



北京理工大学  
BEIJING INSTITUTE OF TECHNOLOGY

OpenSplice

# 软件分析报告

V4.1.16

2018 年 11 月 5 日

# 软件分析报告

软件名称	OpenSplice DDS
指导老师	金福生
项目组长	李林峰
团队成员	马伯乐、沈云汉、尚楠、陈小虎
单    位	北京理工大学计算机学院软件工程系
版  本  号	4.1.16
发布日期	2018 年 11 月 5 日

二〇一八年·北京

# 目录

<b>§1. 文档的目标及范围.....</b>	<b>1</b>
1.1. 文档概述.....	1
1.2. 项目背景.....	1
1.3. 参考材料.....	1
<b>§2. OpenSplice 基础研究.....</b>	<b>2</b>
2.1. 基础.....	2
2.1.1. 数据分发服务 .....	2
2.1.2. 数据分发服务标准.....	2
2.1.3. 数据分发服务简介.....	3
2.1.4. 总结.....	7
2.2. 主题、域和分区.....	7
2.2.1. 主题.....	7
2.2.2. 范围界定信息 .....	12
2.2.3. 内容过滤器.....	13
2.2.4. 总结.....	14
2.3. 读取和写入数据.....	14
2.3.1. 写数据 .....	14
2.3.2. 数据的存取.....	18
2.3.3. 等待与响应.....	21
2.3.4. 总结.....	23
2.4. 服务质量策略.....	24
2.4.1. 数据分发服务的服务质量模型.....	24
2.4.2. 总结.....	28
<b>§3. OpenSplice 软件体系架构.....</b>	<b>29</b>
3.1. OMG-DDS .....	29
3.2. OpenSplice DDS 组成模块.....	30
3.3. OpenSplice 体系结构.....	30
3.3.1. 共享内存 .....	30
3.3.2. 以数据为中心的发布订阅模型架构 .....	31

<b>§4. OpenSplice 设计模式分析.....</b>	<b>34</b>
4.1. 工厂模式.....	34
4.1.1. 工厂模式 .....	34
4.1.2. 在 OSPL 中工厂模式的应用 .....	34
4.1.3. 优点.....	35
4.2. 单例模式.....	35
4.2.1. 单例模式 .....	35
4.2.2. 在 OSPL 中单例模式的应用 .....	35
4.2.3. 优点.....	35
4.3. 代理模式.....	36
4.3.1. 代理模式 .....	36
4.3.2. 在 OSPL 中代理模式的应用 .....	36
4.3.3. 优点.....	37
4.4. 迭代器模式.....	37
4.4.1. 迭代器模式.....	37
4.4.2. 在 OSPL 中迭代器模式的应用 .....	38
4.4.3. 优点.....	38
4.5. 过滤器模式.....	38
4.5.1. 过滤器模式.....	38
4.5.2. 在 OSPL 中过滤器模式的应用 .....	39
4.5.3. 优点.....	39
4.6. 适配器模式.....	39
4.6.1. 适配器模式.....	39
4.6.2. 在 OSPL 中适配器模式的应用 .....	39
4.6.3. 优点.....	40
4.7. 观察者模式.....	40
4.7.1. 观察者模式.....	40
4.7.2. 在 OSPL 中观察者模式的应用 .....	40
4.7.3. 优点.....	41

## §1. 文档的目标及范围

### 1.1. 文档概述

本文档首先就 OpenSplice 进行了基础分析与研究，包括 OSPL DDS 的基础名词解释、写入写出数据、QoS 策略等进行了介绍。然后对 OpenSplice 的软件体系架构与结构进行了一定的分析。最后对 OpenSplice 中用到的设计模式进行解释与举例。

编写该文档便于本组在进行二次开发时有更深入的理解与更合理的利用。

### 1.2. 项目背景

北京理工大学计算机学院软件工程系 2018~2019 学年第一学期为提升学生对软件体系结构与设计模式有更深入的理解，要求学生对 OpenSplice 进行深入的学习与研究，并以此为依据进行二次开发。

### 1.3. 参考材料

[1] OpenSplice\_DDS Tutorial.pdf

[2] OpenSplice\_Tutorial\_C.pdf

[3] CSDN 博客

## §2. OpenSplice 基础研究

### 2.1. 基础

#### 2.1.1. 数据分发服务

Pub/Sub(发布/订阅模式)是一种一对多通信的抽象概念。它提供发布服务器与其订阅者之间的匿名、解耦和异步通信。“Pub/Sub”是当今用于构建和集成分布式应用程序(如社交应用程序、金融交易等)的技术背后的抽象,同时保持组件的松散耦合和独立演化。

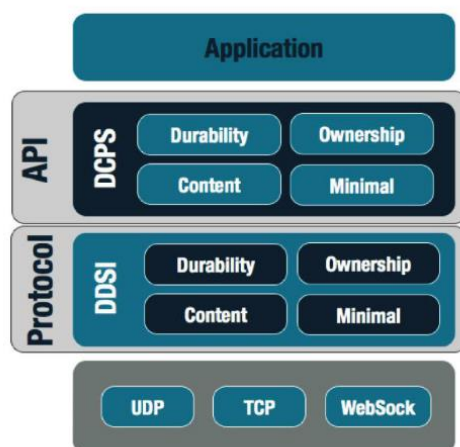
Pub/Sub 抽象的各种实现是随着时间的推移而出现的,以满足不同应用领域的需求。数据分发服务(DDS)是一个对象管理小组(OMG)。Pub/Sub 标准于 2004 年推出,以满足大规模任务和业务关键应用程序的数据共享需求。

DDS 是 Pub/Sub 技术,它适用于无处不在、多标记、高效和安全的数据共享。

#### 2.1.2. 数据分发服务标准

DDS 标准系列包括 DDS v1.4 API 和 DDSI v2.2、ISO/IEC C V1.0 API 和 Java5 V1.0 AP 在 DDS 标准 BELL 中的应用。

DDS 标准



DDS API 标准保证跨不同供应商实现的源代码可移植性,而 DDSI 标准确保来自不同供应商的 DDS 实现之间的在线互操作性。

DDS API 标准定义了几种不同的配置文件(参见 DDS 标准),它们通过内容过滤和查询、时态解耦和自动故障转移来增强实时 Pub/Sub。此外,在 C、C#、Java、JavaScript、CoffeeScript、Scala 和更多版本中都可以使用 API,这些 API 可以根据用户应用程序的需要在部署中混合使用。

DDS 标准于 2004 年被 OMG 正式采用,如今它已经成为一种成熟的 Pub/Sub 技术,可以可靠地、可预测地分发高数据量的数据。在智能电网、智能城市、国防、SCADA、金融交易、空中交通控制和管理、高性能遥测和大规模监控系统等应用领域。它也是物联网工业互联网联盟(物联网)定义的参考通信架构之一。

### 2.1.3. 数据分发服务简介

为了解释 DDS,本分析报告中将开发一个程序作为例子,该示例足够简单,可以容易理解,但又足够复杂,从而说明 DDS 系统的所有主要特征。

这个例子是一个大型建筑的温度监控系统。

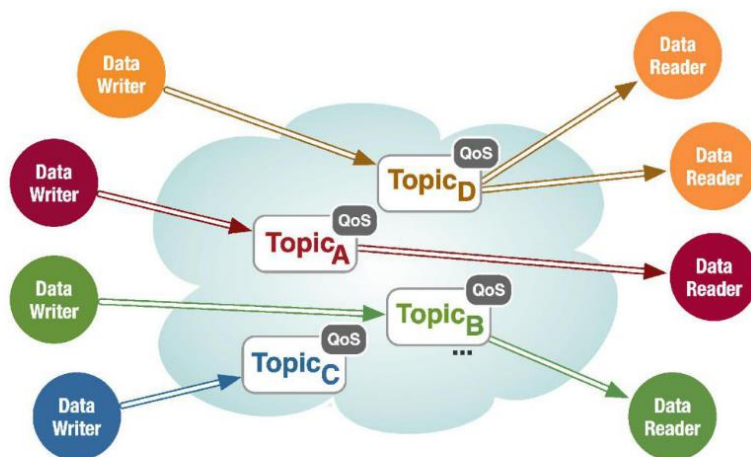
建筑物的每个楼层都有几个房间,每个房间都配备了一套温湿度传感器和一个或多个空调器。该应用旨在对建筑物中的所有元件进行监测,并对每个房间进行温度和湿度控制。

该应用是一种典型的分布式监测和控制应用,其中从分布在整个空间中的多个传感器进行数据遥测,并且传感器数据的处理导致动作被应用于执行器(调理器)。

## 1. 全局数据空间

在“全局数据空间”的基础上的关键抽象是一个完全分布式的 GDS。重要的是要指出,DDS 规范需要实现全局数据空间的完全分布,以避免单点故障和瓶颈。

全局数据空间



出版商和订阅者可以随时加入或离开 GDS,因为它们是动态发现的。发布

服务器和订阅服务器的动态发现由 GDS 执行，不依赖任何类型的集中式注册表，就像在其他 Pub/Sub 技术(如 Java MES)中发现的那样 SAGE 服务(JMS)

最后，GDS 还发现应用程序定义的数据类型，并将它们作为发现过程的一部分进行传播。

这里的要点是，具有动态发现的 GDS 的存在意味着在部署系统时不需要配置。所有的东西都会自动被发现和数据也会开始流动。

此外，由于 GDS 是完全分布式的，所以不必担心某些服务器的崩溃对系统可用性产生不可预测的影响。在 DDS 中没有单点故障，所以尽管应用程序可能崩溃并重新启动，或者断开并重新连接，但是系统作为一个整体继续运行。

## 2. 域参与者

为了做任何有用的事情，DDS 应用程序需要创建域参与者(DP)。DP 允许访问 DDS，它在 DDS 应用程序中被称为域 domain。

创建域参与者的清单显示了如何创建域参与者；请注意，域是由整数标识的。  
创建域参与者

```
// create a Domain Participant, -1 defaults to value defined in configuration file
dds::domain::DomainParticipant dp(-1);

// Creates a domain participant in the domain identified by
// the number 18
dds::domain::DomainParticipant dp2(18);
```

## 3. 主题

在 DDS 中，从发布服务器流向订阅者的数据属于一个主题，该主题表示可以生成或使用的信息单位。

一个主题被定义为：(一个独特的名字、一个标识符、一组服务质量(QOS)策略)这样一个三元组。

后面将详细解释它用于控制与主题(Topic)相关联的非功能属性。

就目前来说，如果 QOS 没有显式设置，那么 DDS 实现将使用标准规定的一些默认值。

可以用定义结构类型的 OMG 接口定义语言(IDL)标准的子集来表示主题类型，但存在不支持任何类型的限制。

可以将 Topic Types 视为用 C 语言的结构体定义的，其属性可以是原语类型。并允许结构的嵌套。

可能有人会疑惑 DDS 是如何关联的。CORBA. DDS 与 CORBA 的唯一共同之处在于它们都使用了 IDL 的子集；除此之外，CORBA 和 DDS 是两个完全不同的标准和两个完全不同但互补的技术。



回到温度控制应用程序, 这里将定义一些主题, 这些主题表示对温度传感器、调理器和安装温度传感器和调理器的房间的读取。上市温度传感器的 IDL 定义提供如何定义温度传感器的主题类型的示例。

温度传感器的 IDL 定义:

```
// TempControl.idl
enum TemperatureScale {
    CELSIUS,
    FAHRENHEIT,
    KELVIN
};

struct TempSensorType {
    short id;
    float temp;
    float hum;
    TemperatureScale scale;
};
#pragma keylist TempSensorType id
```

注意到温度传感器的 IDL 定义还包括一个 `#pragma keylist` 指令。此指令用于指定特定的键。TempSensorType 用具有传感器标识符(id 属性)表示的某个键来指定。在运行时, 每个键值将标识特定的数据流; 更准确地说, 在 DDS 中, 每个键值标识一个 Topic 实例。对于每个实例, 都可以观察它的生命周期并了解感兴趣的内容, 比如何时首次出现在系统中, 或者何时被处理。

Keys, 在识别实例的同时, 也被用来捕获数据关系, 就像在传统的实体关系建模中所做的那样。

Keys, 可以由任意数量的属性组成, 其中一些属性也可以嵌套结构体。

定义 Topic 类型并运行 IDL 预处理器, 生成表示 topic 所需要的计算机语言, DDS Topic 可以通过简单实例化具有正确类型和名称的 Topic Class 的 DDS API Topic 的创建:

```
// Create the topic
dds::topic::Topic<tutorial::TempSensorType> topic(dp, "TTempSensor");
```

