# 指令格式说明

Codeit指令格式设计成形如“Enable -m=1 --position”类linux命令字符串。用户可以自定义解析规则、简写规则和默认值等，系统会对该字符串解析成命令与参数包，。

### 命令字符串的规则 ###

/// 所输入的字符串应该符合以下规则：

/// - 第一个空格以前应为命令名字，命令名字遵从首字母大写

/// - 第一个空格以后的为命令参数，参数可以带值，那么参数名和参数值之间以“=”分割（不可以有空格），也可以不带值，那么系统会取默认值

/// - 参数名若以“-”开始，那么该参数为简写，例如“-a”，“-m=1”

/// - 参数名若以“--”开始，那么该参数为全写，例如“--all”,“--motion\_id=1”

/// - 参数名若以字母开始，那么这是一组复合参数，每个字符都为参数的简写，例如“ap”等同于“-a -p”

///

/// 假设对于命令“Enable”，需要能让用户指定所需使能的电机以及运动模式。那么该命令有以下参数：

/// - “all” 表示所有的电机都要使能，简写为 “a”, 无需默认值

/// - “motion\_id” 表示需要使能的电机ID，简写为“m”，默认值为“0”

/// - “velocity” 表示使能成速度模式，简写为“v”，无需默认值

/// - “position” 表示使能成位置模式，简写为“p”，无需默认值

/// - “current” 表示使能成电流模式，无简写，无需默认值

///

/// 字符串“Enable --all -p” 会被解析成：

///

/// 命令：Enable

///

/// 参数：

/// - all =

/// - position =

///

/// 字符串“Enable -m=1 --position” 会被解析成：

///

/// 命令：Enable

///

/// 参数：

/// - motion = 1

/// - position =

///

/// 字符串“Enable ap” 会被解析成：

///

/// 命令：Enable

///

/// 参数：

/// - all =

/// - position =

///

/// ### 命令解析的规则 ###

///

/// 开发者可以根据需要建立自己的命令解析规则，这主要包括：

/// - 建立参数的简写规则以及默认值

/// - 指定参数之间的关系

///

/// 在上例中，参数之间应该有如下关系：

/// - 参数“all”和“motion\_id”不能共存，用户不能即指定某个电机动，又指定所有的电机都动

/// - 参数“position”、“velocity”，“current”不能共存

/// - 当用户没有指定模式时，系统自动指定模式为“position”

///

/// 那么以下字符串是合法的：

/// - “Enable --all --velocity” ，意味着所有电机都使能成速度模式

/// - “Enable -m=1 -p”,意味着电机1使能成位置模式

/// - “Enable ap”,等同于“Enable -a -p”,将所有电机使能成位置模式

/// - “Enable -a”,将所有电机使能成位置模式

///

/// 以下字符串非法：

/// - “Enable -a -m=1 --position”，因为不能即指定所有电机，又指定电机1

/// - “Enable -p”,因为不知道是使能所有电机还是某一个电机

///

/// 可知，开发者需要根据业务场景来设定参数之间的逻辑与分层关系，在codeit中，用户通过UniqueParam和GroupParam两种节点来指定参数之间的关系，

/// 再加上具体的参数节点，共有三种节点：

/// codeit::core::Param "Param" ：参数节点，无子节点，可以设定简写，可以设定默认值

/// codeit::core::UniqueParam "UniqueParam"：是父节点，各个子节点只能出现一个，无简写，如果有默认值，默认值必须为某个子节点的名字

/// codeit::core::GroupParam "GroupParam"：是父节点，所有子节点必须全部出现，无简写，无默认值

///

/// 那么“enable”命令的节点设置模式可以用以下XML文件来表示：

///

/// ~~~{.xml}

/// <Command name="Enable">

/// <GroupParam>

/// <UniqueParam default="all">

/// <Param name="all" abbreviation="a"/>

/// <Param name="motion\_id" abbreviation="m" default="0"/>

/// </UniqueParam>

/// <UniqueParam default="position">

/// <Param name="position" abbreviation="p"/>

/// <Param name="velocity" abbreviation="v"/>

/// <Param name="current"/>

/// </UniqueParam>

/// </GroupParam>

/// </Command>

/// ~~~

///

/// 以上xml节点确定了enable命令的解析规则，首先该命令包含一个名为“group”的“GroupParam”

