**通信接口及协议配置组件设计方案**

# 引言

## 应用背景

在业务系统开发过程中经常需要处理类型定义、接口通信、协议解析等重复而繁琐的开发工作，这些内容处理起来不仅费时，而且经常容易出错，修改维护也比较困难。通信接口及协议配置组件正是针对这一难题，将繁复的开发任务进行自动化处理；只需输入少量配置信息便可自动生成指定的生成代码，生成代码将受控对象的类型定义、接口驱动调用、通信报文格式解析等技术细节进行内部封装，对外提供一组简单统一的API接口供业务系统开发时使用。

通信接口及协议配置组件（以下简称“通信配置系统”）是一套专用于实现快速创建软件系统底层通信模块的工具软件，通信配置系统主要用来实现：创建配置信息，并根据创建的配置信息自动生成生成代码，生成代码将数据输入输出操作封装成受控对象的方法提供给业务系统使用。在开发业务系统时，开发人员只需直接使用生成代码中的API接口（受控对象的方法）即可实现完整的数据通信功能，而无需直接处理通信协议的底层细节。使用通信配置系统即省去许多繁复的开发工作，又增强了业务系统的健壮性与可维护性，正真使开发人员更集中精力去完成应用系统的业务逻辑。

## 定义

* 通信配置系统

通信配置系统是指实现通信模块、配置信息编辑、解析，及自动生成代码功能的软件系统的统称。通信配置系统的输入内容是配置信息，输出内容是生成代码。

* 配置项

配置项指配置信息中的条目，是按照自定义格式书写的文本代码，每个项目的配置信息均由一系列配置项组成，每一配置项均与面向对象中的域对应。

* 生成代码

生成代码是通信配置系统根据配置内容自动生成的程序代码，是一组C#、C++源代码，生成代码实现对所配置对象的定义，并提供协议通信的API函数。

* 受控对象

受控对象用以指代业务系统中的功能组件，是实现独立功能的分系统；受控对象拥有自己的属性，并使用硬件接口资源进行数据报文收发。受控对象是通信配置系统中的核心处理对象。

* 数据帧

数据帧是按照报文格式排列的一段数据内容，是受控对象进行数据通信时的最小数据单位。

# 总体架构

## 设计原则

通信配置系统的设计原则为：稳定、易用、完备、可扩。

稳定：作为业务系统共用的底层通信模块，通信配置系统必须首先保证其自身的稳定运行，本方案基于成熟的经过实践验证的软件模块构建，这是保证系统整体运行稳定的基石。

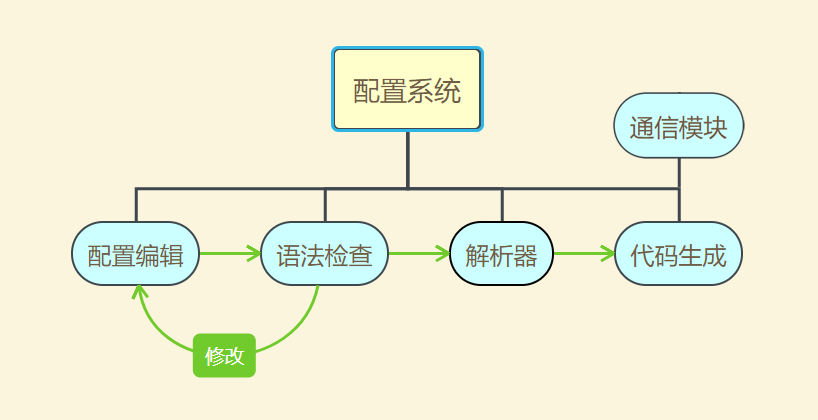
易用：作为配置型工具，通信配置系统需要易于使用，要具备友好的图形化操作界面，且符合用户操作习惯。

完备：作为通用型软件模块，通信配置系统需要适用于协议通信中的各类场景，需具备完善的协议通信处置能力。

可扩：作为可重用代码库，通信配置系统需要具备可扩展的能力，可以很方便的添加其它通信接口，可以与其它平台快速集成。

## 功能模块

通信配置系统由五个主要功能模块组成，分别为：通信模块、配置编辑模块、语法检查模块、解析器和代码生成器。



通信配置系统功能模块

* 通信模块

通信模块提供独立的底层通信接口，实现对硬件通道进行输入输出操作的调用。通信接口类型包括TCP（服务器和客户端）、UDP、串口、CAN和DI/DO。其中CAN通信接口实现对广州致远和研华两个厂家设备驱动的封装，DI/DO接口实现对研华厂家设备驱动的封装。

* 配置编辑模块

配置编辑模块用于创建配置信息，创建配置信息的方式分为可视化编辑和代码编辑两种可选模式，两种模式可以自由切换。

可视化编辑模式下，用户通过选择图形界面上的菜单、工具栏按钮来生成受控对象的各个域（属性、方法和事件）；对每个域内部的详细定义，通过UI表单的方式实现编辑功能。在可视化编辑模式下，用户在图形界面上的每一步操作均会由对应的配置项代码产生，配置信息由通信配置系统自动管理，用户只需按照图形界面上的指引填入相应内容即可生成完整的配置信息，非常适合初级用户使用。

代码编辑模式下，用户的主工作界面是一个WYSIWYG代码编辑器，使用代码编辑器可直接编辑配置信息的原始文本。每个项目的配置信息都是一个独立的配置文件，配置文件由一系列自定义配置项代码组成。代码编辑器实现代码输入、语法高亮显示、复制、粘贴、剪切、撤销、恢复等完善的WYSIWYG编辑功能，对具有一定使用经验的用户，代码编辑模式可大幅提高工作效率。

* 语法检查模块

语法检查模式用来对配置项代码执行语法审核，找出不符合语法规则的配置项代码，并提示给用户，以便进一步修改更正。

配置项代码采用自定义语法规则，在借鉴一些最常用的程序语言规则基础上，针对通信配置系统进行专门的优化设计，语法简洁明了；生成的配置代码文件可读性高，非常易于修改维护和重复使用。

