概述

为了减少DDS用户在新建项目，配置环境所消耗的时间，DDS编译器（下文称为zrddsgen）只需要用户指定相应的IDL文件，然后根据用户在IDL文件中定义的数据类型来生成DDS工程需要的C++/C源代码以及相关IDE的工程文件。
为方便使用，zrddsgen提供模板文件template，此文件夹应与zrddsgen放置在同一目录下，默认若模板存在，将优先使用模板生成样例。用户可通过更改template文件夹下对应的函数模板生成指定样例。目录结构与文件含义如下：

目录	文件	说明
Standard		标准模板目录
	c	c语言模板
	c++	c++模板
	java	java模板
Extension		扩展接口模板目录（目录结构与Standard相同）
Simplify		简化接口模板目录（目录结构与Standard相同）
Makefile		Linux样例的默认Makefile文件
template.mak		Linux样例的默认.mak文件
ZRDDS_QOS_PROFILES.xml		扩展接口与简化接口的配置文件（可自行更改）

模板中特殊字符一般以"$ $"进行包裹，其含义如下：（Makefile与template.mak不建议更改，此处不具体介绍其中特殊字符）

字符名	说明	位置
$fileName$	idl名称	pub与sub模板文件
$exampleName$	最终结构体名称	pub与sub模板文件

若template文件夹丢失或想要恢复默认模板，则可删除文件，重建目录template\Extension\c（c++,java同理），使用zrddsgen编译器重新生成任意idl，模板文件则会重写。
若不采用模板文件，采用zrddsgen默认方式生成模板样例，请将template删除或移至其他路径。

用法

```
zrddsgen -input_idl < path >
         -output_dir < directory >
         -language < language >
         [-example < example_type >]
         [-project < project >]
         [-java_package <package_name>]
         [-type_name_style <type_name_style>]
         [-string_bound < bound >]
         [-sequence_bound < bound >]
         [-enable_unbounded]
         [-help]

         -input_idl |-i         指定输入的IDL文件的路径，支持相对路径，文件扩展名必须为.idl。此参数无默认值，必须给定。
         -output_dir|-d         指定生成文件的输出目录,必须给定。
         -language  |-l         指定使用的语言类型，目前支持C,C++,C#和java，此参数无默认值，必须给定。
         -example   |-e         指定输出的模板类型，目前支持normal，xml-ex，xml-sim。此参数默认不使用
                                 normal：标准接口模板
                                 xml-ex：Qos配置扩展接口模板
                                 xml-sim：Qos配置简化接口模板
         -project   |-p         为生成的代码产生相应的工程配置文件，目前支持
                                 Win32VS2008、Win32VS2010、Win32VS2012、Win32VS2013、
                                 Win32VS2015、Win32VS2012XP、Win32VS2013XP、Win32VS2015XP、Win64VS2010、
                                 Win64VS2012、Win64VS2013、Win64VS2015、LinuxEclipse、LinuxMakefile。
                                 此参数默认不使用。
                                 Win32VS2008：生成Win32下的VS2008的解决方案及项目文件。
                                 Win32VS2010：生成Win32下的VS2010的解决方案及项目文件。
                                 Win32VS2012：生成Win32下的VS2010的解决方案及项目文件。
                                 Win32VS2013：生成Win32下的VS2013的解决方案及项目文件。
                                 Win32VS2015：生成Win32下的VS2015的解决方案及项目文件。
                                 Win32VS2012XP：生成Winxp下的VS2012的解决方案及项目文件。
```

```
                            Win32VS2013XP：生成Winxp下的VS2013的解决方案及项目文件。
                            Win32VS2015XP：生成Winxp下的VS2015的解决方案及项目文件。
                            Win64VS2010：生成Win64下的VS2010的解决方案及项目文件。
                            Win64VS2012：生成Win64下的VS2012的解决方案及项目文件。
                            Win64VS2013：生成Win64下的VS2013的解决方案及项目文件。
                            Win64VS2015：生成Win64下的VS2015的解决方案及项目文件。
                            LinuxMakefile：生成Linux下的带有Makefile的项目文件。
                            （编译命令如下：make -f pubMakefile 编译pubMakefile 默认为Debug版本
                                          make -f subMakefile 编译subMakefile 默认为Debug版本
                                          make -f pubMakefile CONFIG=DEBUG   编译Debug版本
                                          make -f pubMakefile CONFIG=RELEASE 编译Release版本）
    -java_package           指定包名，仅在java语言中有效。此参数默认不启用
    -type_name_style        指定类型名风格。可选项包括dds_standard和zr_style。
                            分别表示内置类型名使用DDS标准风格或者臻融定义风格，默认使用DDS标准风格
    -string_bound           为string指定的默认bound。默认为255。
    -sequence_bound         为sequence指定的默认bound。默认为255。
    -enable_unbounded|-u    启用sequence和string的unbounded。
    -help|-h                打印帮助文档。
```

