rank ="+ sih.value.sample_rank) ;  System.out.println
           ("SampleInfo.instance_state ="+
                                  sih.value.instance_state) ;  if

           (sih.value.valid_data) {

            System.out.println ("Message: subject           ="+ mh.value.subject) ;
            System.out.println ("              subject_id ="+ mh.value.subject_id) ;
            System.out.println ("              从          ="+ mh.value.from) ;
            System.out.println ("              计数        ="+ mh.value.count) ;
            System.out.println ("              文本        ="+ mh.value.text) ;
            System.out.println ("SampleInfo.sample_rank ="+
                                  sih.value.sample_rank) ;
          }
          否则如果 (sih.value.instance_state ==
                       NOT_ALIVE_DISPOSED_INSTANCE_STATE.value) {
            System.out.println ("实例处置") ;
          }
          否则如果 (sih.value.instance_state ==
                       NOT_ALIVE_NO_WRITERS_INSTANCE_STATE.value) {
            System.out.println ("实例未注册") ;
          }
          else {
            System.out.println ("DataReaderListenerImpl :: on_data_available: "+
                                  "收到未知实例状态"+
                                  sih.value.instance_state) ;
          }

        } else if (status == RETCODE_NO_DATA.value) {
          System.err.println ("ERROR: reader received DDS :: RETCODE_NO_DATA! ") ;
        } else {
          System.err.println ("错误: 读取消息: 错误: "+状态) ;
        }
    }

}
```

SampleInfo包含有关消息的元信息，如消息有效性，实例状态等。

10.7 清理OpenDDS Java客户端

应用程序应通过以下步骤清理其OpenDDS环境：

```
        dp.delete_contained_entities () ;
```

清理与该参与者相关的所有主题，订阅者和发布者。

```
dpf.delete_participant (DP) ;
```


DomainParticipantFactory回收与该关联的所有资源
DomainParticipant。

```
TheServiceParticipant.shutdown ();
```

关闭ServiceParticipant。 这清理了所有OpenDDS关联的资源。 清理这些资源是必要的，以防止DCPSInfoRepo在不再存在的端点之间形成关联。

10.8 配置示例

OpenDDS提供了一个基于文件的配置机制。 配置文件的语法与Windows INI文件类似。 这些属性被分成对应于公共和个别传输配置的命名区域。

Messenger示例具有DCPSInfoRepo对象位置和全局传输配置的公共属性：

```
[common] DCPSInfoRepo = file:
//repo.ior DCPSGlobalTransportConfig =
$ file
```

以及具有传输类型属性的传输实例部分：

```
[transport/1]
transport_type=tcp
```

[transport / 1]部分包含名为"1"的传输实例的配置信息。 它被定义为tcp类型。 上面的全局传输配置设置会导致此传输实例被所有读者和作者使用。

见章节7 了解所有OpenDDS配置参数的完整描述。

10.9 运行示例

要运行Messenger Java OpenDDS应用程序，请使用以下命令：

```
$DDS_ROOT/bin/DCPSInfoRepo -o repo.ior

$ JAVA_HOME / bin / java -ea -cp类: $ DDS_ROOT / lib / i2jrt.jar:
$ DDS_ROOT / lib / OpenDDS_DCPS.jar: 类TestPublisher -DCPSConfigFile pub_tcp.ini

