# Codeit基类Object使用说明

/// @defgroup xml\_group Object模块

///

/// \ref codeit::core::Object "Object" 类是codeit最基本的模块，用户可以从该类继承并定义自己的类型。

///

/// ### 添加节点 ###

///

/// 假设现在有如下结构：

///

/// \dot codeit使用树状数据结构

/// digraph G{

/// node[shape = record, fontname = Helvetica, fontsize = 10];

/// family[label = "family"]

/// father[label = "father"];

/// uncle[label = "uncle"];

/// tom[label = "tom"];

/// bob[label = "bob"];

/// family->father->tom[arrowhead = "open"];

/// father->bob[arrowhead = "open"];

/// family->uncle[arrowhead = "open"];

/// }

/// \enddot

///

/// 那么可以使用以下代码来构造该结构：

///

/// ~~~{.cpp}

/// Object family1("family");

/// auto &father1 = family1.add<Object>("father");

/// family1.add<Object>("uncle");

/// father1.add<Object>("tom");

/// father1.add<Object>("bob");

/// ~~~

///

/// ### 与xml交互 ###

///

/// codeit的支持与xml文件的交互，以下可以将上述结构的xml字符串打印出来：

///

/// ~~~{.cpp}

/// std::cout << family1.xmlString() << std::endl;

/// ~~~

///

/// 屏幕输出：

///

/// ~~~

/// <family1 type="Object">

/// <father type="Object">

/// <tom type="Object"/>

/// <bob type="Object"/>

/// </father>

/// <uncle type="Object"/>

/// </family1>

/// ~~~

///

/// Object也可以通过xml文件来构造，如下：

///

/// ~~~{.cpp}

/// Object family2;

/// family2.loadXmlStr(

/// "<family2 type=\"Object\">"

/// " <father type=\"Object\">"

/// " <tom type=\"Object\"/>"

/// " <bob type=\"Object\"/>"

/// " </father>"

/// " <uncle type=\"Object\"/>"

/// "</family2>");

/// ~~~

///

/// family2拥有和family完全相同的结构

///

/// ### 使用自定义类型 ###

///

/// 如果需要保存其他信息，那么需要让用户自己定义类型，假设father和uncle对应Man类型，有age和job两个属性，tom和bob是

/// Child类型，只有age一个属性，那么新的数据结构如下：

///

/// \dot 自定义节点类

/// digraph G{

/// node[shape = record, fontname = Helvetica, fontsize = 10];

/// family[label = "class Family \n name = \"family\""]

/// father[label = "class Man \n name = \"father\" \n age = \"35\" job = \"teacher\""];

/// uncle[label = "class Man \n name = \"uncle\" \n age = \"33\" job = \"policeman\""];

/// tom[label = "class Boy \n name = \"tom\" \n age = \"8\""];

/// bob[label = "class Boy \n name = \"bob\" \n age = \"6\""];

/// family->father->tom[arrowhead = "open"];

/// father->bob[arrowhead = "open"];

/// family->uncle[arrowhead = "open"];

/// }

/// \enddot

///

/// 以下是Man类的代码，为了和xml文件交互，用户务必要写静态函数Type，重载虚函数type, saveXml, loadXml，以及默认构造函数：

///

/// ~~~{.cpp}

/// class Man :public Object

/// {

/// public:

/// static auto Type()->const std::string &{ static const std::string type{ "Man" }; return type; }

/// auto virtual type() const->const std::string& override{ return Type(); }

/// auto virtual saveXml(aris::core::XmlElement &xml\_ele) const->void override

/// {

/// Object::saveXml(xml\_ele);

/// xml\_ele.SetAttribute("age", age\_);

/// xml\_ele.SetAttribute("job", job\_.c\_str());

/// }

/// auto virtual loadXml(const aris::core::XmlElement &xml\_ele)->void override

/// {

/// Object::loadXml(xml\_ele);

/// age\_ = attributeInt32(xml\_ele, "age");

/// job\_ = attributeString(xml\_ele, "job");

/// }

///

/// Man(const std::string &name = "man", int age = 0, const std::string job = "teacher") :Object(name), age\_(age), job\_(job)

/// {

/// registerType<Child>();

/// };

/// private:

/// int age\_;

/// std::string job\_;

/// };

/// ~~~

///

/// 这五个函数的作用为：

/// + Type() : 该函数返回在xml中类型名的字符串，这个字符串不需要和c++代码中的类名保持一致（不过建议保持一致）

/// + type() : 该函数重载基类的函数，返回Type()就可以，用于从指针获取类名

/// + saveXml(aris::core::XmlElement &xml\_ele) ： 该函数定义了c++对象到xml的转换,在实现中应该首先调用基类同名函数

/// + loadXml(const aris::core::XmlElement &xml\_ele) ： 该类型定义了xml到c++对象的转换,在实现中应该首先调用基类同名函数

/// + Man(const std::string &name = "man", int age = 0, const std::string job = "teacher") ： 该函数为默认构造函数，用户务必保证形如Man()的函数可以调用，

/// 一般建议用户在该函数中注册对象可能用到的所有子类型。

///

/// 类似Man类，以下代码定义了Child类和Family类：

///

/// ~~~{.cpp}

/// class Child :public Object

/// {

/// public:

/// static auto Type()->const std::string &{ static const std::string type{ "Child" }; return type; }

/// auto virtual type() const->const std::string& override{ return Type(); }

/// auto virtual saveXml(aris::core::XmlElement &xml\_ele) const->void override

/// {

/// Object::saveXml(xml\_ele);

/// xml\_ele.SetAttribute("age", age\_);

