RAPIDXML Manual

RAPIDXML 中文手册

版本： 1.13

官网： http://rapidxml.sourceforge.net/

作者： Marcin Kalicinski

译者： jhkdiy（申志远）

邮箱： jhkdiy@qq.com

日期： 2012-11-26 开始

2012-12-29 结束

博客： http://blog.csdn.net/jhkdiy

http://www.cnblogs.com/jhkdiy/

http://jhkdiy.blog.51cto.com

简介：

这个号称是最快的 DOM 模型 XML 分析器，在使用它之前我都是用 TinyXML 的，因为它小巧和容易上手，但真正在项目中使用时才发现如果分析一个比较大的 XML 时 TinyXML 还是表现一般，所以我们决定使用 RapidXML 作为替换。当然是为了获取更好的性能，经过我们的初步试验后发现确实比 TinyXML 好，但看到网上关于 rapidxml 的资料零散，而且也缺乏一份较为权威的说明文档，找来找去还是得看官方的英文手册。所以我又下单了，翻译

官方提供的手册，希望给各位朋友提供一些绵薄之力。

我的其他作品：

- 商业开发实战之 VB 篇精彩视频
- VC 专题研究视频全集(VSFlexGrid 应用开发、深入解释 MFC 原理、武装你的开发环境)
- 高清《Win32Asm 与 RadAsm 开发》系列教程打包下载，更新第四季。
- 《Win32Asm 与 Radasm 开发教程》第四部-精彩实例分析 !! 2012-02-28 更新完毕 !!
- 《VC++专题研究》系列视频之-深入解释 MFC 原理
- 《Win32asm 与 Radasm 开发教程》第二阶段的代码、流程图、优秀书籍下载

- Win32汇编开发环境介绍和 RadAsm 简明教程第二版
- 汇编程序员之代码风格指南

目录

RAPIDXML 中文手册 1

简介： 1

我的其他作品： 2

目录 3

1. RapidXml 是什么? 8

1.1 Dependencies And Compatibility（依赖和兼容） 8

1.2 Character Types And Encodings（字符类型和编码）8

1.3 Error Handling （错误处理） 9

1.4 Memory Allocation（内存分配） 9

1.5 W3C Compliance（W3C 兼容） 9

1.6 API Design（API 设计） 9

1.7 Reliability（可靠性） 10

1.8 Acknowledgements（致谢） 10

2. Two Minute Tutorial（两分钟教程） 12

2.1 Parsing(分析) 12

2.2 Accessing The DOM Tree（访问 DOM 树） 12

2.3 Modifying The DOM Tree（修改 DOM 树） 13

2.4 Printing XML（打印 XML） 14

3. Differences From Regular XML Parsers(与常规 XML 分析器的不同之处) 15

3.1 Lifetime Of Source Text（源文本的生命周期）15

3.2 Ownership Of Strings（字符串的所有权）..... 15

3.3 Destructive Vs Non-Destructive Mode（破坏模式和非破坏模式）..... 15

4. Performance（性能）..... 16

4.1 Comparison With Other Parsers（与其它分析器的比较）..... 16

5. Reference（参考）..... 18

类模板 rapidxml::memory_pool..... 23

构造函数 memory_pool::memory_pool..... 24

析构函数 memory_pool::~memory_pool..... 25

函数 memory_pool::allocate_node..... 25

函数 memory_pool::allocate_attribute..... 26

函数 memory_pool::allocate_string..... 27

函数 memory_pool::clone_node..... 28

函数 memory_pool::clear..... 29

函数 memory_pool::set_allocator..... 29

类 rapidxml::parse_error..... 30

构造函数 parse_error::parse_error..... 31

函数 parse_error::what..... 31

函数 parse_error::where..... 32

类模板 rapidxml::xml_attribute..... 32

构造函数 xml_attribute::xml_attribute..... 33

函数 xml_attribute::document..... 33

函数 xml_attribute::previous_attribute..... 34

函数 xml_attribute::next_attribute 35
类模板 rapidxml::xml_base 36
构造函数 xml_base::xml_base 36
函数 xml_base::name 36
函数 xml_base::name_size 37
函数 xml_base::value 37
函数 xml_base::value_size 38
函数 xml_base::name 38
函数 xml_base::name 39
函数 xml_base::value 40
函数 xml_base::value 41
函数 xml_base::parent 42
类模板 rapidxml::xml_document 42
构造函数 xml_document::xml_document 43
函数 xml_document::parse 43
函数 xml_document::clear 44
类模板 rapidxml::xml_node 44
构造函数 xml_node::xml_node 45
函数 xml_node::type 45
函数 xml_node::document 46
函数 xml_node::first_node 46
函数 xml_node::last_node 47

函数 xml_node::previous_sibling 48
函数 xml_node::next_sibling 49
函数 xml_node::first_attribute 50
函数 xml_node::last_attribute 51
函数 xml_node::type 52
函数 xml_node::prepend_node 53
函数 xml_node::append_node 53
函数 xml_node::insert_node 54
函数 xml_node::remove_first_node 54
函数 xml_node::remove_last_node 55
函数 xml_node::remove_node 55
函数 xml_node::remove_all_nodes 56
函数 xml_node::prepend_attribute 56
函数 xml_node::append_attribute 56
函数 xml_node::insert_attribute 57
函数 xml_node::remove_first_attribute 58
函数 xml_node::remove_last_attribute 58
函数 xml_node::remove_attribute 58
函数 xml_node::remove_all_attributes 59
枚举类型 node_type 59
函数 parse_error_handler 60
函数 print 61

函数 print 62
函数 operator<< 63
常数 parse_no_data_nodes 64
常数 parse_no_element_values 64
常数 parse_no_string_terminators 65
常数 parse_no_entity_translation 65
常数 parse_no_utf8 66
常数 parse_declaration_node 66
常数 parse_comment_nodes 67
常数 parse_doctype_node 67
常数 parse_pi_nodes 68
常数 parse_validate_closing_tags 68
常数 parse_trim_whitespace 69
常数 parse_normalize_whitespace 69
常数 parse_default 70
常数 parse_non_destructive 70
常数 parse_fastest 71
常数 parse_full 71
常数 print_no_indenting 72

1. RapidXml 是什么?

RapidXML 是一个试图创建最快的 XML DOM 分析器，当然同时也保留它的可用性、移植性和适当的 W3C 兼容性。它是一个用 C++编写的即用分析器，在相同的数据上它的分析速度接近于 strlen()函数。

整个分析器就包含在单独的一个头文件中，所以不需要任何的构建和链接。当你使用的时候只需将 rapidxml.hpp 文件复制到一个合适的地方（例如你的工程目录下），然后在任何需要它的地方包含一下即可。或许你也会用到 rapidxml_print.hpp 头文件里的打印函数。