$ JAVA_HOME / bin / java -ea -cp类: $ DDS_ROOT / lib / i2jrt.jar:
$ DDS_ROOT / lib / OpenDDS_DCPS.jar: classes TestSubscriber -DCPSConfigFile sub_tcp.ini
```

-DCPSConfigFile命令行参数传递OpenDDS配置文件的位置。

10.10 Java消息服务（JMS）支持

OpenDDS为JMS版本1.1提供部分支持 <http://docs.oracle.com/javaee/6/tutorial/doc/bncdq.html>。 企业Java应用程序可以像标准的Java和C ++应用程序一样使用完整的OpenDDS中间件。

请参阅$ DDS_ROOT / java / jms /目录中的INSTALL文件，以获取有关OpenDDS JMS支持入门的信息，包括先决条件和依赖关系。

C章节11

建模SDK

OpenDDS Modeling SDK是一个建模工具，可供应用程序开发人员使用，将所需的中间件组件和数据结构定义为UML模型，然后使用OpenDDS生成实现该模型的代码。 生成的代码然后可以编译并与应用程序链接，为应用程序提供无缝的中间件支持。

11.1 概观

11.1.1 模型捕获

使用Eclipse插件中包含的图形模型捕获编辑器捕获定义DCPS元素和策略以及数据定义的UML模型。 UML模型的元素遵循DDS规范（OMG: formal / 2015-04-10）中定义的DDS UML平台无关模型（PIM）的结构。

在插件中打开一个新的OpenDDS模型从一个顶层主图开始。 该图包括模型中要包含的任何软件包结构以及模型的本地QoS策略定义，数据定义和DCPS元素。 可以包含零个或多个策略或数据定义元素。 任何给定的模型都可以包含零个或一个DCPS元素定义。

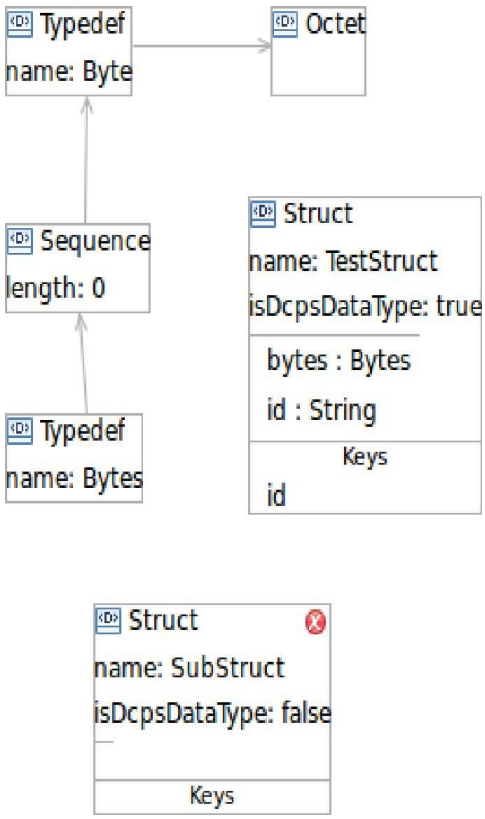


图11-1数据定义的图形建模

仅为QoS策略创建单独的模型，仅支持数据定义或仅支持DCPS元素。 对其他模型的引用允许将外部定义的模型包含在模型中。 这允许在不同的DCPS模型之间共享数据定义和QoS策略，并且将外部定义的数据包括在一组新的数据定义中。

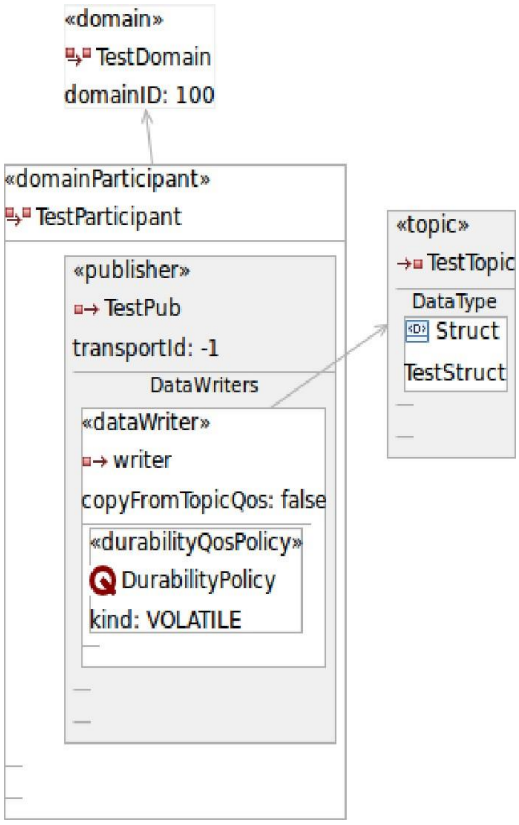


图11-2 DCPS实体的图形建模

11.1.2 代码生成

一旦模型被捕获，源代码可以从它们生成。 然后可以将该源代码编译到链接库中，以便将模型中定义的中间件元素提供给链接库的应用程序。 代码生成是使用单独的基于表单的编辑器完成的。

代码生成的细节对于单独的生成表单是独一无二的，并且与生成代码的模型保持分离。代码生成一次在单个模型上执行，包括定制生成的代码以及在构建时指定用于查找资源的搜索路径。

可以生成模型变体（相同模型的不同定制），然后可以在相同的应用程序或不同的应用程序中创建模型变体。 也可以在构建时指定位置来搜索头文件和链接库。

见部分11.3.2，“生成的代码”的细节。

11.1.3 程序设计

为了使用由模型定义的中间件，应用程序需要链接到生成的代码中。 这是通过头文件和链接库完成的。 支持使用MPC可移植构建工具构建应用程序包含在为模型生成的文件中。

见部分11.3，“开发应用程序”的细节。

11.2 安装和入门

与由开发人员从源代码编译的OpenDDS中间件不同，已编译的Modeling SDK可通过Eclipse更新站点下载。

11.2.1 先决条件

- Java运行时环境（JRE）
  - 版本6更新24（1.6.0_24）在撰写本文时是最新的
  - 从下载*http://www.java.com*
- Eclipse IDE
  - 4.4版本“Luna”，4.4.1在撰写本文时是最新的
  - 从下载 *https://eclipse.org/downloads/packages/release/Luna/SR1A*

11.2.2 安装

1）从Eclipse中，打开“帮助”菜单并选择“安装新软件”。

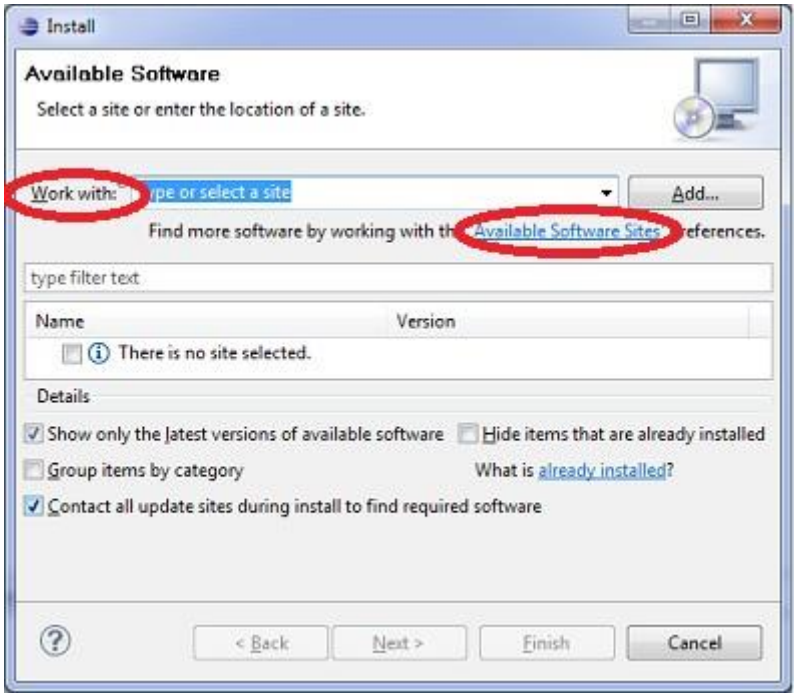


图11-3 Eclipse软件安装对话框

9) 单击可用软件站点的超链接。

10) 应该启用标准的eclipse.org网站（Eclipse Project Updates和Galileo）。 如果他们被禁用，现在启用它们。

11) 使用URL添加名为OpenDDS的新网站条目 *http://www.opendds.org/modeling/eclipse_44*

12) 单击确定关闭首选项对话框并返回到安装对话框。

13) 在“使用”组合框中，选择OpenDDS的新条目。

14) 选择“OpenDDS建模SDK”，然后单击下一步。

15) 查看“安装详细信息”列表，然后单击下一步。 检查许可证，选择接受（如果您接受），然后单击完成。

16) Eclipse将从eclipse.org下载OpenDDS插件和各种依赖的插件。 将会有一个安全警告，因为OpenDDS插件没有签名。 也可能会提示接受来自eclipse.org的证书。

17) Eclipse会提示用户重新启动以使用新安装的软件。

11.2.3 入门

OpenDDS建模SDK包含一个Eclipse透视图。 打开窗口菜单并选择Open Perspective - > Other - > OpenDDS Modeling。

