第3章

数据类型和数据样本

不同的语言、不同的编译器、不同的系统和不同的处理器都会造成数据在存储器中的储存或制定的区别。这种语言、编译器、操作系统、处理器的组合被称为平台(platform)。所有现代中间件必须能够从一个特定的平台(例如 C、gcc. 3. 2. 2、Solaris、Sparc)获取数据,并将其透明地传送至其他平台(例如 Java、JDK1. 6、Windows XP、Pentium)。该过程通常被称为"序列化/还原序列化"或"编码/解码"。

在这个问题上,目前的数据传输产品通常采取以下两种方法中的一个:

- Do nothing。信息仅包括不透明的字节流。例如 JMS Bytes-Message。
- Send everything, every time。使用自描述式消息,该方法走向另一个极端,为每个消息嵌入全反射信息,包括数据类型和域名。例如 JMS MapMessage 和 TIBCO Rendezvous 中的消息。

Do nothing 方法表面上减轻了中间件 API 用户的工作量,但迫使他考虑所有数据编码、调整和补零问题。"Send everything, every time"则会造成每个数据包中发送大量的冗余信息,从而影响性能。

DDS 采取一种中间方法。就像应用程序中的对象属于某些数据类型,相同 DDS 主题上发送的数据样本共享一种数据类型。这些类型确定数据样本中存在的域和它们的组成类型。中间件单独储存和传输来自单独数据样本的元信息,允许其在为用户处理字节顺序和调整问题的同时有效地传输样本。

要用 DDS 发布和订阅数据,用户将执行以下步骤:

(1) 选择一种类型描述用户数据。

用户有很多选择,用户可以在这些选项中选择其中一个,或混合、匹配它们。

① 使用中间件提供的内置类型。如果用户的数据类型需要十分简单,这种选择可能已经足够。如果用户的数据结构高度复杂,或用户需要通过过滤或其他目的检查该数据内的域,该选择可能不适合。

② OMG IDL。这种格式是 DDS 和 CORBA 规范的标准组件,它使用类似 C++ 语言的句法描述数据类型。这种格式的介绍参见 3.3 节"使用 IDL 创建用户数据类型"。

③ XML schema (XSD)。独立或嵌入一个 WSDL 文件。对于单独或连接到网络服务基础结构的使用 DDS 的情况,XSD 应成为被选择的格式。

④ DDS 特定格式中的 XML。这种 XML 格式更为简洁,因此,比 XSD 文件更易读取和编写。它既提供了 XML 的扩展性和易整合性的一般优点,也能充分支持 DDS 特定数据类型和概念。

⑤ 在程序运行时定义一个类型标准。这种方法可能适合拥有动态数据描述需要的应

用程序,即类型改变频繁或无法预先知道的应用程序。

(2) 使用逻辑名称注册用户类型。

如果用户已经选择使用内置类型,而不是自定义类型,那么用户可以略过这一步,中间件已经为用户预注册了内置类型。

(3) 使用用户之前注册的类型名称创建一个 Topic。

如果用户已经选定使用内置类型,而不是自定义类型,用户将使用与那种类型名称对应的 API 常量。

创建和使用 Topic 的内容参见第 6 章。

(4) 创建一个或多个 DataWriter 发布用户数据和创建一个或多个 DataReader 订阅它。

这些对象的具体类型取决于用户选定的具体数据类型,目的是为用户提供一种类型安全的方法。

创建与处理 DataWriters 和 DataReaders 的内容分别参见第 7 章和第 8 章。

无论发布还是订阅数据,用户将需要了解如何创建和删除数据样本,以及如何获取和设置它们的域。

3.1 数据类型概述

DDS 使用的用户数据类型可以是任何一种典型的计算机语言数据类型。它可以是一种结构体、一种联合、一种数值类型、一种枚举型数据或一种类型定义(或类同于计算机语言的数据类型)。

用户应用程序可以拥有任何数量的用户数据类型。它们可以包括下列任意原始数据类型或其他用户数据类型。

DDS 仅能直接读取和编写结构体、联合和数值类型。枚举型数据类型、类型定义和原始类型必须包括在结构体、联合或值类型中。为了使 DataReader 和 DataWriter 能够相互沟通,与其相应主题定义相关的数据类型必须完全相同。

- octet(八位字节)、char(字符型)、wchar(宽字符型);
- short(短整型)、unsigned short(无符号短整型);
- long(长整型)、unsigned long(无符号长整型);
- long long(双长整型)、unsigned long long(无符号双长整型);
- float(浮点型);
- double(双精度型)、long double(长双精度型);
- boolean(布尔型);
- enum(枚举型,有或无确切值);
- bounded and unbounded string and wstring(有限和无限字符串和宽字符串)。

也支持下列数据类型的构造:

- 模块(也称为程序包(package)或名称空间(namespace));
- 指针;
- 原始或用户类型元素阵列;

- 有限/无限元素序列[①]——序列是一种可变长度有序采集，例如矢量或列表；
- typedef(类型定义)；
- bitfield(位域)[②]；
- union(联合)；
- struct(结构体)；
- 数值类型(一种支持继承性和其他面向对象特性的复杂类型)。

如果要通过 DDS 使用数据类型，用户必须使用中间件能够明白的方法定义类型，然后在中间件中注册该类型。这些步骤使 DDS 能够进行序列化、还原序列化以及其他针对特定类型的操作。这些内容将在后续章节中详细介绍。

3.1.1　序列

序列是一个对相同数据类型所有元素的有序采集。

C++ 用户将发现序列在概念上与 STL(标准模板库)中的 deque 类别相似。

序列中的元素可以通过其索引得到访问，就像一个阵列中的元素一样。索引以 0 为起始。与阵列不同的是，序列的大小可以扩展。一个序列有两种与之相关的尺寸，一种是物理尺寸(“最大值”)；另一种是逻辑尺寸(“长度”)。物理尺寸指的是当前序列分配，持有的元素的数量；逻辑尺寸是指序列实际持有的有效元素的数量。其长度范围从 0 到最大值，在现有长度范围外，元素不可在索引中访问。

序列可以声明为有限的或无限的。序列的“界限”是其最大值提取的最大数值。界限是非常重要的，因为它允许 DDS 预分配缓存区，从而持有经过序列化和还原序列化的用户类型样本；当与用户的分布式系统中的其他节点进行通信时，这些缓存区将被用到。如果序列不存在界限，DDS 将不清楚分配多大的缓存区，并会因此在单个样本被读取或写入时将其分配为忙碌状态，这将严重影响用户应用程序的反应时间和确定性。因此，DDS 仅支持有限序列，对于 IDL 文件中发现的任何无限序列，其界限将被默认设置为 100 个元素。

3.1.2　字符串和宽字符串

DDS 对包括单字节字符(IDL string 类型)的字符串和包括宽字符的字符串(IDL wstring)均支持。DDS 支持的宽字符串长度为 4 个字节，其长度不仅足够储存两个字节 Unicode/UTF16 字符，也足够储存 UTF32 字符。

和序列一样，字符串也分为有限和无限的。一个字符串的“界限”是指其最大长度(不算 C 和 C++ 中的尾随字符 NULL)。

3.1.3　类型代码

类型代码是一种类型及其域的名字和定义，它以 TypeCode 对象来表示。类型代码值包括类型代码种类(参见下方的 TCKind 列举)和成员列表。对结构体和阵列这种复合数据类型来说，该列表将递归地包括一个或多个类型代码值。

---

① 序列的序列不能得到直接支持。在这种约束下工作，类型定义内部序列和组成新类型的序列。

② 动态数据不支持包括位域成员的数据类型。

```
enum TCKind {
    TK_NULL,
    TK_SHORT,
    TK_LONG,
    TK_USHORT,
    TK_ULONG,
    TK_FLOAT,
    TK_DOUBLE,
    TK_BOOLEAN,
    TK_CHAR,
    TK_OCTET,
    TK_STRUCT,
    TK_UNION,
    TK_ENUM,
    TK_STRING,
    TK_SEQUENCE,
    TK_ARRAY,
    TK_ALIAS,
    TK_LONGLONG,
    TK_ULONGLONG,
    TK_LONGDOUBLE,
    TK_WCHAR,
    TK_WSTRING,
    TK_VALUE,
    TK_SPARSE
}
```

类型代码指定对应类型表达式,与字符串类型名称相比,它提供了更为可靠的检验方式。

模仿相应 CORBA API,TypeCode 类提供对类型代码信息的访问。

3.2 内置数据类型

DDS 提供了一组内置于中间件的标准数据类型。这些数据类型可以直接使用,它们不需要编写 IDL、使用动态类型 API。

DDS 支持的内置数据类型有 String、KeyedString、Octets 和 KeyedOctets(后两种类型在 Java 和. NET 平台上分别被称为 Bytes 和 KeyedBytes)。

在 C++ 和. NET 中,内置数据类型 API 被按照 DDS 名称分配空间。

3.2.1 注册内置类型

在默认情况下,当 DomainParticipant 创建时,内置数据类型自动注册。用户可以通过将 DomainParticipant 的 DDS. builtin_type. auto_register 属性设置为 0(伪造的)改变该行为。

3.2.2　为内置类型创建主题

如果要为内置数据类型创建 Topic，仅需要使用标准 DomainParticipant 操作 create_topic()或 create_topic_with_profile()(参见 6.1.1 节“创建主题”)；对于 type_name 参数而言，使用 get_type_name()操作返回的值，每个 API 的值参见下面。