语法检查模块不仅能准确找到配置项代码中的语法错误，而且还可以输出非常直观的提示信息，能准确提示出现错误的具体位置（行号和列号），在代码编辑器中对出现语法错误的关键词句进行明显的可视化标示。

* 解析器

解析器将文本格式的配置信息转换为格式化的抽象语法树（AST），从而实现对配置项代码的解释与翻译工作。

通过语法检查后的配置项代码交由解析器进行处理，解析器逐句分析配置项代码的中的符号，创建符号表，解析语法与语义，按照自下而上的顺序生成抽象语法树（AST）。抽象语法树（AST）是下一步代码生成的工作基础。

* 代码生成器

代码生成器用于直接产生C#或C++格式的生成代码。

代码生成器在得到解析器生成的符号表与抽象语法树（AST）后，按照自上而下的顺序遍历全部抽象语法树（AST），并在抽象语法树（AST）的每个节点位置执行通信代码生成任务。执行通信代码生成任务时，代码生成器首先根据当前节点的语法含义动态生成一组程序代码，然后加载C#或C++代码模板，并将动态代码插入到代码模板中。在遍历完全部的抽象语法树（AST）后，代码生成器重新整理生成的代码文件，按照统一的目录结构规则输出完整的生成代码。

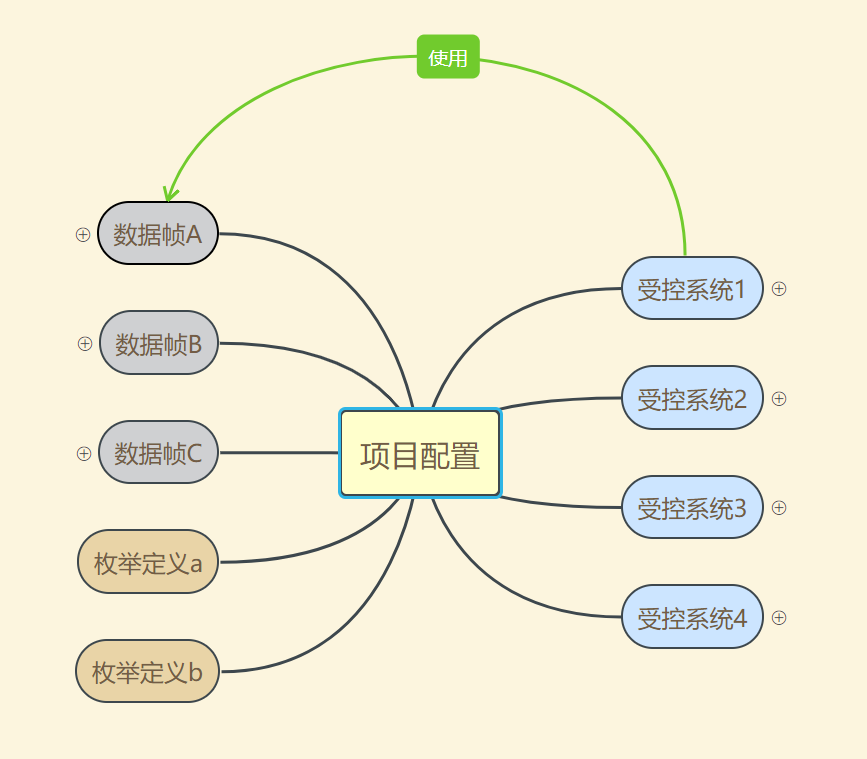
## 输入输出设计

通信配置系统的输入内容是配置信息，输出内容是生成代码；这两部分内容与通信配置系统的使用者直接产生关系，是使用者最关心的内容，本节针对这两部分内容的总体设计进行专门说明。

* + 1. 配置信息

每个项目的配置信息都是一个文本格式的文件，由一组自定义格式的配置项代码编写而成，配置项分为三类：受控对象配置项、数据帧配置项和枚举配置项；其中受控对象配置项是配置信息的最主要内容，数据帧与枚举配置项用以辅助完成对受控对象的配置。

配置信息采用自定义格式编写，其中的关键字与语法规则均为可定制内容，可根据不同的业务特征进行匹配。



配置信息分类

* 受控对象配置项

受控对象配置项用来对受控对象的状态与IO操作进行配置。受控对象配置项按照面向对象模型进行设计，配置项内容与面向对象模型的对应关系如下：

* 每个受控对象对应一个独立的分系统类
* 受控对象的状态值对应分系统类的公有属性，状态值数据类型为值类型或值类型数组
* 受控对象通信时使用的硬件通道对应分系统类的私有成员，使用者无需直接操作这些硬件通道
* 受控对象产生的IO操作对应分系统类的公有方法，对IO操作进行配置时需指定每项IO操作的方向、所使用的硬件通道和数据帧
* 数据帧配置项

数据帧配置项用于配置受控对象通信时使用的报文格式，每一种报文格式对应一个数据帧配置项。

数据帧配置项由一组字段配置项构成，字段配置项按照字节流顺序依次配置。数据帧的每个字段可按照数据类型或按位进行配置，字段类型既与受控对象的状态值类型兼容，又能与已存在的各类报文格式兼容。

对数据帧字段的配置，除配置数值类型外，还可配置计算值、默认值、最大值、最小值、CRC校验规则、动态解析规则等内容。

* 枚举配置项

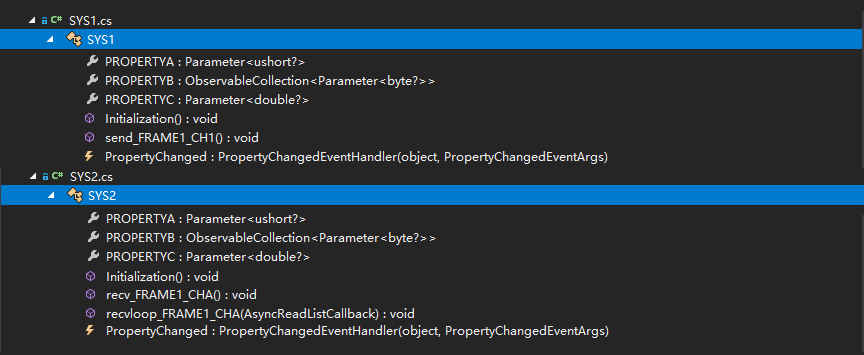
枚举配置项与生成代码中用到的枚举定义完全一致，用以辅助完成受控对象配置与数据帧配置。