## 4. 读写数据

现在已经指定了这些主题, 现在要在 Publishersand 和 Subscriber 之间生成 Topic flow。

DDS 使用用户定义的 Topic Types 规范来生成高效的编码和解码例程, 就好比强类型的 DataReader 和 DataWriter。

创建 DataReader 或 DataWriter 只需要通过用 Topic Type 实例化模板类并传递所需的 Topic 对象来构造对象。

为 TempSensorType 创建 DataReader 之后, 就可以读取系统中分布的温度传感器生成的数据了。同样, 在为 TaveSyror 类型创建数据写入器之后, 就可以准

备编写(发布)数据了。DDS 中写入数据和读取数据的列表显示了写入和读取数据所需的步骤。

DDS 中的数据写入:

```
// create a Domain Participant, -1 defaults to value defined in configuration file
dds::domain::DomainParticipant dp(-1);
```

```
// Create the topic
dds::topic::Topic<tutorial::TempSensorType> topic(dp, "TTempSensor");
```

```
// Create the Publisher and DataWriter
dds::pub::Publisher pub(dp);
dds::pub::DataWriter<tutorial::TempSensorType> dw(pub, topic);
```

```
// Write the data
tutorial::TempSensorType sensor(1, 26.0F, 70.0F, tutorial::CELSIUS);
dw.write(sensor);
```

```
// Write data using streaming operators (same as calling dw.write(...))
dw << tutorial::TempSensorType(2, 26.5F, 74.0F, tutorial::CELSIUS);
```

DDS 中的数据读取:

```
// create a Domain Participant, -1 defaults to value defined in configuration file
dds::domain::DomainParticipant dp(-1);
// create the Topic
dds::topic::Topic<tutorial::TempSensorType> topic(dp, "TTempSensor");
// create a Subscriber
dds::sub::Subscriber sub(dp);
// create a DataReader
dds::sub::DataReader<tutorial::TempSensorType> dr(sub, topic);

while (true) {
    auto samples = dr.read();
    std::for_each(samples.begin(),
                  samples.end(),
                  [](const dds::sub::Sample<tutorial::TempSensorType>& s) {
                      std::cout << s.data() << std::endl;
                  });
    std::this_thread::sleep_for(std::chrono::seconds(1));
}
```

第一个 DDS 应用程序(在 DDS 中读取数据)使用轮询方式每秒钟从 DDS 中读取数据。睡眠是用来避免在循环中旋转过快的, 因为 DDS 读取是非阻塞和返回的。如果没有可用的数据, 则立即获得。

尽管轮询是一种有效的使用方法, DDS 支持另外两种通知应用程序数据可用性的方法: listeners and waitsets。

- Listeners 可以向读取器注册，以便接收数据可用性通知以及其他一些有趣的状态更改，例如违反了 QoS。

- Waitsets，模仿 Unix 风格的选择调用，可以用来等待有趣事件的发生，其中之一可能是数据的可用性。将在本报告后面详细介绍这些协调机制。

由于数据读取器和数据写入器是完全解耦的，所以代码乍一看可能有点令人费解。目前还没解释清楚他们在哪里写数据或者读数据，他们是如何发现彼此的，等等。如本章开头所述，DDS 配备了参与者的动态发现以及用户定义的数据类型。因此，DDS 发现数据生产者和消费者，并注意匹配它们。

#### 2.1.4. 总结

第一章阐述了 DDS 背后的抽象，并介绍了其核心概念。它还展示了如何编写第一个 DDS 应用程序，它在分布式系统上分配温度传感器的值。它需要不到 15 行代码来让应用程序工作，这显示了 DDS 的显著能力。

接下来的章节将介绍更高级的概念，并且所有的 dds 功能都将被演示，同时还将创建一个复杂的可伸缩的、高效的和真实的 Pub/Sub 应用程序。

## 2.2. 主题、域和分区

前一章介绍了 DDS 的基本概念，并详细介绍了编写一个简单的 Pub/Sub 应用程序所需的步骤。本章将深入研究 DDS，从数据管理开始。

### 2.2.1. 主题

主题表示可由 DDS 应用产生或消费的信息单元。主题由名称、类型和一组 QoS 策略定义。

#### 1. 主题类型

DDS 不依赖于编程语言和操作系统(OS)，因此它定义了它的类型系统，并为其类型定义了一个空间和时间高效的二进制编码。

这里将介绍一下可用于定义主题类型的 IDL 子集。主题类型由 IDL 结构加上键组成。该结构可以包含所需的尽可能多的字段，并且每个 field 可以是原语类型(见表 IDL Template Types)、模板类型(参见表 IDL 模板类型)或构造类型(参见表 IDL Constructed types)。

Primitive Types

Primitive Type	Size (bits)
boolean	8
octet	8
char	8
short	16
unsigned short	16
long	32
unsigned long	32
long long	64
unsigned long long	64
float	32
double	64

如表所示本原类型基本类型基本上已经满足需求，除了 `int` 类型，但这不是问题，因为 IDL 整型类型、`long` 和 `long` 与 C99 `int16_t`、`int32_t` 和 `int64_t` 等价。更重要的是：与 `int` 类型不同，`int` 类型可以在不同的平台上具有不同的尺寸，而这些类型有其特有的尺寸。

IDL Template Types:

Template Type	Example
<code>string&lt;length = UNBOUNDED\$&gt;</code>	<pre>string s1; string&lt;32&gt; s2;</pre>
<code>sequence&lt;T,length = UNBOUNDED&gt;</code>	<pre>sequence&lt;octet&gt; oseq; sequence&lt;octet, 1024&gt; oseq1k; sequence&lt;MyType&gt; mtseq; sequence&lt;MyType, \$10&gt;\$ mtseq10;</pre>

在表 IDL Template Types 中，`string` 只能根据其最大长度进行参数化，而 `sequence` 类型可以根据其最大长度和其最大长度进行参数化。序列类型抽象出一个同构的随机访问容器，类似于 C++ 中的 `std: vector` 或 `java.util.Vector`。

最后，需要指出的是，当没有提供最大长度时，类型被假定为具有无限长度，这意味着中间件将根据需要分配尽可能多的内存来存储应用程序提供的值。

表 IDL Constructed Types 显示 DDS 支持三种不同类型的 IDL 构造类型：`enum`，`struct` 和 `union`。

**IDL Constructed Types**

Constructed Type	Example
enum	enum Dimension { 1D, 2D, 3D, 4D};
struct	<pre> struct Coord1D { long x;}; struct Coord2D { long x; long y; }; struct Coord3D { long x; long y; long z; }; struct Coord4D { long x; long y; long z,                 unsigned long long t;}; </pre>
union	<pre> <b>union Coord switch (Dimension) {</b>     case 1D: Coord1D c1d;     case 2D: Coord2D c2d;     case 3D: Coord3D c3d;     case 4D: Coord4D c4d; <b>};</b> </pre>

应该清楚, Topic 类型是一个可以包含 nested structures, unions, enumerations, 和 template types, 以及 primitive types 等结构的结构。此外, 可以定义 DDS 支持的或用户定义的类型的多维数组。

为了把事物联系在一起, 从上面描述的 IDL 类型到特定的编程语言(如 C++, Java, c#)有语言特定的映射。

**2. 主题密钥、实例和示例**

每个主题附带一个相关的密钥集。此密钥集可能是空的, 或者可以包含由主题类型定义的任意数量的属性。用于建立密钥的属性的数量、种类或嵌套级别没有限制。但是它的一些限制是: 键应该是基元类型(见表)。本原类型枚举或字符串。键不能被构造类型(尽管它可能由嵌入构造类型的一个或多个成员组成)、数组或任何类型的序列。

**Keyed and Keyless Topics**

```

enum TemperatureScale {
    CELSIUS,
    FAHRENHEIT,
    KELVIN
};

struct TempSensorType {
    short id;
    float temp;
    float hum;
    TemperatureScale scale;
};
#pragma keylist TempSensorType id

struct KeylessTempSensorType {
    short id;
    float temp;
    float hum;
    TemperatureScale scale;
};
#pragma keylist KeylessTempSensorType

```

回到示例应用程序(温度控制和监控系统), 可以定义基础章节中定义的 TempSensorType 的 Keyless variant。

回到示例 Keyed 和 Keyless Topic 作为 TempSensorTyped 的 id 属性成为它的 key, 除了 KeylessTempSensorType 之外, 它还显示了一个空键集, 它在其 #pragma keylist directive 指令中定义了这一点。

如果创建了与 Keyed 和 Keyless Topic 中声明的类型关联的两个主题, 那么它们之间有什么区别:

```
dds::topic::Topic<tutorial::TempSensorType> topic(dp, "TTempSensor");
dds::topic::Topic<tutorial::KeylessTempSensorType> kltsTopic(dp,
                                                             "KLTempSensorTopic");
```

这两个主题的主要区别在于实例的数量:

- Keyless Topic 只有一个实例, 因此可以认为他是一个单例。
- Key Topic 的每个键值都有一个实例

在面向对象编程语言中, 可以将主题视为定义一个类, 该类的实例是为主题键的每个唯一值创建的。

因此, 如果主题没有键, 我们将得到一个单例。Topic 的实例是运行时实体, DDS 会跟踪它并判断它

- 是否有正在活动的写入者
- 实例首次出现在系统中
- 该实例已被释放(从系统中显式删除)

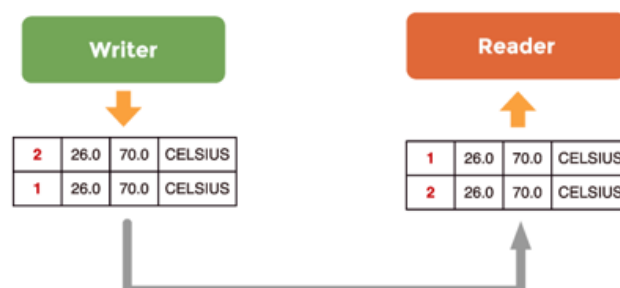
主题实例会影响读取器端数据的组织以及内存的使用。此外, 正如这之后面将看到的, 有一些在实例级别上适用的 QOS。

现在将说明在编写无键 Keyed 和 Keyless Topic 时可能会发生的情况。

如果为 Keyless KLSensorTopic 编写一个示例, 这将修改其他相同实例的值, 即单例, 而不管该实例的内容如何。

另一方面, 为 KeyTempSensorTopic 编写的每个示例将根据 key 属性的值(示例中的 id)修改特定 Topic 实例的值。

Keyless Topic 的数据读取器队列:



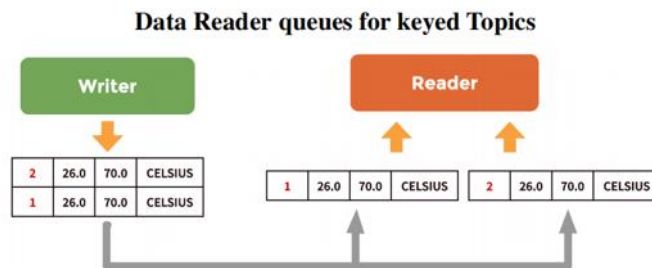
因此，下面的代码是为同一个实例编写两个示例，如无键主题的数据读取器队列中所示。这两个示例将放在同一个读取器队列中：与单例实例关联的队列，如 **Keyless Topic** 的数据读取器队列中所示。

```
dds::pub::DataWriter<tutorial::KeylessTempSensorType> kldw(pub, kltsTopic);
tutorial::KeylessTempSensorType klts(1, 26.0F, 70.0F, tutorial::CELSIUS);
kldw.write(klts);
kldw << tutorial::KeylessTempSensorType(2, 26.0F, 70.0F, tutorial::CELSIUS);
```

如果我们为 **TempSensorTopic** 编写相同的示例，那么最终的结果是完全不同的。在下面的代码片段中编写的两个样本有两个不同的 id 值，分别是 1 和 2；它们是引用两个不同的实例。

```
dds::pub::DataWriter<tutorial::TempSensorType> dw(pub, topic);
tutorial::TempSensorType ts(1, 26.0F, 70.0F, tutorial::CELSIUS);
dw.write(ts);
dw << tutorial::TempSensorType(2, 26.0F, 70.0F, tutorial::CELSIUS);
```