注意：*在下列示例中，用户将看到标记*<BuiltinType>。

对于 C 和 C++ 而言，<BuiltinType>＝String、KeyedString、Octets 或 KeyedOctets。

C API：

```
const char * DDS_<BuiltinType>TypeSupport_get_type_name();
```

指定空间 C++ API：

```
const char * DDS::<BuiltinType>TypeSupport::get_type_name();
```

无名称空间 C++ API：

```
const char * DDS<BuiltinType>TypeSupport::get_type_name();
```

C++ /CLI API：

```
System::String^ DDS:<BuiltinType>TypeSupport::get_type_name();
```

C# API：

```
System.String DDS.<BuiltinType>TypeSupport.get_type_name();
```

下面是主题创建示例，为了简单起见，下列示例中未显示错误处理。

C 示例：

```
DDS_Topic * topic=NULL;

/ * Create a builtin type Topic * /
topic=DDS_DomainParticipant_create_topic(
                                    participant, "StringTopic",
                                    DDS_StringTypeSupport_get_type_name(),
                                    &DDS_TOPIC_QOS_DEFAULT,NULL,
                                    DDS_STATUS_MASK_NONE);
```

指定空间 C++ 示例：

```
using namespace DDS;
⋮
/ * Create a String builtin type Topic * /
Topic * topic=participant->create_topic(
"StringTopic", StringTypeSupport::get_type_name(),
```

```
DDS_TOPIC_QOS_DEFAULT,
NULL, DDS_STATUS_MASK_NONE);
```

C++ /CLI 示例：

```
using namespace DDS;
⋮

/* Create a builtin type Topic */
Topic^ topic=participant->create_topic(
"StringTopic", StringTypeSupport::get_type_name(),
DomainParticipant::TOPIC_QOS_DEFAULT,
                    nullptr, StatusMask::STATUS_MASK_NONE);
```

C# 示例：

```
using namespace DDS;
⋮

/* Create a builtin type Topic */
Topic topic=participant.create_topic(
"StringTopic",  StringTypeSupport.get_type_name(),
                      DomainParticipant.TOPIC_QOS_DEFAULT,
                      null, StatusMask.STATUS_MASK_NONE);
```

3.2.3 字符串内置类型

字符串内置类型的表达式在 C 和 C++ 中为 NULL 终止字符阵列(char *)，在 Java 和 .NET中为不变的字符串对象。这种类型可用于发布和订阅单独字符串。

1. 创建和删除字符串

在 C 和 C++ 中，DDS 提供了一组创建(DDS::String_alloc())、释放(DDS::String_free())和复制(DDS::String_dup())字符串的操作。

当在复制模式中读取/提取将字符串序列作为参数的操作时，如果序列内的字符串要素被初始化为 NULL，DDS 将为其分配内存。

如果元素未被初始化为 NULL，行为将取决于语言：

- 在 Java 和. NET 中，元素的相关内存将被重新分配至每个样本，因为字符串是不可变对象。
- 在 C 和 C++ 中，元素的相关内存必须足够容纳接收数据。内存不足会导致崩溃。

当在 C 和 C++ 中使用 take_next_sample()和 read_next_sample()时，用户必须确保输入的字符串拥有足够的内存，能够持续地接收数据。内存不足会导致崩溃。

2. 字符串数据写入者

字符串 DataWriter API 匹配标准 DataWriter API，不存在扩展。
以下例显示如何使用字符串内置类型 DataWriter 编写简单字符串。
指定空间 C++ 例：

```
#include "nDDS/nDDS_namespace_cpp.h"
using namespace DDS;
⋮

StringDataWriter * stringWriter=…;

/ * Write some data * /
ReturnCode_t  retCode=stringWriter->write(
"Hello World!", HANDLE_NIL);
char * str=DDS::String_dup("Hello World!");
retCode=stringWriter->write(str, HANDLE_NIL);

DDS::String_free(str);
```

3. 字符串数据读取者

字符串 DataReader API 匹配标准 DataReader API，不存在扩展。
以下例显示如何使用复制模式字符串内置类型 DataReader 读取简单字符串。
指定空间 C++ 例：

```
#include "nDDS/nDDS_namespace_cpp.h"
using namespace DDS;
⋮

StringSeq dataSeq;
SampleInfoSeq infoSeq;
StringDataReader * stringReader=…;

/ * Take a print the data * /
ReturnCode_t retCode=stringReader->take(dataSeq, infoSeq,
                                        LENGTH_UNLIMITED,
                                        ANY_SAMPLE_STATE,
                                        ANY_VIEW_STATE,
                                        ANY_INSTANCE_STATE);
for (int i=0; i <data_seq.length(); ++i) {
    if (infoSeq[i].valid_data) {
        StringTypeSupport::print_data(dataSeq[i]);
    }
```

```
}
/* Return loan */
retCode=stringReader->return_loan(dataSeq, infoSeq);
```