* + 1. 生成代码API

生成代码由通信配置系统自动生成，生成代码将数据通信中需要处理的诸多要素和环节进行了隐藏封装，仅公开一组简单的API供业务系统使用；生成代码按照面向对象模式进行设计，API的具体形式为类的公有域。

每个受控对象对应一个分系统类，受控对象的状态值对应分系统类的公有属性，输入输出操作对应分系统类的公有方法，当分系统类的属性值发生改变时，会自动触发相应的消息事件。

生成代码API示例如下：



生成代码API示例

* 公有属性

具有公有访问权限的分系统类属性，由配置项中的受控对象状态值产生，生成代码的使用者通过访问这些属性来获取受控对象的状态值。属性的数据类型全部采用泛型Parameter<T>，在C#代码里，T代表一个值类型的Nullable类型；值类型数组对应的属性类型为ObservableCollection<Parameter<T>>类型。

在C#中，分系统类的属性可以作为WPF控件的数据源进行绑定，方便实现数据驱动模型下的编程开发。

* 公有方法

分系统类的公有方法分两类：系统资源相关的方法及输入输出操作方法。

系统资源相关的方法用于初始化系统资源及资源的释放操作，如打开或关闭硬件端口，这些方法具有统一通用的名称，如Initialization()、Release()等。

输入输出操作方法由受控对象的IO操作配置项产生，根据配置项中指定的IO方向、通道及使用的数据帧自动命名，每一个IO配置项都有对应的方法。

分系统类的公用方法是生成代码最重要的API，开发人员主要使用这些方法来完成底层接口通信，这些方法按照统一的规则进行命名，通常没有输入输出参数，非常易于使用；方法执行过程中出现错误（包括硬件错误）时，生成代码的内部实现会自动将这些错误信息转换为异常对象，抛出给业务系统主线程。

* 公有事件

分系统类的公有事件由代码生成器自动创建，在C#中该事件的类型为PropertyChanged，该事件在分系统类的公有属性值发生改变时自动触发。通过在分系统类的对象上订阅该事件，便可实现对受控对象状态值的动态监控。

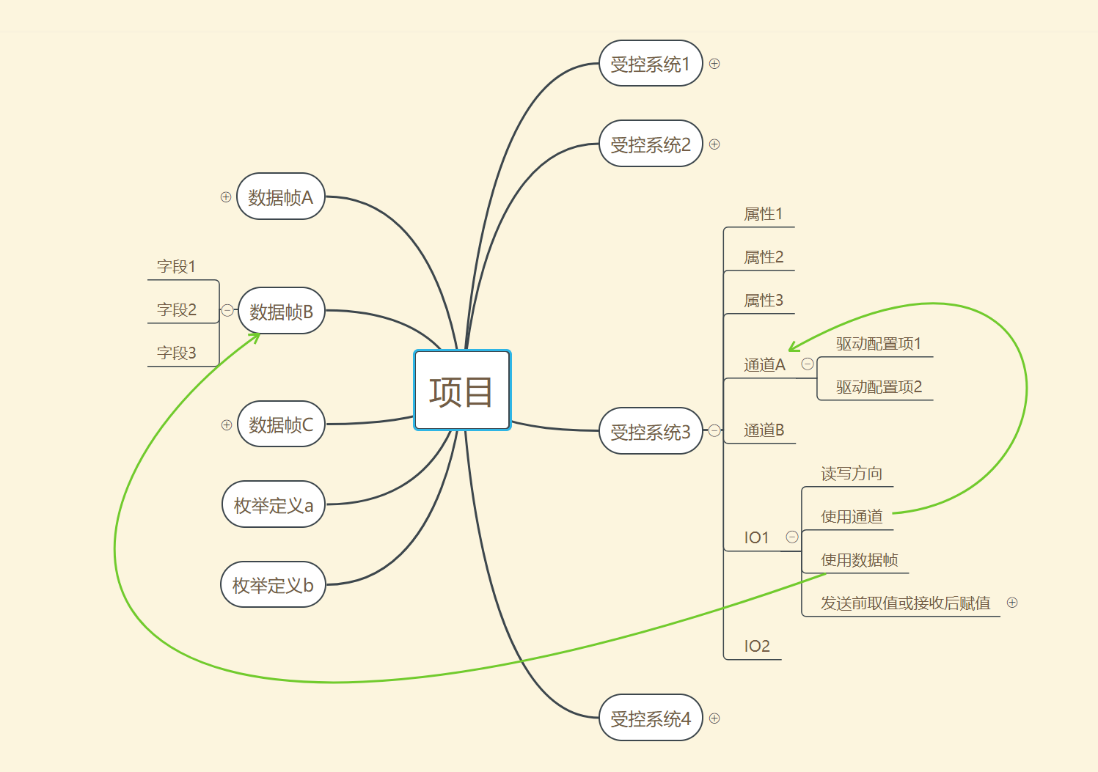
# 详细设计

## 通信模块设计

通信模块提供独立的底层通信接口，实现对硬件通道进行输入输出操作的调用。通信接口类型包括TCP（服务器和客户端）、UDP、串口、CAN和DI/DO。其中CAN通信接口实现对广州致远和研华两个厂家设备驱动的封装，DI/DO接口实现对研华厂家设备驱动的封装。