要开始使用OpenDDS Modeling SDK，请参阅Eclipse中安装的帮助内容。 首先进入“帮助”菜单并选择“帮助内容”。 “OpenDDS建模SDK指南”有一个顶级项目，其中包含所有描述建模和代码生成活动的OpenDDS特定内容。

11.3 开发应用程序

为了使用OpenDDS Modeling SDK构建应用程序，必须了解几个关键概念。 这些概念涉及：

1） 支持库
2） 生成的模型代码
3） 应用程序代码

11.3.1 建模支持库

OpenDDS建模SDK包含一个支持库，可以在这里找到
$ DDS_ROOT /工具/模型/代码生成/模型。 这个支持库与Modeling SDK生成的代码结合在一起，大大减少了构建OpenDDS应用程序所需的代码量。

支持库是由OpenDDS Modeling SDK应用程序使用的C ++库。 大多数开发人员需要支持库中的两个类是Application和Service类。

11.3.1.1 应用程序类

OpenDDS :: Model :: Application类负责OpenDDS库的初始化和定型。 使用OpenDDS的任何应用程序都需要实例化一个Application类的单个实例，而且在使用OpenDDS进行通信时不会破坏Application对象。

Application类初始化用于创建OpenDDS参与者的工厂。 这个工厂需要用户提供的命令行参数。 为了提供它们，Application对象必须提供相同的命令行参数。

11.3.1.2 服务类

OpenDDS :: Model :: Service类负责在OpenDDS Modeling SDK模型中描述的OpenDDS实体的创建。 由于该模型可以是通用的，描述

一个比单个应用程序使用的范围更广的领域，Service类使用延迟实例化来创建OpenDDS实体。

为了正确实例化这些实体，它必须知道：

- 实体之间的关系
- 实体使用的传输配置

11.3.2 生成的代码

OpenDDS Modeling SDK生成模型特定的代码，供OpenDDS Modeling SDK应用程序使用。从一个.codegen文件（它指向一个.opendds模型文件）开始，文件中描述表11-1。Eclipse帮助中记录了生成代码的过程。

表11-1生成的文件

文件名	描述
<MODELNAME>的.idl	数据类型来自模型的DataLib
<ModelName>_T.h	来自模型的DcpsLib的C ++类
<ModelName>_T.cpp	模型的DcpsLib的C ++实现
<ModelName>.mpc	用于生成的C ++库的MPC项目文件
<ModelName>.mpb	MPC基础项目供应用程序使用
<ModelName>_paths.mpb	具有路径的MPC基础项目，请参见部分11.3.3.7
<ModelName>Traits.h	.codegen文件中的传输配置
<ModelName>Traits.cpp	.codegen文件中的传输配置

11.3.2.1 DCPS模型类

DCPS库建模DDS实体之间的关系，包括主题，DomainParticipants，Publishers，订阅者，DataWriters和数据读取器及其相应的域。

对于模型中的每个DCPS库，OpenDDS Modeling SDK将生成一个以DCPS库命名的类。 此DCPS模型类以DCPS库命名，可在代码生成目标目录中的<ModelName> _T.h文件中找到。

模型类包含一个名为Elements的内部类，定义了在库中建模的每个DCPS实体的枚举标识符，以及由库的主题引用的每个类型。 此Elements类包含以下各项的枚举定义：

- DomainParticipants
- 类型
- 主题
- 内容过滤主题
- 多主题
- 出版商
- 认购
- 数据编写者

- 数据读取器

另外，DCPS模型类捕获这些实体之间的关系。 在实例化DCPS实体时，这些关系由Service类使用。

11.3.2.2 特质类

DCPS模型中的实体通过名称引用其传输配置。 Codegen文件编辑器的Model Customization选项卡用于定义每个名称的传输配置。

可以为特定的代码生成文件定义多组配置的配置。 这些配置集被分组到实例中，每个实例都由一个名称来标识。 可以定义多个实例，代表使用该应用程序的模型的不同部署方案。

对于这些实例中的每一个，都会生成Traits类。 traits类提供了在Codegen编辑器中为特定传输配置名称建模的传输配置。

11.3.2.3 Service Typedef

该服务是一个需要两个参数的模板：（1）DCPS模型中的实体模型元素类，（2）传输配置，在Traits类中。 OpenDDS Modeling SDK为DCPS库和传输配置模型实例的每个组合生成一个typedef。 typedef被命名
<实例><DCPSLibraryName>类型。

11.3.2.4 数据库生成的代码

从数据库中生成IDL，由IDL编译器处理。 IDL编译器生成类型支持代码，用于序列化和反序列化数据类型。

11.3.2.5 QoS策略库生成的代码

QoS策略库中没有生成具体的编译单元。 相反，DCPS库存储它所建模实体的QoS策略。此QoS策略稍后将由Service类查询，该类在创建实体时设置QoS策略。

11.3.3 应用程序代码要求

11.3.3.1 必需的标题

除了Tcp.h头文件（用于静态链接），应用程序还需要包含Traits头文件。 这些将包括构建发布应用程序所需的一切。 这是示例发布应用程序MinimalPublisher.cpp的#include部分。