这两个示例被提交到两个不同的队列中，如键控主题的数据读取器队列所示，每个实例都有一个队列。



总之，Topic 应该被认为是面向对象语言中的类，并且每个唯一的键值都标识一个实例。主题实例的生命周期由 DDS 管理，每个 Topic 实例被分配给内存资源；将它看作是读取器端的一个队列。键标识主题中的特定数据流。因此，在示例中，每个 id 值将标识特定的温度传感器。与许多其他 Pub/Sub 技术不同，DDS 允许使用键自动解复用不同的数据流。此外，由于每个温度传感器代表 **TempSensorTopic** 的一个实例，因此可以通过跟踪相关实例的生命周期来跟踪传感器的生命周期。因为它引入了一个新的实例，所以可以检测新传感器何时被添加到系统中；当传感器发生故障时，可以检测到，因为 DDS 可以在没有针对特定实例的写入器时报告。甚至可以通过 DDS 提供的有关状态转换的信息来检测传感器何时崩溃，然后对其进行恢复。

最后，在讨论 DDS 实例之前，强调 DDS 订阅涉及到的 Topic。因此一个订阅服务器接收为该主题生成的所有实例。在某些情况下，这不是需要采取可取的



和有针对性的行动。范围确定将在下一节中讨论。

### 2.2.2. 范围界定信息

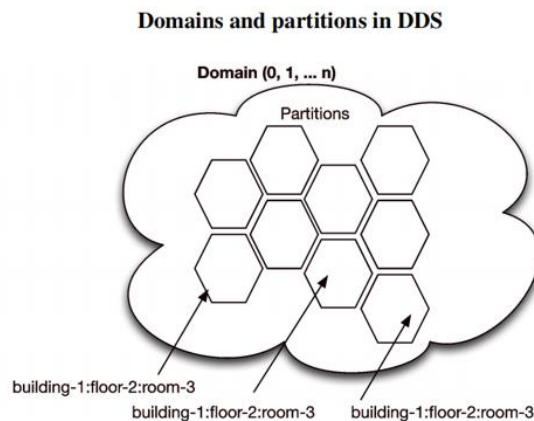
#### 1. Domain

DDS 提供了两种限定信息范围的机制：域和分区。

域建立一个虚拟网络，连接已加入它的所有 DDS 应用程序。除非由用户应用程序显式地中介，否则任何域之间的通信都不可能发生。

#### 2. Partition

可以将域进一步组织为分区，其中每个分区可以表示 topic 的逻辑分组。DDS 分区由名称描述，如 SensorData 分区、Command 分区、LogData 分区等。为了在分区中发布数据或订阅其中包含的主题，必须显式地连接分区。



DDS 提供的连接分区的机制非常灵活，因为发行者或订阅者可以通过提供其全名(如 SensorDataPartition)加入，或者可以加入所有分区。匹配正则表达式，例如 Sens\*或\*data\*。受支持的正则表达式与 POSIX fnMatch 函数所接受的正则表达式相同(参见 POSIX fnMatch)。

概述：分区提供了一种确定范围信息的方法。这种范围界定机制可以用来组织不同的相干集。

分区也可用于分隔 Topic 实例。对于那些具有大量实例(例如大型遥测系统)的应用程序，实例隔离对于优化性能或最小化占用空间是必要的，或金融交易应用程序。参考温度监测和控制系统的例子，可以设计出一种非常自然的数据分割方案，它模拟各种温度传感器的物理位置。为此，我们可以使用由建筑物编号、楼层和安装传感器的房间号组成的分区名称：

```
building-<number>:floor-<level>:room-<number>
```



使用此命名方案，如 dds 中的域和分区所示，在 1 号楼 15 楼 51 室产生的所有主题都属于分区 `building-1:floor-15:room-51`。同样，分区表达式 `building-1:floor-1:room-*` 匹配 1 楼一楼所有房间的所有分区。

简而言之，分区可用于对信息进行范围调整，命名约定(例如用于示例温度控制应用程序的命名约定)可用于模拟分层组织。从平面分区开始的  $n$  个数据。使用相同的技术，可以根据应用程序的需要，在不同的维度或视图中对数据进行切片和访问。

### 2.2.3. 内容过滤器

域和分区对于数据的结构组织是有用的机制，但某些情况下有必要根据其内容来控制接收到的数据呢。内容筛选允许创建限制其实例可能采用的值的主题。

当订阅内容过滤的主题时，应用程序将在所有已发布的值中只接收与主题筛选器匹配的值。筛选器表达式可以对完整的主题进行操作。相对于只能在头上操作(就像在许多其他 Pub/Sub 技术中那样)，如 JMS。过滤器表达式在结构上类似于 SQL 中的 WHERE 语句。

下表列出了 DDS 支持的运算符。

**Legal operators for DDS Filters and Query Conditions**

Constructed Type	Example
=	equal
<>	not equal
>	greater than
<	less than
>=	greater than or equal
<=	less than or equal
BETWEEN	between and inclusive range
LIKE	matches a string pattern

内容过滤的主题从几个不同的角度非常有用。首先，它们将 DDS 使用的内存量限制在与滤波器匹配的实例和样本上。此外，过滤可以通过将检查某些数据属性的逻辑委托给 DDS 来简化此应用程序。例如，如果我们考虑了温度控制应用程序，我们可能会只有在接到通知时，温度或湿度才会超出规定的范围。因此，假设我们想要将温度维持在 20.5 度到 21.5 度之间，并且假设湿度是肯定的在 30% 到 50% 之间，我们可以创建一个内容过滤的主题，当传感器产生超出所需范围的值时，会提醒应用程序。这可以通过使用过滤器表达式来完成。

```
((temp NOT BETWEEN 20.5 AND 21.5)
OR
(hum NOT BETWEEN 30 AND 50))
```

列表内容筛选主题显示了为带有上述表达式的 TempSensor 主题创建内容筛选主题的代码。注意，内容过滤主题是从一个常规主题开始创建的。此外，值得注意的是，筛选器表达式依赖于位置参数%0、%2 等，其实际值通过字符串向量传递。

#### Content Filtered Topic

```
// Create the TTempSensor topic
dds::topic::Topic<tutorial::TempSensorType> topic(dp, "TTempSensor");

// Define the filter expression
std::string expression =
    "(temp NOT BETWEEN %0 AND %1) \
    OR \
    (hum NOT BETWEEN %2 and %3)";

// Define the filter parameters
std::vector<std::string> params =
    {"20.5", "21.5", "30", "50"};

// Create the filter for the content-filtered-topic
dds::topic::Filter filter(expression, params);

// Create the ContentFilteredTopic
dds::topic::ContentFilteredTopic<tutorial::TempSensorType> cfTopic(topic,
                                                                    "CFTTempSensor",
                                                                    filter);

dds::sub::Subscriber sub(dp);
//This data reader will only receive data that matches the content filter
dds::sub::DataReader<tutorial::TempSensorType> dr(sub, cfTopic);
```

## 2.2.4. 总结

本章介绍了 dds 中数据管理的最重要方面：主题-类型和主题实例，以及 dds 提供的用于限定信息范围的各种机制。

信息可以通过域和分区进行结构组织，并且可以使用内容过滤的主题和查询条件创建特殊的视图。

## 2.3. 读取和写入数据

前一章介绍了 DDS 主题、主题实例和示例的定义和语义。它还描述了域和分区以及它们在组织应用程序数据方面所扮演的角色。本章探讨 DDS 提供的数据读写机制。

### 2.3.1. 写数据

如前所述，用 DDS 编写数据就像调用 DataWriter 上的写方法一样简单。但是，为了能够充分利用 DDS，有必要理解写入者与主题实例两者之间的关系。

为了解释主题与主题数据实例之间的差异，我们将主题/主题数据类型与面向对象编程语言(如 Java 或 C++)中的类/对象进行类比。主题数据类型的实例有：

由其独特的键值提供的身份标识，以及自身的生命周期。

主题数据类型的实例生命周期可以通过 **DataWriter** 隐含的语义隐式管理，也可以通过 **DataWriter** API 显式控制。生命周期这个实例转换可能会对本地和远程资源的使用产生影响，因此，了解这方面是很重要的。

## 1. Topic-Instances Life-cycle

在详细了解如何管理生命周期之前，让我们看看存在哪些可能的状态。

- 如果至少有一个 **DataWriter** 显式或隐式(通过写)注册了该主题的数据类型的实例，则该实例是活动的。已注册实例的 **DataWriter** 声明一旦发生，它将致力于发布该实例的潜在更新。因此，**DataWriter** 为实例和至少有一个样本。此主题的 **DataReader** 还将为每个已注册实例维护类似的资源保留。只要实例至少由一个 **DataWriter** 注册，它会被认为是活着的。

- 当没有更多的 **DataWriters** 注册实例时，实例处于 **NOTTIVE\_NOWARS** 状态。这意味着不再有 **DataWriters** 打算更新实例状态和他们中的人都会释放他们以前要求的资源。在这种状态下，**DataReader** 不再期望任何传入的更新，因此它们也可能为实例释放它们的资源。请注意，当 **Writer** 忘记注销它不再打算更新的实例时，它不仅会泄漏它在本地为其保留的资源，而且还会泄漏该资源。所有订阅 **DataReader** 的用户仍然为其保留资源，以期待将来的更新。

- 最后，如果由于某些默认的 QoS 设置而隐式地释放实例，或者通过特定的 **DataWriter** API 调用显式地释放实例，则不对实例进行处理。不活着的人 ATE 表示实例与系统无关，基本上应该从所有存储中删除。与不活着的 **Writer** 的最大不同之处在于后者只表明了这一点。没有人打算更新实例，也没有提到最后一个已知状态的有效性。

例如，当发布应用程序崩溃时，它可能希望在另一个节点上重新启动，并从它所在的域获取其最后的已知状态。同时，它没有意图使其每个实例的最后已知状态无效，或从其域中的所有存储中删除它们。恰恰相反，它希望最后一个已知的状态能够为后期的联合者所用，所以所以在这种情况下，**Writer** 需要确保它的实例在崩溃后从 **ALIVE** 转换成 **NO\_ALIVE\_NO\_WRITERS**，然后，在重新启动发布应用程序之后，它可能会恢复活动。

另一方面，如果应用程序优雅地终止并希望表明它的实例不再是 DDS 全局数据空间所关心的问题，它可能希望将其实例的状态转到 **NOTTIVE\_DRED**，以便域的其他部分知道它可以安全地清除其所有存储中的所有样本。

## 2. 自动化的 Life-cycle 管理

这里将用实例来说明 Life-cycle 管理。如果查看实例生命周期自动管理中的

代码，并假设这是唯一的应用程序写入数据，则三个写操作的结果是创建三个新的主题实例。在系统中用于与 id=1、2、3 相关联的键值(TempSensorType 在第一章中定义为具有一个名为 id 的属性键)。只要该应用程序正在运行，这些实例将处于活动状态，并将自动向编写器注册(可以说是“关联”)。DDS 的默认行为是将主题实例释放一次。DataWriter 对象被销毁，从而将这些实例引向 NOT\_ALIVE\_DISPOSE 状态。默认设置可以被重写以简单地诱导实例的注销，在这种情况下会导致从 ALIVE 到 NO\_ALIVE\_NO\_WRITERS 的转换。

#### Automatic management of Instance life-cycle

```
#include <thread>
#include <chrono>
#include <TempControl_DCPS.hpp>

int main(int, char**) {
    dds::domain::DomainParticipant dp(org::opensplice::domain::default_id());
    dds::topic::Topic<tutorial::TempSensorType> topic(dp, "TempSensorTopic");
    dds::pub::Publisher pub(dp);

    dds::pub::DataWriter<tutorial::TempSensorType> dw(pub, topic);

    //[NOTE #1]: Instances implicitly registered as part
    // of the write.
    // {id, temp hum scale}
    dw << tutorial::TempSensorType(1, 25.0F, 65.0F, tutorial::CELSIUS);
    dw << tutorial::TempSensorType(2, 26.0F, 70.0F, tutorial::CELSIUS);
    dw << tutorial::TempSensorType(3, 27.0F, 75.0F, tutorial::CELSIUS);

    std::this_thread::sleep_for(std::chrono::seconds(10));

    //[NOTE #2]: Instances automatically unregistered and
    // disposed as result of the destruction of the dw object

    return 0;
}
```