## 配置项内容设计

配置内容分为受控对象配置项、数据帧配置项和枚举配置项三类，其中核心的配置项是受控对象配置项，数据帧与枚举配置项用以辅助对受控对象进行配置，配置项的组成关系如下图：



配置项结构图

以下对各类配置项进行逐一说明：

* + 1. 受控对象配置项

受控对象配置项主要描述三方面内容：受控对象有哪些状态值需要管理，受控对象使用什么硬件接口进行通信，以及通信时按照什么样的报文格式收发数据。

* 受控对象属性

受控对象配置项中，每项属性代表一项需要管理的受控对象状态值。向受控对象配置项中增加属性配置时，需要指定属性名称和数值类型。

属性的名称为一个字符串标识，以字母打头，后跟若干个下划线、字母或数字的组合，通信配置系统使用者需确保该名称标识与业务系统中的标识没有冲突。

数值类型为以下选项之一：bool、byte、sbyte、ushort、short、uint、int、ulong、long、float、double或以上类型的数组。对于数组类型还需配置数组的长度，数组长度可以为固定的整数值或运行时动态计算的值。

* 通道

通道是受控对象传输数据时使用的硬件资源，向受控对象配置项中增加通道配置时，需要指定通道的类型、通道名称及通道参数。

可选的通道类型有：can、com、tcpserver、tcpclient、udp、di、do，通道名称是使用者指定的字符串标识，每一种通道类型都有各自不同的通道参数。

* can类型通道的参数

|  |  |
| --- | --- |
| 序号 | 名称 |
| 1. | 通道号 |
| 2. | 波特率 |
| 3. | 验收码 |
| 4. | 屏蔽码 |
| 5. | 工作模式(正常工作/自发自收) |
| 6. | 过滤模式(单滤波/双滤波) |
| 7. | 发送超时 |

* com口类型通道的参数

|  |  |
| --- | --- |
| 序号 | 名称 |
| 1. | 端口 |
| 2. | 波特率 |
| 3. | 奇偶校验 |
| 4. | 数据位长 |
| 5. | 停止位 |

* tcpserver通道的参数

|  |  |
| --- | --- |
| 序号 | 名称 |
| 1. | 监听IP |
| 2. | 监听端口 |
| 3 | 待测设备地址 |
| 4 | 待测设备端口 |

* tcpclient通道的参数

|  |  |
| --- | --- |
| 序号 | 名称 |
| 1. | 设备IP地址 |
| 2. | 端口 |
| 3 | 连接超时 |

* upd通道的参数

|  |  |
| --- | --- |
| 序号 | 名称 |
| 1. | 本地地址 |
| 2. | 本地端口 |
| 3 | 远端地址 |
| 4 | 远端端口 |

* di通道的参数

|  |  |
| --- | --- |
| 序号 | 名称 |
| 1. | 通道号 |
| 2. | 最小信号值 |

* do通道的参数

|  |  |
| --- | --- |
| 序号 | 名称 |
| 1. | 通道号 |
| 2. | 最小信号值 |

* IO 操作

IO操作是受控对象进行数据传输的动作命令，每项IO操作配置代表受控对象进行一次数据输入输出时执行的具体动作。向受控对象配置项中增加IO操作配置时，需要指定IO操作的方向、所使用的通道、使用的数据帧（即具体的报文格式）以及相关赋值取值动作。

操作方向必须配置为发送或接收其中之一，分别代表数据输出和输入数据。

IO操作使用的通道只能引用当前受控对象中已配置通道的名称。

IO操作使用的数据帧（即具体的报文格式）只能引用当前项目中已配置数据帧的名称。

取值动作是指在发送数据前，获取受控对象的属性值，并传递给数据帧的对应字段；赋值动作是指在接收完数据帧后，将完成解析的数据帧字段值赋值给受控对象属性。进行赋值取值动作配置时需要指定受控对象属性与数据帧字段间的对应关系。

* + 1. 数据帧配置项

数据帧配置项用以配置传输协议中规定的报文数据格式，数据帧配置项是受控对象完成数据输入输出的必备条件；每一个数据帧配置项均会产生与之对应的数据帧类，该类内部封装了报文解析的功能。

增加数据帧配置项时只需要指定数据帧的名称即可，数据帧配置项的具体内容由一组字段配置项构成，字段配置项按照字节流顺序依次配置。每项字段配置项需要指定字段名称、字段类型、字段属性及字段相关操作。

* 字段类型与属性

数据帧中的字段类型有：integer、real、text、block四种；分别代表整型、浮点数、字符串和字节块，每种类型均有各自不同的属性。虽然只有四种字段类型，但通过与字段属性的灵活组合，可以配置出各种复杂的报文格式。