1.1 Dependencies And Compatibility（依赖和兼容）

RadpidXML除了使用到标准C++库 (<cassert>, <cstdlib>, <new> 和 <exception>, 除非异常被关闭了)的很小一个子集外没有关联其它东西了。它能被任何标准编译器所编译，并已在 Visual C++ 2003, Visual C++ 2005, Visual C++ 2008, gcc 3, gcc 4, 和 Comeau 4.3.3. 中测试通过。也就是说在这些编译器上没出现任何的警告信息，即使在最高警告级别上。

1.2 Character Types And Encodings（字符类型和编码）

RadpidXML 并不知道字符集的类型（译注：因为它是基于模板，可以自定义字符类型），所以在窄字符和宽字符下都能工作。当前版本并不完全支持 UTF-16和 UTF-32，所以使用宽字符稍微有点不适合。不管怎样，如果数据的字节顺序与当前系统匹配，它能成功分析包含 UTF-16或 UTF-32 的 wchar_t 字符串。UTF-8是完全支持的，包括所有的数值字符引用，这些字符会被扩展到适当的 UTF-8字节顺序（除非你打开 parse_no_utf8 flag 标记）。

要注意 RapidXml 不会对 name() 和 value() 函数返回的字符串进行解码，因为函数会

根据源代码文件本身的编码模式来编码（译注：也就是说如果你的源代码文件编码格式为ANSI，则 name()函数返回的字符串编码也是 ANSI，如果源代码文件的编码格式为 UTF-8，字符串编码也是 UTF-8）。Rapidxml 能解释和扩展下列的字符应用 :' & " < > &#...；其它字符引用不会被展开。

1.3 Error Handling （错误处理）

默认情况下，RapidXml 使用 C++异常来报告错误，如果你不喜欢这个行为，可以使用 RAPIDXML_NO_EXCEPTIONS 定义来制止异常代码，请参考 parse_error 类和 parse_error_handler() 函数。

1.4 Memory Allocation（内存分配）

RadpidXml 使用一个特别的内存池对象来分配所有的节点和属性，因为直接使用 new 操作符来分配太慢了，池中的内存分配可用 memory_pool::set_allocator() 函数来自定义，详细信息请看 memory_pool 类。

1.5 W3C Compliance（W3C 兼容）

RapidXml 并不是一个与 W3C 完全一致的分析器，主要原因是忽略了 DOCTYPE 声明。还有一些其它原因，如兼容性不是很好，但是，它能成功分析和生成与 W3C 文件组（有关 XML 处理器的规格设计超过1000个文件）定义的完整 XML 树。即使在不良的情况下它也能对空白符进行正常化处理和对一部分字符集实体进行替换。

1.6 API Design（API 设计）

为了尽可能的减少代码量，RadpidXML API 已经很简约了，并且为了能在它嵌入的环境中更容易使用，一些附加的功能被分割在不同的头文件中：rapidxml_utils.hpp 和 rapidxml_print.hpp。这两个头文件并不是必须的，并且当前也没有相关文档（但是代码中有注释）。

1.7 Reliability（可靠性）

RadpidXML 很健壮，因为它通过了一大堆的单元测试。已经特别小心地确保分析器的稳定性，不管你扔什么文本给它。其中的一项单元测试用了随机产生100,000个 XML 文档错误，RapidXml 也能创建它（还未更正）。当这些错误被引入后 RapidXml 还是能通过这项测试，并且不会被破坏和进入死循环。

其它的单元测试也和 RapidXMl 进行肉搏，稳定的 XML 分析器，和其它为了符合一大推文档和功能的测试。

另外 RapidXml 进行了1000多个 W3C 兼容性文件测试，并且都能取得正确的结果。其它的测试包括对 API 函数的测试，并且测试各种各样的分析模式。

1.8 Acknowledgements（致谢）

I would like to thank Arseny Kapoulkine for his work on pugixml, which was an inspiration for this project. Additional thanks go to Kristen Wegner for creating pugxml, from which pugixml was derived. Janusz Wohlfeil kindly ran RapidXml speed tests on hardware that I did not have access to, allowing me to expand performance

comparison table.

2. Two Minute Tutorial（两分钟教程）

2.1 Parsing(分析)

下面的代码示范了 RapidXml 分析一个以 0 结尾的字符串 text：

```
using namespace rapidxml;
xml_document<> doc;      // 字符类型默认为 char
doc.parse<0>(text);       // 0表示默认分析标记
```

doc 对象现在是一个 DOM 树，代表已分析的 XML。因为所有的 RadpidXml 接口都包含在命名空间 rapidxml 中，用户要么加入此命令空间，要么使用完整的命名（译注：即要使用 doc 必须使用 rapidxml::xml_docment<> doc），类 xml_document 代表一个 DOM 层次的根节点。除了是公共的继承点，它还是一个 xml_node 和 memory_pool。xml_document::parse()函数的模板参数是用来指定分析标记的，它可以用来对分析器进行微调。但要注意，标记必须是一个编译器的常数。

2.2 Accessing The DOM Tree（访问 DOM 树）

要访问 DOM 树，使用 xml_node 和 xml_attribute 类中的方法即可：

```
cout << "Name of my first node is: " << doc.first_node()->name() << "\n";
xml_node<> *node = doc.first_node("foobar");
cout << "Node foobar has value " << node->value() << "\n";
for (xml_attribute<> *attr = node->first_attribute();
     attr; attr = attr->next_attribute())
{
    cout << "Node foobar has attribute " << attr->name() << " ";
```

```
    cout << "with value " << attr->value() << "\n";
}
```

2.3 Modifying The DOM Tree（修改 DOM 树）

分析器生成的 DOM 树完全可以修改。可以增加和删除节点和属性，还有他们的内容。下面的例子创建了一个 HTML 文档，并有个单独的链接到 google.com 网站：

```
xml_document<> doc;
xml_node<> *node = doc.allocate_node(node_element, "a", "Google");
doc.append_node(node);
xml_attribute<> *attr = doc.allocate_attribute("href", "google.com");
node->append_attribute(attr);
```

其中一个特点是节点和属性都不拥有它们的名称和值。这是因为通常它们都只保存指向源文本的指针。所以，当关联一个新的名称或值给节点时，要特别小心字符串的生命周期。最简单的方法是通过 xml_document 内存池来分配字符串。在上面的例子中其实是不需要的，因为我们只关联了字符常量。但是下面的例子就使用 memory_pool::allocate_string()函数来分配节点名称（拥有与文档一样的生命周期），并将它关联到一个新的节点：

```
xml_document<> doc;
char *node_name = doc.allocate_string(name);            // 分配字符串并将名称复制进去
xml_node<> *node = doc.allocate_node(node_element, node_name);  // 为 node_name 设置节点名称
```