3.2.4 关键字字符串内置类型

关键字字符串内置类型成对表达(关键字、值),此处关键字和值是字符串。该类型可用于发布和订阅关键字字符串,该类型的语言特定表达式如下。

C/C++ 表达式(无名称空间):

```
struct DDS_KeyedString {
    char * key;
    char * value;
};
```

1. 创建和删除关键字字符串

DDS 提供了一组创建/删除关键字字符串的构造函数/析构函数。

如果用户想控制 C/C++ 的关键字字符串结构中的“值”和“关键字”的内存,使用 DDS::String_alloc()、DDS::String_dup()和 DDS::String_free()操作。

2. 关键字字符串数据写入者

关键字字符串 DataWriter API 可通过以下方法得到扩展。

```
DDS::ReturnCode_t DDS::KeyedStringDataWriter::dispose(
                        const char * key,
                        const DDS::InstanceHandle_t * instance_handle);
```

```
DDS::ReturnCode_t DDS::KeyedStringDataWriter::dispose_w_timestamp(
                        const char * key,
                        const DDS::InstanceHandle_t * instance_handle,
                        const struct DDS::Time_t * source_timestamp);
```

```
DDS::ReturnCode_t DDS::KeyedStringDataWriter::get_key_value(
                        char * key,
                        const DDS::InstanceHandle_t * handle);
```

```
DDS::InstanceHandle_t DDS::KeyedStringDataWriter::lookup_instance(
                            const char * key);
```

```
DDS::InstanceHandle_t DDS::KeyedStringDataWriter::register_instance(
                            const char * key);
```

```
DDS::InstanceHandle_t
DDS_KeyedStringDataWriter::register_instance_w_timestamp(
                            const char * key,
                            const struct DDS_Time_t * source_timestamp);
```

```
DDS::ReturnCode_t
DDS::KeyedStringDataWriter::unregister_instance(
                             const char * key,
                             const DDS::InstanceHandle_t * handle);
```

```
DDS::ReturnCode_t
DDS::KeyedStringDataWriter::unregister_instance_w_timestamp(
                             const char * key,
                             const DDS::InstanceHandle_t * handle,
                             const struct DDS::Time_t * source_timestamp);
DDS::ReturnCode_t DDS::KeyedStringDataWriter::write (
                        const char * key,
                        const char * str,
                        const DDS::InstanceHandle_t * handle);
```

```
DDS::ReturnCode_t DDS::KeyedStringDataWriter::write_w_timestamp(
                        const char * key,
                        const char * str,
                        const DDS::InstanceHandle_t * handle,
                        const struct DDS::Time_t * source_timestamp);
```

这些操作用于为数据写入者和实例管理操作输入的参数的格式提供最大的灵活性。对于其他信息和所有支持语言中操作的完整描述参见网上的相关文献。

以下例显示如何使用关键字字符串内置类型 DataWriter 和一些扩展 API，编写关键字字符串。

指定空间 C++ 例：

```
#include "nDDS/nDDS_namespace_cpp.h"
using namespace DDS;
⋮

KeyedStringDataWriter * stringWriter=…;

/ * Write some data using the KeyedString * /
KeyedString * keyedStr=new KeyedString(255, 255);
strcpy(keyedStr->key, "Key 1");
strcpy(keyedStr->value, "Value 1");

ReturnCode_t  retCode=stringWriter->write(keyedStr, HANDLE_NIL);
```

```
delete keyedStr;

/* Write some data using individual strings */
retCode=stringWriter->write("Key 1", "Value 1", HANDLE_NIL);

char* str=String_dup("Value 2");

retCode=stringWriter->write("Key 1", str, HANDLE_NIL);

String_free(str);
```

3. 关键字字符串数据读取者

关键字字符串 DataReader API 可通过以下方法得到扩展。

```
DDS::ReturnCode_t DDS::KeyedStringDataReader::get_key_value(
                        char* key,
                        const DDS::InstanceHandle_t* handle);
```

```
DDS::InstanceHandle_t DDS::KeyedStringDataReader::lookup_instance(
                            const char* key);
```

其他信息和所有支持语言中操作的完整描述参见网上的相关文献。

复制操作中内存的考虑：

(1) 对于使用复制语义进行的读取/提取操作，例如 read_next_sample()和 take_next_sample()，如果初始化为 NULL，DDS 将为域“值”和“关键字”分配内存。

(2) 如果域未被初始化为 NULL，行为将取决于语言。

- 在 Java 和.NET 中，域“值”和“关键字”相关内存将被重新分配至每个样本。
- 在 C 和 C++ 中，域“值”和“关键字”相关内存必须足够容纳接收数据。内存不足会导致崩溃。