每种字段类型均具有多种字段属性，用以配置报文解析的各种不同方式，所有属性均为可选配置项，属性的默认值通常都是协议解析种最常用的选择方式。

**integer 字段属性**包括：signed、bitcount、default、byteorder、encoded、repeated等。

signed属性标识整数是否有符号。

bitcount属性标识字段按位计算的长度。

default属性用来设置字段的默认值。

byteorder属性标识字节排列顺序，字节排列顺序分为大端序（先高字节后低字节）与小端序（先低字节后高字节）两种。

encoded属性标识字段的编码方式，编码方式分为原码（primitive）、反码（inversion）和补码（complement）三种。

repeated属性标识当前字段重复出现的次数。

**real字段属性**包括：isdouble、byteorder、encoded、repeated等。

isdouble属性标识浮点数的是否为双精度（double），isdouble配置为false时表示单精度浮点数（float）。

byteorder属性标识字节排列顺序，字节排列顺序分为大端序（先高字节后低字节）与小端序（先低字节后高字节）两种。

encoded属性标识字段的编码方式，编码方式分为原码（primitive）、反码（inversion）和补码（complement）三种。

repeated属性标识当前字段重复出现的次数。

**text字段属性**包括：tail、alignedlen、repeated等。

tail属性标识字符串的结尾标志。

alignedlen属性标识字符串进行整字节对齐时的字节长度。

repeated属性标识当前字段重复出现的次数。

**block字段属性**包括：type、bytesize、repeated等。

type属性标识block字段所对应的嵌套报文格式

bytesize属性标识当前字段的字节长度

repeated属性标识当前字段重复出现的次数

* 字段操作

字段操作用于对协议解析中的数值转换、计算、动态解析、验证规则等操作进行配置。

大部分解析过程中的数值转换均由解析器自动处理，需要显式配置的数值转换操作为toenum，用于执行将字段值转换为枚举类型的值。

计算操作用于运行时执行表达式计算，表达式是由加、减、乘、除、括号、字段名称组成的计算公式，表达式的结果有运行时动态计算得到。

动态解析用于执行运行时动态选择报文解析方法的操作，如根据某一字段的值针对当前字段选择不同的解析方式。

验证规则用于判断字段的值是否符合一定的规则，其中包括：max、min、check，分别用于配置最大值、最小值、校验规则。其中check校验规则内置的校验算法如下表，此外check校验算法还支持自定义crc算法扩展。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 内置check算法 | | | | |
| 校验模式 | 多项式 | 初始值 | 数据位序 | 结果处理 |
| sum8 | 8位校验和 | | | |
| xor8 | 8位异或值 | | | |
| sum16 | 16位校验和，低字节在前，高字节在后 | | | |
| sum16\_false | 16位校验和，高字节在前，低字节在后 | | | |
| xor16 | 16位异或值，低字节在前，高字节在后 | | | |
| xor16\_false | 16位异或值，高字节在前，低字节在后 | | | |
| sum32 | 32位校验和，低字节在前，高字节在后 | | | |
| sum32\_false | 32位校验和，高字节在前，低字节在后 | | | |
| xor32 | 32位异或值，低字节在前，高字节在后 | | | |
| xor32\_false | 32位异或值，高字节在前，低字节在后 | | | |
| crc4\_itu | 0x03 | 0x00 | 低位在前，高位在后 | 与0x00异或 |
| crc5\_epc | 0x09 | 0x09 | 低位在后，高位在前 | 与0x00异或 |
| crc5\_itu | 0x15 | 0x00 | 低位在前，高位在后 | 与0x00异或 |
| crc5\_usb | 0x05 | 0x1F | 低位在前，高位在后 | 与0x1F异或 |
| crc6\_itu | 0x03 | 0x00 | 低位在前，高位在后 | 与0x00异或 |
| crc7\_mmc | 0x09 | 0x00 | 低位在后，高位在前 | 与0x00异或 |
| crc8 | 0x07 | 0x00 | 低位在后，高位在前 | 与0x00异或 |
| crc8\_itu | 0x07 | 0x00 | 低位在后，高位在前 | 与0x55异或 |
| crc8\_rohc | 0x07 | 0xFF | 低位在前，高位在后 | 与0x00异或 |
| crc8\_maxim | 0x81 | 0x00 | 低位在前，高位在后 | 与0x00异或 |
| crc16\_ibm | 0x8005 | 0x0000 | 低位在前，高位在后 | 与0x0000异或 |
| crc16\_maxim | 0x8005 | 0x0000 | 低位在前，高位在后 | 与0xFFFF异或 |
| crc16\_usb | 0x8005 | 0xFFFF | 低位在前，高位在后 | 与0xFFFF异或 |
| crc16\_modbus | 0x8005 | 0xFFFF | 低位在前，高位在后 | 与0x0000异或 |
| crc16\_ccitt | 0x1021 | 0x0000 | 低位在前，高位在后 | 与0x0000异或 |
| crc16\_ccitt\_false | 0x1021 | 0xFFFF | 低位在后，高位在前 | 与0x0000异或 |
| crc16\_x25 | 0x1021 | 0xFFFF | 低位在前，高位在后 | 与0xFFFF异或 |
| crc16\_xmodem | 0x1021 | 0x0000 | 低位在后，高位在前 | 与0x0000异或 |
| crc16\_dnp | 0x3D65 | 0x0000 | 低位在前，高位在后 | 与0xFFFF异或 |
| crc32 | 0x04C11DB7 | 0xFFFFFFFF | 低位在前，高位在后 | 与0xFFFFFFFF异或 |
| crc32\_mpeg\_2 | 0x04C11DB7 | 0xFFFFFFFF | 低位在后，高位在前 | 与0x00000000异或 |
| crc64 | 0x42F0E1EBA9EA3693 | 0x0000000000000000 | 低位在后，高位在前 | 与0x00000000  00000000异或 |
| crc64\_we | 0X42F0E1EBA9EA3693 | 0xFFFFFFFFFFFFFFFF | 低位在后，高位在前 | 与0xFFFFFFFF  FFFFFFFF异或 |

* + 1. 枚举配置项

枚举配置项与C#、C++中枚举类型定义方式一致，主要用于辅助完成受控对象与数据帧的配置。

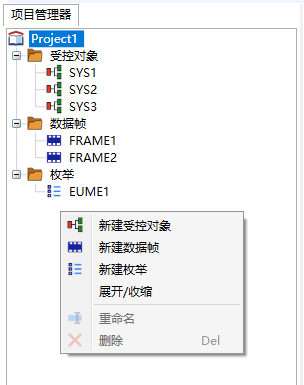
## 配置项编辑方式

编辑配置项的方式分为可视化编辑和代码编辑两种可选模式，两种模式可以自由切换。

* + 1. 可视化编辑模式

可视化编辑模式中，对所有配置项的编辑均通过windows系统中的标准控件进行，通过直观易操作的界面实现全部配置项的编辑功能。用户通过选择图形界面上的菜单、工具栏按钮控制配置项的添加、删除、重命名，配置项的具体内容通过UI表单中的输入框、复选框、下拉框等控件组合进行控制。

使用者在图形界面上的每一步操作均会自动生成相应的配置项代码，自动生成的配置项代码均为语法正确的完整代码，可视化编辑模式下不会产生任何语法错误。自动代码生成由通信配置系统后台自动执行，可视化编辑模式下使用者无需接触配置代码。