### 3. 显示的 Life-cycle 管理

主题-实例生命周期也可以通过 DataWriter 区上定义的 API 显式地管理。在这种情况下可以完全控制实例何时注册、未注册和释放。

当应用程序经常写入实例并要求最低延迟写入时，主题实例注册是一个很好的做法。本质上，显式注册实例的行为允许中间件保留资源并优化实例查找。Topic 实例注销提供了一种方法，用于告诉 DDS 应用程序是通过编写特定的主题实例来完成的，因此可以安全地释放与本地相关的所有资源。最后，处理主题实例给出了一种与 DDS 通信的方式，即实例不再与分布式系统相关，因此，只要有可能，就会分配给指定的资源。IC 实例应该在本地和远程释放。主题实例生命周期的显式管理展示了 DataWriter API 如何用于注册、注销和释放 Topic 实例。

为了显示完整的生命周期管理，默认 DataWriter 行为已经更改，以便在未注册时不会自动释放实例。此外，为了保持代码紧凑，它利用了新的 c11 自动功能，将其留给编译器从右侧返回类型推断左侧类型。

主题实例生命周期的显式管理显示了一个应用程序，该应用程序编写属于四个不同主题实例的四个示例，分别为 id=1、2、3 的实例。我的例子 D=1、2、3 是通过调用 `DataWriter::RegistryInstant` 方法显式注册的，而 id=0 的实例是自动注册的，这是 `DataWrite` 上写入的结果。

为了显示不同的可能的状态转换，带有 id=1 的主题实例被显式地取消注册，从而导致它转换到 `NOT_ALIVE_NO_WRITER` 状态；主题-实例具有 id=2。显式地释放它，从而导致它转换到 `NOTEAY_RESTED` 状态。最后，具有 id=0，3 的主题实例将被自动取消注册，这是由于销毁了对象分别为 `dw` 和 `dwi3`，对状态 `NOT_ALIVE_NO_WRITER` 的访问。

同样，如前所述，在本例中，`Writer` 已被配置为确保在取消注册时不会自动处理主题实例。

```
#include <iostream>
#include <TempControl_DCPS.hpp>

int main(int, char**) {
    dds::domain::DomainParticipant dp(org::opensplice::domain::default_id());
    dds::topic::Topic<tutorial::TempSensorType> topic(dp, "TempSensorTopic");
    dds::pub::Publisher pub(dp);

    //[NOTE #1]: Avoid topic-instance dispose on unregister
    dds::pub::qos::DataWriterQos dwqos = pub.default_datawriter_qos()
    << dds::core::policy::WriterDataLifecycle::ManuallyDisposeUnregistered();

    //[NOTE #2]: Creating DataWriter with custom QoS.
    // QoS will be covered in detail in article #4.
    dds::pub::DataWriter<tutorial::TempSensorType> dw(pub, topic, dwqos);

    tutorial::TempSensorType data(0, 24.3F, 0.5F, tutorial::CELSIUS);
    dw.write(data);

    tutorial::TempSensorType key;
    short id = 1;
    key.id(id);

    //[NOTE #3] Registering topic-instance explicitly
    dds::core::InstanceHandle h1 = dw.register_instance(key);
    id = 2;
    key.id(id);
    dds::core::InstanceHandle h2 = dw.register_instance(key);
    id = 3;
    key.id(id);
    dds::core::InstanceHandle h3 = dw.register_instance(key);

    dw << tutorial::TempSensorType(1, 24.3F, 0.5F, tutorial::CELSIUS);
    dw << tutorial::TempSensorType(2, 23.5F, 0.6F, tutorial::CELSIUS);
    dw << tutorial::TempSensorType(3, 21.7F, 0.5F, tutorial::CELSIUS);

    // [NOTE #4]: unregister topic-instance with id=1

    dw.unregister_instance(h1);
    // [NOTE #5]: dispose topic-instance with id=2
    dw.dispose_instance(h2);
    //[NOTE #6]:topic-instance with id=3 will be unregistered as
    // result of the dw object destruction

    return 0;
}
```

#### 4. Keyless Topics

正如在主题中所解释的，域和分区无键主题就像单例，在某种意义上是只有一个例子。因此，对于无键主题，状态转换与数据编写器的生命周期相关联。

主题实例生命周期的显式管理。

#### 5. Blocking or Non-Blocking Write?

数据的写入是否会被阻塞是此时可能出现的一个问题。简短的回答是，写是非阻塞的；但是，正如稍后将看到的，在某些情况下，根据设置，写可能会阻塞。在这种情况下，阻塞行为是避免数据丢失所必需。

### 2.3.2. 数据的存取

DDS 提供了一种根据样本的内容和状态选择样本的机制，并提供了另一种机制来控制是否需要读取或提取样本(从缓存中删除)。

#### 1. Read vs. Take

DDS 通过 `DataReader` 类提供数据访问，它有两个共有函数成员：`Read` 和 `Take`。

由 `DataReader::Read` 方法实现的 `Read` 语句允许访问 `DataReader` 接收的数据，而无需将其从缓存中删除。这意味着数据将通过适当的读取调用保持可读性。

由 `DataReader::Take` 方法实现的采取语义允许 DDS 通过从其本地缓存中删除 `DataReader` 来访问 `DataReader` 接收的数据。这意味着一旦数据被获取，不再适用于后续的读取或获取操作。

读和取操作提供的语义使我们能够使用 DDS 作为分布式缓存或类似于队列系统，或者两者兼而有之。这是一个功能强大的组合，很少在同一个中间件平台中找到。这是 DDS 在各种系统中使用的原因之一，有时用作高性能的分布式缓存，有时用作高性能的消息传递技术，而在其他时候则用作梳子。两个人中的一员。此外，在使用主题对分布式状态建模时，读取语义是有用的，在建模分布式事件时使用的是取语义。

#### 2. Data and Meta-Data

本章的第一部分展示了如何使用 `DataWriter` 来控制主题实例的生命周期。主题实例生命周期以及描述接收到的数据示例可供 `DataReader` 使用，并可用于通过 `Read` 和 `Take` 来选择数据访问。具体来说，`DataWriter` 接收的每个数据样本都有一个相关的 `SampleInfo` 描述该示例的属性。这些属性包括以下方面的信息：

- **Sample State**，它的状态可以为 `READ` 或者 `NOT_READ`，这取决于示例是



否已被读取。

- **Instance State**，如前所述，该实例的状态可以为为 `ALIVE`、`NOT_ALIVE_NO_WRITE-RS` 或者 `NOT_ALIVE_DISPOSED`。

- **View State**，视图状态可以是 `NEW`，也可以是 `NOT_NEW`，这取决于这是否是为给定主题实例收到的第一个示例。

`SampleInfo` 还包含一组计数器，这些计数器允许我们确定主题实例执行某些状态转换的次数。

最后，`SampleInfo` 包含数据的时间戳和指示关联数据是否有效的标志。后一个标志很重要，因为 DDS 可能会生成有效的样本。包含无效数据的信息通知有关状态转换的信息，例如正在释放的实例。

### 3. Selecting Samples

无论数据是从 DDS `read` 还是从 DDS 中 `take`，都使用相同的机制来表示示例选择。因此，为了简洁起见，下面的示例使用 `read` 操作；如果想使用 `take` 语句，则只需要将所有出现的 `read` 替换成 `take` 即可。

- **State-based selection** 基于视图状态、实例状态和示例状态的值。
- **Content-based selection** 是基于示例的内容。

#### State-based Selection

例如，要获取接收到的所有数据，无论视图、实例和示例状态如何，使用如下语句 `read(或者 take)`：

```
dds::sub::LoanedSamples<tutorial::TempSensorType> samples;
samples = dr.select().state(dds::sub::status::DataState::any()).read();
```

另一方面，若要只想 `read(或者 take)` 尚未被读取的示例，请按以下方式使用 `read(或者 take)`：

```
samples = dr.select().state(dds::sub::status::SampleState::not_read()).read();
```

若要读取新的有效数据，意味着没有仅包含有效 `SampleInfo` 的示例，请按以下方式使用 `read(或者 take)`：

```
samples = dr.select().state(dds::sub::status::DataState::new_data()).read();
```

最后，若要仅读取首次在系统中出现的实例相关的数据，请按以下方式使用 `read(或者 take)`：

```
dds::sub::status::DataState ds;
ds << dds::sub::status::SampleState::not_read()
  << dds::sub::status::ViewState::new_view()
  << dds::sub::status::InstanceState::alive();

samples = dr.select().state(ds).read();
```

注意：这种读取只能获取到每个实例的第一个示例。

这虽然看上去有些古怪，但对于所有需要在系统中首次出现新实例时都需要执行特殊操作的应用程序来说，这是非常有用的。一个例子可能是一架新的飞机进入一个新的控制区域；在这种情况下，系统需要做一些特定状态转换所特有的事情。

值得一提的是，如果省略了状态，可以这样使用 `read`(或者 `take`) 语句：

```
auto samples2 = dr.read();
```

这相当于使用 `NOT_READ_SAMPLE_STATE`，`ALIVE_INSTANCE_STATE` 以及 `ANY_VIEW_STATE`。

最后，应该指出的是，状态使数据能够根据其元信息进行选择。

### Content-based Selection

通过查询支持基于内容的选择。虽然查询的概念似乎与内容过滤的概念重叠，但其基本思想是不同的。

过滤是控制数据读取器接收的数据：不匹配过滤器的数据没有插入到数据读取器缓存中。另一方面，查询是关于选择数据读取器中已经存在的数据。

#### Content Query

```
// Define the query expression
std::string expression =
    "(temp NOT BETWEEN (%0 AND %1)) \
    OR \
    (hum NOT BETWEEN (%2 and %3))";

// Define the query parameters
std::vector<std::string> params = {"20.5", "21.5", "30", "50"};

dds::sub::Query query(dr, expression, params);

auto samples = dr.select().content(query).read();
```

查询表达式支持的语法与用于定义筛选表达式的语法相同；为了方便起见，在选项卡中对此进行了总结。

#### Legal operators for content query

Constructed Type	Example
<code>=</code>	equal
<code>&lt;&gt;</code>	not equal
<code>&gt;</code>	greater than
<code>&lt;</code>	less than
<code>&gt;=</code>	greater than or equal
<code>&lt;=</code>	less than or equal
<code>BETWEEN</code>	between and inclusive range
<code>LIKE</code>	matches a string pattern

查询的执行完全由用户控制，并在 `read` or `take` 操作的上下文中执行，如 ListingB 所示[ Listing: DDS: Query]



### Instance-based Selection

在某些情况下，我们可能只想查看来自特定主题实例的数据。由于实例是由其关键属性的值标识的，因此可能会倾向于使用内容 `FILT`。要区分他们。虽然这会非常好，但它并不是选择实例的最有效方法。`DDS` 提供了另一种机制，它允许用户比内容过滤更有效地确定感兴趣的实例。本质上，每个实例都有一个相关联的实例句柄；这可以非常有效地用于访问来自给定实例的数据。

基于 listing Instance-based selection 显示了如何做到这一点。

### Instance-based selection

```
tutorial::TempSensorType key;
key.id() = 123;
auto handle = dr.lookup_instance(key);

auto samples = dr.select().instance(handle).read();
```

## 4. Iterators or Containers(迭代器或容器)

到目前为止所显示的例子是从 `DDS` ‘借出’ 数据：换句话说，我们不必为样本提供存储。这种阅读方式的优点是它允许“零拷贝”读取。但是，如果要将数据存储在我们选择的容器中，则可以使用基于迭代器的读取和接受操作。

基于迭代器的读/取 API 既支持前向迭代器，也支持反向插入迭代器。API 允许用户 `read`(或 `take`)任何想要的结构中的数据，只要能够为它找一个前进的迭代器或一个循环迭代器。在这里将重点介绍基于前向迭代器的 API；反向插入非常类似。

```
// Forward iterator using array.
dds::sub::Sample<tutorial::TempSensorType> samples[MAXSAMPLES];
unsigned int readSamples = dr.read(&samples, MAXSAMPLES);

// Forward iterator using vector.
std::vector<dds::sub::Sample<tutorial::TempSensorType> > fSamples(MAXSAMPLES);
readSamples = dr.read(fSamples.begin(), MAXSAMPLES);

// Back-inserting iterator using vector.
std::vector<dds::sub::Sample<tutorial::TempSensorType> > biSamples;
uint32_t readBiSamples = dr.read(std::back_inserter(biSamples));
```