IDL语言语法

在使用zrddsgen前，你需要编写相应的IDL文件来描述你的数据类型。比如：

```
struct myDataType
{
    long value;
};
```

IDL的主要用途在于定义数据类型，它的语法和C语言中定义struct的语法基本一致，都是通过struct等关键字来定义数据类型名字，大括号中可以定义数据类型有哪些成员以及成员是否为关键字，大括号末尾以分号作为定义的结束。

基础数据类型

基础数据类型可以直接作为struct成员的数据类型，目前支持的基础数据类型列表如下：

类型	说明
octet	1字节无符号整型
char	1字节整型
boolean	true/false
short	2字节整型
unsigned short	2字节无符号整型
long	4字节整型
unsigned long	4字节无符号整型
float	4字节单精度浮点型
long long	8字节长整型
unsigned long long	8字节无符号长整型
double	8字节双精度浮点型
enum	4字节枚举类型
string	字符串类型
struct	复杂结构体类型，支持继承

union	复杂联合类型
数组	以上类型的定长数组类型
sequence	以上类型（除数组）的变长数组类型

String & Sequence

除了基础数据类型外，IDL中还有string以及sequence两个类型，它们的用法如下：

```
struct SequenceAndString
{
    string defaultBoundStr;
    string<100> bound100Str;
    sequence<long> defaultLongSequence;
    sequence<short, 100> bound100ShortSequence;
};
```

如例子中所示，sequence和string都有一个大小的上界，你可以在IDL中自行指定上界，如果不指定的话则会使用默认值，默认值可以使用zrddsgen提供的参数（-string_bound和-sequence_bound）来进行设置。
Sequence是对类型数组的一个简单封装，提供了一些基本的操作方法，具体可见OMG对于Sequence的介绍。

struct嵌套

IDL支持将先前使用struct定义的一个类型作为另一个struct的成员的数据类型，如下：

```
struct A
{
    long value;
};
struct B
{
    A a;
};
```

union

IDL支持定义复杂联合类型union，用法如下:

```
union A switch(long)
{
    case 1:
    case 2:
        long value;
    case 3:
        string<100> bound100Str;
}
```

union类型同样支持嵌套使用，用法同struct

enmu

IDL中的enmu定义方式如下：

```
enum TestEnum
{
    DOMAIN_TEST,
    INNER_PROCESS_OTO_TEST,
    INTER_PROCESS_OTO_TEST,
    INTER_NODE_OTO_TEST,
    INNER_PROCESS_OTM_TEST,
    INTER_PROCESS_OTM_TEST,
    INTER_NODE_OTM_TEST,
```

```
    INNER_PROCESS_MTM_TEST,
    INTER_PROCESS_MTM_TEST,
    INTER_NODE_MTM_TEST,
    UNKNOWM_TEST
};
```

typedef

IDL中的typedef的语法与C++中的基本相同，一般用于简化一个复杂的类型名，如sequence或者一个类型数组，如下：

```
typedef sequence<long, 100> SeqLong100; // 简化sequence
typedef long LongArr[10]; // 定义一个类型数组
```

Java语言不支持对基础数据类型进行typedef且若idl文件中typedef类型处于最外层，则需将其置于包内（可通过在外层添加module或编译时指定java_package实现。
Java语言不支持typedef数组类型。

数组

IDL支持定义成员变量时给定相关的数组维度，如下：

```
struct myDataType
{
    long valueArr[100];
    sequence<long> seqLongArr[10];
};
```

数组可以是任何先前提及的类型。

注释及特殊标记

IDL中使用`//`作为单行注释，比如上文提及的typedef中示例。
为了在IDL中增加ZRDDS所使用的一些概念，我们在IDL中增加了一些扩展标记。
目前支持的扩展标记列表如下：