查看 参考 小节可看到这个接口的描述。

2.4 Printing XML（打印 XML）

你可以将 xml_document 和 xml_node 对象输出为一个 XML 字符串。使用 print()函数或操作符 operator<<，在 rapidxml_print.hpp 头文件中可找到这些定义。

```
using namespace rapidxml;
xml_document<> doc;      // 字符类型默认为 char
// ... 一些填充文档的代码
// 使用标准字符流 operator <<
std::cout << doc;

// 用 print 函数来输出流，指定了打印参数
print(std::cout, doc, 0);    // 0 表示默认

// 通过输出迭代器来打印字符串
std::string s;
print(std::back_inserter(s), doc, 0);

// 通过输出迭代器来输出到内存缓存里
char buffer[4096];                          // 你必须保证缓存足够大!
char *end = print(buffer, doc, 0);          // end 包含了指向最后一个字符的指针
*end = 0;                                   // 在 XML 后面添加字符终止符
```

3. Differences From Regular XML Parsers(与常规XML分析器的不同之处)

RapidXml 是一个原位分析器，能提供非常快的分析速度。原位的意思是分析器并不做字符串的拷贝，而是在 DOM 层次中放置指向源文本的指针。

3.1 Lifetime Of Source Text（源文本的生命周期）

原位分析需要源文本的生命周期与 XML 文档对象一样长。如果源文本销毁了，DOM 树里的节点名称和值也会跟着销毁，而且，空白符处理，字符集转换，和0结尾的字符串都需要在分析期间被修改（但不是破坏性的）。这就使得 Rapidxml 进行下一步分析时字符串不再有效了。

但是在大多数情况下，这些都不是严重的问题。

3.2 Ownership Of Strings（字符串的所有权）

Rapidxml 生成的节点和属性都不拥有它们自己的名称和值字符串。它们只是保留了指向名称和值的指针。也就是说，当你用 xml_base::name(const Ch *) 或 xml_base::value(const Ch *) 函数时必须非常小心的设置这些值。要小心保证传递进来的字符串拥有与节点和属性一样长的生命周期。也可以简单地通过使用文档的内存池来分配这些字符串。使用 memory_pool::allocate_string()函数可达到此目的。

3.3 Destructive Vs Non-Destructive Mode（破坏模式和非破坏模式）

默认情况下，当分析器进行分析处理时会修改源文本。这时因为需要做字符集的转换、空白符常规化、字符串 0 终止。

可能在某些情况下这个行为不好，例如源文本在一个只读的内存中，或者是从文件中直接映射过来的，通过使用适当的分析标记（parse_non_destructive），源文本修改模式可以被关闭。但是，因为 Rapidxml 做了原位分析，该标记明显有下列的副作用：

- 不作空白符的正常化处理
- 不作实体引用转换
- 名称和值都是非0终止的，你必须使用 xml_base::name_size() 和 xml_base::value_size()函数来告诉他们结束。

4. Performance（性能）

RadpidXML 通过下列几项技术来实现它的速度：

- 原位分析，当创建 DOM 树时，RadpidXML 并不拷贝节像节点名称和值之类的字符串数据，而是存储了指向源文本的指针。
- 使用模板元编程技术。这使得它将很多工作都移到了编译期，C++编译器为任何您使用的分析标记生成不同的分析代码拷贝。在每一个拷贝中，所有可能的判断都会在编译期间确定，并省略不使用的代码。
- 分析期间广泛使用了查找表。
- 在流行的 CPU 上做了 C++手工优化。

这就产生一个非常小巧和快速的代码：可以根据需要裁剪和扩展的分析器。

4.1 Comparison With Other Parsers（与其它分析器的比较）

下面的表格比较了 RadpidXML 和其它分析器的速度，和 strlen()函数在这些数据上的执行速度。在一个现代 CPU 上（2007以后），你可能期望分析速度能达到每秒1GB。作为一个经验，分析速度比 Xerces DOM 快50-100倍，比 TinyXML 快30-60倍，比 pugxml 快3-12倍，比 pugixml 快5%-30%，这是我知道的最快 XML 分析器。测试文件是一个真实的，50kB，密集适当的 XML 文件。

- 所有的计时都是用兼容奔腾的 CPU 指令 RDTSC 来完成的。
- 没有使用配置优化。
- 所有的分析器都在它们最快模式下运行。
- 结果都是用每个字符的 CPU 周期来算，所以与 CPU 的频率不相干。
- 结果是大量运行后的最小值，这是为了将操作系统的活动、任务切换、中断处理等的影响降到最低。
- 测试文件的一项单独测试只用了大约10分之一毫秒，所以大量运行可以避免最后一个测试被中断的缺陷，并可以获得比较纯净的结果。

Platform	Compiler	strlen()	RapidXml	pugixml 0.3	pugxml	TinyXml
Pentium 4	MSVC 8.0	2.5	5.4	7.0	61.7	298.8
Pentium 4	gcc 4.1.1	0.8	6.1	9.5	67.0	413.2
Core 2	MSVC 8.0	1.0	4.5	5.0	24.6	154.8
Core 2	gcc 4.1.1	0.6	4.6	5.4	28.3	229.3
Athlon XP	MSVC 8.0	3.1	7.7	8.0	25.5	182.6
Athlon XP	gcc 4.1.1	0.9	8.2	9.2	33.7	265.2

Pentium 3	MSVC 8.0	2.0	6.3	7.0	30.9	211.9
Pentium 3	gcc 4.1.1	1.0	6.7	8.9	35.3	316.0