## 5. 阻塞式或非阻塞式 Read/Take

`DDS` 的读取和读取始终是非阻塞的。如果没有可读取的数据，则调用将立即返回。同样，如果数据少于请求，则调用将收集可用的数据并立即返回。读/取操作的非阻塞性质确保了轮询数据的应用程序能够安全地使用这些操作。

### 2.3.3. 等待与响应

与 `DDS` 协调的一种方法是让应用程序轮询数据，经常执行读或取。轮询可能是某些应用程序类的最佳方法，最常见的例子是执行控制循环或循环执行的控制应用程序。但是，一般来说，应用程序可能希望得到数据可用性的通知，或者

可能是能够等待它的可用性，而不是轮询它。DDS 通过等待集和侦听器支持同步和异步协调。

## 1. Waitsets

DDS 提供了一种用于等待条件的通用机制。支持的一种条件是 ReadConditions，它可以用于等待一个或多个 DataReader 上的可用性数据。这个功能由 Waitset 类提供，它可以看作是 unix select 的面向对象版本。

### Using WaitSet to wait for data availability

```
// Create the WaitSet
dds::core::cond::WaitSet ws;
// Create a ReadCondition for our DataReader and configure it for new data

dds::sub::cond::ReadCondition rc(dr, dds::sub::status::DataState::new_data());
// Attach the condition
ws += rc;

// Wait for new data to be available
ws.wait();
// Read the data
auto samples = dr.read();
std::for_each(samples.begin(),
               samples.end(),
               [](const dds::sub::Sample<tutorial::TempSensorType>& s) {
                   std::cout << s.data() << std::endl;
               });
```

如果我们想要等待温度样本可用，我们可以在 DataReader 上创建 ReadCondu，并通过创建 WaitSet 并向其附加 ReadCondu，使其等待新数据，如使用 WaitSet 等待数据可用性所示。

在这一点上，我们可以同步数据的可用性，有两种方法可以做到这一点。一种方法是调用 Waitset::wait 方法，该方法返回活动条件列表。然后对活动条件扫描进行迭代，并可以访问它们相关的数据表。另一种方法是调用 Waitset: 分派，这在单独的示例中得到了演示。

作为自己迭代条件的替代方法，DDS 条件可以与函子对象相关联，然后在条件为 t 时用于执行特定于应用程序的逻辑。被偷了。DDS 均衡化机制允许我们将任何想要的东西绑定到一个事件，这意味着我们可以将一个函数、一个类方法甚至一个 lambda 函数作为一个函子绑定到条件上。

然后以同样的方式将条件附加到 Waitset，但在本例中将调用 Waitset::dispatch 方法，这将导致基础设施在解除阻塞之前自动调用与每个触发条件关联的函子，如使用 WaitSet 向传入数据分派所示。注意，在从 Waitset::dispatch 方法返回之前，函子的执行发生在应用程序线程的上下文中。

**Using WaitSet to dispatch to incoming data**

```

// Create the WaitSet
dds::core::cond::WaitSet ws;
// Create a ReadCondition for our DataReader and configure it for new data
dds::sub::cond::ReadCondition rc(dr,
    dds::sub::status::DataState::new_data(),
    [] (const dds::sub::ReadCondition& srcCond) {
        dds::sub::DataReader<tutorial::TempSensorType> srcReader = srcCond.data_reader();
        // Read the data
        auto samples = srcReader.read();
        std::for_each(samples.begin(),
            samples.end(),
            [] (const dds::sub::Sample<tutorial::TempSensorType>& s) {
                std::cout << s.data() << std::endl;
            });
    });
// Attach the condition
ws += rc;

// Wait for new data to be available
ws.dispatch();

```

**2. 监听器**

另一种发现数据何时被读取的方法是利用 DDS 引发的事件并异步通知注册处理程序。因此，如果我们希望一个处理程序被通知对于数据的可用性，我们将适当的处理程序与 DataReader 引发的 on\_data\_Available 事件连接起来。

**Using a listener to receive notification of data availability**

```

class TempSensorListener :
public dds::sub::NoOpDataReaderListener<tutorial::TempSensorType>
{
public:
    virtual void on_data_available(
        dds::sub::DataReader<tutorial::TempSensorType>& dr) {
        auto samples = dr.read();
        std::for_each(samples.begin(), samples.end(),
            [] (const dds::sub::Sample<tutorial::TempSensorType>& s) {
                std::cout << s.data().id() << std::endl;
            });
    }
};

TempSensorListener listener;
dr.listener(&listener, dds::core::status::StatusMask::data_available());

```

使用侦听器接收数据可用性通知的清单显示了如何做到这一点。NoOpDataReaderListener 是 API 提供的实用工具类，它提供了一个简单的实现作为侦听器的一部分定义的所有操作的命名。这样，就只能覆盖那些与应用程序相关的内容。

值得指出的是，处理程序代码将在中间件线程中执行。因此，在使用侦听器时，应该尽量减少在侦听器中花费的时间。

**2.3.4. 总结**

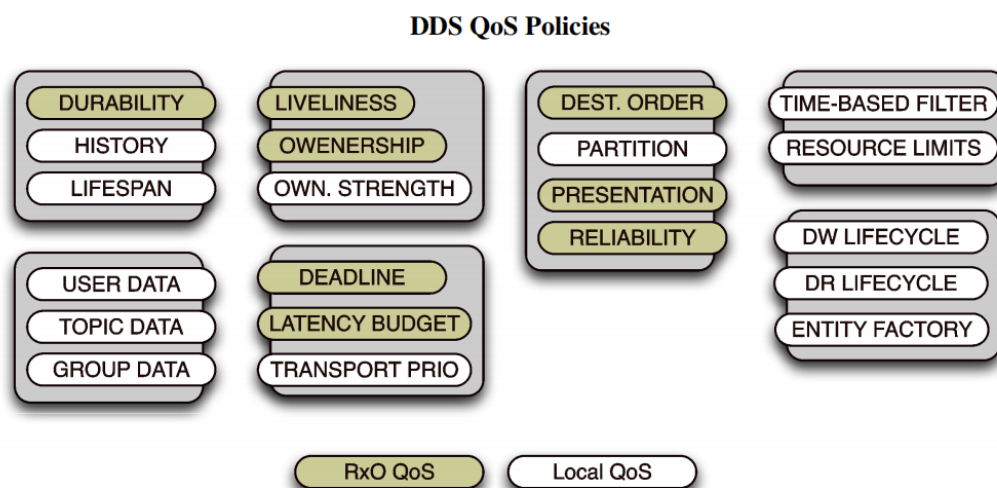
本章介绍了用 DDS 编写和读取数据所涉及的各个方面。它描述了主题-实例生命周期，解释了如何通过 DataWriter 和显示来管理它。对 DataReader 可用的所

有方法进行大小写处理。它解释了等待集和侦听器，以及如何使用它们来接收数据何时可用的指示。

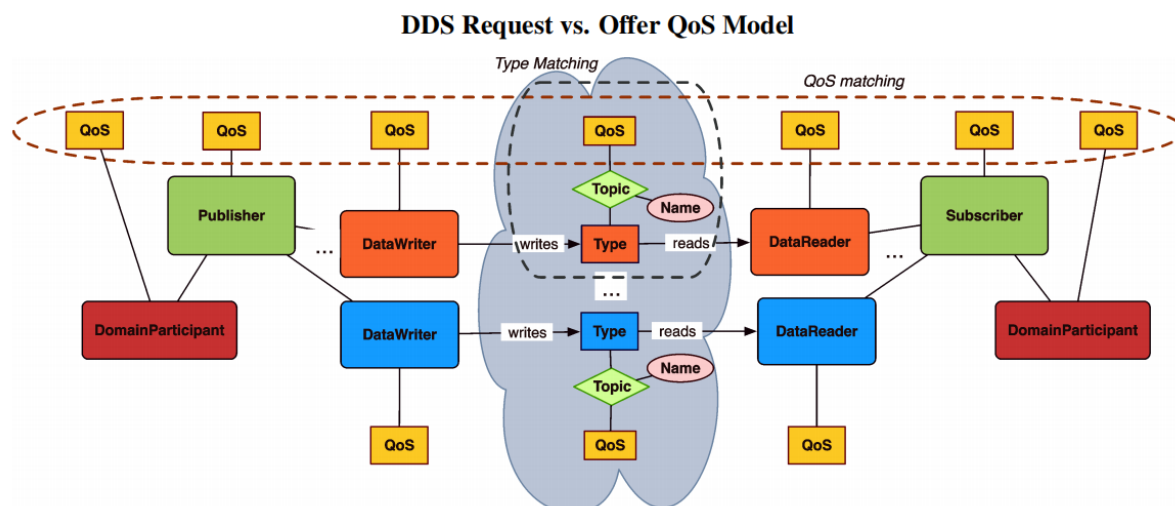
## 2.4. 服务质量策略

### 2.4.1. 数据分发服务的服务质量模型

DDS 提供应用策略来控制广泛的非功能属性，如数据可用性、数据传递、数据及时性和资源使用。



可以通过可用的 QoS 策略来控制实体的语义和行为，例如主题、数据读取器和数据写入器。控制端到端属性的策略是不利的。作为订阅匹配的一部分。



DDS 使用了一种“请求与提供”的 QoS 匹配方法，如图 DDS 请求与提供 QoS 模型所示，其中数据读取器匹配数据写入器的当且仅当只有当它为给定主题

请求的 QoS 不超过数据编写器产生数据的 QoS 时。订阅与主题类型和名称以及数据编写者和读者提供和请求的 QoS 相匹配。这种 DDS 匹配机制确保：

- 由于主题类型匹配，类型被保存在端到端，并且
- 端到端 QoS 不变量也被保留。

本章的其余部分描述了 DDS 中最重要的 QoS 策略。

## 1. 数据有效性

DDS 提供了以下 QoS 策略，用于控制域参与者对数据的有效性：

• 持久性策略控制写入域中全局数据空间的数据的生存期。支撑耐久性水平包括：

– VOLATILE：它规定一旦数据被发布，它就不会由 DDS 维护以交付给迟加入的应用程序；

– TRANSIENT\_LOCAL：它指定发布者在本地存储数据，以便晚加入订阅者在发行者仍然活着的情况下获得最后发布的项目；

– TRANSIENT：确保 GDS 将信息保存在任何出版商的本地范围之外，供迟加入的订阅者使用；

– PERSISTENT：，这确保 GDS 持久地存储信息，以便即使在整系统关闭和重新启动之后，也可以将其提供给后期的连接者。

持久性是通过依赖一个持久性服务来实现的，该服务的属性是通过非易失性主题的持久性服务 QoS 来配置的。

• LIFESPAN QoS 策略控制数据样本有效的时间间隔。默认值是 *infinite*，可选值是可以考虑数据的时间跨度。红色有效。

• HISTORY QoS 策略控制必须为 Reader 或 Writer 存储的数据样本数量(即，相同主题的后续写入)。可能的值是最后一个示例，最后一个 *n* 个示例。或者所有的样本。

这些 DDS 数据可用性 QoS 策略使应用在时间和空间上解耦。它们还使这些应用程序能够在高度动态的环境中进行协作，其特点是连续地加入发布者和订阅者离开。这些属性在系统(SOS)中特别重要，因为它们增加了组件部件的解耦。

## 2. 数据传输

DDS 提供了以下 QoS 策略，这些策略控制数据的传递方式以及发布者如何在数据更新中声明独占权利：

• PRESENTATION QoS 策略控制如何向订阅者显示对信息模型的更改。这种 QoS 控制了数据更新的顺序和一致性。应用它的作用域由访问作用域定义，该范围可以是 INSTANCE， TOPIC， 或 GROUP 级别之一。

- **RELIABILITY** QoS 策略控制与数据扩散相关的可靠性水平。可能的选择是 **RELIABLE** 和 **BEST\_EFFORT** 的分配。

- **PARTITION** QoS 策略控制 DDS 分区(由字符串名称表示)与发布服务器/订阅服务器的特定实例之间的关联。这种关联为 DDS 实现提供了一个抽象, 允许隔离由不同分区生成的通信量, 从而提高了整个系统的可伸缩性和性能。