项目管理的可视化界面

* + 1. 代码编辑模式

代码编辑模式中，主工作界面是一个所见即所得的代码编辑器，在代码编辑器中加载完整的配置代码内容，通过代码编辑器可直接修改配置信息的原始文本。

代码编辑器除支持中英文输入，支持文本内容复制、粘贴、剪切功能，支持按修改步骤执行撤销、恢复功能。代码编辑器最左侧一列显示配置代码行号；代码行号右侧为配置代码内容，根据自定义语法规则配置代码的不同关键词显示不同的高亮颜色，方便使用者识别，以增强配置代码的可读性。



配置代码编辑器

## 配置项处理

在生成完整的配置项后，通信配置系统便可以进行配置项处理，配置项处理由两个主要有两个环节：语法检查与配置项解析。

* + 1. 语法检查

配置代码语法检查的功能实现分两个步骤：首先进行符号识别，然后进行语法语义合规性判别。

符号识别采用正则表达式扫描器，对配置代码中的所有符号进行扫描识别，并记录出现的具体位置，对无法识别的符号均记录为输入错误。

语法语义判别通过BNF语法分析，对配置代码中的完整配置项进行逐项识别，识别完成后按照识别的语义进行合规性判别，对出现的不合规语法或语义冲突均记录为一项语法错误。

对整个检查过程中发现的问题均记录具体的行号和列号，并在图形界面上输出明确提示信息。

* + 1. 配置项解析

配置项解析的任务由解析器完成，主要任务是将合格的配置代码转换为抽象语法树（AST），主要执行过程是自下而上构建抽象语法树（AST）的过程。

通过语法检查后的配置项代码交由解析器进行处理，解析器逐句解析配置项代码的中的符号，并创建符号表，同时按照配置项语法递归生成语法树的组成项，全部配置项解析完成后便生成了完整的抽象语法树（AST）。

配置项解析主要工作内容为：受控对象配置项解析、数据帧解析、枚举解析；其中受控对象配置项解析是抽象语法树（AST）的树根。解析时先进行数据帧和枚举解析，对数据帧中的逐个字段按照配置内容解析成对应的数据类型，并计算出数据帧打包后的字节大小，对动态计算的内容则需解析为逆波兰表达式；枚举配置项解析结果是整数值与枚举项标识对照表。

受控对象配置项解析是解析器的核心任务，对受控对象的解析需要交叉引用之前解析过程中的结果。解析受控对象配置项时，先解析属性和通道配置项，构建受控对象属性列表和通道资源列表，最后解析IO操作配置项，每项IO操作均会引用到前面各项解析结果。完成受控对象配置项解析后，便生成了完整的抽象语法树（AST）。

配置项解析是通信配置系统产生生成代码的基础，配置项解析应确保准确。该过程由通信配置系统后台自动调用，对使用者是透明的。

## 代码生成

生成代码由代码生成器根据配置项代码解析的结果自动生成，代码生成器在得到解析器生成的符号表与抽象语法树（AST）后，按照自上而下的顺序遍历全部抽象语法树（AST），并在抽象语法树（AST）的每个节点位置执行通信代码生成任务。执行通信代码生成任务时，代码生成器首先根据当前节点的语法含义动态生成一组程序代码，然后加载C#或C++代码模板，并将动态代码插入到代码模板中。在遍历完全部的抽象语法树（AST）后，代码生成器重新整理生成的代码文件，按照统一的目录结构规则输出完整的生成代码。

由通信配置系统自动生成的生成代码中包含分系统类代码、数据帧类代码和枚举定义代码，分别由配置项中的数控对象配置项、数据帧配置项和枚举配置项对应产生。

* + 1. 分系统类代码

每个受控对象配置项会生成一个分系统类定义。

受控对象配置项中的属性配置（即受控对象的状态值）对应生成分系统类的公有属性，属性数据类型为泛型类Parameter<T>， 其中T为配置项中指定的基础类型的Nullable表示。例如：配置项中指定受控对象属性数据类型为long，则对应生成的分系统类属性数据类型为Parameter<long?>。如果配置项中的属性数据类型为基础数据类型的数组，对应生成的分系统类属性数据类型为ObservableCollection<Parameter<T>>，ObservableCollection类型可以实现对数组成员值进行修改时自动触发PropertyChanged事件。

受控对象配置项中的通道配置对应生成分系统类的通道私有变量成员，这些私有成员按照配置项进行命名并在类构造函数中进行初始化，通道私有成员仅能在分系统类内部使用，供IO操作实现中使用。

受控对象配置项中的IO操作配置对应生成分系统类的公有函数。对于数据输出操作，先将分系统类属性赋值给对应的数据帧对象，然后调用数据帧对象的打包接口获取字节流，最后将获取到的字节流在指定的接口上发送出去；对于数据输入操作，先将指定接口上读取到的字节流传递给特定数据帧对象的解包接口，完成数据帧解包后再将数据帧的字段值分别赋值给分系统中对应的属性，如果属性值发生改变则自动触发PropertyChanged事件。

受控对象的构造函数中实现通道的初始化工作，代码实现中会根据配置项中的内容，将通道属性值传递给底层的硬件驱动程序，并按照驱动程序要求的顺序对硬件进行初始化操作。

受控对象的析构函数中实现通道资源的释放工作，代码实现中会按照驱动程序要求的顺序释放相关的硬件资源。

* + 1. 数据帧类代码

数据帧配置项对应生成数据帧类代码，主要功能是报文格式解析，分系统类执行IO操作时需要使用数据帧类进行报文的解包与拆包，在业务系统代码中通常不会直接使用数据帧类代码。