(*) 结果都是用原文本每个字符的 CPU 周期来算。

5. Reference (参考)

本节列出了所有的类、函数、常数等等，并在每个具体类中描述。

template rapidxml::memory_pool

constructor memory_pool()

destructor ~memory_pool()

function allocate_node(node_type type, const Ch *name=0, const Ch *value=0, std::size_t name_size=0, std::size_t value_size=0)

function allocate_attribute(const Ch *name=0, const Ch *value=0, std::size_t name_size=0, std::size_t value_size=0)

function allocate_string(const Ch *source=0, std::size_t size=0)

function clone_node(const xml_node< Ch > *source, xml_node< Ch > *result=0)

function clear()

function set_allocator(alloc_func *af, free_func *ff)

class rapidxml::parse_error

constructor parse_error(const char *what, void *where)

function what() const

function where() const

class template rapidxml::xml_attribute

constructor xml_attribute()

function document() const

function previous_attribute(const Ch *name=0, std::size_t name_size=0, bool case_sensitive=true) const

function next_attribute(const Ch *name=0, std::size_t name_size=0, bool case_sensitive=true) const

class template rapidxml::xml_base

constructor xml_base()

function name() const

function name_size() const

function value() const

function value_size() const

function name(const Ch *name, std::size_t size)

function name(const Ch *name)

function value(const Ch *value, std::size_t size)

function value(const Ch *value)

function parent() const

class template rapidxml::xml_document

constructor xml_document()

function parse(Ch *text)

function clear()

class template rapidxml::xml_node

constructor xml_node(node_type type)

function type() const

function document() const

function first_node(const Ch *name=0, std::size_t name_size=0, bool case_sensitive=true) const

function last_node(const Ch *name=0, std::size_t name_size=0, bool case_sensitive=true) const

function previous_sibling(const Ch *name=0, std::size_t name_size=0, bool case_sensitive=true) const

function next_sibling(const Ch *name=0, std::size_t name_size=0, bool case_sensitive=true) const

function first_attribute(const Ch *name=0, std::size_t name_size=0, bool case_sensitive=true) const

function last_attribute(const Ch *name=0, std::size_t name_size=0, bool case_sensitive=true) const

function type(node_type type)

function prepend_node(xml_node< Ch > *child)

function append_node(xml_node< Ch > *child)

function insert_node(xml_node< Ch > *where, xml_node< Ch > *child)

function remove_first_node()

function remove_last_node()

function remove_node(xml_node< Ch > *where)

function remove_all_nodes()

function prepend_attribute(xml_attribute< Ch > *attribute)

function append_attribute(xml_attribute< Ch > *attribute)

function insert_attribute(xml_attribute< Ch > *where, xml_attribute< Ch > *attribute)

function remove_first_attribute()

function remove_last_attribute()

function remove_attribute(xml_attribute< Ch > *where)

function remove_all_attributes()

namespace rapidxml

enum node_type

function parse_error_handler(const char *what, void *where)

function print(OutIt out, const xml_node< Ch > &node, int flags=0)

function print(std::basic_ostream< Ch > &out, const xml_node< Ch > &node, int flags=0)

function operator<<(std::basic_ostream< Ch > &out, const xml_node< Ch > &node)

constant parse_no_data_nodes

constant parse_no_element_values

constant parse_no_string_terminators

constant parse_no_entity_translation

constant parse_no_utf8

constant parse_declaration_node

constant parse_comment_nodes

constant parse_doctype_node

constant parse_pi_nodes

constant parse_validate_closing_tags

constant parse_trim_whitespace

constant parse_normalize_whitespace

constant parse_default

constant parse_non_destructive

constant parse_fastest

constant parse_full

constant print_no_indenting

类模板 rapidxml::memory_pool

定义在 rapidxml.hpp 中

是 xml_document 的基类

描述：

此类被分析器用来创建新的节点和属性，无需额外的动态内存分配。更多情况下，你根本不需要直接使用此类。但是，如果你想手工创建节点或修改节点的名称或值，则鼓励你使用 xml_document 中的 memory_pool 来分配内存。这样不仅仅比使用 new 操作符要快，而且他们的生命期也与文档一样长，内存管理也可能更简单。

调用 allocate_node() 或 allocate_attribute() 可从内存池中获取新节点或属性。你也可以调用 allocate_string() 来分配字符串。这些字符串能用在节点的名称和值中，而不必担心他们的生命期。注意这里没有 free() 函数 --所有的分配都会在 clear() 函数调用或内存池被销毁后释放。

也可以创建一个独立的 memory_pool，并用它来分配节点，但是它们的生命期不会跟任何文档相关了。

内存池管理 RAPIDXML_STATIC_POOL_SIZE 个静态分配的内存字节。除非静态内存用完，否则不会进行动态内存分配。当静态内存确实用完时，内存池会通过全局 new[] h 和 delete[] 来分配 RAPIDXML_DYNAMIC_POOL_SIZE 大小的内存块。此行为可通过分配函数来自定义。使用 set_allocator() 函数来设置它们。

对于节点的分配，属性和字符串用 RAPIDXML_ALIGNMENT 来对齐字节，该值是目标

架构的默认指针大小（译注：此架构是指 x86架构或其它 CPU 架构，x86的32位系统中指针为4个字节）。

如果要从分析器中获取最好的性能，最重要的是只从一个单一的、连续的内存块中分配所有节点，因此，当在两个独立的内存块中跳转时会发生缓存未命中，这样可能会降低分析速度。如果需要，你可以用 RAPIDXML_STATIC_POOL_SIZE、RAPIDXML_DYNAMIC_POOL_SIZE 和 RAPIDXML_ALIGNMENT 来在内存浪费和性能之间取得平衡。只需在 rapidxml.hpp 头文件之前定义这些值就可以完成。

参数：

Ch

用于创建节点的字符类型（译注：这里是指整个待创建或分析的字符串究竟是普通的 char 类型还是其它，例如还可以是宽字符的 wchar_t）。

构造函数 memory_pool::memory_pool

原型

```
memory_pool();
```

描述

用默认分配函数构造一个空的内存池。

析构函数 memory_pool::~memory_pool

原型

~memory_pool();

描述

销毁内存池并释放所有的内存。这会导致池中被节点使用的内存会被释放掉，从池中分配的所有节点也不再有效。

函数 memory_pool::allocate_node

原型

xml_node<Ch>* allocate_node(node_type type, const Ch *name=0, const Ch *value=0, std::size_t name_size=0, std::size_t value_size=0);

描述

从池中分配一个新节点，可传递名称和值给它。如果分配不能被满足，此函数会抛出 std::bad_alloc 。如果用 RAPIDXML_NO_EXCEPTIONS 定义屏蔽了异常，则函数会调用 rapidxml::parse_error_handler() 函数。

参数

type

被创建节点的类型

name

节点名称，0表示不传递。

value

节点的值，0表示不传递。

name_size

节点名称的大小，0表示从名称字符串中自动计算长度。

value_size

节点值的大小，0表示从名称字符串中自动计算长度。

返回值

指向已分配节点的指针，该指针永远不会为 NULL。

函数 memory_pool::allocate_attribute

原型

xml_attribute<Ch>* allocate_attribute(const Ch *name=0, const Ch *value=0, std::size_t name_size=0, std::size_t value_size=0);

描述

从池中分配新的属性，可传递名称和值给它，如果分配不能被满足，此函数会抛出 std::bad_alloc 。如果用 RAPIDXML_NO_EXCEPTIONS 定义屏蔽了异常 ,则函数会调用 rapidxml::parse_error_handler() 函数。

