msgbuf 用户手册

V2.0.4

gaobo.benny

2013/1/22

目录

1. msgbuf 是什么？

msgbuf 是一个面向异构平台环境网络通信、平台无关、支持多种编程语言的结构化数据序列化(serializing)工具，当前版本支持的语言包括 Java(Java5 以上)、C#和 C++。

首先 msgbuf 是面向异构平台环境下的网络数据通信应用，如果通信的双方都是用 Java 语言开发，那问题似乎很简单，Java 标准类库中的 ObjectOutputStream 和 ObjectInputStream 就能很好地满足要求。但是当通信的双方基于不同的硬件体系结构，并且采用不同的开发语言时，问题就会变得复杂很多，32 位或者 64 位体系的硬件，硬件采用字节序(BigEndian/LittleEndian)，不同语言基础数据类型的差异，都会成为异构平台环境下网络数据通信需要解决的问题。

应用程序对于网络通信的要求可能会很高，不是简单地传递几个整数值就能满足应用开发的要求。应用需要更丰富的数据表达形式，例如：

- 向量：Java 中的 List、C++中的 stl vector、C#中的 IList，用以存储 0 到任意个数的基础数据类型或者对象。
- 数组：与向量类似，不同的是长度固定，而 vector 可以动态扩展。
- Key-Value 映射：Java 中的 Map，C++中的 stl map、C#中的 IDictionary。

其次，msgbuf 尽可能多地提供主流平台和主流开发语言的支持，尤其针对移动平台的开发环境的特点进行了设计和适配，当前能够支持 Android SDK Java、Android NDK C++、Unity3D C#、以及 iOS Objective-C++。

虽然移动平台的硬件发展速度很快，但当前相较普通的 PC 或者服务器硬件平台来说，CPU 和 MEM 的资源还是受限很大，msgbuf 使用对象池（Objects Pool）方式管理对象，使数据对象得到重用，极大地减少了对象的创建次数，以利于移动应用的稳定运行。

与 XML 不同的是，msgbuf 使用了二进制格式进行序列化，并且支持数据压缩，从而使数据量更小，处理速度也更快。

用户定义了应用数据的结构以后，就可以用 msgbuf 生成相应 Java、C#或者 C++代码，对数据进行序列化、网络传递、反序列化。

msgbuf 早期版本中支持过 J2ME 和 Objective-C。J2ME 已经逐渐退出历史舞台；而 apple XCode 可以混合 Object-C 和 C/C++共同开发 iOS 应用，有了 C++的支持后，Objective-C 的支持似乎必要性不大；所以这两个语言的支持在新版本中不再提供。

2. 名词定义

MDL：消息定义语言（Message Definition Language）

MDL 是 msgbuf 自定义的一种用于定义消息数据结构的语言，语法类似 Java。

MessageBean：消息类

每个消息的数据结构称为一个 MessageBean，Bean 这个单词和概念来自于 Java，为了保持概念和名字的一致性，msgbuf 在各种语言的实现中，都沿用了 MessageBean 这个词。

在所有语言的实现中，都有 MessageBean 这个类，它是所有消息类的基类。

Command：指令

Client 与 Server 之间交互的每一次通信，都是希望触发对方的一个处理，也可以理解调用对方的一个方法（类似远程方法调用 Remote Method Call），或者称为发送一个指令（Command）给对方。

一个指令典型地包括：

- 指令编号（CommandId），用以唯一地标识一个指令。
- 指令参数，是一个 MessageBean。

CommandId：指令编号

CommandId 用以唯一地标识一个指令，例如：用户登录、用户退出、购买道具等，CommandId 可以理解为远程方法调用（RMI）中的“方法名”，在远程方法调用的实现中，用 login、logout、buyProperties 这样的方法名来唯一标识一个处理。msgbuf 与之不同的是使用**整型数值**作为这个唯一标识。

MessageCompiler

使用 msgbuf，首先要在一个或者数个文本文件中，用 MDL 编写消息的数据结构定义。这些消息定义的文本文件建议的后缀名是”.mdl”，但不强制，可以任意后缀名。

定义好消息数据结构以后，调用 msgbuf 提供的编译工具 MessageCompiler 对这些消息定义文件进行处理，根据需要生成 Java、C++或者 C#的代码，放到应用工程中使用。

应用开发中，如果消息定义有修改，必须再次调用 MessageCompiler 进行重新编译生成。

Client 和 Server 必须确保使用相同的消息定义文件。

Calibur

如上面对 msgbuf 的描述，msgbuf 仅仅是一个对象序列化/反序列化的工具，序列化出来的数据可以通过网络发送，也可以存储在文件中，msgbuf 本身并不提供网络通信的功能。

Calibur 是一个 C/S 结构应用的 framework，在 Server 端包括线程调度、事件分发等功能，在 Client 端提供 Java、C#和 C++的客户端 API 库，实现与 Calibur Server 之间的通信。

Calibur 在数据传输时，使用 msgbuf 作为对象数据在网络传递时序列化和反序列化的组件。

3. 安装 msgbuf

环境要求

msgbuf 的工具功能使用 Java 开发，在安装 msgbuf 的系统环境中需要安装了 Java 运行环境，要求 Java6 以上的 Java 运行环境。

安装

msgbuf 的发行包是一个 zip 格式的压缩文件，例如：mc-2.0.3.zip。只需要把这个压缩文件解压到任意目录即可。解压后，会产生如下的子目录：

目录	文件	说明
bin	mc.bat messagecompiler-2.0.3.jar	消息编译工具 MessageCompiler 的批处理文件 mc.bat 等。
lib		
java	msgbuf-2.0.3.jar	msgbuf Java 版的运行时刻 jar 包。
csharp	Calibur.dll	C#运行时刻 DLL，包括 Calibur C# Client 和 msgbuf。
cpp		
android	libmsgbuf.so	Android C++运行时刻动态链接库。
doc	msgbuf user manual.pdf	存放 msgbuf 的用户手册等文档。

使用 msgbuf 之前，最好把 msgbuf 安装目录下 bin 目录路径设置到系统的 PATH 路径集中，这样在命令行可以直接调用 MessageCompiler 工具——mc.bat 脚本。

4. 支持的数据类型

基本数据类型

字节：**byte**

字节数	1
取值范围	-128 ~ 127
Java 类型	byte
C#类型	byte
C++类型	unsigned char

布尔值：**boolean/bool**

字节数	1
取值范围	true/false
Java 类型	boolean
C#类型	bool
C++类型	bool

短整型整数：**short/int16**

字节数	2
取值范围	-2^{15} ~ $2^{15}-1$
Java 类型	short
C#类型	short/Int16
C++类型	short/Int16_t

整数：**int/int32**

字节数	4
取值范围	-2^{31} ~ $2^{31}-1$
Java 类型	int
C#类型	int/Int32
C++类型	int/Int32_t

长整型整数：long/int64

字节数	8
取值范围	$-2^{63} \sim 2^{63}-1$
Java 类型	long
C#类型	long/Int64
C++类型	long long/Int64_t

单精度浮点数：float

字节数	4
取值范围	$2^{-149} \sim (2-2^{-23})\cdot 2^{127}$ 参考 Java2 API Documents
Java 类型	float
C#类型	float/single
C++类型	float

双精度浮点数：double

字节数	8
取值范围	$2^{-1074} \sim (2-2^{-52})\cdot 2^{1023}$ 参考 Java2 API Documents
Java 类型	long
C#类型	long/Int64
C++类型	long long/Int64_t

字符串：string

字节数	变长
取值范围	N/A
Java 类型	String
C#类型	string
C++类型	std::string

对于基础数据类型中的整型类型(short/int/long)，msgbuf 在序列化时，使用了 google 的 varint 编码格式，varint 是一种变长的格式，根据数值的大小占用不同的字节数，当一个非负整型的值小于 127 时，只会占用 1 个字节。如果采用定长的格式，一个 long 类型的数值，不管值多小，都必须固定占用 8 个字节的空间。所以建议 MDL 定义中，除非特别需要，可以不再使用短整型(short)类型。

消息类 MessageBean

消息类可以理解为一个“类(class)”，可以包含不限定个数的属性，属性数据类型可以是上面列出的基础数据类型，也可以是其他消息类，还可以是下面描述的各种容器类型（数组、向量、map）。

消息类可以继承其他消息类，继承后，消息类包含了父消息类的所有属性。与 Java 和 C#一样的是，MDL 仅支持单继承，即只能继承一个父类消息。

下面的 MDL 片段中，演示了如何定义普通消息、复合消息和派生消息。

```
/**
 * 普通消息，仅包含基础数据类型的字段。
 */
message BaseClass {
    int errorCode;
    string errorMessage;
}

/**
 * 复合消息，包含了其他消息。
 */
message Composite {
    int xxId;
    BaseClass baseInfo;
}

/**
 * 派生消息，继承自其他消息。
 * 消息Inheritor继承了BaseClass，自动包含了BaseClass的errorCode
 * 和errorMessage属性。
 */
message Inheritor extends BaseClass {
    long xxId;
    string name;
}
```

在 msgbuf 支持的所有目标编程语言中，每个消息都被生成一个类(class)，类名是 MDL 定义的消息名字（命令行用-prefix 指定类名前缀时例外），所有消息类都派生自一个叫做 MessageBean 的基类。

数组

支持除了 string 之外的基础数据类型的数组，即 byte、boolean、short、

int、long、float 和 double 的数组。msgbuf 不支持 string 和消息的数组，原因是在 C++这样的静态语言中，string 和消息都是 class。如果要表示不固定个数的 string 和消息，应该用下面介绍的 vector 来代替。