数据帧类的代码按照数据帧字段配置内容逐一生成数据帧类的属性成员，并将每项属性成员在字节流中的位置记录在一个位置表中。数据帧类代码自动实现解包函数和打包函数，解包函数的输入参数为字节流，函数实现时按照位置表中的记录从字节流中取出数据并赋值给对应的字段属性；打包函数的输出为字节流，执行打包函数时，首先会创建一个内存缓冲区，然后按照位置表中的记录将字段属性的值逐一放入缓冲区的相应位置，组后将有效的缓冲区内容作为函数返回值输出。

在数据帧解包与打包报文的过程中需要处理几种特殊情况：

一是计算公式的处理：

在解包与打包过程中遇到计算公式时，会自动调用数据帧类的静态成员函数cacl进行数值计算，cacl函数实现公式动态计算功能，cacl函数的输入值为一个逆波兰式，返回值为公式计算结果数值。

二是动态解析的处理：

在解包与打包过程中需要进行动态解析时，会按照配置项中的内容对应生成一组switch case语句，用来动态选择字段解析方式。

三是数据验证的处理：

在解包过程中当字段定义由min、max属性时，会自动判断解包后的字段值是否落在规定的范围内，出现异常时会自动记录在数据验证表中（通过修改代码模板可以改为抛出异常）。

数据帧类以静态成员函数的方式实现常用的crc校验函数，当解包过程中的字段配置有crc时会调用相应的校验算法，验证失败时会自动记录在数据验证表中（通过修改代码模板可以改为抛出异常）；当打包过程中的字段配置有crc时会自动调用相应的校验算法生成校验字段值。

* + 1. 枚举定义代码

枚举配置项对应生成枚举定义代码，枚举定义具有公共访问权限，枚举定义代码在独立的代码文件中，在数据帧类和分系统类的实现中可以公有同一组枚举定义。

## 异常处理

## 通信方式扩展

# 系统实现

## 开发与运行环境

通信配置系统开发采用Visual Studio 2017，其中配置编辑模块采用C# WPF开发，语法检查与解析器为C++开发的动态链接库，代码生成器是C#开发的.Net类库。其中.Net Framework使用4.6.2版本，同时支持更高版本。

## 关键技术路线

实现一个通信配置系统通常会使用XML、JASON格式或自定义格式，本方案选用自定义格式实现配置代码，在语法检查和解析器的实现上相对比较其它技术路线要复杂，但却有其它技术所不具备的许多优势。

使用XML、JASON的主要优势在于：通用性强，大部分开发环境下都有可用的解析器，数据交互采用纯文本格式，适合系统间进行数据交互；其缺点也很明显：效率较低、人工输入较为复杂，不具备动态解析与计算功能。而采用自定义格式实现的配置代码，则可以针XML的不足，进行专门实现。通过分析用户业务需求，为能生成更加通用的基础通信类库，以应对各种可能出现的复杂业务情况，本方案选用自定义格式配置代码进行实现，其中配置代码解析器选用凯云公司自主研发的成熟商业软件类库（ETest软件的底层代码库）。

# 关于凯云

凯云联创（北京）科技有限公司（简称：凯云科技）核心业务是为军方、航空航天、中电、兵器、船舶、核工业、核物理、院校、交通、水利水电、通信以及金融等行业提供软件测试工具与测试服务，并承接软件开发、系统集成及信息技术服务。自主研发了超过40项拥有全部知识产权的软硬件产品,是国家级高新技术企业、北京市双软认定企业。

凯云科技自创建以来，始终致力于国产自主可控软件测试类产品的研究、开发及培训推广，为国内军用、工业、民用以及院校培训教学提供具有国际领先水平、通用性强、可靠性高、应用成本低的通用嵌入式系统测试及管理环境，同时积极引进国内外先进技术。主要产品覆盖软件测试业务全过程，其中有自主研发的嵌入式系统半实物仿真测试平台、嵌入式系统测试教学实训平台、软件测试项目管理系统、测试数据生成与管理软件等，引进产品包括各类源代码分析测试、性能及功能测试等工具。

基于长期在嵌入式系统测试领域的研究与创新，凯云科技目前已发展为国内嵌入式系统测试类产品的主要供应商，得到各行业用户的认可与支持。我们将一如既往坚持“以客户为中心，以创新产品和服务创造价值”，与用户携手，为国内各行业装备研发水平与技术创新能力提高作出卓越贡献！

本方案中使用的配置代码解析器为凯云公司自主研发的代码解析器，是ETest软件的重要组成模块。

通用嵌入式系统测试平台（Embedded System Interface Test Studio,简称：ETest）是针对嵌入式系统进行实时、闭环、非侵入式测试的自动化测试平台。该平台提供了嵌入式系统的半实物仿真测试环境，支持各类通讯接口，适用于嵌入式系统在设计、仿真、开发、调试、测试、集成验证和维护等各阶段配置项级别和系统级别的动态测试与验证。

ETest提供了针对嵌入式系统的硬件在环半实物仿真测试环境，通过模拟待测系统的外部环境并产生信号输入到待测系统，同时获取并分析待测系统的输出信号，实现针对嵌入式系统的自动化功能黑盒测试。

ETest具有适用范围广、通用性强、自动化测试程度高、扩展性好、携带方便、配置灵活、操作简单以及使用成本低等特点，能充分满足航空航天、武器装备、工业控制、汽车电子、仪器仪表等各领域嵌入式系统的测试需求。

# 实操示例

本部分内容将按照通信配置系统的方案设计，给出一个具体的实操案例，以示范通信配置系统的操作使用流程。

## 模拟场景说明

本节将通过示例代码对通信配置系统的使用步骤进行详细说明，为方便阅读，本示例代码中的配置项关键词均使用小写字母，用户自定义标识使用大写字母单词。

示例代码将构建项目PROJECT1，并配置两个受控对象SYS1、SYS2和一个数据帧MSG1；最后通过通信配置系统自动生成代码（以C#为例）。

受控对象SYS1的配置项如下：

* 属性：PROPERTY1 数据类型：long
* 属性：PROPERTY2 数据类型：bool[8]
* 属性：PROPERTY3 数据类型：double
* 通道：CH1 通道类型：com
* IO操作：使用通道CH1发送数据帧MSG1