```
#ifdef ACE_AS_STATIC_LIBS
#include <dds/DCPS/transport/tcp/Tcp.h>
#万一

#include“model / MinimalTraits.h”
```

11.3.3.2 异常处理

建议建模SDK应用程序同时捕获CORBA :: Exception对象和std :: exception对象。

```
int ACE_TMAIN (int argc, ACE_TCHAR * argv [])
{
  尝试{
    // 创建和使用OpenDDS Modeling SDK (请参阅下面的部分)
  catch (const CORBA :: Exception&e) {
    // 处理异常并返回非零值
  } catch (const OpenDDS :: DCPS :: Transport :: Exception&te) {
    // 处理异常并返回非零值
  catch (const std :: exception&ex) {
    // 处理异常并返回非零值
  }
  返回0;
}
```

11.3.3.3 实例化

如上所述，OpenDDS Modeling SDK应用程序必须在其生命周期内创建一个OpenDDS :: Model :: Application对象。 这个Application对象又被传递给由traits头文件中的一个typedef声明指定的Service对象的构造函数。

然后该服务用于创建OpenDDS实体。 要创建的特定实体是使用Elements类中指定的枚举标识符之一来指定的。 该服务为创建实体提供了这个接口：

```
DDS :: DomainParticipant_var参与者 (Elements :: Participants :: Values部分) ;  DDS :: TopicDescription_var主题
 (Elements :: Participants :: Values部分,
                                      Elements :: Topics :: Values topic) ;  DDS ::
Publisher_var publisher (Elements :: Publishers :: Values publisher) ;  DDS ::
Subscriber_var订阅者 (Elements :: Subscribers :: Values订阅者) ;  DDS :: DataWriter_var
writer (Elements :: DataWriters :: Values writer) ;  DDS :: DataReader_var阅读器
 (Elements :: DataReaders :: Values阅读器) ;
```

需要注意的是，该服务还会在必要时创建所需的任何中间实体，例如DomainParticipants，Publishers，Subscribers和Topics。

11.3.3.4 发布者代码

使用上面显示的writer () 方法，MinimalPublisher.cpp继续：

```
int ACE_TMAIN (int argc, ACE_TCHAR * argv [])
{
  尝试{
    OpenDDS :: Model :: Application应用程序 (argc, argv) ;  MinimalLib ::
    DefaultMinimalType模型 (应用程序, argc, argv) ;

    使用OpenDDS :: Model :: MinimalLib :: Elements;
    DDS :: DataWriter_var writer = model.writer (Elements :: DataWriters :: writer) ;
```

剩下的就是将DataWriter缩小到特定类型的数据写入器，并发送样本。