- **DESTINATION\_ORDER** QoS 策略控制发布者对给定主题的某些实例进行更改的顺序。DDS 允许根据源或目的地排序不同的更改。

- 当有多个 **Writer** 时, **OWNERSHIP** QoS 策略控制 **Writer** ‘拥有’ 对一个主题的写访问, 并且所有权是独占的。只有拥有最高 **OWNERSHIP\_STRENGTH** 的 **Writer** 才能把数据看清楚。如果共享 **OWNERSHIP** QoS 策略值, 则多个 **Writer** 可以同时更新一个主题。因此, **OWNERSHIP** 有助于管理相同数据的复制发布者。

这些 DDS 数据传递 QoS 策略控制数据的可靠性和可用性, 从而允许在正确的时间将正确的数据交付到正确的地点。DDS 内容感知配置文件提供了更详细的选择正确数据的方法, 它允许应用程序根据其内容选择感兴趣的信息。这些 QoS 策略在 SOS 中特别有用, 因为它们可以用来精细地调整数据的传递方式和传递对象, 因此不仅限制了所用资源的数量, 而且还将独立数据的干扰程度降到最低。

### 3. 数据的及时性

DDS 提供了以下 QoS 策略来控制分布式数据的及时性属性:

- **DEADLINE** QoS 策略允许应用程序定义数据的最大到达时间。DDS 可以配置为在错过截止日期时自动通知应用程序。

- **LATENCY\_BUDGET** QoS 策略为应用程序提供了一种向 DDS 通报与传输数据相关的紧迫性的方法。延迟预算指定了 **dds** 必须发布的时间段。传播信息。这个时间段从发布服务器写入数据的那一刻开始, 直到它在订阅服务器的数据缓存中可用, 以便 **Reader** 使用。

- **TRANSPORT\_PRIORITY** QoS 策略允许应用程序控制与主题或主题实例关联的重要性, 从而允许 DDS 实现优先处理更重要的数据。相对于不太重要的数据。这些 QoS 策略有助于确保及时提供重建共享操作画面所需的关键任务信息。

这些 DDS 数据的及时性、QoS 策略提供了对数据的时态属性的控制。这些属性在 SOS 中特别相关, 因为它们可以用于定义和控制时态。各子系统数据交换的各个方面, 同时确保最优地利用带宽。

## 4. 资源

DDS 定义了以下 QoS 策略来控制网络 and 计算资源，这对于满足数据传播的需求是必不可少的：

- **TIME\_BASED\_FILTER** QoS 策略允许应用程序指定数据样本之间的最小到达时间，从而表示它们以最大速率消耗信息的能力。以较快的速度生产的样品不会交付。此策略有助于 DDS 实现优化连接在 1 上的订户的网络带宽、内存和处理能力。模拟带宽网络或计算能力有限的网络。

- **RESOURCE\_LIMITS** QoS 策略允许应用程序控制最大可用存储以保存主题实例和相关数量的历史样本。DDS 的 QoS 策略支持各种构成以网络为中心的关键任务信息管理的元素和操作场景。通过控制这些 QoS 策略，可以将 DDS 从连接窄带和噪声的无线电链路的低端嵌入式系统扩展到连接高速光纤网络的高端服务器。

这些 DDS 资源 QoS 策略提供对本地和端到端资源(如内存和网络带宽)的控制。这些属性在 SOS 中特别重要，因为它们是特征性的。主要由异构的子系统、设备和网络连接组成，这些子系统、设备和网络连接通常需要向下采样，以及对所用资源的总体限制。

## 5. 配置

上面描述的 QoS 策略提供了对数据传递、可用性、及时性和资源使用的最重要方面的控制。DDS 还支持我们的定义和分布。通过以下 QoS 策略指定的引导信息：

- **USER\_DATA** QoS 策略允许应用程序将一系列 octet 关联到域参与者、数据读取器和数据写入器。然后，通过内置的主题分发这些数据。这个 QoS 策略通常用于分发安全凭据。

- **TOPIC\_DATA** QoS 策略允许应用程序将八进制序列与主题关联起来。这些引导信息是通过内置的主题来分发的。这种 QoS 策略的常见用法是用其他信息或元信息(如 IDL 类型代码或 xml 架构)扩展主题。

- **GROUP\_DATA** QoS 策略允许应用程序将一系列 octet 与发布者和订阅者关联起来；此引导信息是通过内置主题分发的。典型用途此信息的目的是允许对订阅匹配进行额外的应用程序控制。

这些 DDS 配置 QoS 策略为在 SOS 中运行的应用程序的引导和配置提供了一种有用的机制。这种机制在 SOS 中特别重要，因为它提供了一个完整的-提供配置信息的分发手段。



## 6. QoS 设置

到目前为止，所有代码示例都依赖于默认的 QoS 设置，因此不必定义所需的 QoS。在 dds 实体上设置 QoS 显示了如何 DDS 系统的 QoS 实现与设置。

### Setting QoS on DDS entities

```
// create a Domain Participant, -1 defaults to value defined in configuration file
dds::domain::DomainParticipant dp(-1);

dds::topic::qos::TopicQos topicQos
    = dp.default_topic_qos()
    << dds::core::policy::Durability::Transient()
    << dds::core::policy::Reliability::Reliable();

dds::topic::Topic<tutorial::TempSensorType> topic(dp, "TempSensor", topicQos);

dds::pub::qos::PublisherQos pubQos
    = dp.default_publisher_qos()
    << dds::core::policy::Partition("building-1:floor-2:room:3");

dds::pub::Publisher pub(dp, pubQos);

dds::pub::qos::DataWriterQos dwqos = topic.qos();
dds::core::policy::TransportPriority transportPriority(10);
dwqos << transportPriority;

dds::pub::DataWriter<tutorial::TempSensorType> dw(pub, topic, dwqos);
```

除了一个用于显式创建 QoS 的 API 之外，DDS 还提供了 QoSProvider 的概念，以便能够将 QoS 的定义具体化并使其成为部署时的关注点。下面的清单显示了如何使用 QoS 提供程序从文件中获取 QoS 定义。

### Setting QoS on DDS entities using the QoSProvider

```
dds::core::QosProvider qp("file://defaults.xml", "DDS DefaultQosProfile");

// create a Domain Participant, -1 defaults to value defined in configuration file
dds::domain::DomainParticipant dp(-1);

dds::topic::qos::TopicQos topicQos = qp.topic_qos();

dds::topic::Topic<tutorial::TempSensorType> topic(dp, "TempSensor", topicQos);

dds::pub::qos::PublisherQos pubQos = qp.publisher_qos();
dds::pub::Publisher pub(dp, pubQos);

dds::pub::qos::DataWriterQos dwqos = qp.datawriter_qos();
dds::pub::DataWriter<tutorial::TempSensorType> dw(pub, topic, dwqos);
```

### 2.4.2. 总结

本章解释了 QoS 在 DDS 中的作用，并说明了如何使用各种策略来控制通信、数据可用性和资源使用等最重要的方面。密码例子也说明了设置是非常简单的，使用 QoSProvider 可以极大地帮助将 QoS 的选择作为一个部署让步器。



### §3. OpenSplice 软件体系架构

OpenSplice 采用了 DDS(数据分发服务)的体系结构。DDS 是一种发布/订阅体系架构，它强调以数据为中心，提供丰富的 QoS 服务质量策略，能保障数据进行实时、高效、灵活地分发，可满足各种分布式实时通信应用需求。

#### 3.1. OMG-DDS

OpenSpliceDDS 是建立在 OMG-DDS 上的第二代模型，在解释 OpenSpliceDDS 前本文先对 OMG-DDS 加以阐释：

OMG-DDS 服务指定了一组连贯的配置文件，每个 DDS 配置文件都添加了定义 DDS 提供的服务级别的不同功能，以实现这种“在正确的时间、正确的地点”的正确数据模式：

1. 最小配置文件(Minimum Profile)-这个基本配置文件利用众所周知的发布/订阅模式，在共享的多个发布者和订阅者之间实现高效的信息传播。对所谓的“话题”感兴趣。主题是用 OMG 的 IDL 语言表示的基本数据结构。此文件还包括 QoS 框架，该框架允许中间件“匹配”请求并提供服务质量参数。

2. 所有权配置文件(Ownership Profile)-此配置文件允许每个发布者表示“强度”，从而为相同信息的复制发布者提供支持，这样只有“最有利”的信息才能提供给有兴趣的各方。

3. 内容订阅配置文件(Content Subscription Profile)-这个“内容意识”配置文件提供了强大的功能来表达对特定信息内容的细粒度兴趣(内容过滤器)。此配置文件还允许应用程序通过使用众所周知的 SQL 语言的子集来指定投影视图和数据聚合以及对订阅的“主题”的动态查询同时保留对信息访问的实时要求。

4. 持久性配置文件(Persistence Profile)-这个“持久性”配置文件提供了透明和容错的“非易失性”数据，这些数据可以表示持久的“设置”或“状态”保留在瞬态发布者的范围之外。

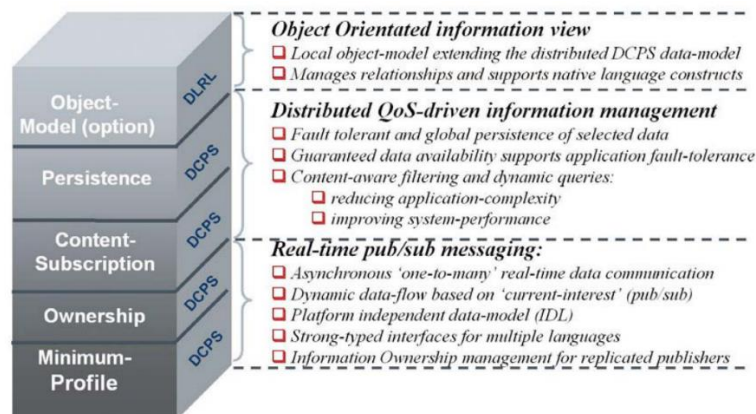


Figure 1 OMG DDS Layers

### 3.2. OpenSplice DDS 组成模块

OpenSpliceDDS 是这一成功产品的第二代 COTS 演进，它由几个模块组成，这些模块涵盖了 OMG 的完整规范，并通过集成的生产力工具套件提供了对整个生命周期的支持——

1. OpenSpliceDDS 核心模块包括提供基本发布-订阅消息功能的“最小”和“所有权”配置文件。最低配置文件的目的是满足实时消息传递的需求，在这些需求中，性能和低占用空间是必不可少的。所有权配置文件为复制的发布者(Publisher)提供了基本的支持，在这种情况下，发布的数据的“所有权”取决于指示发布信息质量的“强度”。

2. OpenSpliceDDS 内容订阅和持久性配置文件提供了附加的信息管理功能，这是确保高信息可用性和强大的“内容感知”功能(消息过滤)的关键。这样就可以为嵌入到大规模容错系统的所有小规模的系统提供无与伦比的性能。

### 3.3. OpenSplice 体系结构

OpenSplice 采用了一种以数据为中心的 P/S(Publish/Subscribe)体系结构(即以数据为中心的发布/订阅通信体系结构)，这种体系结构能在应用程序之间高效地分发数据。网络节点只需要订阅它们想要的信息，或者发布它们能提供的信息。从逻辑上讲，消息是在通信节点间直接传递的。OpenSplice 是一种能够适用于实时操作系统的面向消息的传输中间件，它采用“可插拔”的服务系统，通过共享内存与各个节点上的应用“互联”，亦即通过共享内存来实现数据通讯。

#### 3.3.1. 共享内存

共享内存指在多处理器的计算机系统中，可以被不同中央处理器访问的大容量内存。由于多个 CPU 需要快速访问存储器，这样就要对存储器进行缓存。任何一个缓存的数据被更新后，由于其他处理器也可能要存取，共享内存就需要立即更新，否则不同的处理器可能用到不同的数据。