如果要定义一个 int 类型的数组，定义方式如下：

```
// 定义一个整数的数组
int[] intArray;
```

在 Java 中，会生成为：`int[] intArray;`

在 C#中，会生成为：`int[] intArray;`

在 C++中，会生成为：`msgbuf::varray<int> *intArray;`

在 Java 和 C#这样的动态语言中，数组也是对象，数组的长度可以通过数组对象的属性获得，Java 的.length 和 C#的.Length。而 C++中，数组本质是指针，指针所指向的数组元素个数是无法直接得到的，必须用另外的参数指定。msgbuf 在 C++的实现中，自定义了一个带有长度信息的 msgbuf::varray 类，用这个类来封装数组，让调用者可以得到数组的长度。

向量：vector

向量用于表示不固定长度的数据集合，可以支持所有基础数据类型以及 string 和消息，例如：

```
// 定义一个整数的向量
vector<int> intvec;

// 定义一个string的向量
vector<string> msgs;

// 定义一个消息的向量
vector<Base> baseList;
```

Java 映射

```
// 定义一个整数的向量
java.util.List<Integer> intvec;

// 定义一个 string 的向量
java.util.List<String> msgs;

// 定义一个消息的向量
java.util.List<Base> baseList;
```

可以用实现了 java.util.List 接口的类来赋值，如 ArrayList、LinkedList

等，msgbuf 内部使用了 java.util.ArrayList。

C#映射

```
// 定义一个整数的向量
System.Collections.Generic.IList<int> intvec;

// 定义一个 string 的向量
System.Collections.Generic.IList<string> msgs;

// 定义一个消息的向量

System.Collections.Generic.IList<Base> baseList;
```

vector 可以用实现了 System.Collections.Generic.IList 泛型接口的类来赋值，msgbuf 内部使用了 System.Collections.Generic.List。

C++映射

```
// 定义一个整数的向量
std::vector<int> intvec;

// 定义一个 string 的向量
std::vector<string> msgs;

// 定义一个消息的向量

std::vector<Base> baseList;
```

Key-Value 对：map

map 用于表示一个“键(Key)”与其对应的“值(Value)”的对应关系，例如姓名与联系方式的对应关系。map 是一个简单但强大的数据结构，可以简化应用逻辑开发中数据检索的实现。

msgbuf 能够使用基础数据类型（包括 string）作为“键(Key)”，不允许消息作为“键”。“值(Value)”则可以定义为基础数据类型或者消息。

```
// 定义联系方式
message PersonalInformation {
    // 固定电话号码
    string tel;
    // 移动电话号码
    string mobile;
```

```
    // 电子邮件
    string email;
    // 通信地址
    string address;
}

message Contact {
    // 定义姓名与联系方式的map
    map<string, Contact> contactMap;
}
```

Java 映射

Java 中，contactMap 被生成为：

```
java.util.Map<String, Contact> contactMap;
```

可以用实现了 java.util.Map 接口的类来赋值，如 HashMap、TreeMap 等，msgbuf 内部使用了 java.util.HashMap。

C#映射

C#中，contactMap 被生成为：

```
IDictionary<string, Contact> contactMap;
```

可以用实现了 System.Collections.Generic.IDictionary 泛型接口的类来赋值，msgbuf 内部使用了 System.Collections.Generic.Dictionary。

C++映射

C++中，contactMap 被生成为：

```
std::map<string, Contact> contactMap;
```

5. 用 MDL 定义消息和指令

msgbuf 使用文本文件格式来定义消息和指令，一个这样的文件称为一个 MDL 文件，建议使用.mdl 作为统一的文件后缀名。

定义消息

```
// 定义联系方式
message PersonalInformation {
    // 固定电话号码
    string tel;
    // 移动电话号码
    string mobile;
    // 电子邮件
    string email;
    // 通信地址
    string address;
}
```

与 Java 中一个 class 的定义类似，用“message <消息名> {”标识开始定义一个消息，用“}”标识消息定义结束。

“消息名”不能重复，建议首字母大写，与大部分开发语言对于类定义的规范一致。

定义指令

```
command {
    // 玩家登录游戏
    1000 ReqLogin RspLoginResult(compressed);

    // 玩家在场景中行走
    2001 NotifyPosition void;

    // 服务器系统广播
    9999 void PushSystemBroadcast;
}
```

MDL 文件中，用“command {”标识开始定义指令，用“}”标识指令定义结束。

一个指令包括以下信息：

数据项	要求	数据类型	说明
指令编号	必填	非负整数	Client 与 Server 之间相互调用的唯一标识，不能重复。
输入消息	必填	已定义消息名称，或者 void、null。	
压缩标志	选填	(compressed)	如果选填，输入消息在网络传递时会进行压缩。
输出消息	必填	已定义的消息名，或者 void、null。	
压缩标志	选填	(compressed)	如果选填，输出消息在网络传递时会进行压缩。

Client 与 Server 之间通信的模式可以分为三类：

- **类型一**：Client 发送指令及其输入消息给 Server，Server 进行处理后，返回处理结果给 Client。典型的是用户登录游戏的处理，Client 把玩家 id 和密码发给 Server， Server 检查玩家 id 是否存在、密码是否正确等，并进行登录处理，处理完后把处理成功与否的消息发回给 Client。定义方式如下：

```
// 玩家登录游戏
1000 ReqLogin RspLoginResult(compressed);
```

玩家登录游戏的指令编号 1000；输入消息是 ReqLogin，这个消息中包含了玩家 id 和密码等必要信息；输出消息是 RspLoginResult，这个消息中包含了登录成功与否的信息。

一般登录成功时，Server 要把玩家的基础信息(HP/MP)、社区信息(称号、在线好友列表)、钱包信息、任务列表等等放在 RspLoginResult 中发给 Client，数据量比较大，对于这种数据量比较大的消息，可以设定(compressed)选项，指定在网络传递时进行数据压缩，减少用户的数据流量。

- **类型二**：Client 单向通知 Server 一个信息，并不需要 Server 给予响应。例如玩家在场景中行走时，定时地通知 Server 自己所在的位置，这个通知不需要 Server 给予什么响应，当然 Server 收到玩家位置时，需要检查是否有作弊加速，认定有作弊加速时进行响应处理，不过这个处理通过其他的机制和消息实现，不在每次同步位置的通信时处理。

```
// 玩家在场景中行走
2001 NotifyPosition void;
```

对于这种 Server 不给予响应消息的指令，把输出消息定义为“void”。

- **类型三**：Server 单向通知 Client 一个信息，例如 Server 定期维护

时，在 Server 关闭前一段时间通知 Client：Server 即将关闭，请玩家尽快 logout。

```
// 服务器系统广播
9999 void PushSystemBroadcast;
```

对于这种 Server 单向通知 Client 的消息，把输入消息定义为“void”。

特殊的消息 null

当 Client 与 Server 之间的通信仅仅需要传递一个指令号，没有任何参数数据时，可以把参数消息指定为 null。

null 与 void 的区别：void 是什么都没有，null 是有消息，但是仅有指令号。

MDL 中包含其他 MDL

消息并不需要全部定义在一个 MDL 文件中，可以根据应用的功能模块划分，分别定义在多个 MDL 文件中，由多个开发者分别维护，利于管理。在总装的 MDL 中，用 import 指令把这些分散的 MDL 包含进来。

```
// 引入基础数据结构MDL
import base.mdl;

// 引入战斗系统消息定义MDL
import fight.mdl;

// 引入任务系统消息定义MDL
import ext/mission.mdl;
```

import 的 MDL 可以带有路径，不需要所有 MDL 存放在相同的目录下，可以根据开发管理需要进行分目录存放。

如果一个 MDL 有 import 其他 MDL，那么 import 部分必须放置在它的一开始部分，即所有 message 和 command 定义的前面。

注释

MDL 中支持一般开发语言的两种注释方式，多行注释包在“/*”和“*/”中，双斜杠“//”注释到行尾。

注释并不强制，但是写好的注释非常重要，因为 MDL 本身就是作为 Client 与 Server 之间交互的接口，Client 与 Server 通常不是由同一个工程师负责开发，那在接口层把每一个数据项的定义写清楚，对于项目中信息沟通、减少

错误能起到好的作用。例如以下的定义和注释就是好的方式。

```
/*
 * 玩家登录请求消息
 */
message ReqLogin {
    // 玩家通行证帐号
    string passport;

    // 密码
    string password;
}

/*
 * 玩家登录处理响应
 */
message RspLoginResult {
    // 登录处理处理结果。0:成功 1:通行证帐号不存在 2:密码错误
    int errorCode;

    // 玩家基础信息，当errorCode=0时存放了玩家基础信息，其他情况为null。
    RoleBase roleBase;

    // 玩家钱包信息，当errorCode=0时存放了玩家钱包信息，其他情况为null。
    Wallet wallet;
}
```

在 MDL 中写的所有 message、field、command 的注释，全都会被生成到目标语言(Java/C#/C++)的 MessageBean 类源代码中，同时这些注释还用于生成更利于阅读的 HTML 文档。

MDL 保留关键字

MDL 保留的关键字不多，共计有：import message vector boolean bool byte short int long float double string map null void compressed extends command。

但是为了预备未来支持更多的种目标编程语言，MDL 还保留了 C++、C#、Objective-C、Java、Python、PHP 语言的关键字，详细见【附录一：MDL 保留的关键字】。所有这些关键字都不能用作消息名和字段名。