```
    data1 :: MessageDataWriter_var msg_writer = data1 ::
        MessageDataWriter :: _ narrow (writer) ;
    data1 ::消息消息;
    // 填充消息并发送message.text =“最糟糕的电影
    永远”。
    DDS :: ReturnCode_t error = msg_writer-> write (message, DDS :: HANDLE_NIL) ;  if
     (error! = DDS :: RETCODE_OK) {
      // 处理错误
    }
```

总的来说，我们的发布应用程序MinimalPublisher.cpp看起来像这样：

```
#ifdef ACE_AS_STATIC_LIBS
#include <dds/DCPS/transport/tcp/Tcp.h>
#万一

#include“model / MinimalTraits.h”

int ACE_TMAIN (int argc, ACE_TCHAR * argv [])
{
  尝试{
    OpenDDS :: Model :: Application应用程序 (argc, argv) ;  MinimalLib ::
    DefaultMinimalType模型 (应用程序, argc, argv) ;

    使用OpenDDS :: Model :: MinimalLib :: Elements;
    DDS :: DataWriter_var writer = model.writer (Elements :: DataWriters :: writer) ;

    data1 :: MessageDataWriter_var msg_writer = data1 ::
        MessageDataWriter :: _ narrow (writer) ;
    data1 ::消息消息;
    // 填充消息并发送message.text =“最糟糕的电影
    永远”。
    DDS :: ReturnCode_t error = msg_writer-> write (message, DDS :: HANDLE_NIL) ;  if
     (error! = DDS :: RETCODE_OK) {
      // 处理错误
    }
  catch (const CORBA :: Exception&e) {
    // 处理异常并返回非零值
  catch (const std :: exception&ex) {
    // 处理异常并返回非零值
  }
  返回0;
}
```

注意这个最小的例子忽略了日志和同步，这些问题不是OpenDDS Modeling SDK特有的。

11.3.3.5 用户代码

订阅者代码与发布者非常相似。 为了简单起见，OpenDDS Modeling SDK订户可能希望利用称为OpenDDS :: Modeling :: NullReaderListener的Reader Listener的基类。 NullReaderListener实现整个DataReaderListener接口并记录每个回调。

订户可以通过从NullReaderListener派生类并重写感兴趣的接口（例如on_data_available）来创建监听器。

```
#ifdef ACE_AS_STATIC_LIBS
#include <dds/DCPS/transport/tcp/Tcp.h>
#万一

#include"model / MinimalTraits.h"
#include <model/NullReaderListener.h>

类ReaderListener: public OpenDDS :: Model :: NullReaderListener {public:
  virtual void on_data_available (DDS :: DataReader_ptr reader)
                            ACE_THROW_SPEC ( ( (CORBA :: SystemException) )
    {data1 :: MessageDataReader_var reader_i =
      DATA1 :: MessageDataReader :: _窄 (读取器) ;

    if (! reader_i) {
      // 处理错误
      ACE_OS :: exit (-
      1) ;
    }

    data1 ::消息消息;
    DDS :: SampleInfo信息;

    // 阅读，直到没有更多的消息while
     (true) {
      DDS :: ReturnCode_t错误= reader_i-> take_next_sample (msg, info) ;  if
       (error == DDS :: RETCODE_OK) {
        if (info.valid_data) {
          std :: cout <<"消息: "<< msg.text.in () << std :: endl;
        }
      } else {
        如果 (错误! = DDS :: RETCODE_NO_DATA) {
          // 处理错误
        }
        打破;
      }
    }
  }
};
```

在主函数中，从服务对象中创建一个数据读取器：

```
    DDS :: DataReader_var reader = model.reader (Elements :: DataReaders :: reader) ;
```

自然，DataReaderListener必须与数据读取器关联才能获得回调。

```
    DDS :: DataReaderListener_var侦听器 (新的ReaderListener) ;
    reader-> set_listener (listener, OpenDDS :: DCPS :: DEFAULT_STATUS_MASK) ;
```

剩余的用户代码与OpenDDS Modeling SDK应用程序具有相同的要求，因为它必须通过OpenDDS :: Modeling :: Application对象初始化OpenDDS库，并使用适当的DCPS模型Elements类和traits类创建Service对象。

下面是一个示例订阅应用程序MinimalSubscriber.cpp。

```
#ifdef ACE_AS_STATIC_LIBS
#include <dds/DCPS/transport/tcp/Tcp.h>
#万一

#include"model / MinimalTraits.h"
#include <model/NullReaderListener.h>

类ReaderListener: public OpenDDS :: Model :: NullReaderListener {public:
  virtual void on_data_available (DDS :: DataReader_ptr reader)
                           ACE_THROW_SPEC ( ( (CORBA :: SystemException) )
    {data1 :: MessageDataReader_var reader_i =
      DATA1 :: MessageDataReader :: _窄 (读取器) ;

    if (! reader_i) {
      // 处理错误
      ACE_OS :: exit (-
      1) ;
    }

    data1 ::消息消息;
    DDS :: SampleInfo信息;