参数

name

属性的名称，0表示不传递名称。

value

属性的值，0表示不传递值。

name_size

属性名的大小，0表示从名称字符串中自动计算长度。

value_size

属性值的大小，0表示从名称字符串中自动计算长度。

返回值

指向已分配属性的指针，该指针永远不会为 NULL。

函数 memory_pool::allocate_string

原型

Ch* allocate_string(const Ch *source=0, std::size_t size=0);

描述

从池中分配给定大小的字符数组，也可复制一份字符串给它。如果分配不能被满足，此函数会抛出 std::bad_alloc 。如果用 RAPIDXML_NO_EXCEPTIONS 定义屏蔽了异常，则函数会调用 rapidxml::parse_error_handler() 函数。

参数

source

用来初始化已分配内存的字符串，0表示不初始化。

size

分配多少字符，为0表示从源字符串中自动计算大小，如果 size 非0，必须传递以 null 为终止符的 source 并。

返回值

指向已分配的字符数组指针，此指针永远不会为 NULL。

函数 memory_pool::clone_node

原型

xml_node<Ch>* clone_node(const xml_node< Ch > *source, xml_node< Ch > *result=0);

描述

复制一个 xml_node 并复制它的所有字节点和属性。节点和属性从当前内存池中分配，但名称和值不复制，它们共享克隆和源文本。可以将结果节点指定为第二个参数，在这种情况下它的内容会被源节点的

克隆体替换，这对于想复制整个文档时很有用。

参数

source

要克隆的节点。

result

要放置的结果节点，0表示自动分配结果节点。

返回值

指向已克隆的节点指针，该指针永远不会为 NULL。

函数 memory_pool::clear

原型

void clear();

描述

清空内存池，这会引发池中已分配的节点被释放，任何在池中分配的节点和字符串都不再有效。

函数 memory_pool::set_allocator

原型

void set_allocator(alloc_func *af, free_func *ff);

描述

设置或重置内存池的用户自定义内存分配函数。此函数只能在池中没有任何分配前调用，否则结果是未定义的。分配函数在失败时绝对不能返回无效指针，而是应该抛出异常，停止程序，或者使用 longjmp()函数跳转到程序的其它位置。如果返回无效指针，结果是未定义的。

用户自定义的分配函数必须是下列的原型：

```
void *allocate(std::size_t size);
void free(void *pointer);
```

参数

af

分配函数，0表示还原为默认函数。

ff

释放函数，0表示还原为默认函数。

类 rapidxml::parse_error

定义在 rapidxml.hpp

描述

分析异常错误，此异常是被分析器在发生异常时抛出的。使用 what()函数能获取易读的错误信息。使用 where()函数可获取指向原文本中检测到错误的位置的指针。

如果不允许分析器抛出异常，可以在包含 rapidxml.hpp 之前用 RAPIDXML_NO_EXCEPTIONS 宏来屏蔽。这会时分析器调用 rapidxml::parse_error_handler()函数来替换抛出异常，从此函数必须由用户来定义。

此类继承自 std::exception 类。

构造函数 parse_error::parse_error

原型

parse_error(const char *what, void *where);

描述

构造一个错误分析器。

函数 parse_error::what

原型

virtual const char* what() const;

描述

获取易读的错误描述。

返回

指向以 null 结尾的错误描述指针。

函数 parse_error::where

原型

Ch* where() const;

描述

获取错误发生后的字符数据指针，Ch 的字符类型与产生错误的 xml_document 一致。

返回值

指向在分析字符串中发生错误的位置指针。

类模板 rapidxml::xml_attribute

定义在 n rapidxml.hpp

从 xml_base 继承

描述

此类作为 XML 文档的节点属性，每个属性都有名称和值字符串，可通过 name()和 value()函数来调用（继承自 xml_base）。注意当分析完后，名称和值都会指向正在分析的源文本内部。因此，这个文本在内存中必须与属性有相同的生命期。

参数

Ch

要使用的字符集

构造函数 xml_attribute::xml_attribute

原型

```
xml_attribute();
```

描述

用指定类型构造一个空属性，如果是手工分配属性请考虑使用 xml_document 的 memory_pool 。

函数 xml_attribute::document

原型

```
xml_document<Ch>* document() const;
```

描述

获取属性所属的文档。

返回

指向包含此属性的文档，0表示没有父文档。

函数 xml_attribute::previous_attribute

原型

xml_attribute<Ch>* previous_attribute(const Ch *name=0, std::size_t name_size=0, bool case_sensitive=true) const;

描述

获取前一个属性，可匹配属性名。

参数

name

要寻找的属性名，0直接表示的上一个属性，不管名称。如果 name_size 非0则此字符串结尾不能用0终止。

name_size

名称的大小，为字符数，0表示从字符串中自动计算大小。

case_sensitive

用于比较名称的大小写，对 ASCII 而言只能用忽略大小写比较。

返回值

指向找到属性的指针，0表示未找到。

函数 xml_attribute::next_attribute

原型

xml_attribute<Ch>* next_attribute(const Ch *name=0, std::size_t name_size=0, bool case_sensitive=true) const;

描述

获取下一个属性，可匹配属性名称。

参数

name

要寻找的属性名称，0表示直接返回下一个属性，不管名称。如果 name_size 非0则此字符串结尾不能用0终止。

name_size

名称的大小，为字符数，0表示从字符串中自动计算大小。

case_sensitive

用于比较名称的大小写，对 ASCII 而言只能用忽略大小写比较。

返回值

指向找到属性的指针，0表示未找到。

类模板 rapidxml::xml_base

定义在 rapidxml.hpp

xml_attribute xml_node 的基类

描述

xml_attribute xml_node 的基类，实现了公共函数：name(), name_size(), value(), value_size() 和 parent().