受控对象SYS2的配置项如下：

* 属性：PROPERTYA 数据类型：long
* 属性：PROPERTYB 数据类型：bool[8]
* 属性：PROPERTYC 数据类型：double
* 通道：CHA 通道类型：com
* IO操作：使用通道CHA接收数据帧MSG1

数据帧MSG1的字段配置项如下：

* 第一个字段SEGMENTA 字段类型：64位整型
* 第二个字段SEGMENTB 字段类型：长度为8的bool型数组
* 第三个字段SEGMENTC 字段类型：双精度浮点
* 第四个字段SEGMENTD 字段类型：校验字段 校验算法：crc16\_ibm

说明：MSG1的前三个字段与SYS1、SYS2的属性一一对应。

## 配置项代码

受控对象SYS1、SYS2和数据帧MSG1的完整配置代码如下：

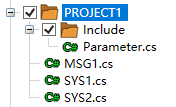
1. //项目配置
2. project @PROJECT1
3. {
4. //数据帧格式配置
5. frame @MSG1
6. {
7. //数据帧内部字段配置
8. integer @SEGMENTA signed=**true** bitcount=**64**;
9. integer @SEGMENTB bitcount=**1** repeated=**8**;
10. real @SEGMENTC isdouble=**true**;
11. integer @SEGMENTD bitcount=**16** check=**crc16\_ibm**;
12. }


16. //分系统定义
17. system @SYS1
18. {
19. //分系统属性定义
20. long @PROPERTY1;
21. bool @PROPERTY2[8];
22. double @PROPERTY3;
24. //通道配置
25. channel @CH1
26. {
27. device = **COM1**;
28. }
30. //通过通道CH1发送数据帧MSG1
31. send  @MSG1 on @CH1
32. {
33. @SEGMENTA = @PROPERTY1;
34. @SEGMENTB = @PROPERTY2;
35. @SEGMENTC = @PROPERTY3;
36. }
37. }
39. system @SYS2
40. {
41. long @PROPERTYA;
42. bool @PROPERTYB[8];
43. double @PROPERTYC;
45. //通道配置
46. channel @CHA
47. {
48. device = **COM2**;
49. }
51. // 在通道CHA上接收数据帧MSG1
52. recv @MSG1 on @CHA
53. {
54. @PROPERTYA = @SEGMENTA;
55. @PROPERTYB = @SEGMENTB;
56. @PROPERTYC = @SEGMENTC;
57. }
58. }
60. }

在通信配置系统中输入以上完整配置代码后，选择工具栏中的输出按钮，即可自动生成相应的生成代码。

## 生成代码

由以上配置代码生成的生成代码文件目录如下：



生成代码文件目录

生成代码的输出文件均位于PROJECT1目录下，其中Include目录包含一些固定的程序代码文件，如Parameter<T>的程序代码实现文件等；由配置代码生成的C#程序代码文件有SYS1.cs、SYS2.cs、MSG1.cs，分别对应受控对象SYS1、SYS2和数据帧MSG1。

以下对生成的C#程序代码文件逐一进行说明：

受控对象SYS1配置项生成的程序代码文件SYS1.cs内容如下：

1. **namespace** PROJECT1.SYS1
2. {
3. **public** **class** SYS1:INotifyPropertyChanged
4. {
6. **public** **void** Initialization()
7. {
9. CH1 = **new** CANChannel();
10. var \_cfg = **new** Dictionary<**string**, **object**>();
11. \_cfg.Add("config1", "value1");
12. \_cfg.Add("config2", "value2");
13. CH1.Open(\_cfg);
15. CH2 = **new** COMChannel();
16. \_cfg.Clear();
17. \_cfg.Add("configa", "valuea");
18. \_cfg.Add("configb", "valueb");
19. CH2.Open(\_cfg);
21. }
22. **private** Parameter<**ushort**?> \_PROPERTYA = **new** Parameter<**ushort**?>();
23. **private** ObservableCollection<Parameter<**byte**?>> \_PROPERTYB =**new** ObservableCollection<Parameter<**byte**?>>(**new** Parameter<**byte**?>[4]);
24. **private** Parameter<**double**?> \_PROPERTYC = **new** Parameter<**double**?>();
25. **public** Parameter<**ushort**?> PROPERTYA
26. {
27. **get**
28. {
29. **return** \_PROPERTYA;
30. }
31. **set**
32. {
33. **if**(\_PROPERTYA!=value)
34. {
35. \_PROPERTYA = value;
36. PropertyChanged.Invoke(**this**, **new** PropertyChangedEventArgs(nameof(**this**.PROPERTYA)));
37. }
38. }
39. }
41. **public** ObservableCollection<Parameter<**byte**?>> PROPERTYB
42. {
43. **get**
44. {
45. **return** \_PROPERTYB;
46. }
47. }
49. **public** Parameter<**double**?> PROPERTYC
50. {
51. **get**
52. {
53. **return** \_PROPERTYC;
54. }
55. **set**
56. {
57. **if** (\_PROPERTYC != value)
58. {
59. \_PROPERTYC = value;
60. PropertyChanged.Invoke(**this**, **new** PropertyChangedEventArgs(nameof(**this**.PROPERTYA)));
61. }
62. }
63. }
65. **private** CANChannel CH1;
66. **private** COMChannel CH2;
68. **public** **void** send\_FRAME1\_CH1()
69. {
70. var data = **new** FRAME1();
71. data.SEGMENTA = \_PROPERTYA.TValue ?? 0;
72. **for**(**int** i=0; i<4; i++)
73. {
74. data.SEGMENTB[i] = \_PROPERTYB[i].TValue ?? 0;
75. }
76. data.SEGMENTC = \_PROPERTYC.TValue ?? 0;
77. CH1.WriteFrame(data);
78. }
80. **public** **event** PropertyChangedEventHandler PropertyChanged;
81. }
82. }