    // 阅读，直到没有更多的消息while
     (true) {
      DDS :: ReturnCode_t错误= reader_i-> take_next_sample (msg, info) ;  if (error
      == DDS :: RETCODE_OK) {
        if (info.valid_data) {
          std :: cout <<"消息: "<< msg.text.in () << std :: endl;
        }
      } else {
        如果 (错误! = DDS :: RETCODE_NO_DATA) {
          // 处理错误
        }
        打破;
      }
    }
  }
};

int ACE_TMAIN (int argc, ACE_TCHAR * argv [])
{
  尝试{
    OpenDDS :: Model :: Application应用程序 (argc, argv) ;  MinimalLib ::
    DefaultMinimalType模型 (应用程序, argc, argv) ;

    使用OpenDDS :: Model :: MinimalLib :: Elements;
```


```
DDS :: DataReader_var reader = model.reader (Elements :: DataReaders :: reader) ;  DDS ::
DataReaderListener_var侦听器 (新的ReaderListener) ;
```

```
    reader-> set_listener (listener, OpenDDS :: DCPS :: DEFAULT_STATUS_MASK) ;

    // 调用on_data_available，如果有样本正在等待listener-> on_data_available
     (reader) ;

    // 此时应用程序可以等待外部“停止”指示
    // 例如阻塞直到用户用Ctrl-C终止程序。

  catch (const CORBA :: Exception&e) {e._tao_print_exception (“main
     () : ”中捕获的异常） 返回-1;
  catch (const std :: exception&ex) {
    // 处理错误返回-
    1;
  }
  返回0;
}
```

11.3.3.6 MPC项目

为了使用OpenDDS Modeling SDK支持库，OpenDDS Modeling SDK MPC项目应该从dds_model项目库继承。 这是非建模SDK项目继承的dcpsexe基础的补充。

```
项目 (*发布者) : dcpsexe, dds_model {
  // 项目配置
}
```

生成的模型库将在目标目录中生成MPC项目文件和基础项目文件，并负责构建模型共享库。 OpenDDS建模应用程序必须（1）将生成的模型库包含在其构建中；（2）确保其项目在生成的模型库之后构建。

```
项目 (*发布者) : dcpsexe, dds_model {
  // 项目配置库+最小\
  在+ =最小之后
}
```

这两种方法都可以通过从模型库的项目库中继承来完成。

```
项目 (* Publisher) : dcpsexe, dds_model, Minimal {
  // 项目配置
}
```

请注意，Minimal.mpb文件现在必须在项目文件创建期间由MPC找到。 这可以通过-include命令行选项来完成。

使用这两种形式之一，MPC文件必须告诉构建系统在哪里查找生成的模型库。

```
项目 (* Publisher) : dcpsexe, dds_model, Minimal {
  // 项目配置libpaths + =
  model
}
```

此设置基于在Codegen文件编辑器中提供给目标文件夹的设置。

最后，像任何其他MPC项目一样，它的源文件必须包含在内：

```
  Source_Files
    { MinimalPublisher.c
    pp
  }
```

最终的MPC项目对于发布者而言是这样的：

```
项目 (* Publisher) : dcpsexe, dds_model, Minimal {
  exename   = publisher
  libpaths += model

  Source_Files
    { MinimalPublisher.c
    pp
  }
}
```

对于用户来说也是类似的：

```
项目 (* Subscriber) : dcpsexe, dds_model, Minimal {
  exename   =订阅者
  libpaths + =模型

  Source_Files
    { MinimalSubscriber.c
    pp
  }
}
```

11.3.3.7 模型之间的依赖关系

最后一点考虑 – 生成的模型库本身可能依赖于其他生成的模型库。 例如，可能会有一个外部数据类型库生成到不同的目录。

这种可能性可能会导致项目文件的大量维护，因为模型会随着时间的推移而改变依赖关系。 为了帮助克服这种负担，生成的模型库在一个名为<ModelName> _paths.mpb的单独MPB文件中记录了所有外部参考模型库的路径。 继承此路径的基础项目将继承所需的设置以包含依赖模型。

我们的完整MPC文件如下所示：

```
项目 (* Publisher) : dcpsexe, dds_model, Minimal, Minimal_paths {
```

```
exename    =出版商
```


```
  libpaths + =模型

  Source_Files
    { MinimalPublisher.c
    pp
  }
}

项目 (* Subscriber) : dcpsexe, dds_model, Minimal, Minimal_paths {
  exename   =订阅者
  libpaths + =模型