所有保留的关键字，包括 MDL、C++、C#、Objective-C、Java、Python、PHP 关键字，都不能用作消息名和字段名。

需要特别注意的是 id 这个可能被广泛使用的标识符，因为它是 Objective-C 语言里的保留关键字，所以也不能用了。建议 MDL 中定义为 xxxId

的形式，例如 missionId，propId。

MDL 关键字不区分大小写，message 和 MESSAGE 都是关键字 message。

完整的例子

下面列出了 msgbuf 内部用于开发测试的测试用例 MDL，包括两个 MDL 文件，base.mdl 中定义了所有的消息；test.mdl 中 import 了 base.mdl，并且定义了所有的指令。本文后续的 MesageBuffer 使用说明中，都以这两个 MDL 的内容为基础。

base.mdl

```
/*
 * Message Compiler 测试用例
 */

/**
 * 道具
 */
message Prop {
    /** 道具编号 */
    int propId;
    /** 道具名称 */
    string name;
    /** 道具类型 */
    byte type;
    /** 道具显示图标编号 */
    short icon;
    /** 道具数量 */
    long quantity;
    /** 道具图像PNG流 */
    byte[] image;
}

message CommandResponse {
    // 错误码
    int errorCode;

    // 错误信息
    string errorMessage;
}
```

```
/**
 * 玩家背包
 * 服务器下发玩家背包中的详细物品列表
 */
message Backpack {
    /** 角色ID */
    int folderNum;

    /** 背包中的物品列表 */
    vector<Prop> props;
}

/**
 * 地图数据
 */
message SceneMap extends CommandResponse {
    /** 地图编号 */
    int mapId;

    /** 地图数据 */
    vector<byte> data;

    /** 测试单个Bean */
    Prop prop;
}

/**
 * 玩家登陆请求
 */
message Login {
    /** 通行证帐号 */
    String passport;

    /** 密码 */
    String password;
}

/**
 * 玩家登陆结果
 */
message LoginResponse extends CommandResponse {
    /** 玩家角色id */
    long playerId;
```

```
    /** 玩家的背包信息 */
    Backpack backpack;
}

/**
 * 玩家走动
 */
message Walk {
    /** 当前位置X坐标 */
    int x;

    /** 当前位置Y坐标 */
    int y;
}

/**
 * 系统广播
 */
message Broadcast {
    /** 系统广播内容 */
    string msg;
}

/**
 * 登出游戏响应
 */
message LogoutResponse extends CommandResponse {
    // 有礼貌地说再见
    String grace;
}

/**
 * 测试所有可能的数据类型
 */
message AllType extends CommandResponse {
    byte byteValue;
    vector<byte> byteVector;
    byte[] byteArray;

    boolean boolValue;
    vector<boolean> boolVector;
    boolean[] boolArray;

    short shortValue;
```

```
    vector<short> shortVector;
    short[] shortArray;

    int intValue;
    vector<int> intVector;
    int[] intArray;

    long longValue;
    vector<long> longVector;
    long[] longArray;

    float floatValue;
    vector<float> floatVector;
    float[] floatArray;

    double doubleValue;
    vector<double> doubleVector;
    double[] doubleArray;

    string stringValue;
    vector<string> stringVector;

    Prop prop;
    vector<Prop> propVector;

    map<int, Prop> folders;
}
```

test.mdl

```
import base.mdl;

// 指令定义
command {
    /** 玩家登录游戏服务器 */
    1000 Login       LoginResponse;

    /** 玩家logout */
    1001 null     LogoutResponse;

    /** 玩家行走（无响应）*/
    1003 Walk     void;
```

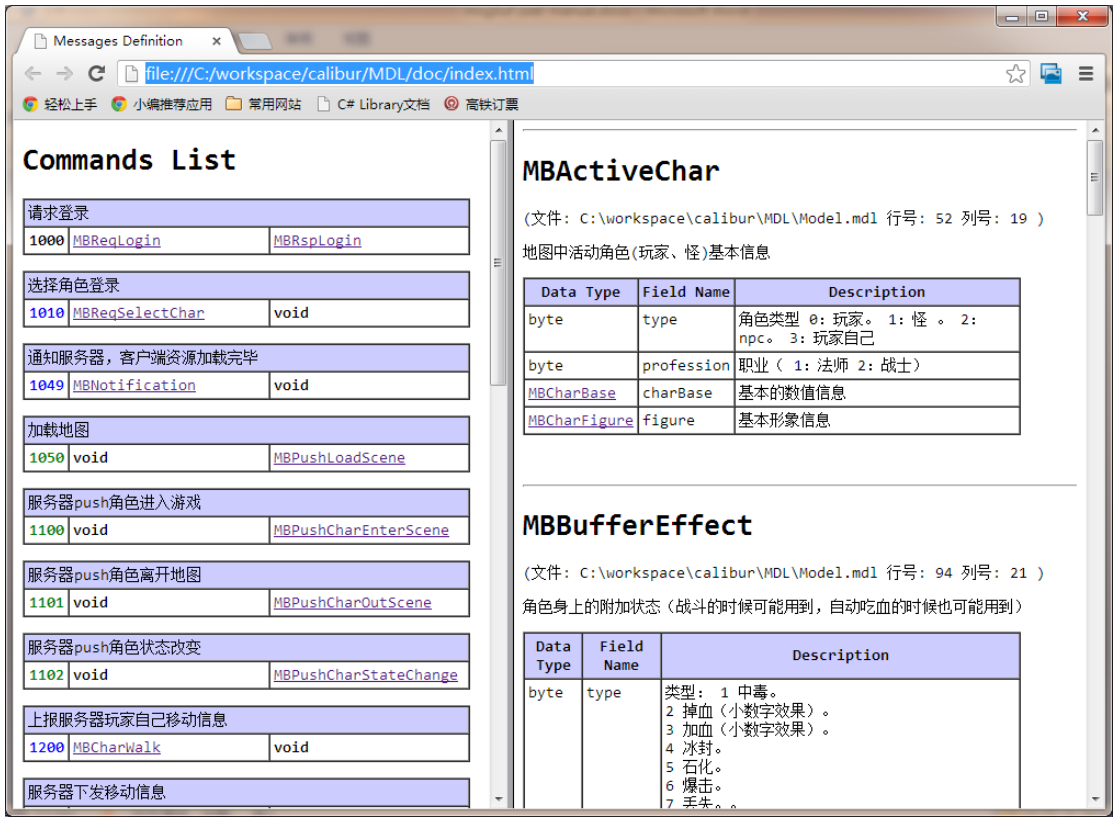
```
    /** 服务器Push系统广播 */
    1004 void     Broadcast;

    /** 测试指令 */
    2000 AllType     AllType(compressed);
}
```

6. 为 MDL 生成 HTML 文档

msgbuf 可以为 MDL 生成可读性更好的 HTML 文档，利用 HTML 超链接的便捷跳转，可以快速地查阅指令、消息之间的交叉引用和嵌套信息。

生成的 HTML 文档样式如下，左侧分栏是以指令编号为索引的指令信息列表，右侧分栏是 message 的详细说明。



MDL 中，command、message 和字段的所有注释信息都会生成到 HTML 中，所以在 MDL 中编写详细的注释十分重要。

生成 HTML 文档的命令如下：（黄色斜体字是需要根据具体环境修改的内容）

```
c:> mc.bat -doc -prefix MB -mdlencoding gb2312 -encoding utf-8 -outdir
c:\workspace\DemoGame\mdl\doc  C:\workspace\DemoGame\mdl\test.mdl
```

- -doc：必选，无参数，指定生成 HTML 文档。
- -outdir：必选，参数是指定把 HTML 文档生成到哪个路径下。
- -prefix：可选，参数是生成消息类名的前缀，例如设定了-prefix 为 MB，则 Login 这个消息实际生成的消息类名为 MBLogin。
- -mdlencoding：可选，参数是 MDL 文件的字符集编码。不指定时，

MessageCompiler 用系统环境缺省的字符集编码读入 MDL 文件。在简体中文版 Windows XP/7/8 系统中，系统缺省的字符集编码是 gb2312；Linux 系统中缺省是 iso-8859-1，可以设置环境变量 LANG 来指定。

- -encoding：可选，参数是生成 HTML 文档的字符集编码。不指定时，用系统环境缺省的字符集编码。如果工程要考虑多国语言支持，最好把-encoding 设置称为 UTF-8。
- 总装 MDL 文件路径名：必选，指定总装的 MDL 文件路径。

执行命令后，会在-outdir 选项指定的目录下生成三个 HTML 文件：index.html、Message.html、Navigation.html。用浏览器打开 index.html 即可看到生成的内容。

7. C#语言映射

下面的命令行例子中，假定 base.mdl 和 test.mdl 的存放目录是 **D:\mdl**；假定 C#工程的源代码目录是 **D:\workspace\DemoGame**。实际使用时，请替换成实际的路径。

编译 MDL 生成 C#代码

命令行编译

在命令行执行以下命令：（黄色斜体字是需要根据具体环境修改的）

```
C:> mc.bat -csharp -override -package Protocol -srcdir D:\workspace\DemoGame -prefix MB -mdlencoding utf-8 -encoding utf-8 D:\mdl\test.mdl
```

- -csharp：必选，无参数，指定生成 C#代码。
- -override：可选，无参数，强制覆盖上次生成的代码。MessageCompiler 在生成代码时，每个 message 会生成一个 C#的 class，class 的名字就是 MDL 中定义的 message 的名字。并会根据 message 的定义生成一个电子摘要(Digest)，如果不指定-override 选项，则会比对本次生成的 Digest 与上次的是否相同，如果相同，就不再生成。而指定了-override 的话，就不管内容是否有改动，都强制重新生成。
- -package：必选，参数是生成出 C#消息类的 namespace。这个 namespace 由开发者自己指定合适的名字，本例中设定为 Protocol。
- -srcdir：必选，参数是目标工程的源代码目录。注意生成的消息类源代码不是直接放在这个目录下，而是会在这个目录下创建一个名字是-package 选项参数的子目录下。本例中，目标工程源代码目录设定为 D:\workspace\DemoGame，-package 设定为 Protocol，则消息类的源代码被放置在 D:\workspace\DemoGame\Protocol 目录下。
- -prefix：可选，参数是生成消息类名的前缀，例如设定了-prefix 为 MB，则 Login 这个消息实际生成的消息类名为 MBLogin。
- -mdlencoding：可选，参数是 MDL 文件的字符集编码。不指定时，MessageCompiler 用系统环境缺省的字符集编码读入 MDL 文件。在简体中文版 Windows XP/7/8 系统中，系统缺省的字符集编码是 gb2312；Linux 系统中缺省是 iso-8859-1，可以设置环境变量 LANG 来指定。