参数

Ch

要使用的字符集。

构造函数 xml_base::xml_base

原型

xml_base();

函数 xml_base::name

原型

Ch* name() const;

描述

获取节点的名称 ,解析节点关联类型的名称。注意当使用了 rapidxml::parse_no_string_terminators 来分析时名称字符串不能以0结尾，请使用 name_size() 函数来确定名称长度。

返回值

节点的名称，空字符串表示节点没有名称。

函数 xml_base::name_size

原型

std::size_t name_size() const;

描述

获取节点名称的大小，不包括终止符。此函数不管名称是否是以0结尾的都能正常地工作。

返回值

节点名称的大小，以字符个数为单位（译注：返回值不以字节为单位，因为不同字符集用不同的存储空间来存储字符，有时一个字符会占用两个字节）。

函数 xml_base::value

原型

Ch* value() const;

描述

获取节点的值，根据节点类型来解释值。注意，当使用了 rapidxml::parse_no_string_terminators 选项来分析时节点值是非0终止的。

使用 value_size()函数来确定值的长度。

返回值

节点的值，如果节点没有值则返回空字符串。

函数 xml_base::value_size

原型

std::size_t value_size() const;

描述

获取节点值的大小，不包括终止字符。此函数不管节点值是否有0终止都能正常工作。

返回值

节点值的大小，以字符数为单位。

函数 xml_base::name

原型

void name(const Ch *name, std::size_t size);

描述

将节点的名称设置一个非0终止的字符串，可查看 字符串的关系。

注意节点并不拥有它自己的名称和值，它只是存储了指向它们的指针。在销毁的时候并不会删除或释放指针。管理字符串的生命期是用户的责任。最简单的方式还是使用文档提供的 memory_pool 来分配字符串，这样当文档销毁时字符串也会自动释放掉。

名称的大小必须被指定，因为名称并没有用0来终止，使用 name(const Ch*)函数可以自动计算长度（但字符串必须是用0终止的）。

参数

name

要设置的节点名称，无需用0来终止。

size

名称的大小，用字符数计算，不包括0终止符。

函数 xml_base::name

原型

void name(const Ch *name);

描述

用0终止符的字符串来设置节点的名称，查看 字符串关系 和 xml_node::name(const Ch *, std::size_t).

参数

name

要设置的节点名称，必须以0终止。

函数 xml_base::value

原型

void value(const Ch *value, std::size_t size);

描述

用一个非0终止的字符串来设置节点的值，请查看 字符串的关系。

注意节点并不拥有它自己的名称和值，它只是存储了指向它们的指针。在销毁的时候并不会删除或释放指针。管理字符串的生命期是用户的责任。最简单的方式还是使用文档提供的 memory_pool 来分配字符串，这样当文档销毁时字符串也会自动释放掉。

值的大小必须被指定，因为名称并没有用0来终止，使用 value(const Ch*)函数可以自动计算长度（但

字符串必须是用0终止的)。

如果一个元素有 node_data 类型的子节点时，它会在打印时优先于值。如果你想通过元素的值来操作元素，请在用分析器创建节点的时候使用 rapidxml::parse_no_data_nodes 分析标记。

参数

value

要设置的节点值，不需要用0终止。

size

值的大小，用字符数来计算，不包含0终止符。

函数 xml_base::value

原型

void value(const Ch *value);

描述

用一个以0终止的字符串来设置节点的值，请查看 字符串的关系和 xml_node::value(const Ch *, std::size_t).

参数

value

要设置的节点值，必须以0为终止符。

函数 xml_base::parent

原型

xml_node<Ch>* parent() const;

描述

获取节点的父亲

返回值

指向节点的父节点，0表示没有父节点。

类模板 rapidxml::xml_document

定义在 rapidxml.hpp

继承自 xml_node memory_pool

描述

此类提供了 DOM 层次的 root 节点。它也是一个 xml_node 和一个共有继承的 memory_pool 。使用 parse()函数可以从一个以0为终止符的字符串中创建一个 DOM 树。可通过 xml_document 的相关函数来为 parse()函数分配节点和属性，这也是从 memory_pool 中继承的。要访问一个文档的根节点，

请使用文档本身，如果它是一个 xml_node 的话。

参数

Ch

要使用的字符类型

构造函数 xml_document::xml_document

原型

```
xml_document();
```

描述

构造一个空的 XML 文档。

函数 xml_document::parse

原型

```
void parse(Ch *text);
```

描述

根据给定的分析标记分析一个以0为终止符的 XML 字符串。传递过来的字符串会被分析修改，除非使用 rapidxml::parse_non_destructive 标记，字符串必须有与文档一样长的存留时间。在异常错误方面，rapidxml::parse_error 可能会被抛出。

文档可能会被分析好几次，每次重新调用 parse 都会移除之前的节点和属性，但并不清空内存池。

参数

text

要分析的 XML 数据，指针为非 const 表示此数据会被分析器修改。

函数 xml_document::clear

原型

void clear();

描述

通过删除所有节点和清空内存池来清除文档。池中文档所拥有的所有节点将会销毁。

类模板 rapidxml::xml_node

定义在 rapidxml.hpp

继承自 xml_base

xml_document 的基类

描述

此类为 XML 文档提供一个节点，每个节点可以通过 name()和 value()函数关联名称和值字符串，通

过节点的关联类型来解释名称和值，节点的类型可用 type()函数来确定。

注意当分析完后，节点的名称和值，不管怎样，都会在分析时指向原文本的内部。因此，在内存中的文本必须与节点有同样的生命期。

参数

Ch

要使用的字符类型

构造函数 xml_node::xml_node

原型

xml_node(node_type type);

描述

用指定类型构造一个空节点，若要手工分配节点最好还是使用文档的内存池。

参数

type

构造节点的类型。

函数 xml_node::type

原型

node_type type() const;

描述

获取节点的类型

返回值

节点的类型。

函数 xml_node::document

原型

xml_document<Ch>* document() const;

描述

获取包含此节点的 xml 文档对象。

返回值

指向包含此节点文档，0表示此节点无父文档。

函数 xml_node::first_node

原型

xml_node<Ch>* first_node(const Ch *name=0, std::size_t name_size=0, bool case_sensitive=true) const;

描述

获取第一个子节点，可用节点名匹配。

参数

name

要查找节点的名称，0表示不理会名称返回第一个节点，如果 name_size 为非0则名称字符串不能用0来终止。

name_size

名称的大小，用字符数计算，0表示自动从字符串计算。

case_sensitive

可以比较名称的大小写，非大小写比较只能工作在 ASCII 字符下。

返回值

指向要找的子节点，0表示未找到。

函数 xml_node::last_node

原型

xml_node<Ch>* last_node(const Ch *name=0, std::size_t name_size=0, bool

case_sensitive=true) const;

描述

获取最后一个子节点，可用节点名匹配。如果节点没有孩子结果是未定义的。可使用 first_node()来测试一个节点是否有孩子。

参数

name

要查找节点的名称，0表示不理会名称返回第一个节点，如果 name_size 为非0则名称字符串不能用0来终止。

name_size

名称的大小，用字符数计算，0表示自动从字符串计算。

case_sensitive

可以比较名称的大小写，非大小写比较只能工作在 ASCII 字符下。

返回值

指向要找的子节点，0表示未找到。

函数 xml_node::previous_sibling

原型

xml_node<Ch>* previous_sibling(const Ch *name=0, std::size_t name_size=0, bool

case_sensitive=true) const;