以下例显示如何使用关键字字符串内置类型 DataReader 读取关键字字符串。

指定空间 C++ 例：

```
#include "nDDS/nDDS_namespace_cpp.h"
using namespace DDS;
⋮

KeyedStringSeq dataSeq;
SampleInfoSeq infoSeq;
KeyedStringDataReader* stringReader=…;

/* Take a print the data */
ReturnCode_t retCode=stringReader->take(dataSeq, infoSeq,
```

```
        LENGTH_UNLIMITED,
        ANY_SAMPLE_STATE,
        ANY_VIEW_STATE,
        ANY_INSTANCE_STATE);
    for (int i=0; i <data_seq.length(); ++i) {
        if (infoSeq[i].valid_data) {
            KeyedStringTypeSupport::print_data(&dataSeq[i]);
        }
    }
    /* Return loan */
    retCode=stringReader->return_loan(dataSeq, infoSeq);
```

DDS 的其他内置数据类型可以参考网上的相关文献。

3.2.5　管理内置数据类型的内存

当编写一个样本时，DataWriter 将序列化该样本，并将结果储存在预分配缓冲池的一个缓存区中。相同的方式，当接收一个样本时，DataWriter 将其还原序列化，并将结果储存在预分配样本池的一个样本中。

对于内置数据类型来说，缓存区/样本的最大长度是未知的，其取决于使用内置类型的应用程序的性质。例如，与使用相同内置数据类型发布市场数据值的市场数据应用程序相比，使用 keyed octets 内置类型发布图像流的视频监控应用程序将需要更大的缓存区。

如果要在适应应用程序类型的同时优化内存使用，用户可以使用 DDS 提供的 QoS 策略对内存进行管理，根据每个单独 DataWriter 或 DataReader 的基本需要配置内置类型的最大长度。有关内置数据类型内存管理的详细说明，可以参考网上的相关文献。

表 3-1 列出了 DDS 支持的内置类型及其属性。当属性在 DomainParticipant 中得到定义时，它们可以应用于属于 DomainParticipant 的所有 DataWriter 和 DataReader 中，除非它们在 DataWriter 和 DataReader 中被更新。

表 3-1　为每个 DataWriter 和 DataReader 分配内置数据类型大小设置属性

内置类型	属　性	描　述
string	DDS. builtin_type . string. alloc_size	由 DataWriter 发布或由 DataReader 接收的 string 的最大值(包括 NULL 终止字符) 默认值：如定义，为 DDS. builtin_type. string. max_size，否则为 1024
keyed-string	DDS. builtin_type . keyed_string. alloc_ key_size	DataWriter 或 DataReader 使用的关键字的最大值(包括 NULL 终止字符)。 默认值：如定义，为 DDS. builtin_type. keyed_string. max_key_size，否则为 1024
	DDS. builtin_type . keyed_string. alloc_ size	由 DataWriter 发布或由 DataReader 接收的 string 的最大值(包括 NULL 终止字符)。 默认值：如定义，为 DDS. builtin_type. keyed_string. max_size，否则为 1024

续表

内置类型	属　　性	描　　述
octets	DDS. builtin_type . octets. alloc_size	由 DataWriter 或 DataReader 发布的 octet 序列的最大值。 默认值：如定义，为 DDS. builtin_type. octets. max_size，否则为 2048
keyed-octets	DDS. builtin_type . keyed_octets. alloc_ key_size	由 DataWriter 发布或由 DataReader 接收的键的最大值（包括 NULL 终止字符）。 默认值：如定义，为 DDS. builtin_type. keyed_octets. max_key_size，否则为 1024
	DDS. builtin_type . keyed_octets. alloc_ size	由 DataWriter 或 DataReader 发布的 octet 序列的最大值。 默认值：如定义，为 DDS. builtin_type. keyed_octets. max_size，否则为 2048

注意：这些属性必须与 DomainParticipant 中对应的 *. max_size 属性一致。alloc_size 属性的值必须小于或等于在 DomainParticipant 中拥有相同名称前缀的 max_size 的属性。

用户也可以使用 XML QoS 配置文件设置内置数据类型的最大值。例如，以下 XML 代码显示了如何为 DataWriter 设置 string 内置数据类型的最大值。

```
<DDS>
    <qos_library name="BuiltinExampleLibrary">
        <qos_profile name="BuiltinExampleProfile">
            <datawriter_qos>
                <property>
                    <value>
                        <element>
                        <name>DDS.builtin_type.string.alloc_size</name>
                        <value>2048</value>
                        </element>
                    </value>
                </property>
            </datawriter_qos>
            <datareader_qos>
                <property>
                    <value>
                        <element>
                        <name>DDS.builtin_type.string.alloc_size</name>
                        <value>2048</value>
                        </element>
                    </value>
                </property>
            </datareader_qos>
        </qos_profile>
    </qos_library>
</DDS>
```