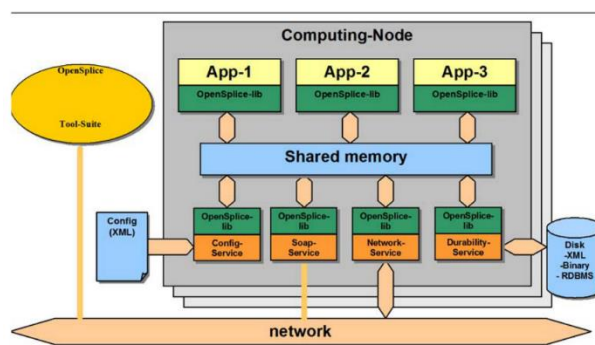


Figure 2 OpenSplice DDS Pluggable Service Architecture

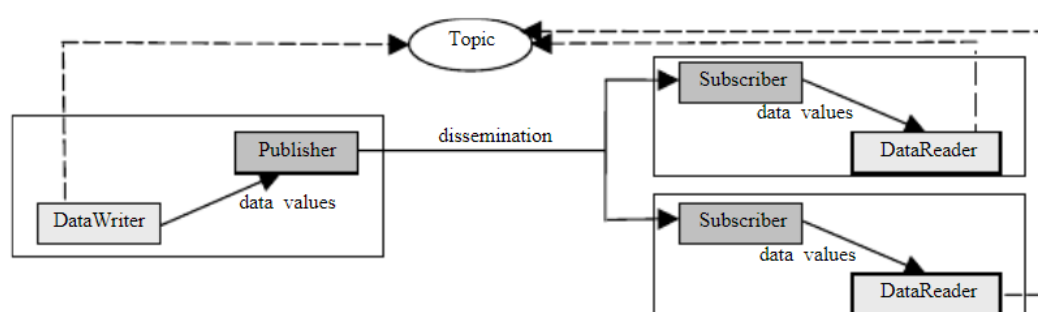
在 OpenSplice 中, 为了确保可伸缩性、灵活性和可扩展性, 其内部有一个架构, 利用共享内存进行“互连”, 它不仅仅连接了驻留在一个计算节点中的所有应用程序, 还包括一个可配置和可扩展的服务集的“主机”。

OpenSpliceDDS 通过使用了共享内存架构, 使数据在任何机器上只显示一次, 而且智能管理仍然为每个订阅者提供自己对这些数据的私人“视图”。这允许订阅者的数据缓存被视为一个单独的“数据库”, 可以进行内容过滤、查询等。这种共享内存体系结构导致极佳的可伸缩性和最佳性能。

OpenSpliceDDS 中间件可以通过指定服务以及将这些服务配置为最优的方式, 轻松地“动态地”配置这些服务。易于维护的 XML 文件用于配置所有 OpenSplice 服务, 还可以通过 MDA 工具集支持 OpenSplice DDS 配置, 允许系统/网络建模和自动生成适当的 xml 配置文件。

### 3.3.2. 以数据为中心的发布订阅模型架构

#### 1. 名词解释



Topic——应用程序利用名称来指定它想要读或写的数据库

Publisher——负责分发数据的对象

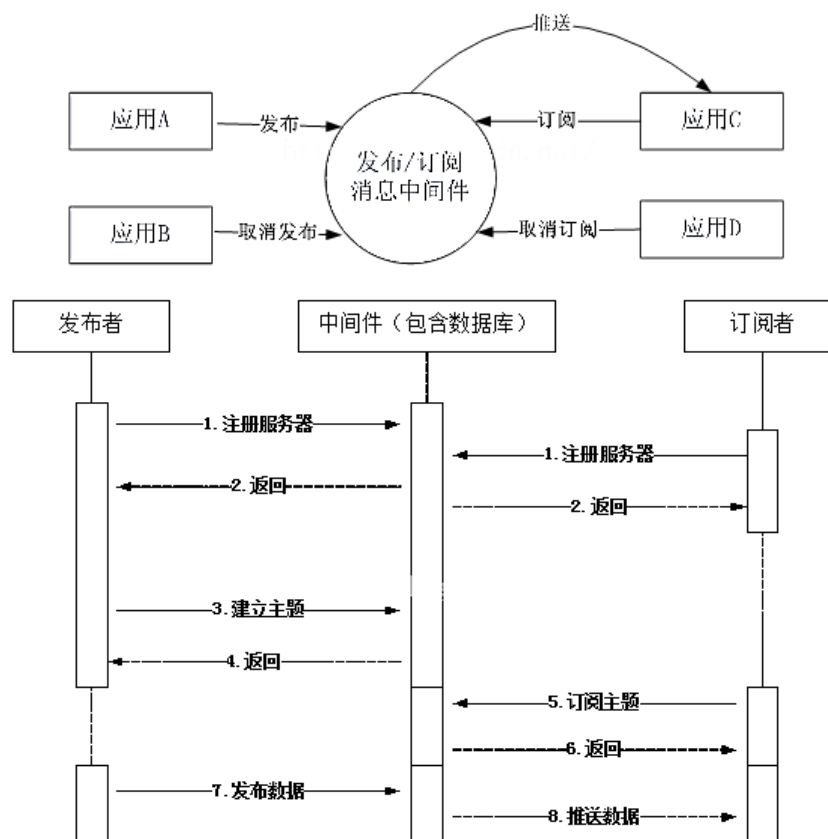
DataWriter——与 Publisher 通信, 专注于一种数据类型

Subscriber——负责接收已发布的数据

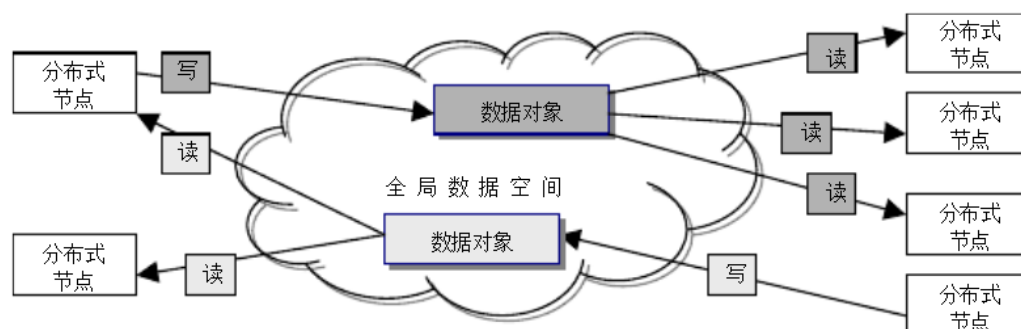
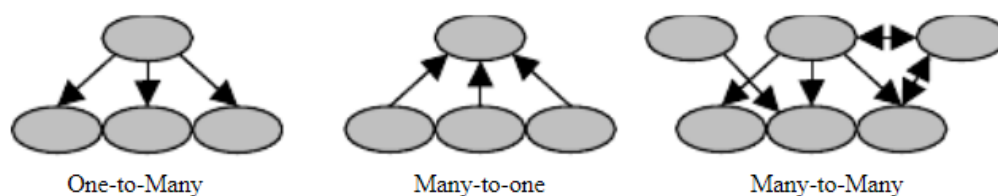
DataReader——应用数据获取 Subscriber 的数据

#### 2. 架构核心

发布者注册它们可以提供的数据库, 而订阅者则订阅它们感兴趣的数据库; 发布者发送数据库时, 只需关心它要传递的特定数据库; 订阅者在接收信息时, 只需知道它想接收的特定数据库。



发布者与订阅者的数量没有限定，可以是一对多、多对一或多对多的任意形式。



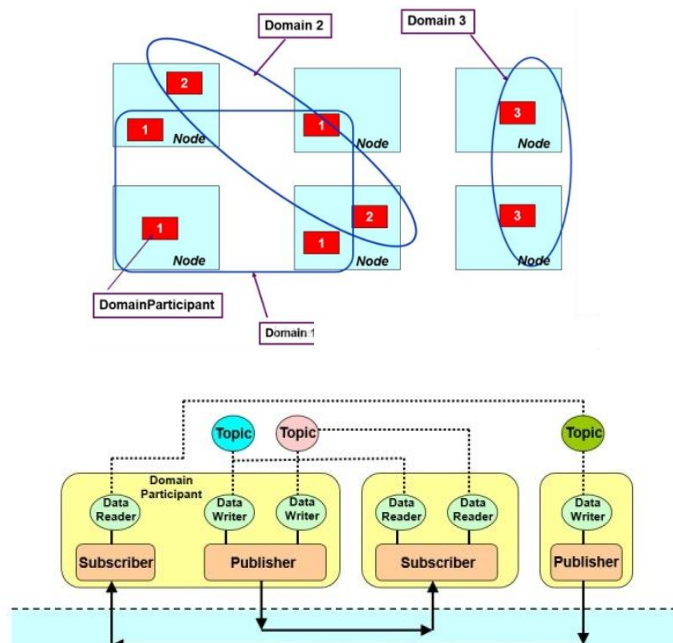
### 3. 模型特点

- i. 低耦合→若干个发布者和订阅者，不严格界定通信角色。
- ii. 正确的数据→订阅者只需要声明感兴趣的信息
- iii. 正确的时间→只要数据有效，发布者可以发布
- iv. 高效率→数据从发布者流向订阅者，不需要服务器参与。

### 4. 在 OpenSpliceDDS 中使用发布订阅体系架构

让我们在了解 DDS 如何使用发布订阅体系架构前先解释三个名词：

- i. Domain: DDS 的基本结构，由域号唯一标识。同一域内实体才能通信。不同域内实体无联系。
- ii. Domain Participant: 作为数据分发服务的入口点，包含若干发布者，订阅者和注册主题，负责创建，删除和管理这些实体。
- iii. DataType: 表示一条信息的定义，通常在 IDL 中声明为结构化数据类型  
主题(topic)=结构化的数据类型+键表+特定 Qos



任何 DDS 应用都需首先获取 domain participant，然后通过 domain participant 获取其他服务，如 publisher，subscriber，topic 等。

发布者(publisher)至少包含一个 DataWriter → 通过 writer()发布消息同时其必须绑定 topic。

订阅者(subscriber)至少包含一个 DataReader 同时其必须绑定 topic。

主题(Topic)是由 datawriter 和 datareader 约定的,要求其需要有确定的 datatype 且通信双方 Qos 必须匹配。

## §4. OpenSplice 设计模式分析

### 4.1. 工厂模式

#### 4.1.1. 工厂模式

工厂模式分为简单工厂、工厂方法和抽象工厂三类。

简单工厂不是一个真正的模式，但它和抽象工厂、工厂方法一样经常被用于封装创建对象的代码。

工厂方法定义了一个创建对象的接口，但由于子类决定实例化的类是哪一个，工厂方法让类把实例化推迟到了子类。

抽象工厂是 OSPL 中用到的最多的工厂模式。抽象工厂模式是所有形态的工厂模式中最为抽象和最具一般性的一种形态。抽象工厂模式是指当有多个抽象角色时，使用的一种工厂模式。抽象工厂模式可以向客户端提供一个接口，使客户端在不必指定产品的具体的情况下，创建多个产品族中的产品对象。根据里氏替换原则，任何接受父类型的地方，都应当能够接受子类型。因此，实际上系统所需要的，仅仅是类型与这些抽象产品角色相同的一些实例，而不是这些抽象产品的实例。换言之，也就是这些抽象产品的具体子类的实例。工厂类负责创建抽象产品的具体子类的实例。

#### 4.1.2. 在 OSPL 中工厂模式的应用

```
void DDSEntityManager::createParticipant(const char *partitionName)
{
    domain = DOMAIN_ID_DEFAULT;
    dpf = DomainParticipantFactory::get_instance();
    checkHandle(dpf.in(), "DDS::DomainParticipantFactory::get_instance");
    participant = dpf->create_participant(domain, PARTICIPANT_QOS_DEFAULT, NULL,
        STATUS_MASK_NONE);
    checkHandle(participant.in(),
        "DDS::DomainParticipantFactory::create_participant");
    partition = partitionName;
}
```

此处是最重要的用例之一。DomainParticipantFactory 用于生产 DomainParticipant，此处用到了工厂模式



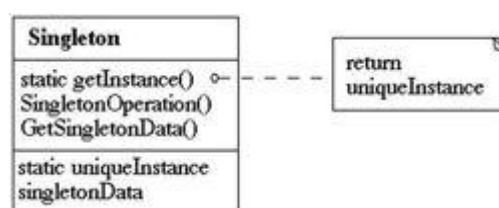
### 4.1.3. 优点

1. 抽象工厂模式隔离了具体类的生产，使得客户并不需要知道什么被创建。
2. 当一个产品族中的多个对象被设计成一起工作时，它能保证客户端始终只使用同一个产品族中的对象。
3. 增加新的具体工厂和产品族很方便，无须修改已有系统，符合“开闭原则”。

## 4.2. 单例模式

### 4.2.1. 单例模式

一种常用的软件设计模式。在它的核心结构中只包含一个被称为单例的特殊类。通过单例模式可以保证系统中，对于应用该模式的类，每一个类中只有一个实例。即一个类只有一个对象实例。