- -encoding：可选，参数是生成 C#源代码的字符集编码。不指定时，用系统环境缺省的字符集编码。如果工程要考虑多国语言支持，最好把-encoding 设置称为 UTF-8。
- 总装 MDL 文件路径名：必选，指定总装的 MDL 文件路径。

执行以上命令行后，如果一切正常，将会看到如下的运行信息：

```
警告：[base.mdl:47] 定义了消息 SceneMap ，但没有任何地方使用它。
生成 D:\workspace\DemoGame\Protocol\ObjectFactory.cs ... 完成。
生成 Login 到 D:\workspace\DemoGame\Protocol\Login.cs ... 完成。
生成 Walk 到 D:\workspace\DemoGame\Protocol\Walk.cs ... 完成。
生成 LogoutResponse 到 D:\workspace\DemoGame\Protocol\LogoutResponse.cs ... 完成。
生成 Prop 到 D:\workspace\DemoGame\Protocol\Prop.cs ... 完成。
生成 AllType 到 D:\workspace\DemoGame\Protocol\AllType.cs ... 完成。
生成 Backpack 到 D:\workspace\DemoGame\Protocol\Backpack.cs ... 完成。
生成 NullMessageBean 到 D:\workspace\DemoGame\Protocol\NullMessageBean.cs ... 完成。
生成 Broadcast 到 D:\workspace\DemoGame\Protocol\Broadcast.cs ... 完成。
生成 LoginResponse 到 D:\workspace\DemoGame\Protocol\LoginResponse.cs ... 完成。
生成 SceneMap 到 D:\workspace\DemoGame\Protocol\SceneMap.cs ... 完成。
生成 CommandResponse 到 D:\workspace\DemoGame\Protocol\CommandResponse.cs ... 完成。
源代码已经成功生成到目标目录 D:\workspace\DemoGame 下。
```

命令行成功执行后，会在指定目录下为每一个消息生成一个消息类的 C#源代码文件，例如消息 Backpack 会生成到 Backpack.cs 中，同时还会额外地生成两个类 NullMessageBean.cs 和 ObjectFactory.cs。

NullMessageBean.cs 定义了公共的空消息类，即 command 定义中指定为 null 的消息。

ObjectFactory.cs 是所有消息类的单例(singleton)工厂(factory)模式类，负责管理指令编号与消息的映射关系，以及用对象池的方式管理所有消息类的实例。

在 C#目标工程中，导入进命令行编译创建的所有消息类后，就可以使用这些消息类。

常见错误提示信息

命令行指定的 MDL 或者 import 的 MDL 路径名错误

```
打开消息定义文件D:\workspace\DemoGame\mdl\basea.mdl错误，请检查文件路径及访问权限。
```

语法错：关键字敲错

```
文件：D:\workspace\DemoGame\mdl\base.mdl
```

```
行号：9
列号：1
错误：在"mesage"附近语法错。
```

语法错：关键字被用作消息名或者字段名

```
文件：D:\workspace\DemoGame\mdl\base.mdl
行号：49
列号：6
错误：在"id"附近语法错。
```

消息名字重复

```
文件：D:\workspace\DemoGame\mdl\base.mdl
行号：36
列号：14
错误：消息名字重复，Prop 在 base.mdl:9 已经定义过。
```

使用了未定义的消息名

```
文件：D:\workspace\DemoGame\mdl\base.mdl
行号：41
列号：9
错误：引用了未定义的消息类型 Mission。
```

消息类的 namespace

C#的类路径用 namespace 进行管理，生成消息类的 namespace 就是命令行选项中-package 指定的名字。因此应用中要使用生成的消息类时，需要用“using Protocol;”语句来指定使用 Protocl 这个 namespace。

同时，还需要用“using MessageBuffer;”语句指定使用 msgbuf 的运行时刻支撑库。所以应该是：

```
using MessageBuffer; // 使用 msgbuf 运行时刻支撑库
using Protocol; // 使用 MDL 生成的那些消息类
```

消息对象实例的创建和回收

msgbuf 对消息类的实例进行强制对象池管理，生成的消息类代码中把消息类的构造函数修饰符设定为 internal，因此调用者无法用 new 操作符来创建新的消息类实例。

要创建消息类实例，只能通过消息类的静态属性 Instance 获得。

```
// 得到一个 Prop 实例（从对象池获得）
Prop mbProp = Prop.Instance;
```

非常重要：消息类实例创建后，当使用完成时，必须调用实例的 Release()进行释放，也就是归还到对象池中，以循环使用。如果没有调用 Release()，将会造成内存漏洞，直到耗尽所有内存。

```
mbLogin.Release(); // 释放 mbLogin
mbLogin = null; //非强制，但最好这样，避免自己重复使用。
```

非常重要：调用了 Release()方法释放后的实例，调用者不能再继续使用同一个实例，这个实例可能在其他地方循环使用了，调用者再使用同一个实例的话，可能把其他逻辑的数据给覆盖了。

非常重要：对于一个实例，只能调用一次 Release()方法，多次调用可能会导致数据覆盖等错误。

非常重要：对象释放时，只需要调用最顶层消息对象实例的 Release()方法，自动会循环、递归地把顶层消息中引用的所有消息实例就进行 Release()。

使用 using 块自动释放消息类对象

所有消息类对象都实现了 C# IDisposable 接口，在 Dispose()方法中调用了 Release()方法释放消息类对象到对象池，也就是说可以使用 using 块进行自动的消息类对象释放。

```
using (MBLogin mbLogin = MBLogin.Instance) { // 获得消息类MBLogin的实例
    // 使用消息类
    mbLogin.passport = "john";
    mbLogin.password = "123456";
    session.SendCommand(1000, mbLogin);
} // 结束using块时，会自动调用mbLogin.Dispose()方法释放mbLogin
```

对象池活动监控

C#代码中，可以调用 ObjectFactory.ObjectPool.ToString()方法得到对象池内部状态信息，输出到屏幕可以看到类似以下的信息：

```
         Prop active=   0 idle=  10
      AllType active=   0 idle=   2
        Login active=   0 idle=   2
    Broadcast active=   0 idle=   1
 LoginResponse active=   0 idle=   1
```

idle 的是对象池中空闲的对象数，active 是活动的对象数，应监控这两

个数值，确保 idle/active 的数量都不大，如果发现某个数持续增长不减少，那就说明代码中存在没有 Release 的实例，应检查相应消息类的 Release 处理。

输出先按照消息类活动对象数量（active）倒序排序，再按照空闲对象数量（idle）倒序排序。

消息类字段的读操作

读操作指获得消息类中定义的各个字段的值，例如当 Client 收到 Server 发来的消息数据时，读取消息中各字段的值。

MDL 中消息中定义的每一个字段，在消息类中，都会有一个同名的 public 成员相对应，用 mbLogin.passport 这样的方式就可以取值。

需要注意的是，对于对象类型的字段：string/array/vector/map，msgbuf 支持传递 null，所以消息类中的这些字段值可能等于 null。

消息类字段的写操作

当 Client 要往 Server 发送数据时，需要根据协议约定向消息类中各字段赋值。基础数据类型字段的赋值与容器类型字段(array/vector/map)的赋值有所不同。

基础数据类型字段的赋值

基础数据类型包括：byte、bool、short、int、long、float、double、string。赋值形式比较简单。

```
// 获得一个AllType消息的实例。
AllType mb = AllType.Instance;

// 为整数型字段赋值
mb.intValue = 100;

// 为bool型字段赋值
mb.boolValue = true;

// 为字符串类型字段赋值
mb.stringValue = "Hello world! 世界，你好！";
```

容器类型字段的赋值

vector

vector 类型的字段，例如 AllType 类中的 `vector<Prop> propList;` 字段，在生成的消息类内部会创建一个缺省的 List<Prop>容器，调用者可以通过与字段名相同但首字母大写的 Property 获得这个容器，向其中添加元素。这样可以使调用者无需自己创建容器类对象，减少对象的创建次数，降低 C# 垃圾回收器的压力。

```
/*
 * 为vector<Prop> propList赋值
 * 直接赋值到AllType内部预先创建的List<Prop>容器中。
 * PropList是AllType定义的一个C# Property，注意首字母大写。
 */
mb.PropList.Add(prop1);
mb.PropList.Add(prop2);
```

对于 C#中类的字段（field 或者称为 attribute）与属性 Property 意义和规范的差异，请参考 Microsoft C#编程指南中对属性（Property）的定义。

map

map 类型的字段，例如 AllType 中的 `map<int, Prop> folders;` 字段，在生成的消息类内部会创建一个缺省的 Distionary<int, Prop>容器，调用者可以通过与字段名相同但首字母大写的 Property 获得这个容器，向其中添加元素。

```
/*
 * 为map<int, Prop> folders赋值
 * 直接复制到AllType内部预先创建的Dictionary<int, Prop>容器中。
 */
mb.Folders[0] = prop1;
mb.Folders[1] = prop2;
```

数组（array）

array 类型的字段，例如 AllType 中的 `int[] intArray;`，赋值方式与上面描述的基础数据类型字段赋值相同。

```
// 为int[] intArray赋值
int[] ary = new int[20];
for (int i = 0; i < 20; ++i) {
    ary[i] = i;
}
mb.intArray = ary;
```

数组没有像 vector 和 map 那样预先创建容器，是因为数组一旦创建以后，长度是不可变的，多次通信时，数组数据的长度不一定相同，所以不能复用。

格式化输出消息类数据

在 Client 与 Server 的联合调试阶段，相互发送数据的正确性是 debug 的重要检查项目，消息类的 ToString()方法能够把消息内容以 nice 的方式输出。

```
// 格式化输出mb的内容
Console.WriteLine(mb.ToString());
```