描述

获取前一个相邻节点，可用节点名匹配，如果节点没有父亲此行为是未定义的。使用 parent()可测试节点是否有父亲。

参数

name

要查找的相邻节点名称，0表示不理会名称直接返回第一个相邻节点，如果 name_size 为非0则名称字符串不能用0来终止。

name_size

名称的大小，用字符数计算，0表示自动从字符串计算。

case_sensitive

可以比较名称的大小写，非大小写比较只能工作在 ASCII 字符下。

返回值

指向找到的相邻节点，0表示未找到。

函数 xml_node::next_sibling

原型

xml_node<Ch>* next_sibling(const Ch *name=0, std::size_t name_size=0, bool

case_sensitive=true) const;

描述

获取下一个相邻节点，可用节点名匹配，如果节点没有父亲此行为是未定义的。使用 parent()可测试节点是否有父亲。

参数

name

要查找的相邻节点名称，0表示不理会名称直接返回第一个相邻节点，如果 name_size 为非0则名称字符串不能用0来终止。

name_size

名称的大小，用字符数计算，0表示自动从字符串计算。

case_sensitive

可以比较名称的大小写，非大小写比较只能工作在 ASCII 字符下。

返回值

指向找到的相邻节点，0表示未找到。

函数 xml_node::first_attribute

原型

xml_attribute<Ch>* first_attribute(const Ch *name=0, std::size_t name_size=0, bool

case_sensitive=true) const;

描述

获取节点的第一个属性，可用属性名匹配。

参数

name

要查找的属性名称，0表示不理会名称直接返回第一个属性，如果 name_size 为非0则字符串不能用0来终止。

name_size

名称的大小，用字符数计算，0表示自动从字符串计算。

case_sensitive

可以比较名称的大小写，非大小写比较只能工作在 ASCII 字符下。

返回值

指向找到的第一个属性，0表示未找到。

函数 xml_node::last_attribute

原型

xml_attribute<Ch>* last_attribute(const Ch *name=0, std::size_t name_size=0, bool case_sensitive=true) const;

描述

获取节点的最后一个属性，可用属性名匹配。

参数

name

要查找的属性名称，0表示不理会名称直接返回属性，如果 name_size 为非0则字符串不能用0来终止。

name_size

名称的大小，用字符数计算，0表示自动从字符串计算。

case_sensitive

可以比较名称的大小写，非大小写比较只能工作在 ASCII 字符下。

返回值

指向找到的属性，0表示未找到。

函数 xml_node::type

原型

void type(node_type type);

描述

设置节点的类型。

参数

type

要设置的节点类型。

函数 xml_node::prepend_node

原型

void prepend_node(xml_node< Ch > *child);

描述

在起始处加入一个新节点，此节点将会成为第一个节点，而且现存的子节点都会向后移动一个位置。

参数

child

要添加的节点。

函数 xml_node::append_node

原型

void append_node(xml_node< Ch > *child);

描述

在尾处追加一个新节点，此节点会成为最后一个节点。

参数

child

要追加的节点。

函数 xml_node::insert_node

原型

```
void insert_node(xml_node< Ch > *where, xml_node< Ch > *child);
```

描述

在指定的一个位置插入一个新节点，包括指定节点在内的所有节点会向后移动一个位置。

参数

where

插入子节点的地方，0表示在末端插入。

child

要插入的节点。

函数 xml_node::remove_first_node

原型

void remove_first_node();

描述

移除第一个子节点，如果节点没有孩子，行为是未定义的。可用 first_node()函数来测试节点是否有孩子。

函数 xml_node::remove_last_node

原型

void remove_last_node();

描述

移除节点的最后一个子节点，如果节点没有孩子，行为是未定义的。可用 first_node()函数来测试节点是否有孩子。

函数 xml_node::remove_node

原型

void remove_node(xml_node< Ch > *where);

描述

删除指定节点的子节点。

函数 xml_node::remove_all_nodes

原型

void remove_all_nodes();

描述

移除所有的子节点（不是属性）

函数 xml_node::prepend_attribute

原型

void prepend_attribute(xml_attribute< Ch > *attribute);

描述

在节点的起始处添加一个新属性。

参数

attribute

要添加的属性

函数 xml_node::append_attribute

原型

void append_attribute(xml_attribute< Ch > *attribute);

描述

在节点的末尾追加一个新属性。

参数

attribute

要追加的属性。

函数 xml_node::insert_attribute

原型

void insert_attribute(xml_attribute< Ch > *where, xml_attribute< Ch > *attribute);

描述

在指定位置插入一个新属性，包含位置属性在内的所有属性都会向后移动一个位置。

参数

where

要插入属性的位置，0表示插入到最后位置。

attribute

要插入的属性。

函数 xml_node::remove_first_attribute

原型

void remove_first_attribute();

描述

移除节点的第一个属性，如果节点没有属性，此行为是未定义的。可用 first_attribute()函数来测试节点是否有属性。

函数 xml_node::remove_last_attribute

原型

void remove_last_attribute();

描述

移除节点的最后一个属性，如果节点没有属性，此行为是未定义的。可用 first_attribute()函数来测试节点是否有属性。

函数 xml_node::remove_attribute

原型

void remove_attribute(xml_attribute< Ch > *where);

描述

从节点中删除指定的属性。

参数

where

指向要删除属性。

函数 xml_node::remove_all_attributes

原型

void remove_all_attributes();

描述

删除节点的所有属性。

枚举类型 node_type

描述

枚举由分析器提供的所有节点类型。使用 xml_node::type() 函数可查询节点类型。

值

node_document

一个文档节点，名称和值都为空。

node_element

一个元素节点，Name 包含元素名，Value 包含节点的第一块数据。

node_data

一个数据节点，Name 为空，Value 包含数据文本。

node_cdata

一个 CDATA 节点，Name 为空，Value 包含数据文本。

node_comment

一个注释节点，Name 为空，Value 包含数据文本。

node_declaration

一个声明节点，Name 和 Value 都为空，声明参数(version,encoding 和 standalone)在节点的属性中。

node_doctype

一个 DOCTYPE 节点，Name 为空，Value 包含 DOCTYPE 文本。

node_pi

一个 PI 节点，Name 包含目标，Value 包含指令。

函数 parse_error_handler

原型

```
void rapidxml::parse_error_handler(const char *what, void *where);
```

描述

当用 RAPIDXML_NO_EXCEPTIONS 定义屏蔽了异常后，此函数可以被调用来通知用户发生了错误. 这必须由用户定义。

此函数不能 return，如果这样做，结果是未定义的。

最简单的定义可像这样：

```
void rapidxml::parse_error_handler(const char *what, void *where)

{ std::cout << "Parse error: " << what << "\n"; std::abort(); }
```