/// }

/// auto virtual loadXml(const aris::core::XmlElement &xml\_ele)->void override

/// {

/// Object::loadXml(xml\_ele);

/// age\_ = attributeInt32(xml\_ele, "age");

/// }

///

/// Child(const std::string &name = "child", int age = 0) :Object(name), age\_(age) {};

///

/// private:

/// int age\_;

/// };

/// class Family :public Object

/// {

/// public:

/// static auto Type()->const std::string &{ static const std::string type{ "Family" }; return type; }

/// auto virtual type() const->const std::string& override{ return Type(); }

///

/// Family(const std::string &name = "family") :Object(name)

/// {

/// registerType<Man>();

/// };

/// };

/// ~~~

///

/// 定义完类型后，就可以由代码来构造上述框图中的数据结构：

/// ~~~{.cpp}

/// Family family3("family3");

/// auto &father3 = family3.add<Man>("father", 35, "teacher");

/// family3.add<Man>("uncle", 33, "policeman");

/// father3.add<Child>("tom", 8);

/// father3.add<Child>("bob", 6);

///

/// std::cout << family3.xmlString() << std::endl;

/// ~~~

///

/// 以上代码输出：

/// ~~~

/// <family3 type="Family">

/// <father type="Man" age="35" job="teacher">

/// <tom type="Child" age="8"/>

/// <bob type="Child" age="6"/>

/// </father>

/// <uncle type="Man" age="33" job="policeman"/>

/// </family3>

/// ~~~

///

/// ### 注册新类型 ###

/// 在aris中，如果要使用xml转换的功能，需要提前注册。上文中类型注册发生在构造函数里，这样每个对象都知道自己在xml文件里可能使用到的类型的信息。

/// 现在假设Family，Man，Child三个类型位于基础库中，而用户想要继续扩展一个Boy类：

///

/// ~~~{.cpp}

/// class Boy :public Child

/// {

/// public:

/// static auto Type()->const std::string &{ static const std::string type{ "Boy" }; return type; }

/// auto virtual type() const->const std::string& override{ return Type(); }

///

/// Boy(const std::string &name = "boy", int age = 0) :Child(name, age) {};

/// };

/// ~~~

///

/// 此时因为Family类型的构造函数内并未注册过Boy类，因此需要手动注册。手动注册有两种：

/// + 全局注册：注册后所有对象都可以在xml中使用

/// + 局部注册：针对某个对象来注册其可能使用的子类

///

/// 局部注册优先于全局注册，以下为全局注册，注册并不针对某个具体的对象，注册后添加了一个名为bill的Boy类型节点：

///

/// ~~~{.cpp}

/// Family family5;

/// aris::core::Object::registerTypeGlobal<Boy>();

/// family5.loadXmlStr(

/// "<family5 type=\"Family\">"

/// " <father type=\"Man\" age=\"35\" job=\"teacher\">"

/// " <tom type=\"Child\" age=\"8\"/>"

/// " <bob type=\"Child\" age=\"6\"/>"

/// " <bill type=\"Boy\" age=\"3\"/>"

/// " </father>"

/// " <uncle type=\"Man\" age=\"33\" job=\"policeman\"/>"

/// "</family5>");

/// std::cout << family5.xmlString() << std::endl;

/// ~~~

///

/// 以下为局部注册，注册只对family6对象有效：

///

/// ~~~{.cpp}

/// Family family6;

/// family6.registerType<Boy>();

/// family6.loadXmlStr(

/// "<family6 type=\"Family\">"

/// " <father type=\"Man\" age=\"35\" job=\"teacher\">"

/// " <tom type=\"Child\" age=\"8\"/>"

/// " <bob type=\"Child\" age=\"6\"/>"

/// " <bill type=\"Boy\" age=\"3\"/>"

/// " </father>"

/// " <uncle type=\"Man\" age=\"33\" job=\"policeman\"/>"

/// "</family6>");

/// std::cout << family6.xmlString() << std::endl;

/// ~~~

///

/// ### BIG 5的行为 ###

///

/// C++中的big 5是指默认构造函数，拷贝构造函数，移动构造函数，拷贝赋值函数和移动赋值函数。

/// 这5个函数的使用地方为：

///

/// 父节点调用 | Default ctor | Copy ctor | Move ctor | Copy assignment | Move assignment |

/// --------: | : -------- : | : ----- : | : ----- : | : ----------- : | : ----------- : |

/// 子节点行为 | Default ctor | Copy ctor | 无，左侧直接接管右侧子节点 | 如果左右侧类型一致，则拷贝赋值；如果类型不一致，删除左侧，并拷贝构造； | 如果左侧和右侧类型一致，则移动赋值；如果类型不一致，删除左侧，并移动构造； |

/// 子节点地址 | 创建 | 创建 | 左侧使用右侧 | 如果左右侧类型一致，则左侧地址不变；如果类型不一致，左侧地址改变； | 如果左右侧类型一致，则左侧地址不变；如果类型不一致，左侧地址改变； |

///

///

///

///

///

///

/// @{

///

**1、en (--all/--motion\_id=i)**

(1)en:使能指令，enable的缩写

(2)--all/--motion\_id=i：两个互斥的可选参数（互斥参数用"/"隔开，可选参数用"()"包含，后续指令类似），可以选定1个或者全部电机，--all为默认参数，即指令en

和指令en --all等效；--motion\_id=i表示选择第i+1个电机

(3)指令和参数之间、参数与参数之间，都用一个空格隔开

例如，

en：使能全部电机

en --motion\_id=0：使能第1个电机