能够看到类似下面的输出信息：

```
message LoginResponse {
    base : message CommandResponse {
        Int32 errorCode = 0
        String errorMessage = null
    }
    Int64 playerId = 1
    Backpack backpack = null
}
```

消息类的继承

消息类不能被应用层的类继承，因为所有消息类的构造方法都是 internal 的。同时 msgbuf 的对象池实例管理方式，也希望规避由于继承和方法重载实现的不当，可能给对象创建和回收带来的错误。

对于希望把消息类数据用于应用逻辑控制的场合，可以使用组合模式(Composite)和代理方法(Delegate)，也就是应用类中包含消息类的字段，对外提供消息类方法的代理方法。例如：

```
public class AppClass {
    private Prop mbProp; // 这是一个消息类，非继承，组合！

    // 这是一个代理方法
    public int getPropId() {
        return mbProp.propId;
    }

    // ... ...
}
```

上例的方式下，也不要忘记在 mbProp 不再使用时，调用 mbProp.Release()进行释放。

消息类的序列化(serialize)与反序列化(deserialize)

msgbuf 序列化(serialize)和反序列化(deserialize)的载体是 C#标准库中的 System.IO.Stream，也就是说可以把消息类写入到这两个类的的各种派生类中，例如文件流、压缩流、网络流等等。

每个生成的消息类中，均包涵以下两个方法：

```
    /**
     * 向指定的流中写入MessageBean
     *
     * @param output 用于写入MessageBean的System.IO.Stream对象
     */
    public override void Serialize(Stream output);

    /**
     * 从指定的流中读入MessageBean
     *
     * @param format 用于读入MessageBean的System.IO.Stream对象
     */

     public override MessageBean Deserialize(Stream input);
```

调用这两个方法就可以对消息类进行序列化和反序列化处理，如下例所示。

```
/*************************************
 * 序列化输出(Serialize Output)
 *************************************/
MBLogin mbLogin = MBLogin.Instance;
// 为mbLogin的各字段赋值
mbLogin.passport = "john";
mbLogin.password = "123456";
// 把mbLogin写入到一个文件流
FileStream fos = new FileStream("c:\\tmp\\login.dat",
       FileMode.OpenOrCreate);
mbLogin.Serialize(fos);

/*************************************
 * 序列化输入(Serialize Input)
 *************************************/
// 从文件流中读入mbLogin
FileStream fis = new FileStream("c:\\tmp\\login.dat",
       FileMode.Open);
MBLogin mbLogin = MBLogin.Instance;
```

```
mbLogin.Deserialize(fis);
```

使用 Calibur C# Client API 与服务器通信

关于 Calibur 是什么，请参考【名词定义】章的【Calibur】项。Calibur C# Client API 全部包含在 Calibur.dll 中，使用前应先把 Calibur.dll 正确地导入到项目工程中。

Calibur.dll 存放在 msgbuf 安装目录中的 lib\csharp 子目录中，它已经包含了 msgbuf 的 C#实现，除了 Calibur.dll 之外，无需再导入其他 msgbuf 相关的 dll。

凡是用到 Calibur C# Client API 的代码中，需要 using 三个 namespace：MessageBuffer、Calibur、Protocol（应用定义的消息类 namespace）。

```
using MessageBuffer;
using Calibur;
using Protocol;
```

与 Server 创建会话（session）

与 Server 通信的第一步是创建 session，session 建立好后，就通过 session 对象进行网络消息的发送和接收。

```
/*
 * 创建Session对象，参数是msgbuf生成的ObjectFactory实例。
 */
Session session = new Session(ObjectFactory.Instance);

/*
 * 尝试与指定服务器地址建立Session。
 * Open()返回0表示成功建立；非0表示失败。
 */
if (session.Open("123.4.5.6", 10000) == 0) {
    // Session成功建立了
    // ... ...
} else {
    // Session建立失败
    // 友好地提示用户去检查网络信号
    // ... ...
}
```

创建 Session 实例

首先创建一个 Session 的实例，Session 的构造函数需要一个 msgbuf 生成的 ObjectFactory 实例，因为 Client 可能不仅只跟一个 Server 通信，例如

登录服务器和游戏服务器，登录服务器和游戏服务器可能都分别定义了自己的指令和消息，与这两个服务器通信的指令和消息都是完全不同的两个集合，指定不同的 ObjectFactory 参数，就可以分别创建与登录服务器的 Session 和与游戏服务器的 Session。

与服务器建立会话

接下来调用 Session.Open()方法尝试与 Server 建立会话，第一个参数指定 Server 的 ip 地址，当前仅支持"123.4.5.6"这样的 IPv4 地址，还不支持 IPv6 和域名；第二个参数指定 Server 的端口号。

会话可能会有很多原因建立失败，例如：服务器 IP 地址不可达、服务器应用没启动等。Session.Open()方法返回值是 int 型的，当返回 0 时，表示会话成功建立；非 0 时表示失败，值是错误码。当前可能的错误返回码如下：

-1	Session 已经处于正常连接状态，也就是说对已经建立好的一个 Session 对象又调用了一次 Open()。
-2	ip 地址参数错误。
-3	服务器拒绝连接，可能由于服务器所在网络的防火墙配置、服务器系统配置或者服务器应用没有启动等原因造成的拒绝连接。

Session 建立成功后，应用可以保留这个 Session 对象，利用这个对象与服务器之间发送和接收指令。

发送指令到服务器

```
Login mbLogin = Login.Instance;
mbLogin.passport = "john";
mbLogin.password = "123456";
try {
    // 调用SendCommand()方法发送指令
    session.SendCommand(1000, mbLogin);
} catch (SessionLostException sle) { // 网络层异常，session已经丢失。
    // 重要：关闭session，终止session内部的收发线程
    session.Close();
    // 其他异常处理
    // ... ...
} finally {
    // 确保释放mbLogin，否则内存漏洞。
    mbLogin.Release();
}
```

- session.SendCommand()的第一个参数是整型的指令编号，第二个参数是 MDL 里定义的输入消息类。

- session.SendCommand()方法的调用，应当放在 try...catch 中，因为这个方法会抛出 SessionLostException。
- 当 catch 到 SessionLostException 时，应调用 session.Close()方法终止 session 对象内部的网络数据读/写线程，并进行内部清理。
- 最好在 finally 块中，调用发送消息的 Release()方法，这样不管 SendCommand()是否成功，都能保证 MessageBean 被 Release。
- session.SendCommand()方法是异步的，方法返回调用者时，并不表示数据已经被发送出去，只是把消息对象放在了发送队列上等待 session 内部发送线程真正进行发送。
- session.SendCommand()一旦返回，作为参数的 MessageBean 实例可以立即调用 Release()方法释放，session 内部不会再使用它。

接收服务器送达的指令

```
MessageBean mbean;
while (true) { // 假设这个是游戏的主循环
    // 各种绘制及逻辑处理
    // ... ...

    mbean = null;
    try {
        // 接收指令
        mbean = session.RecvCommand();
    } catch (SessionLostException sle) {
        // 重要：关闭session，终止session内部的收发线程
        session.Close();
        // 其他异常处理
        // ... ...
    }

    if (mbean != null) { // 收到了指令
        // 根据指令编号调度响应的处理
        // ...
    }
}
```

- session.RecvCommand()方法是从 session 内部的接收队列上取得一个已经接收到的指令。
- session.RecvCommand()方法会抛出 SessionLostException，因此必须放在 try ... catch 块中。
- catch 到 SessionLostException 时，第一件事是调用

session.Close()关闭 session。

- session.RecvCommand()返回给应用的任何 MessageBean，都必须在使用完后进行调用 MessageBean.Release()方法释放，否则内存漏洞。

8. Java 语言映射

下面的命令行例子中，假定 base.mdl 和 test.mdl 的存放目录是 **D:\mdl**；假定 C#工程的源代码目录是 **D:\workspace\DemoGame\src**。实际使用时，请替换成实际的路径。

编译 MDL 生成 Java 代码

命令行编译

在命令行执行以下命令：（黄色斜体字是需要根据具体环境修改的）

```
C:> mc.bat -j2se -override -package com.example.demo.protocol -srcdir
D:\workspace\DemoGame\src -prefix MB -mdlencoding utf8 -encoding utf8
D:\mdl\test.mdl
```

- -j2se：必选，无参数，指定生成 Java。
- -override：可选，无参数，强制覆盖上次生成的代码。MessageCompiler 在生成代码时，每个 message 会生成一个 Java 的 class，class 的名字就是 MDL 中定义的 message 的名字。并会根据 message 的定义生成一个电子摘要(Digest)，如果不指定 -override 选项，则会比对本次生成的 Digest 与上次的是否相同，如果相同，就不再生成。而指定了-override 的话，就不管内容是否有改动，都强制重新生成。
- -package：必选，参数是生成出 Java 消息类的 package。这个 package 由开发者自己指定合适的名字，本例中设定为 com.example.demo.protocol。
- -srcdir：必选，参数是目标工程的源代码目录。注意生成的消息类源代码不是直接放在这个目录下，而是会这个目录下根据 Java package 规范的子目录中。本例中，目标工程源代码目录设定为 D:\workspace\DemoGame\src，-package 设定为 com.example.demo.protocol，则消息类的源代码被放置在 D:\workspace\DemoGame\src\com\example\demo\protocol 目录下。
- -prefix：可选，参数是生成消息类名的前缀，例如设定了-prefix 为 MB，则 Login 这个消息实际生成的消息类名为 MBLogin。
- -mdlencoding：可选，参数是 MDL 文件的字符集编码。不指定时，MessageCompiler 用系统环境缺省的字符集编码读入 MDL 文件。在简体中文版 Windows XP/7/8 系统中，系统缺省的字符集编码是

gb2312；Linux 系统中缺省是 iso-8859-1，可以设置环境变量 LANG 来指定。

- -encoding：可选，参数是生成 Java 源代码的字符集编码。不指定时，用系统环境缺省的字符集编码。如果工程要考虑多国语言支持，最好把-encoding 设置称为 UTF-8。
- 总装 MDL 文件路径名：必选，指定总装的 MDL 文件路径。