以下扩展标记使用于struct、union（标记于struct或union之前，使用时不需要添加//, 前三种为DDS数据类型可扩展性需求）

扩展标记	说明
@Extensibility(EXTENSIBLE_EXTENSIBILITY)	默认类型，该类型表示该类型可进行尾部扩展（裁剪），即新的数据结构可在老的数据结构最后添加成员或者删除成员，序列化与反序列化的方式为按照成员的顺序依次将成员的值进行CDR编码/解码。若报文内容比所需的成员内容长时，多余的内容将被忽略，且不报错；若报文内容不足以反序列化所有成员时，未反序列化到的成员将设置为初始值
@Extensibility(MUTABLE_EXTENSIBILITY)	该类型表示该类型可进行任意结构/数量的变化，配置该类型时，需在IDL文件中为每个成员设置唯一的ID标识，序列化采用扩展的CDR方式，将ID与内容一并序列化，在接收端反序列化时先反序列化ID，再根据本地ID所对应成员进行成员的反序列化，这种方式灵活但是序列化/反序列化的过程以及内容将变复杂。ID的值默认从0开始依次递增，复杂结构嵌套时，计算ID的方式为将成员展开累积；使用在成员后面添加 `@ID (1)` 标注来设置ID值，设置ID值后，如后续的成员未设置，则从该ID值开始累积计数
@Extensibility(FINAL_EXTENSIBILITY)	该类型表示该类型禁止扩展，通信双方的数据结构必须一致，序列化与反序列化的方式为按照成员的顺序依次将成员的值进行CDR编码/解码，遇到报文不足或者多余的报文时，DDS将报序列化或者反序列化出错的日志；
@Nested	指定为中间值，不为此结构生成TypeSupport和相关的数据写者和数据读者，常用于嵌套结构中

可扩展数据类型的示例参见如下：

```
@Extensibility(EXTENSIBLE_EXTENSIBILITY)
struct ExtensibleOldType
```

```
{
    long x;
    long y;
};

@Extensibility(EXTENSIBLE_EXTENSIBILITY)
struct ExtensibleNewType
{
    long x;
    long y;
    long z;
    float angle;
};

@Extensibility(MUTABLE_EXTENSIBILITY)
struct MutableOldType
{
    long x;
    long y;
};

@Extensibility(MUTABLE_EXTENSIBILITY)
struct MutableNewType
{
    long z; @ID (2)
    long x; @ID (0)
    float angle; @ID (3)
    long y; @ID (1)
};
```

以下扩展标记使用于结构成员（标记于成员之后，使用时需要添加//）

扩展标记	说明
@key	指定关键字(也可以使用@Key，不需要//)
@shared	指定为指针数据类型
@spare	指定为稀疏数据类型，在传输中，若此类型传输值与上次相同，则不进行重复传输，序列化时会比较数据成员值，只能在Extensibility(MUTABLE_EXTENSIBILITY)时使用，并且ZRDDS中需要开放此功能
@ID	指定成员序号，通常与@Extensibility(MUTABLE_EXTENSIBILITY)合用
@bitwidth	指定成员位域，使用方式为@bitwidth(x),x为位域数据长度

示例

```
@Extensibility(MUTABLE_EXTENSIBILITY)
struct A
{
    long thisIsKey; //@key
    long thisIsNotKey; // @ID(3)
    sequence<char> charSeq;//@shared
};
```

注意事项：

（1）稀疏数据功能只能在用户指定输出的编程语言为C++时使用。
（2）稀疏数据类型只有在Extensibility为MUTABLE_EXTENSIBILITY才能生效
（3）稀疏数据类型需要与 MUTABLE_EXTENSIBILITY类型结合使用
（4）位域功能只能在用户指定输出的编程语言为C或C++时使用。
（5）位域功能只支持char、octet、long、unsigned long类型的数据使用，当其它数据类型使用位域功能时，将编译失败并打印“InvalidMemberTypeError”。
（6）位域数据需要定义在所有其他数据类型之前（因为idl中声明的顺序与生成的C++/C语言中的成员顺序一致，从位域的使用目的来看，希望用户把

位域声明在前面）。

（7）位域类型的长度不能超过原始数据的最大长度，char和octet不能大于8，unsigned long 和long不能超过32，否则将编译失败并打印“InvalidBitWidthError”。

（8）位域类型不支持数组和sequence类型，否则编译失败并打印Unsupported bitWidth type。

（9）位域不能声明为key，否则报错“InvalidKeyMemberTypeError”。