/// 这说明了这个节点以下的所有参数（unique1和unique2）都必须有值，而unique1和unique2都是“UniqueParam”

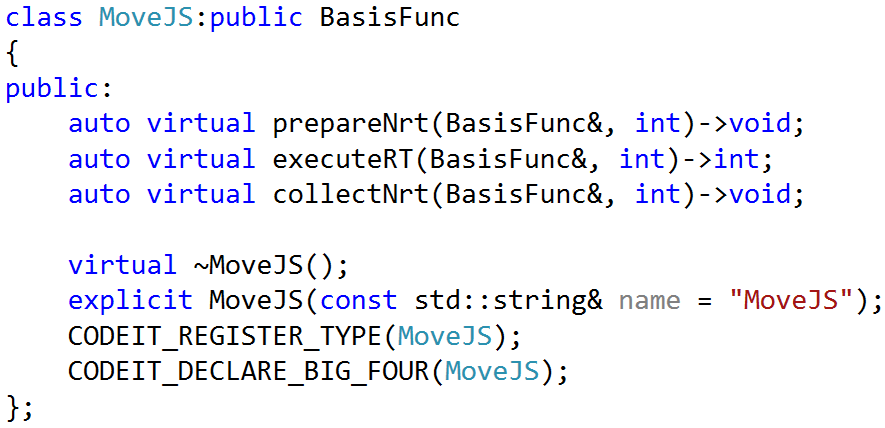
/// 这说明这两个节点下的所有子节点只能出现一个，于是all和motion\_id不能共存，position、velocity和current也不能共存。

/// unique2有名为“default”的参数，它指向position，这意味着当用户没有指定unique2中的参数时，parser会自动选择position作为

/// 默认参数。

# 新增指令

Codeit系统中增加指令即是要增加一个类，该类处于命名空间codeit::function下。该类继承于基类codeit::function::BasisFunc。类的命名遵从首字母大写规则。比如新增一个MoveJS指令，则需编写一个MoveJS类，该类包含的成员如下：



该类包含了五个函数以及两个宏，其中两个宏用于注册该指令，五个函数除了一个构造函数和一个虚构函数外，其他三个函数是该指令的主要功能函数，系统接收到该指令后，会依次执行这三个功能函数：

* prepareNrt：运行在非实时线程内，该函数只运行一次。每个指令的prepareNrt函数

一定会执行，在prepare中可以通过选项来决定是否执行 executeRT 或 collectNrt。其中 prepare 可以抛异常，在此情况下后面两个函数一定不执行。