执行以上命令行后，如果一切正常，将会看到如下的运行信息：

```
警告：[base.mdl:47] 定义了消息 SceneMap ，但没有任何地方使用它。
生成 D:\workspace\DemoGame\src\com\example\demo\protocol\ObjectFactory.java ... 完成。
生成 Login 到 D:\workspace\DemoGame\src\com\example\demo\protocol\Login.java ... 完
成。
生成 Walk 到 D:\workspace\DemoGame\src\com\example\demo\protocol\Walk.java ... 完成。
生成 LogoutResponse 到
D:\workspace\DemoGame\src\com\example\demo\protocol\LogoutResponse.java ... 完成。
生成 Prop 到 D:\workspace\DemoGame\src\com\example\demo\protocol\Prop.java ... 完成。
生成 AllType 到 D:\workspace\DemoGame\src\com\example\demo\protocol\AllType.java ...
完成。
生成 Backpack 到
D:\workspace\DemoGame\src\com\example\demo\protocol\Backpack.java ... 完成。
生成 NullMessageBean 到
D:\workspace\DemoGame\src\com\example\demo\protocol\NullMessageBean.java ... 完成。
生成 Broadcast 到
D:\workspace\DemoGame\src\com\example\demo\protocol\Broadcast.java ... 完成。
生成 LoginResponse 到
D:\workspace\DemoGame\src\com\example\demo\protocol\LoginResponse.java ... 完成。
生成 SceneMap 到
D:\workspace\DemoGame\src\com\example\demo\protocol\SceneMap.java ... 完成。
生成 CommandResponse 到
D:\workspace\DemoGame\src\com\example\demo\protocol\CommandResponse.java ... 完成。
源代码已经成功生成到目标目录 D:\workspace\DemoGame\src 下。
```

命令行成功执行后，会在上面描述的目录下为每一个消息生成一个消息类的 Java 源代码文件，例如消息 Backpack 会生成到 Backpack.java 中，同时还会额外地生成两个类 NullMessageBean.java 和 ObjectFactory.java。

NullMessageBean.java 定义了公共的空消息类，即 command 定义中指定为 null 的消息。

ObjectFactory.java 是所有消息类的单例(singleton)工厂(factory)模式类，负责管理指令编号与消息的映射关系，以及用对象池的方式管理所有消息类的实例。

在 Java 目标工程中，导入进命令行编译创建的所有消息类后，就可以使用这些消息类。

常见错误提示信息

命令行指定的 MDL 或者 import 的 MDL 路径名错误

```
打开消息定义文件D:\workspace\DemoGame\mdl\basea.mdl错误，请检查文件路径及访问权限。
```

语法错：关键字敲错

```
文件：D:\workspace\DemoGame\mdl\base.mdl
行号：9
列号：1
错误：在"mesage"附近语法错。
```

语法错：关键字被用作消息名或者字段名

```
文件：D:\workspace\DemoGame\mdl\base.mdl
行号：49
列号：6
错误：在"id"附近语法错。
```

消息名字重复

```
文件：D:\workspace\DemoGame\mdl\base.mdl
行号：36
列号：14
错误：消息名字重复，Prop 在 base.mdl:9 已经定义过。
```

使用了未定义的消息名

```
文件：D:\workspace\DemoGame\mdl\base.mdl
行号：41
列号：9
错误：引用了未定义的消息类型 Mission。
```

消息对象实例的创建和回收

通常方式

msgbuf Java 映射的实现中，支持通过普通的 new 方式创建 MessageBean，由 JVM 的垃圾回收机制来回收不用的 MessageBean 实例。

```
// 用通常的方式创建 MessageBean 实例
Login mbLogin = new Login();
```

对象池方式

msgbuf 内部实现了一个对象池，用对象池可以减少对象创建次数，降低 GC 压力，提高性能。但是编码的要求提高了，对象的创建和释放都必须得到妥善的处理，否则就造成内存漏洞。

要创建对象池方式管理的 MessageBean 实例，需要用消息类提供的静态工厂方法 getInstance()来创建。

```
// 得到一个 Login 的实例
Login mbLogin = Login.getInstance();
```

非常重要：用 MessageBean.getInstance()方法得到的实例，当使用完成时，必须调用实例的 release()进行释放，也就是归还到对象池中，以循环使用。如果没有调用 release()，将会造成内存漏洞，直到耗尽所有内存。

```
mbLogin.release(); // 释放 mbLogin
mbLogin = null; // 非强制，但最好这样，避免自己重复使用。

mbProp.release(); // 释放 mbProp
mbProp = null; // 非强制，但最好这样，避免自己重复使用。
```

非常重要：调用了 release()方法释放后的实例，调用者不能再继续使用同一个实例，这个实例可能在其他地方循环使用了，调用者再使用同一个实例的话，可能把其他逻辑的数据给覆盖了。

非常重要：对于一个实例，只能调用一次 release()方法，多次调用可能会导致数据覆盖等错误。

非常重要：对象释放时，只需要调用最顶层消息对象实例的 release()方法，自动会循环、递归地把顶层消息中引用的所有消息实例就进行 Release()。

对象池活动监控

Java 代码中，可以调用 `ObjectFactory.getInstance()getMessageBeanPool().toString()` 方法得到对象池内部状态信息，输出到屏幕可以看到类似以下的信息：

```
          Prop actives:0   idles:10
       AllType actives:0   idles:2
         Login actives:0   idles:1
 LoginResponse actives:0   idles:1
     Broadcast actives:0   idles:1
```

idles 的是对象池中空闲的对象数，actives 是活动的对象数，应监控这两个数值，确保 idles/activse 的数量都不大，如果发现某个数持续增长不减少，那就说明代码中存在没有 release 的实例，应检查相应消息类的 release 处理。

输出先按照消息类活动对象数量（actives）倒序排序，再按照空闲对象数量（idles）倒序排序。

消息类字段的 getter/setter

按照 Java 标准规范，MessageBean 的每个字段都有相应的 getter/setter 方法。

对于对象类型的字段：string/array/vector/map，msgbuf 支持传递 null，所以消息类中的这些字段值可能等于 null。

基础数据类型字段的写操作

当 Client 要往 Server 发送数据时，需要根据协议约定向消息类中各字段赋值。基础数据类型字段就用相应的 setter 方法赋值即可。

内部预创建容器类对象的使用

容器类型(vector/map)的字段，在生成的消息类内部会为其创建一个缺省的容器(ArrayList/HashMap)，例如调用者可以通过 newPropList()方法获得 AllType 内部预先创建的 ArrayList<Prop>对象，直接向里面添加元素。

```
// 为vector<Prop> propList赋值
// 直接赋值到AllType内部预先创建的List<Prop>容器中。
List<Prop> propList = mb.newPropList();
propList.Add(prop1);
propList.add(prop2);
```

数组长度不可变，所以数组类型没有预先分配的容器。

消息类的继承

消息类可以被继承，不过不建议继承，建议使用组合模式和代理方法，也就是应用类中包含消息类的字段，对外提供消息类方法的代理方法。例如：

```
public class AppClass {
    private Prop mbProp; // 这是一个消息类，非继承，组合！

    // 这是一个代理方法
    public int getPropId() {
        return mbProp.propId;
    }
    // … …
}
```

eclipse 开发环境中，菜单项 Source->Generate Delegate Methods…功能可以方便地用一个类的成员生成代理方法，继承带来的好处不多，增加了对象创建和 copy。

上例的方式下，如果 mbProp 是对象池方式管理的实例，也不要忘记在 mbProp 不再使用时，调用 mbProp.release()进行释放。

消息类的序列化(serialize)与反序列化(deserialize)

msgbuf 序列化(serialize)和反序列化(deserialize)的载体是 Java 标准库中的 java.io.InputStream 和 java.io.OutputStream，也就是说可以把消息类写入到这两个类的的各种派生类中，例如文件流、压缩流、网络流等等。

每个生成的消息类中，均包涵以下两个方法：

```
    /**
     * 向指定的流中写入MessageBean
     *
     * @param output 用于写入MessageBean的java.io.OutputStream对象
     */
    public void serialize(OutputStream output);

    /**
     * 从指定的流中读入MessageBean
     *
     * @param format 用于读入MessageBean的java.io.InputStream对象
     */

     public MessageBean deserialize(InputStream input);
```

调用这两个方法就可以对消息类进行序列化和反序列化处理，如下例所示。

```
/**************************************
 * 序列化写出到流(Serialize Output)
 **************************************/
MBLogin mbLogin = MBLogin.getInstance();
// 为mbLogin的各字段赋值
mbLogin.passport = "john";
mbLogin.password = "123456";
// 把mbLogin写入到一个文件流
FileOutputStream fos = new FileOutputStream("c:\\tmp\\login.dat");
mbLogin.serialize(fos);

/**************************************
 * 从流中反序列化读入(Deserialize Input)
```

```
 ************************************/
// 从文件流中读入mbLogin
FileInputStream fis = new FileInputStream("c:\\tmp\\login.dat");
MBLogin mbLogin = MBLogin.getInstance();
mbLogin.deserialize(fis);
```

9. C++语言映射

下面的命令行例子中，假定 base.mdl 和 test.mdl 的存放目录是 **D:\mdl**；假定 C++工程的源代码目录是 **D:\workspace\DemoGame\src**。实际使用时，请替换成实际的路径。

编译 MDL 生成 C++代码

命令行编译

在命令行执行以下命令：（黄色斜体字是需要根据具体环境修改的）

```
C:> mc.bat -cpp -package protocol -srcdir D:\workspace\DemoGame\src -prefix MB -mdlencoding utf-8 -encoding utf-8 D:\ mdl\test.mdl
```