  Source_Files
    { MinimalSubscriber.c
    pp
  }
}
```

C章节12

记录器和重播器

12.1 概观

OpenDDS的记录器功能允许应用程序记录在任意主题上发布的样本，而无需事先知道该主题使用的数据类型。

类似地，Replayer功能允许将这些记录的样本重新发布回相同或其他主题。 使这些功能与其他数据读取器和写入器不同的是，它们能够使用任何数据类型，即使在应用程序中是未知的

建立时间。 实际上，样本被看作是每个样本包含一个不透明的字节序列。

本章的目的是描述OpenDDS的公共API来启用记录/重放用例。

12.2 API结构

在OpenDDS :: DCPS命名空间中定义了两个新的用户可见类（与DDS实体对应的行为有些类似），以及关联的Listener接口。
听众可以可选地由应用程序来实现。 Recorder类的作用类似于DataReader，Replayer类的作用类似于DataWriter。

Recorder和Replayer分别使用底层的OpenDDS发现和传输库，就好像它们是DataReader和DataWriter一样。 域中的常规OpenDDS应用程序将“看到”记录器对象，就像它们是远程DataReader和Replayers，就好像它们是DataWriters一样。

12.3 使用模式

应用程序根据需要创建任意数量的记录器和重放器。 这可能基于使用内置主题来动态发现哪些主题在域中处于活动状态。 创建记录器或重播器需要应用程序提供主题名称和类型名称（如在DomainParticipant :: create_topic（）中）以及相关的QoS数据结构。 记录器需要SubscriberQos和DataReaderQos，而Replayer需要PublisherQos和DataWriterQos。 这些值用于发现的读写器匹配。 有关记录器和重播器如何使用QoS，请参阅下面关于QoS处理的部分。 以下是创建记录器所需的代码：

```
OpenDDS::DCPS::Recorder_var recorder =
      service_participant-> create_recorder (domain_participant,
                                             topic.in
                                              () ,
                                             sub_qos,
                                             dr_qos,
                                             recorder_listener) ;
```

数据样本通过RecorderListener使用简单的“每个样本一个回调”模型提供给应用程序。 该示例作为OpenDDS :: DCPS :: RawDataSample对象提供。 该对象包括该数据样本的时间戳以及编组样本值。 以下是用户定义的记录器侦听器的类定义。

```
类MessengerRecorderListener: public OpenDDS :: DCPS :: RecorderListener
{
上市:
```

```
  MessengerRecorderListener () ;

  virtual void on_sample_data_received (OpenDDS :: DCPS :: Recorder *,
                                        const OpenDDS :: DCPS :: RawDataSample&sample) ;

  virtual void on_recorder_matched (OpenDDS :: DCPS :: Recorder *,
                                    const DDS :: SubscriptionMatchedStatus&status) ;
};
```

应用程序可以将数据存储在合适的地方（内存，文件系统，数据库等）。 在以后的任何时候，应用程序都可以将相同的样本提供给为相同主题配置的Replayer对象。 确保主题类型匹配是应用程序的责任。 以下是一个示例调用，可以向所有与播放器主题相关的读者重播样本：