### 4.2.2. 在 OSPL 中单例模式的应用

```
DDS::DomainParticipantFactory_ptr
DDS::DomainParticipantFactory::get_instance(
)
{
    // Static initializer uses the 'Construct on first use' idiom, which is guaranteed to
    // It is not guaranteed to be thread-safe though, although on most compilers it is.
    static DomainParticipantFactory *theFactory = new DDS::DomainParticipantFactory();
    return DDS::DomainParticipantFactory::_duplicate(theFactory);
}
```

### 4.2.3. 优点

#### 1. 实例控制

单例模式会阻止其他对象实例化其自己的单例对象的副本，从而确保所有对象都访问唯一实例。

## 2. 灵活性

因为类控制了实例化过程，所以类可以灵活更改实例化过程。

## 4.3. 代理模式

### 4.3.1. 代理模式

给某一个对象提供一个代理对象，并由代理对象控制对原对象的引用。通俗的来讲代理模式就是我们生活中常见的中介。

代理模式分为远程代理、虚拟代理和保护代理。

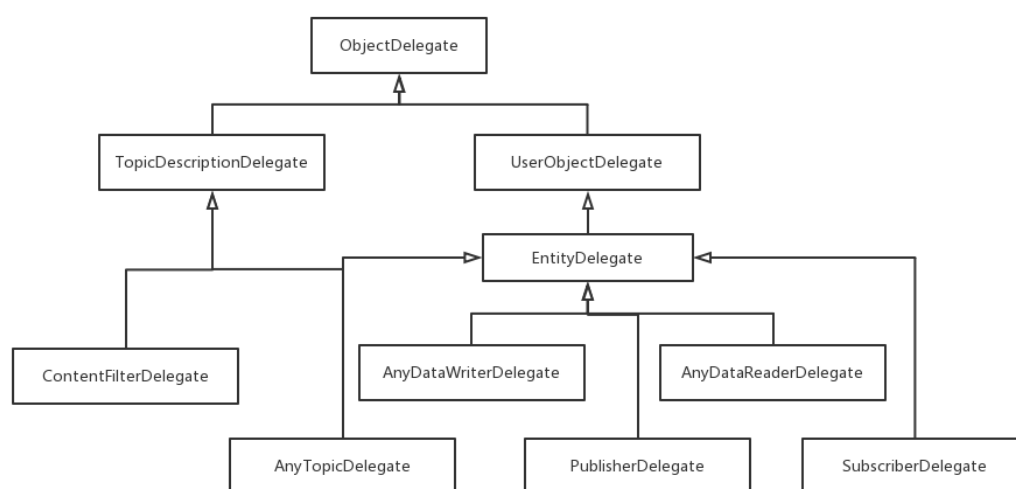
远程代理是远程对象的本地代表。

虚拟代理作为创建开较大的对象的代表，虚拟代理经常直到我们真正需要一个对象的时候才创建它。当对象在创建前和创建中时，由虚拟代理来扮演对象的替身。对象创建后，代理就会将请求直接委托给对象。

保护代理是为一个对象提供一个替身，以便可以访问这个对象。OSPL 中用到的就是这种代理模式。

### 4.3.2. 在 OSPL 中代理模式的应用

代理模式应用有很多，涉及到以下很多类：





这是代理模式在 OSPL 中的底层接口

```
115     template <typename D>
116     □ const D& Value<D>::delegate() const
117     {
118         return d_;
119     }
120
121     template <typename D>
122     □ D& Value<D>::delegate()
123     {
124         return d_;
125     }
```

### 4.3.3. 优点

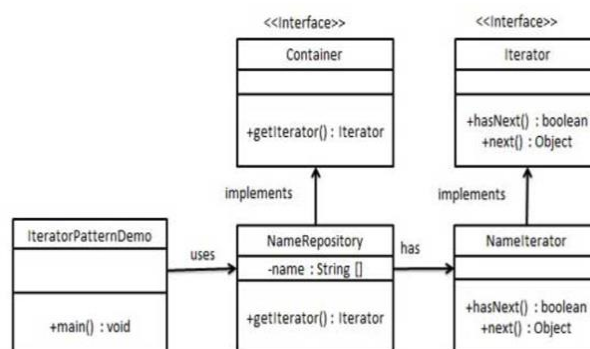
**1. 开闭原则，增加功能：**代理类除了是客户类和委托类的中介之外，我们还可以通过给代理类增加额外的功能来扩展委托类的功能，这样做我们只需要修改代理类而不需要再修改委托类，符合代码设计的开闭原则。代理类本身并不真正实现服务，而是同过调用委托类的相关方法，来提供特定的服务。真正的业务功能还是由委托类来实现，但是可以在业务功能执行的前后加入一些公共的服务。

**2. 中介隔离作用：**在某些情况下，一个客户类不想或者不能直接引用一个委托对象，而代理类对象可以在客户类和委托对象之间起到中介的作用，其特征是代理类和委托类实现相同的接口。

## 4.4. 迭代器模式

### 4.4.1. 迭代器模式

提供一种方法顺序访问一个聚合对象中的各种元素，而又不暴露该对象的内部表示。



#### 4.4.2. 在 OSPL 中迭代器模式的应用

在 OSPL 中 `DataReader` 类提供了两种方法去遍历收到的信息，先序遍历和倒序遍历(主要用于在尾部插入新消息)。

```
// --- Forward Iterators: --- //

template<typename SamplesFWIterator>
uint32_t read(SamplesFWIterator sfit, uint32_t max_samples)
{
    return dr_>read(sfit, max_samples, *this);
}
```

```
// --- Back-Inserting Iterators: --- //

template<typename SamplesBIIterator>
uint32_t read(SamplesBIIterator sbit)
{
    return dr_>read(sbit, *this);
}
```

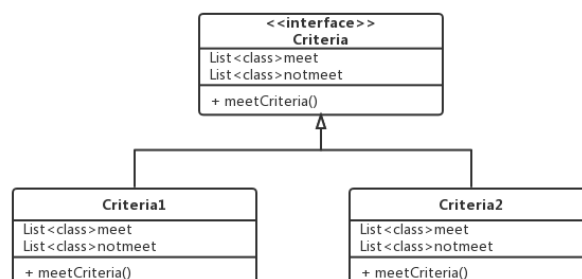
#### 4.4.3. 优点

1. 这种模式用于顺序访问集合对象的元素，不需要知道集合对象的底层表示。
2. 可以提供多种遍历方式。

### 4.5. 过滤器模式

#### 4.5.1. 过滤器模式

也叫标准模式，这种模式允许开发人员使用不同的标准来过滤一组对象，通过逻辑运算以解耦的方式把它们连接起来。



### 4.5.2. 在 OSPL 中过滤器模式的应用

在 ContentFilterTopic 类中，OSPL 提供了两种不同的筛选器分别用于根据不同的参数或表达式筛选需要的信息。

```
public:
    /**
     * Get the filter expression.
     *
     * @return the filter expression
     */
    const std::string& filter_expression() const;

    /**
     * Get the filter expression parameters.
     *
     * @return the filter parameters as a sequence
     */
    const dds::core::StringSeq filter_parameters() const;
```

### 4.5.3. 优点

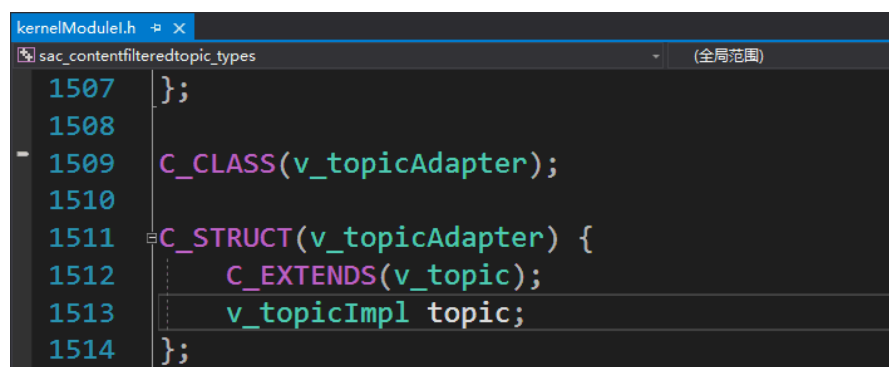
方便依据各种不同条件快速过滤出符合要求的数据信息。

## 4.6. 适配器模式

### 4.6.1. 适配器模式

将一个类的接口转换成客户希望的另一个接口。适配器模式让那些接口不兼容的类可以一起工作。适配器模式可分为类适配器、对象适配器，实际开发中对对象适配器的应用更多。

### 4.6.2. 在 OSPL 中适配器模式的应用



```
kernelModule.h  x
sac_contentfilteredtopic_types  (全局范围)
1507 };
1508
1509 C_CLASS(v_topicAdapter);
1510
1511 C_STRUCT(v_topicAdapter) {
1512     C_EXTENDS(v_topic);
1513     v_topicImpl topic;
1514 };
```

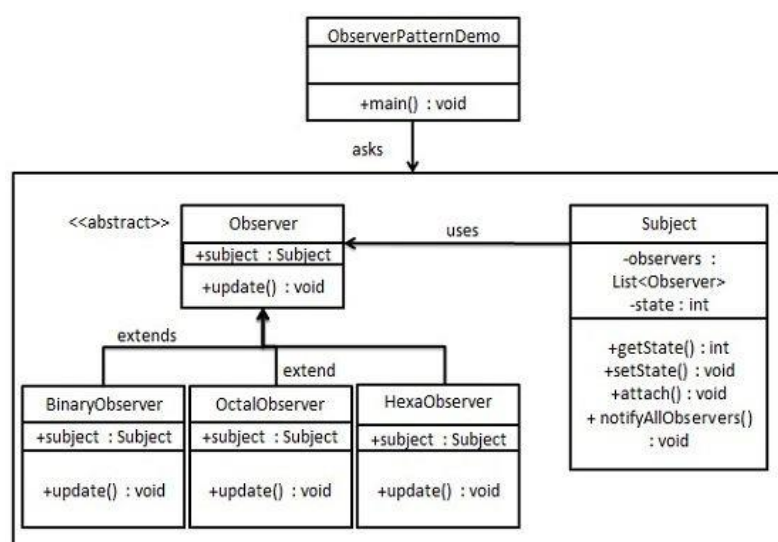
### 4.6.3. 优点

1. 将目标类和适配者类解耦，通过引入一个适配器类来重用现有的适配者类，无需修改原有结构。
2. 增加了类的透明性和复用性，将具体的业务实现过程封装在适配者类中，对于客户端类而言是透明的，而且提高了适配者的复用性，同一适配者类可以在多个不同的系统中复用。
3. 灵活性和扩展性都非常好，通过使用配置文件，可以很方便的更换适配器，也可以在不修改原有代码的基础上 增加新的适配器，完全复合开闭原则。

## 4.7. 观察者模式

### 4.7.1. 观察者模式

当对象间存在一对多关系时，则使用观察者模式。



### 4.7.2. 在 OSPL 中观察者模式的应用

```

C_CLASS(v_observer);

C_STRUCT(v_observer) {
    C_EXTENDS(v_observable);
    c_cond cv;
    c_voidp eventData;
    c_ulong eventFlags;
    c_ulong eventMask;
    c_mutex mutex;
    c_long waitCount;
};
  
```

在 OSPL 中提供了一个观察者(observer), 并被很多其他数据类型继承, 如 v\_entity, v\_statusCondition 等。

```
C_STRUCT(v_statusCondition) { C_STRUCT(v_entity) {  
    C_EXTENDS(v_observer);  
};
```

### 4.7.3. 优点

1. 具体主题和具体观察者是松耦合关系。由于主题接口仅仅依赖于观察者接口, 因此具体主题只是知道它的观察者是实现观察者接口的某个类的实例, 但不需要知道具体是哪个类。同样, 由于观察者仅仅依赖于主题接口, 因此具体观察者只是知道它依赖的主题是实现主题接口的某个类的实例, 但不需要知道具体是哪个类。

2. 观察者模式满足“开-闭原则”。主题接口仅仅依赖于观察者接口, 这样, 就可以让创建具体主题的类也仅仅是依赖于观察者接口, 因此, 如果增加新的实现观察者接口的类, 不必修改创建具体主题的类的代码。同样, 创建具体观察者的类仅仅依赖于主题接口, 如果增加新的实现主题接口的类, 也不必修改创建具体观察者类的代码。