- -cpp：必选，无参数，指定生成 C++代码。
- -package：必选，参数是生成出 C++消息类的 namespace。这个 namespace 由开发者自己指定合适的名字，本例中设定为 protocol。
- -srcdir：必选，参数是目标工程的源代码目录，编译生成的 C++源代码就存放在这个目录下。
- -prefix：可选，参数是生成消息类名的前缀，例如设定了-prefix 为 MB，则 Login 这个消息实际生成的消息类名为 MBLogin。
- -mdlencoding：可选，参数是 MDL 文件的字符集编码。不指定时，MessageCompiler 用系统环境缺省的字符集编码读入 MDL 文件。在简体中文版 Windows XP/7/8 系统中，系统缺省的字符集编码是 gb2312；Linux 系统中缺省是 iso-8859-1，可以设置环境变量 LANG 来指定。
- -encoding：可选，参数是生成 C#源代码的字符集编码。不指定时，用系统环境缺省的字符集编码。如果工程要考虑多国语言支持，最好把-encoding 设置称为 UTF-8。
- 总装 MDL 文件路径名：必选，指定总装的 MDL 文件路径。

执行以上命令行后，如果一切正常，将会看到如下的运行信息：

```
警告：[base.mdl:47] 定义了消息 SceneMap ，但没有任何地方使用它。
生成头文件 D:\workspace\DemoGame\src\protocol.h ...
生成类实现文件 D:\workspace\DemoGame\src\protocol.cpp ...
源代码已经成功生成到 D:\workspace\DemoGame\src 目录下。
```

命令行成功执行后，会在指定目录下，以-package 指定的 namespace 为文件名，生成一个.h 头文件和一个.cpp 源代码文件。本例-package 的参数为 protocol，则生成了 protocol.h 和 protocol.cpp。

protocol.h 中是所有用户定义的消息类、空消息（NullMessageBean）和消息类对象工厂（ObjectFactory）的申明；protocol.cpp 是这些类方法的实现。

其中消息类对象工厂类 ObjectFactory 是所有消息类的单例(singleton)工厂(factory)模式类，负责管理指令编号与消息的映射关系，以及用对象池的方式管理所有消息类的实例。

在 C++目标工程中，导入进 protocol.h 和 protocol.cpp 后，就可以使用这些消息类。

常见错误提示信息

命令行指定的 MDL 或者 import 的 MDL 路径名错误

```
打开消息定义文件D:\workspace\DemoGame\mdl\basea.mdl错误，请检查文件路径及访问权限。
```

语法错：关键字敲错

```
文件：D:\workspace\DemoGame\mdl\base.mdl
行号：9
列号：1
错误：在"mesage"附近语法错。
```

语法错：关键字被用作消息名或者字段名

```
文件：D:\workspace\DemoGame\mdl\base.mdl
行号：49
列号：6
错误：在"id"附近语法错。
```

消息名字重复

```
文件：D:\workspace\DemoGame\mdl\base.mdl
行号：36
列号：14
错误：消息名字重复，Prop 在 base.mdl:9 已经定义过。
```

使用了未定义的消息名

```
文件：D:\workspace\DemoGame\mdl\base.mdl
行号：41
列号：9
错误：引用了未定义的消息类型 Mission。
```

消息类的 namespace

C++的类路径用 namespace 进行管理，生成消息类的 namespace 就是命令行选项中-package 指定的名字。因此应用中要使用生成的消息类时，需要用 `using namespace protocol;` 语句来指定使用 protocl 这个 namespace，如果类名与其他 namespace 冲突时，需要用 protocol::前缀来特别指定。

同时，还需要用 `using namespace msgbuf;` 语句指定使用 msgbuf 的运行时刻支撑库。所以应该是：

```
using namespace msgbuf; // 使用 msgbuf 运行时刻支撑库
using namespace protocol; // 使用 MDL 生成的那些消息类
```

消息对象实例的创建和回收

msgbuf 对消息类的实例进行强制对象池管理，生成的消息类代码中把消息类的构造函数修饰符设定为 private，因此调用者无法用变量声明以及 new 操作符来创建新的消息类实例，如下的方式编译时都会报错：

```
// 变量声明创建类对象，编译报错！
AllType allType1;

// 用new操作符创建对象，编译报错！
AllType *allType2 = new AllType();
```

gcc 编译器的报错信息为：

```
../src/protocol.h:402:2: error: 'protocol::AllType::AllType(int)' is private
../src/test.cpp:94:10: error: within this context
../src/protocol.h:403:10: error: 'virtual protocol::AllType::~AllType()' is private
../src/test.cpp:94:10: error: within this context
```

要创建消息类实例，只能通过消息类的静态方法 instance()获得。

```
// 得到一个 AllType 实例（从对象池获得）
AllType allType = AllType::instance();
```

非常重要：消息类实例创建后，当使用完成时，必须调用消息类对象指针的 release()方法进行释放，也就是归还到对象池中，以循环使用。如果没有调用 release()，将会造成内存漏洞，直到耗尽所有内存。

```
mbLogin->release(); // 释放mbLogin
mbLogin = NULL; //非强制，但最好这样，避免自己重复使用。
```

非常重要：调用了 release()方法释放后的实例，调用者不能再继续使用同一个实例，这个实例可能在其他地方循环使用了，调用者再使用同一个实例的话，可能把其他逻辑的数据给覆盖了。

非常重要：对于一个实例，只能调用一次 release()方法，多次调用可能会导致数据覆盖等错误。

非常重要：对象释放时，只需要调用最顶层消息对象实例的 release()方法，自动会循环、递归地把顶层消息中引用的所有消息实例就进行 release()。

对象池活动监控

C++代码中，可以调用用如下语句输出 msgbuf 内部对象池状态套屏幕。

```
ObjectFactory::instance()->pool()->print(std::cout);
```

可以看到类似以下的信息：

```
          Prop active=   0 idle=  10
       AllType active=   0 idle=   2
         Login active=   0 idle=   2
     Broadcast active=   0 idle=   1
 LoginResponse active=   0 idle=   1
```

idle 的是对象池中空闲的对象数，active 是活动的对象数，应监控这两个数值，确保 idle/active 的数量都不大，如果发现某个数持续增长不减少，那就说明代码中存在没有 release 的实例，应检查相应消息类的 release 处理。

输出先按照消息类活动对象数量（active）倒序排序，再按照空闲对象数量（idle）倒序排序。

消息类字段的读操作

读操作指获得消息类中定义的各个字段的值，例如当 Client 收到 Server 发来的消息数据时，读取消息中各字段的值。

MDL 中消息中定义的每一个字段，在消息类中，都会有一个同名的 public 成员相对应，用 mbLogin->passport 这样的方式就可以取值。

需要注意的是，对于对象类型的字段：string/array/vector/map，消息类中全部定义为指针，所以这些字段值可能等于 NULL。

消息类字段的写操作

当 Client 要往 Server 发送数据时，需要根据协议约定向消息类中各字段赋值。基础数据类型字段的赋值与对象类型字段(string/array/vector/map)

的赋值有所不同。

对于对象类型的字段：string/array/vector/map，消息类内部都有预先分配的对象，以利于减少内存分配和释放(new/delete)的次数，如何使用这些预分配对象，在接下来的内容中描述。

基础数据类型字段的赋值

基础数据类型包括：byte、bool、short、int、long、float、double。赋值形式比较简单。

```
// 获得一个AllType消息的实例。
AllType *mb = AllType::instance();

// 为整数型字段赋值
mb->intValue = 100;

// 为bool型字段赋值
mb->boolValue = true;

// 为float型字段赋值
mb->floatValue = 3.14;
```

string 类型字段的赋值

用已有的 string 对象进行赋值

```
// 代码中原有的string对象
std::string msg = "Hello world! 世界，你好！";

// 赋值到MessageBean中
allType->stringValue = &msg;
```

赋值到消息类中预创建的 string 对象

```
// 得到内部预创建string的指针
std::string *ptr = allType->pStringValue;

// 赋值
ptr->append("Hello World! 世界，你好！");
```

数组(array)类型字段的赋值

C++中普通的数组本质是指针，因而仅通过指针本身是无法获得数组元

素个数的，msgbuf 定义了一个 msgbuf::varray 类对数组进行封装，通过 varray 可以获得数组元素的个数。

```
// 获得消息类AllType的实例
AllType *allType = AllType::instance();

// 获得AllType中预创建的数组对象指针
varray<int> *pary = allType->pIntArray;

// 重设数组元素个数为你希望的size
pary->resize(10);

// 为每个元素赋值
for (int i = 0; i < pary->size(); ++i) {
    (*pary)[i] = i;
}
```

vector 类型字段的赋值

使用已有的 std::vector 对象赋值

```
// 获得消息类AllType的实例
AllType *allType = AllType::instance();

// 代码中已有的std::vector对象
std::vector<Prop*> vec;

// 向vector中添加元素1
Prop *prop1 = Prop::instance(); // 当然还需要为prop1的成员赋值
vec.push_back(prop1);

// 向vector中添加元素2
Prop *prop2 = Prop::instance(); // 当然还需要为prop2的成员赋值
vec.push_back(prop2);

// 赋值到AllType中
allType->propVector = &vec;
```

赋值到消息类中预创建的 std::vector 对象

```
// 获得消息类AllType的实例
AllType *allType = AllType::instance();

//获得AllType中预创建的vector对象指针
```

```
std::vector<Prop*> *ptr = allType->pPropVector;

// 向vector中添加元素1
Prop *prop1 = Prop::instance(); // 当然还需要为prop1的成员赋值
ptr->push_back(prop1);

// 向vector中添加元素2
Prop *prop2 = Prop::instance(); // 当然还需要为prop2的成员赋值
ptr->push_back(prop2);
```

map 类型字段的赋值

使用已有的 std::map 对象赋值

```
// 获得消息类AllType的实例
AllType *allType = AllType::instance();

// 代码中已有的std::map对象
std::map<int, Prop*> folders;

// 向vector中添加元素1
Prop *prop1 = Prop::instance(); // 当然还需要为prop1的成员赋值
folders[123] = prop1;

// 向vector中添加元素2
Prop *prop2 = Prop::instance(); // 当然还需要为prop2的成员赋值
folders[456] = prop2;

// 赋值到AllType中
allType->folders = &folders;
```