每个指令可根据需求，在prepareNrt()中对option()设定一些选项，比如

**option() |= NOT\_RUN\_EXECUTE\_FUNCTION | NOT\_RUN\_COLLECT\_FUNCTION**

表示该指令不会执行executeRT和collectNrt函数。Option可选项有：

**enum Option : std::uint64\_t**

**{**

**NOT\_PRINT\_CMD\_INFO = 0x01ULL << 0,**

**NOT\_PRINT\_EXECUTE\_COUNT = 0x01ULL << 1,**

**NOT\_LOG\_CMD\_INFO = 0x01ULL << 2,**

**NOT\_RUN\_EXECUTE\_FUNCTION = 0x01ULL << 3,**

**NOT\_RUN\_COLLECT\_FUNCTION = 0x01ULL << 4,**

**WAIT\_FOR\_EXECUTION = 0x01ULL << 5,**

**WAIT\_IF\_CMD\_POOL\_IS\_FULL = 0x01ULL << 6,**

**WAIT\_FOR\_COLLECTION = 0x01ULL << 7,**

**};**

每个指令可根据需求，在prepareNrt()中对各个电机设定一些选项

**for (auto& option : motorOptions())**

**option |= NOT\_CHECK\_ENABLE | NOT\_CHECK\_POS\_MAX**

**| NOT\_CHECK\_POS\_MIN;**

表示该指令运行过程中，不需要检测电机是否上伺服，不需要检测位置是否超限。MotorOption提供了如下选项：

enum MotorOption : std::uint64\_t

{

USE\_TARGET\_POS = 0x01ULL << 16,

USE\_TARGET\_VEL = 0x01ULL << 17,

USE\_TARGET\_TOQ = 0x01ULL << 18,

USE\_OFFSET\_VEL = 0x01ULL << 19,

USE\_OFFSET\_TOQ = 0x01ULL << 20,

NOT\_CHECK\_MODE = 0x01ULL << 22,

NOT\_CHECK\_ENABLE = 0x01ULL << 23,

NOT\_CHECK\_POS\_MIN = 0x01ULL << 24,

NOT\_CHECK\_POS\_MAX = 0x01ULL << 25,

NOT\_CHECK\_POS\_CONTINUOUS = 0x01ULL << 26,

NOT\_CHECK\_POS\_CONTINUOUS\_SECOND\_ORDER = 0x01ULL << 28,

NOT\_CHECK\_POS\_FOLLOWING\_ERROR = 0x01ULL << 30,

NOT\_CHECK\_VEL\_MIN = 0x01ULL << 31,

NOT\_CHECK\_VEL\_MAX = 0x01ULL << 32,

NOT\_CHECK\_VEL\_CONTINUOUS = 0x01ULL << 33,

NOT\_CHECK\_COLLISION = 0x01ULL << 34,

NOT\_CHECK\_VEL\_FOLLOWING\_ERROR = 0x01ULL << 35,

CHECK\_NONE = NOT\_CHECK\_MODE | NOT\_CHECK\_ENABLE | NOT\_CHECK\_POS\_MIN | NOT\_CHECK\_POS\_MAX | NOT\_CHECK\_POS\_CONTINUOUS | NOT\_CHECK\_POS\_CONTINUOUS\_SECOND\_ORDER | NOT\_CHECK\_POS\_FOLLOWING\_ERROR |

NOT\_CHECK\_VEL\_MIN | NOT\_CHECK\_VEL\_MAX | NOT\_CHECK\_VEL\_CONTINUOUS | NOT\_CHECK\_VEL\_FOLLOWING\_ERROR,

};

* executeRT：运行在实时线程内，该函数可周期性运行，直到函数返回非正值才结

束。

* collectNrt：不要报错，这个相当于析构函数，只要 prepare 不抛异常而且未设置

NOT\_RUN\_COLLECT\_FUNCTION，就一定会执行。

# 多模型下指令格式说明

若系统中存在N个模型，比如有两只机械臂，codeit支持同时输入N条子指令，子指令数目必须与模型数目相同。比如en指令可同时对所有模型起作用。

例如，

en：使能第一个模型的电机

{en}：使能所有模型的电机

{en||en}:共有两个模型，同时使能这两个模型的电机；当模型数与指令数不同时，会抛出异常

{en --motion\_id=0||en --motion\_id=1}：使能模型1的第1个电机与模型2的第二个电机

**指令格式说明：**

指令名称遵从首字母大写规则，指令参数遵从字母小写规则，且单词间用下划线分割。当指令参数需要使用VAR定义的类型变量时，需要类型名称后加”\_var”

当需要使用配置文件里variablePool()包含的类型参数时，直接使用名称即可，且这两种方式只能取一种。例如使用speed类型，则指令的参数格式需进行如下配置：

<UniqueParam default="speed">

<Param name="speed" default="v100"/>

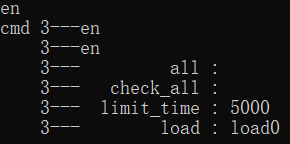
<Param name="speed\_var" default="v100"/>

</UniqueParam>

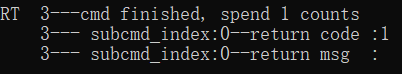
# 指令运行返回值说明

指令运行完后，运行结果会显示在控制台窗口内；如果指令是从其他Interface发过来的，则运行结果（也可包含用户想要的其他控制系统的信息）也会沿着原路径返回信息给用户。

以发送en指令为例，在控制台窗口输入 ”en”，指令输入后会先显示指令信息：



指令成功运行完成后，会出现如下信息，其中return code为1表示该指令的executeRT()运行了



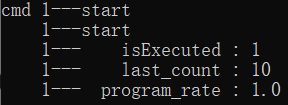
若指令运行失败（在非实时核运算出现异常，比如把指令错误输成了”enn”）



（在实时核运算中出现了错误），则会出现如下信息，其中return code会为负数，通过查看错误码可了解到具体的错误原因，return msg也会打印出错误信息，信息的语言类型取决于控制系统设定的语言。



如果该指令无需运行executeRT()，比如start指令，指令输入后会先显示指令信息：



运行成功后