下面是一个例，为一个字符串设置最大长度。

指定空间 C++ 例：

```
#include "nDDS/nDDS_namespace_cpp.h"
using namespace DDS;
⋮
Publisher * publisher=…;
Topic * stringTopic=…;
DataWriterQos writerQos;

ReturnCode_t retCode=participant->get_default_datawriter_qos(writerQos);

retCode=PropertyQosPolicyHelper::add_property (
    &writerQos.property,
    "DDS.builtin_type.string.alloc_size", "1000",
    BOOLEAN_FALSE);

DataWriter * writer=publisher->create_datawriter(
                                        stringTopic, writerQos,
NULL, STATUS_MASK_NONE);
StringDataWriter * stringWriter=StringDataWriter::narrow(writer);
```

3.2.6　内置数据类型的类型代码

与内置数据类型相关的类型代码是由以下 IDL 类型定义生成的：

```
module DDS {
/ * String * /
struct String {
string<max_size>value;
};
/ * KeyedString * /
struct KeyedString {
string<max_size>key;                    //@key
string<max_size>value;
};
/ * Octets * /
struct Octets {
sequence<octet, max_size>value;
};
/ * KeyedOctets * /
struct KeyedOctets {
string<max_size>key;                    //@key
sequence<octet, max_size>value;
};
};
```

包括在类型代码定义中的字符串和序列的最大大小(max_size)可通过使用表 3-2 中的属性按照每个 DomainParticipant 的实际情况进行配置。

表 3-2 为每个 DomainParticipant 分配内置数据类型大小设置属性

内置类型	属 性	描 述
string	DDS. builtin_type. string. max_size	由 DataWriter 发布或由 DataReader 接收的属于一个 DomainParticipant 的 string 的最大值(包括 NULL 终止字符)。 默认值：1024
keyed-string	DDS. builtin_type. keyed_string. max_key_size	DataWriter 和 DataReader 使用的属于一个 DomainParticipant 的键的最大值(包括 NULL 终止字符)。 默认值：1024
	DDS. builtin_type. keyed_string. alloc_size	由 DataWriter 发布或由 DataReader 接收的属于一个使用内置类型 DomainParticipant 的 string 的最大值(包括 NULL 终止字符)。 默认值：1024
octets	DDS. builtin_type. octets. alloc_size	DataWriter 和 DataReader 发布的属于一个 DomainParticipant 的 octet 序列的最大值(包括 NULL 终止字符)。 默认值：1024
keyed-octets	DDS. builtin_type. keyed_octets. alloc_key_size	由 DataWriter 发布或由 DataReader 接收的属于一个 DomainParticipant 的关键字的最大值(包括 NULL 终止字符)。 默认值：1024
	DDS. builtin_type. keyed_octets. alloc_size	DataWriter 和 DataReader 发布的属于一个 DomainParticipant 的 octet 序列的最大值(包括 NULL 终止字符)。 默认值：2048

3.3 使用 IDL 创建用户数据类型

用户可使用 IDL(接口定义语言)在文本文件中创建用户数据类型。IDL 不依赖于编程语言,所以,相同文件可用于在 C、C++ 、C++ /CLI 和 Java 中生成代码。

DDS 仅使用 IDL 句法的一个子集。IDL 最初由 OMG 定义,以便在企业设置中使用 CORBA 客户/服务器应用程序。在高性能、以数据为中心的嵌入式应用程序环境下,并非所有可用语言描述的结构体都可用。这包括定义诸如 interface 这样的方法和函数原型的结构体。

IDL 规范保留了一些关键字,参见表 3-3。

表 3-3 保留的 IDL 关键字

IDL 关键字				
abstract	emits	local	pseudo	typeid
alias	enum	long	public	typename
any	eventtype	mirrorport	publishes	typeprefix
attribute	exception	module	raises	union

续表

IDL 关键字				
boolean	factory	multiple	readonly	unsigned
case	false	native	sequence	uses
char	finder	object	setraises	valuebase
component	fixed	octet	short	valuetype
connector	float	oneway	string	void
const	getraises	out	struct	wchar
consumes	home	port	supports	wstring
context	import	porttype	switch	
custom	in	primarykey	true	
default	inout	private	truncatable	
double	interface	provides	typedef	

对于 C 和 C++ 来说，DDS 对原始类型使用 typedef，而不是语言关键字。例如，DDS_Long 代替 long，或者 DDS_Double 代替 double。这确保了无论采用何种平台[①]，类型的大小都是相同的。

3.3.1　可变长度类型

当 DDS 用可变长度类型为数据结构体生成代码时（字符串和序列），它包括创建、初始化和终结（破坏）那些对象的函数。这些支持函数将恰当地初始化指针，并分配和解除分配可变长度类型使用的内存。所有 DDS API 假设传送给它们的数据结构都得到恰当的初始化。