赋值到消息类中预创建的 std::map 对象

```
// 获得消息类AllType的实例
AllType *allType = AllType::instance();

// 获得AllType中预创建的map对象指针
std::map<int, Prop*> *ptr = allType->pFolders;

// 向vector中添加元素1
Prop *prop1 = Prop::instance(); // 当然还需要为prop1的成员赋值
(*ptr)[123] = prop1;

// 向vector中添加元素2
Prop *prop2 = Prop::instance(); // 当然还需要为prop2的成员赋值
```

```
    (*ptr)[456] = prop2;    ptr->push_back(prop2);
```

格式化输出消息类数据

在 Client 与 Server 的联合调试阶段，相互发送数据的正确性是 debug 的重要检查项目，消息类的 print(ostream&)方法，可以把消息内容格式化输出到指定的输出流中，例如 std::cout。

```
// 格式化输出mb的内容
mb->print(std::cout);
```

能够看到类似下面的输出信息：

```
message LoginResponse {
    base : message CommandResponse {
        Int32 errorCode = 0
        String errorMessage = null
    }
    Int64 playerId = 1
    Backpack backpack = null
}
```

消息类的继承

消息类不能被应用层的类继承，因为所有消息类的构造方法都是 private 的。同时 msgbuf 的对象池实例管理方式，也希望规避由于继承和方法重载实现的不当，可能给对象创建和回收带来的错误。

对于希望把消息类数据用于应用逻辑控制的场合，可以使用组合模式(Composite)和代理方法(Delegate)，也就是应用类中包含消息类的字段，对外提供消息类方法的代理方法。例如：

```
class AppClass {
public:
    Prop *mbProp; // 这是一个消息类，非继承，组合!

    // 这是一个代理方法
    int getPropId() {
        return mbProp->propId;
    }

    // ... ...
};
```

上例的方式下，也不要忘记在 mbProp 不再使用时，调用 mbProp->release()进行释放。

消息类的序列化(serialize)与反序列化(deserialize)

msgbuf 序列化(serialize)和反序列化(deserialize)的载体是 C++标准库中的 std::istream 和 std::ostream，也就是说可以把消息类写入到这两个类的的各种派生类中，例如文件流、压缩流、网络流等等。

每个生成的消息类中，均包涵以下两个 public 方法：

```
/**
 * 向指定的流中写入MessageBean
 *
 * @param output 用于写入MessageBean的std::ostream引用
 */
virtual void serialize(ostream &output);

/**
 * 从指定的流中读入MessageBean
 *
 * @param input 用于读入MessageBean的std::istream引用
 */
 virtual void deserialize(istream &input);
```

调用这两个方法就可以对消息类进行序列化和反序列化处理，如下例所示。

```
/**************************************
 * 序列化输出(Serialize Output)
 **************************************/
Login *mbLogin = Login::instance();
// 为mbLogin的各字段赋值
mbLogin->passport = "john";
mbLogin->password = "123456";
// 把mbLogin写入到一个文件流
ofstream ofs("c:\\tmp\\login.dat", ios::trunc | ios::binary);
mbLogin->serialize(ofs);

/**************************************
 * 序列化输入(Serialize Input)
 **************************************/
// 从文件流中读入mbLogin
ifstream ifs("c:\\tmp\\login.dat", ios::binary);
```

```
    Login *mbLogin = Login::instance();
    mbLogin->deserialize(ifs);
```

10. 附录一：MDL 保留的关键字

MDL 关键字

import message vector boolean bool byte short int long float double string map null void compressed extends command

C#关键字

abstract as base bool break byte case catch char checked class const continue decimal default delegate do double else enum event explicit extern false finally fixed float for foreach goto if implicit in int interface internal is lock long namespace new null object operator out override params private protected public readonly ref return sbyte sealed short sizeof stackalloc static string struct switch this throw true try typeof uint ulong unchecked unsafe ushort using virtual void volatile while add alias ascending async await descending dynamic from get global group into join let orderby partial remove select set value var where yield

C++关键字

alignas alignof and and_eq asm auto bitand bitor bool break case catch char char16_t char32_t class compl const constexpr const_cast continue decltype default delete do double dynamic_cast else enum explicit export extern false float for friend goto if inline int long mutable namespace new noexcept not not_eq nullptr operator or or_eq private protected public register reinterpret_cast return short signed sizeof static static_assert static_cast struct switch template this thread_local throw true try typedef typeid typename union unsigned using virtual void volatile wchar_t while xor xor_eq

Objective-C

auto BOOL break Class case bycopy char byref const id continue IMP default in do inout double nil else NO enum NULL extern oneway float out for Protocol goto SEL if self inline super int YES long interface register end restrict implementation return protocol short class signed public sizeof protected static private struct property

switch try typedef throw union catch unsigned finally void synthesize volatile dynamic while selector _Bool atomic _Complex nonatomic _Imaginery retain alloc autorelease release

Java 关键字

abstract continue for new switch assert default goto package synchronized boolean do if private this break double implements protected throw byte else import public throws case enum instanceof return transient catch extends int short try char final interface static void class finally long strictfp volatile const float native super while

Python 关键字

and del from not while as elif global or with assert else if pass yield break except import print class exec in raise continue finally is return def for lambda try

PHP 关键字

__halt_compiler abstract and array as break callable case catch class clone const continue declare default die do echo else elseif empty enddeclare endfor endforeach endif endswitch endwhile eval exit extends final for foreach function global goto if implements include include_once instanceof insteadof interface isset list namespace new or print private protected public require require_once return static switch throw trait try unset use var while xor __CLASS__ __DIR__ __FILE__ __FUNCTION__ __LINE__ __METHOD__ __NAMESPACE__ __TRAIT__

11. 附录二：google varints 编码

Base 128 Varints

Varints 是使用一个或者多个 byte 来序列化整数的方法，特点是越小的数值使用越少的字节数来编码.。

在一个 Varint 中，除了最后一个 byte，其他的每个 byte 的最高有效位(most significant bit 简称 msb)都被置位，这个 msb 被用来表示接下来的一个 byte 也是当前整数编码的一部分. byte 的低 7 位 bit 用来存储该整型数二进制补码的 7 个 bit 分组，整数的低 7bit 被存储在第一个 byte 中.

32 位整数使用 4 字节存储，32 位的整数值 1 同样要使用 4 个字节，比较浪费空间。Varint 采用变长字节的方式存储整数，将高位为 0 的字节去掉，节约空间

高位为 0 的字节去掉以后，用来存储整数的每一个字节，其最高有效位（most significant bit）用作标识位，0 表示这是整数的最后一个字节，1 表示不是最后一个字节；其他 7 位用于存储整数的数值。字节序采用 little-endian

示例：

整数 1，Varint 的二进制值为 0000 0001。因为 1 个字节就足够，所以最高有效位为 0，后 7 位则为 1 的原码形式

整数 300，Varint 需要 2 字节表示，二进制值为 1010 1100 0000 0010。第一个字节最高有效位设为 1，最后一个字节最高有效位设为 0。解码过程如下：

a). 首先每个字节去掉最高有效位，得到：010 1100 000 0010

b). 按照 little-endian 方式处理字节序，得到：000 0010 010 1100

c). 二进制值 100101100 即为 300

ZigZag 编码

Varint 对于无符号整数有效，对负数无法进行压缩，protocol buffer 对有符号整数采用 ZigZag 编码后，再以 varint 形式存储

对 32 位有符号数，ZigZag 编码算法为 (n << 1) ^ (n >> 31)，对 64 位有符号数的算法为(n << 1) ^ (n >> 63)

注意：32 位有符号数右移 31 位后，对于正数所有位为 0，对于负数所有位为 1。

编码后的效果是 0=>0, -1=>1, 1=>2, -2=>3, 2=>4……，即将无符号数编码为有符号数表示，这样就能有效发挥 varint 的优势了。

12. 附录三：理解字节序

BigEndian/LittleEndian

字节序是特定电脑硬件存放多字节数据(short, int, long, float, double 等)时的字节顺序，Big Endian 是指低地址存放最高有效字节（MSB），而 Little Endian 则是低地址存放最低有效字节（LSB）。

例如一个 4 字节(32 位)的 int 型整数，值为 0x123456，它在 Big Endian 和 Little Endian 的硬件上，存储的方式如下：

物理内存地址	Big Endian	Little Endian
低地址	12	78
↓	34	56
	56	34
高地址	78	12

Java 虚拟机统一采用了 Big Endian 作为内部数据存储字节序，同时 Big Endian 也是绝大部分网络协议采用的字节序。所以通常把把 Big Endian 方式称之为网络字节序。当两台采用不同字节序的主机通信时，在发送数据之前都必须经过字节序的转换成为网络字节序后再进行传输。这个转换很简单，就是根据数据长度，按字节进行次序颠倒。

BigEndian 和 LittleEndian 名词的来源

这两个术语来自于 Jonathan Swift 的《格利佛游记》其中交战的两个派别无法就应该从哪一端——小端还是大端——打开一个半熟的鸡蛋达成一致。

“endian”这个词出自《格列佛游记》。小人国的内战就源于吃鸡蛋时是究竟从大头(Big-Endian)敲开还是从小头(Little-Endian)敲开，由此曾发生过六次叛乱，其中一个皇帝送了命，另一个丢了王位。”

我们一般将 endian 翻译成“字节序”，将 Big Endian 和 Little Endian 称作“大尾”和“小尾”。

在那个时代，Swift 是在讽刺英国和法国之间的持续冲突，Danny Cohen 这位网络协议的早期开创者，第一次使用这两个术语来指代字节顺序，后来这个术语被广泛接纳了。