参数

what

可读的错误描述。

where

指向检测到错误字符位置。

函数 print

原型

```
OutIt rapidxml::print(OutIt out, const xml_node< Ch > &node, int flags=0);
```

描述

在给定的输出 iterator 中打印 XML。

参数

out

要打印的输出 iterator。

node

要打印的节点，传递 xml_document 会打印整个文档。

flags

控制打印的标记。

返回值

指向打印文本的最后一个 iterator 指针。

函数 print

原型

std::basic_ostream<Ch>& rapidxml::print(std::basic_ostream< Ch > &out, const xml_node< Ch > &node, int flags=0);

描述

在给定的输出流中打印 XML。

参数

out

要打印的输出流。

node

要打印的节点，传递 xml_document 会打印整个文档。

flags

控制打印的标记。

返回值

输出流。

函数 operator<<

原型

std::basic_ostream<Ch>& rapidxml::operator<<(std::basic_ostream< Ch > &out, const xml_node< Ch > &node);

描述

在给定的输出流中打印格式化的 XML。使用默认的打印标记，使用 print()函数可自定义打印过程。

参数

out

要打印的输出流。

node

要打印的节点。

返回值

输出流。

常数 parse_no_data_nodes

原型

```
const int parse_no_data_nodes = 0x1;
```

描述

指示分析器不创建数据节点的分析标记，数据节点的文本仍然会放置在父元素中，除非同时指定了 rapidxml::parse_no_element_values 标记，可用 | 操作符来组合其它标记。

查看 xml_document::parse() 函数。

常数 parse_no_element_values

原型

```
const int parse_no_element_values = 0x2;
```

描述

此分析标记指示分析器不要使用父元素的第一个数据节点文本。可用 | 操作符组合其它标记。注意，一个元素节点的子数据节点会在打印时覆盖它的值。也就是说，如果元素超过一个或更多的子数据节点并有一个值时 ,该值会被忽略。如果你想使用元素值来处理数据时使用 rapidxml::parse_no_data_nodes 标记可防止创建数据节点。

查看 xml_document::parse() 函数。

常数 parse_no_string_terminators

原型

const int parse_no_string_terminators = 0x4;

描述

分析标记指示分析器不再原文本中放置0终止符。默认情况下是会放置0终止符的，会修改原文本。可用 | 操作符组合其它标记。

查看 xml_document::parse() 函数。

常数 parse_no_entity_translation

原型

const int parse_no_entity_translation = 0x8;

描述

分析标记指示分析器不转换源文本的实体。默认情况下会转换，并修改原文本。可用 | 操作符组合其它标记。

查看 xml_document::parse() 函数。

常数 parse_no_utf8

原型

const int parse_no_utf8 = 0x10;

描述

分析标记指示分析器关闭 UTF-8处理并假定为8位的文本字符。默认 UTF-8处理是开启的，可用 | 操作符组合其它标记。

查看 xml_document::parse() 函数。

常数 parse_declaration_node

原型

const int parse_declaration_node = 0x20;

描述

分析标记指示分析器创建 XML 声明节点，默认不创建声明节点。可用 | 操作符组合其它标记。

查看 xml_document::parse() 函数。

常数 parse_comment_nodes

原型

const int parse_comment_nodes = 0x40;

描述

分析标记指示分析器创建 XML 注释节点，默认不创建注释节点。可用 | 操作符组合其它标记。

查看 xml_document::parse() 函数。

常数 parse_doctype_node

原型

const int parse_doctype_node = 0x80;

描述

分析标记指示分析器创建 DOCTYPE 节点，默认不创建 DOCTYPE 节点。W3C 规格中运行最多一个 DOCTYPE 节点，Rapidxml 支持此文档并可超过一个。可用 | 操作符组合其它标记。

查看 xml_document::parse() 函数。

常数 parse_pi_nodes

原型

const int parse_pi_nodes = 0x100;

描述

分析标记指示分析器创建 PI 节点，默认不创建 PI 节点。可用 | 操作符组合其它标记。

查看 xml_document::parse() 函数。

常数 parse_validate_closing_tags

原型

const int parse_validate_closing_tags = 0x200;

描述

分析标记指示分析器验证关闭 tag 名字。如果未设置，名称的关闭 tag 对分析器来说无关紧要。默认不验证关闭 tag 名字。可用 | 操作符组合其它标记。

查看 xml_document::parse() 函数。

常数 parse_trim_whitespace

原型

const int parse_trim_whitespace = 0x400;

描述

分析标记指示分析器去除数据节点的空白符。默认不去除。此标记不会引发分析器去修改源文本，可用 | 操作符组合其它标记。

查看 xml_document::parse() 函数。

常数 parse_normalize_whitespace

原型

const int parse_normalize_whitespace = 0x800;

描述

分析标记指示分析器压缩所有的数据节点中的空白符为一个单一的空格字符。去除前后空白符可用 rapidxml::parse_trim_whitespace 标记，默认空白符不会被正常化。如果指定了此标记，原文本会被修改。可用 | 操作符组合其它标记。

查看 xml_document::parse() 函数。

常数 parse_default

原型

const int parse_default = 0;

描述

此分析标记为分析器的默认行为，此值总是为0，所以可以很简单的组合其它标记。正常情况下无需通过 ~ 符号来关闭其它标记。也就是说每个标记都是默认标记的负数。例如，如果标记为 rapidxml::parse_no_utf8，表示默认情况下开启了 utf-8，并使用此标记来屏蔽此标记。

查看 xml_document::parse() 函数。

常数 parse_non_destructive

原型

const int parse_non_destructive = parse_no_string_terminators | parse_no_entity_translation;

描述

A combination of parse flags that forbids any modifications of the source text. This also results in faster parsing. However, note that the following will occur:

此组合标记表示严禁修改原文本。这也是使得的分析速度更快。但无论如何，请注意下面的事情：

- 节点的名称和值不会被0终止，你必须使用 xml_base::name_size()和 xml_base::value_size()函数来确定名称和值的结束。
- 实体不会被转换。
- 空白符不会被正常化。

查看 xml_document::parse() 函数。

常数 parse_fastest

原型

const int parse_fastest = parse_non_destructive | parse_no_data_nodes;

描述

此组合标记是最快的分析标记，不会牺牲主要数据。查看 xml_document::parse() 函数。

常数 parse_full

原型

const int parse_full = parse_declaration_node | parse_comment_nodes | parse_doctype_node | parse_pi_nodes | parse_validate_closing_tags;

描述

此组合标记是最大范围的数据处理。也是最慢的分析。查看 xml_document::parse() 函数。

常数 print_no_indenting

原型

const int print_no_indenting = 0x1;

描述

打印分析标记指示打印器制止缩排 XML。查看 xml_document::parse() 函数。