对于可变长度类型来说，在编写样本时（无论类型是否具有硬界限），数据以实际长度（代替最大长度）在导线上传输。

1. 序列

C、C++、C++/CLI 和 C# 用户可以分配很多不同来源的内存，例如来自堆阵、堆栈或来自某种传统的分配符。在这些语言中，序列提供了内存“所有权（ownership）”的概念。序列可以拥有分配给它的内存或从其他来源借来的内存。如果序列拥有其内存，它将自我管理其下的储存器缓存区。当序列的最大长度改变时，序列将按照需要释放和重新分配缓存区。然而，如果序列是由用户代码使用借用的内存创建的，那么其内存不属于自己，无法释放或重新分配。因此，用户无法为借用内存的序列设置最大值。

如上所述，在 IDL 中，序列可以声明为有限或无限。序列的“界限”是其最大大小的最

① 对于每个数据类型，线上发送的字节数是由 CDR（公共数据表达式）标准决定的。关于 CDR 的详细内容，参见 CORBA（公共对象请求代理结构）规范 3.1 版。

大值。

2. 字符串和宽字符串

对于 Java 和.NET 用户来说,一个 IDL 字符串是一个 String 对象:它是不可变的,且清楚所拥有的长度。然而,C 和 C++ 用户必须小心,因为没有办法确定多少内存被分配至一个字符指针"string";能确定的是字符串当前的逻辑长度。在某些情况下,DDS 可能需要将一个字符串复制到用户代码提供的结构体中。DDS 不会释放提供给它的字符串内存,因为它不知道内存由何处分配而来。

因此,在 C 和 C++ API 中,DDS 执行以下约定:

- 一个字符串的内存归包括该字符串的结构体"所拥有"。调用一个为类型提供的终结函数将释放所有递归包括的字符串。如果用户以特殊方式分配了一个有限的字符串,用户必须在调用类型的 finalize()方法之前小心地清理其所拥有的内存并将指针分配至 NULL,这样,DDS 将略过该字符串。
- 用户必须为 DDS 提供一个非 NULL 字符串指针,以方便其复制,否则 DDS 无法记录错误。
- 当用户在其数据结构中提供了一个非 NULL 字符串指针,DDS 将在无须执行任何额外内存分配的情况下复制到提供的内存。注意,如果用户为 DDS 提供了未经初始化的指针或分配的字符串过短,用户可能会造成内存损坏或程序崩溃。DDS 永远不会试图复制长度超过目标字符串长度的字符串。然而,用户应用程序必须确保它分配的所有字符串足够长。

3.3.2 值类型

值类型与结构体相似,但是它提供了对额外面向对象特性的支持,例如继承性。它与 Java 对象相似。

在 CORBA 环境中熟悉值类型的读取者可参见表 3-4,了解哪些值类型相关的 IDL 关键词能够得到支持,以及在 DDS 环境下它们的行为。

表 3-4 值类型支持

方　　面	DDS 中的支持水平
继承性	来自其他值类型的单一继承性
公有成员	支持
私有成员	代码生成时成为公有成员
自定义关键字	忽略(值类型在没有关键词的情况下得到解析,相关代码生成)
抽象值类型	无代码生成(值类型得到解析,但无代码生成)
操作	无代码生成(值类型得到解析,但无代码生成)
Truncatable 关键词	忽略(值类型在没有关键词的情况下得到解析,相关代码生成)

3.4　与用户数据类型动态互动

3.4.1　类型代码概述

数据类型格式是一个数据类型及其域的名称和定义，它由 TypeCode 对象表达。一个数据类型代码值包括一个数据类型代码类（参见下面的 TCKind 枚举）和成员列表。对于诸如结构体和阵列这样的复合类型来说，该列表将递归地包括一个或多个数据类型代码值。

```
enum TCKind {
    TK_NULL,
    TK_SHORT,
    TK_LONG,
    TK_USHORT,
    TK_ULONG,
    TK_FLOAT,
    TK_DOUBLE,
    TK_BOOLEAN,
    TK_CHAR,
    TK_OCTET,
    TK_STRUCT,
    TK_UNION,
    TK_ENUM,
    TK_STRING,
    TK_SEQUENCE,
    TK_ARRAY,
    TK_ALIAS,
    TK_LONGLONG,
    TK_ULONGLONG,
    TK_LONGDOUBLE,
    TK_WCHAR,
    TK_WSTRING,
    TK_VALUE,
    TK_SPARSE
}
```

数据类型代码指定匹配数据类型表达式，并提供一个比字符串类型更可靠的数据格式。TypeCode 类模仿 CORBA API，提供对类型代码信息的访问。

注意：如果用户将以默认 SQL 过滤器使用 ContentFilteredTopics，type-code 支持必须得到启用。用户可以禁用数据类型代码和使用自定义过滤器，具体描述可参见创建 ContentFilteredTopics 这部分内容。