```
replayer->写（样品）；
```

由于存储的数据取决于数据结构的定义，因此不能跨OpenDDS的不同版本或OpenDDS参与者使用的不同版本的IDL使用。

12.4 QoS处理

缺乏关于数据样本的详细知识使得在Replayer方面使用许多正常的DDS QoS属性变得复杂。属性可以分为几类：

- 支持的
  - 生动活泼
  - 基于时间的过滤器
  - 寿命
  - 耐用性（暂时的本地水平，详见下文）
  - 演示文稿（仅限主题级别）
  - 运输优先权（直通运输）
- 不支持
  - 截止日期（仍然用于读者/作家匹配）
  - 历史
  - 资源限制
  - 耐用性服务
  - 所有权和所有权实力（仍然用于读者/作者匹配）
- 影响读者/写作者匹配和内置主题，否则被忽略
  - 划分
  - 可靠性（仍然由运输谈判使用）

- 目的地顺序
- 延迟预算
- 用户/组数据

12.4.1 耐久性细节

在记录器方面，瞬态本地耐久性与任何正常的DataReader一样工作。 从匹配的DataWriters接收持久数据。 在Replayer方面有一些差异。 与正常的DDS DataWriter不同的是，Replayer没有缓存/存储任何数据样本（只是将其发送到传输）。 由于实例未知，根据通常的历史和资源限制规则存储数据样本是不可能的。 相反，暂时的本地持久性可以通过一个“拉”模型来支持，当一个新的远程DataReader被发现时，中间件调用ReplayerListener上的一个方法。 应用程序然后可以在Replayer上调用应该发送到新加入的DataReader的任何数据样本的方法。 确定这些样本是留给应用程序的。

C章节13

安全档案

13.1 概观

安全配置文件配置允许OpenDDS用于具有可用操作系统和标准库函数的有限集合的环境，并且只需要在系统启动时进行动态内存分配。

OpenDDS安全配置文件（以及ACE中的相应功能）是针对Open Group的FACE规范2.1版开发的 http://www.opengroup.org/face/tech-standard-2.1）。 它可以与对FACE传输服务的支持一起使用来创建符合FACE的DDS应用程序，也可以用于未写入FACE传输服务API的通用DDS应用程序。 这后面的用例就是这样描述的

开发人员指南的一部分。 有关前用例的更多信息，请参阅源代码分发中的文件FACE / README.txt，或者联系我们sales@objectcomputing.com （商业支持）或opendds-main@lists.sourceforge.net （社区支持）。

13.2 OpenDDS的安全配置文件子集

配置安全配置文件时，OpenDDS的以下功能不可用：

- DCPSInfoRepo及其关联的库和工具
- 传输类型：tcp，udp，多播，共享内存
  - rtps_udp传输类型可用（使用UDP单播或多播）
- OpenDDS监视器库和监视GUI

在开发安全配置文件时，以下DDS合规性配置文件被禁用：

- content_subscription
- ownership_kind_exclusive
- object_model_profile
- persistence_profile

参见章节1.3.3 有关合规性概况的更多详情。 在安全配置文件构建中启用任何这些合规性配置文件可能会导致编译时或运行时错误。

要构建OpenDDS安全配置文件，请将命令行参数“--safety-profile”与您的平台或配置所需的任何其他参数一起传递给配置脚本。 在配置脚本中启用安全配置文件时，上面列出的四个合规配置文件默认为禁用。 见部分1.3 以及源代码分发中的INSTALL文件以获取有关配置脚本的更多信息。

13.3 ACE的安全配置文件配置

OpenDDS使用ACE作为其平台抽象库，而在OpenDDS的安全配置文件配置中，必须在ACE中启用以下安全配置文件配置之一：

- FACE Safety Base（始终使用内存池）
- FACE安全扩展与内存池
- 使用标准C ++动态分配扩展FACE安全

OpenDDS的配置脚本将自动配置ACE。 通过命令行参数“--safety-profile = base”选择Safety Base配置文件。 否则，“ - 安全配置文件”（无等号）配置将默认为使用内存池进行安全扩展。

使用标准C ++动态分配配置的安全扩展不是由配置脚本自动生成的，但可以在configure（和make make之前）生成文件“build / target / ACE_wrappers / ace / config.h”后对其进行编辑。 删除ACE_HAS_ALLOC_HOOKS的宏定义以禁用内存池。

ACE的安全配置文件配置已经在Linux和LynxOS-178版本2.3.2 +补丁上进行了测试。 其他平台也可能工作，但可能需要额外的配置。

13.4 运行时配置选项

OpenDDS使用的内存池可以通过在配置文件的[common]部分设置值来配置。 见部分7.2 以及表格的pool_size和pool_granularity行表7-2.

13.5 运行ACE和OpenDDS测试

配置和构建OpenDDS安全配置文件之后，请注意，顶层有两个子目录，每个子目录都包含一些二进制工件：

- build / host具有构建时代码生成器tao_idl和opendds_idl
- build / target包含安全配置文件ACE和OpenDDS的运行时库以及OpenDDS测试

因此，测试需要相对于build / target子目录。 源代码 - 在生成的文件build / target / setenv.sh中获取所有需要的环境变量。

ACE测试不是默认构建的，但是一旦建立了这个环境，构建它们所需要的就是生成makefile并运行make：

1. cd $ACE_ROOT/tests
2. $ACE_ROOT/bin/mwc.pl -type gnuace
3. 使

通过更改到$ ACE_ROOT / tests目录并使用run_test.pl来运行ACE测试。 通过配置所需的任何“-Config XYZ”选项（使用run_test.pl -h查看可用的配置选项）。

通过更改为$ DDS_ROOT并使用bin / auto_run_tests.pl运行OpenDDS测试。 默认情况下，通过“-Config OPENDDS_SAFETY_PROFILE”，“-Config SAFETY_BASE”（如果使用安全基础），“Config RTPS”和-Config选项，默认情况下为：“-Config DDS_NO_OBJECT_MODEL_PROFILE -Config DDS_NO_OWNERSHIP_KIND_EXCLUSIVE -Config DDS_NO_PERSISTENCE_PROFILE
-Config DDS_NO_CONTENT_SUBSCRIPTION“。

或者，可以使用该测试目录下的run_test.pl运行单个测试。 将相同的一组-Config选项传递给run_test.pl。

13.6 使用内存池应用

当构建时启用内存池时，由OpenDDS中的代码或ACE（由OpenDDS调用的方法）所做的所有动态分配都将通过该池。 由于池是通用动态分配器，所以应用程序代码也可能需要使用该池。 由于这些API是OpenDDS的内部函数，因此在将来的版本中可能会更改。

类OpenDDS :: DCPS :: MemoryPool（dds / DCPS / MemoryPool.h）包含池实现。 但是，大多数客户端代码不应该直接与之交互。 类SafetyProfilePool（dds / DCPS / SafetyProfilePool.h）将池调整到ACE_Allocator接口。 PoolAllocator <T>（PoolAllocator.h）将池调整为C ++ Allocator概念（C ++ 03）。 由于PoolAllocator是无状态的，它依赖于ACE_Allocator的单例。 当OpenDDS配置内存池时，ACE_Allocator的单例实例将指向SafetyProfilePool类的一个对象。

使用C ++标准库类的应用程序代码可以直接使用PoolAllocator，也可以使用PoolAllocator.h中定义的宏（例如OPENDDS_STRING）。

分配动态内存的原始（无类型）缓冲区的应用程序代码可以直接使用SafetyProfilePool，也可以通过ACE_Allocator :: instance（）单例使用。

从堆分配对象的应用程序代码可以使用PoolAllocator <T>模板。

应用程序开发人员编写的类可以派生自PoolAllocationBase（请参阅PoolAllocationBase.h），以继承类范围的运算符new和delete，从而将这些类的所有动态分配重定向到池。