3.4.2 定义新类型

创建 TypeCode 与用户静态定义数据类型的方式类似：用户使用一些名称定义数据类型本身，然后为其添加成员，每个成员都拥有自己的名称和类型。

例如，考虑以下静态定义的类型。它可运用在 C、C++ 或 IDL 中，句法大致相同。

```
struct MyType {
    long my_integer;
    float my_float;
    bool my_bool;
string<128>my_string;                          //@key
};
```

以下是用户在 C++ 中定义相同数据类型的方法：

```
DDS_ExceptionCode_t ex=DDS_NO_EXCEPTION_CODE;
DDS_StructMemberSeq structMembers;             // ignore for now
DDS_TypeCodeFactory * factory=DDS_TypeCodeFactory::get_instance();
DDS_TypeCode * structTc=factory->create_struct_tc(
"MyType", structMembers, ex);   //If structTc is NULL, check 'ex' for more information
structTc->add_member("my_integer",
DDS_TYPECODE_MEMBER_ID_INVALID,
    factory->get_primitive_tc(DDS_TK_LONG),
    DDS_TYPECODE_NONKEY_MEMBER, ex);
structTc->add_member("my_float", DDS_TYPECODE_MEMBER_ID_INVALID,
    factory->get_primitive_tc(DDS_TK_FLOAT),
    DDS_TYPECODE_NONKEY_MEMBER, ex);
structTc->add_member("my_bool", DDS_TYPECODE_MEMBER_ID_INVALID,
    factory->get_primitive_tc(DDS_TK_BOOLEAN),
    DDS_TYPECODE_NONKEY_MEMBER, ex);
structTc->add_member("my_string", DDS_TYPECODE_MEMBER_ID_INVALID,
    factory->create_string_tc(128),
    DDS_TYPECODE_KEY_MEMBER, ex);
```

对于以上方法和常数更为详细的文献，包括例代码，用户可以参见 DDS 的网上文献，它适用于所有支持的编程语言，格式为 HTML 和 PDF。

3.5 使用数据样本

现在用户应当了解了如何定义和使用数据类型，无论用户使用内置于中间件的简单数据类型（参见 3.2 节）、动态定义类型（参见 3.5.2 节）或是 IDL、XML 文件生成的代码。

用户可以选定一个或多个数据类型以便使用，本节将帮助用户了解如何创建和操作那些类型的对象。

3.5.1　具体类型的对象

如果用户使用一种内置数据类型，或决定从一个 IDL 或 XML 文件生成自定义类型，用户的 DDS 数据类型与用户应用程序中的任何其他数据类型相似：一个拥有域、方法和其他用户可直接与之互动的成员的类或结构体。

在 C 和 C++ 中，用户从工厂中创建和删除用户拥有的对象，就像用户从工厂创建 DDS 对象一样。这里，工厂是一个被称为类型支持的单例对象，从这些工厂分配的对象得到深度分配和完全初始化。

```
/* In the generated header file: */
struct MyData {
char * myString;
};
/* In your code: */
MyData * sample=MyDataTypeSupport_create_data();
char * str=sample->myString;                /* empty, non-NULL string */
/* … */
MyDataTypeSupport_delete_data(sample);
```

和在 C 中一样，在 C++ 中，用户使用 TypeSupport 工厂创建和删除对象。

```
MyData * sample=MyDataTypeSupport::create_data();
char * str=sample->myString;                  //empty, non-NULL string
//…
MyDataTypeSupport::delete_data(sample);
```

3.5.2　动态定义数据类型的对象

如果使用一个可在运行时动态定义的数据类型，用户将使用 DynamicData 类提供的映射 API 获取和设置用户对象的域。

考虑以下类型定义：

```
struct MyData {
long myInteger;
};
```

对于静态定义类型来说，用户将从 TypeSupport 工厂创建对象。对于如何创建或获取一个 TypeCode 以及之后如何在此基础上创建 DynamicDataTypeSupport，参见 3.4.2 节“定义新类型”。

对于更多关于 DynamicData 和 DynamicDataTypeSupport 类的信息，参见网上(HTML)的文献。

在 C++ 中：

```
DDSDynamicDataTypeSupport * support=…;
DDS_DynamicData * sample=support->create_data();
DDS_ReturnCode_t success=sample->set_long("myInteger",
    DDS_DYNAMIC_DATA_MEMBER_ID_UNSPECIFIED, 5);
//Error handling omitted
DDS_Long theInteger=0;
success=sample->get_long(&theInteger, "myInteger",
    DDS_DYNAMIC_DATA_MEMBER_ID_UNSPECIFIED);
//Error handling omitted
//"theInteger"now contains the value 5 if no error occurred
```