Tokit库

使用手册

洪坤安(psydollar@gmail.com)

©2014年10月8日  TOKIT

[1. Tokit库简介 1](#_Toc402382983)

[1.1简介 1](#_Toc402382984)

[1.2 Tokit可以做到什么 1](#_Toc402382985)

[1.3 Tokit的由来 2](#_Toc402382986)

[1.4如何使用Tokit 2](#_Toc402382987)

[1.5示例 6](#_Toc402382988)

[2. 主流技术简介 10](#_Toc402382989)

[2.1 使用xml文件 10](#_Toc402382990)

[2.2. excel转csv文件 18](#_Toc402382991)

[2.3. 直接存在数据库中 19](#_Toc402382992)

[2.4 流程小结 20](#_Toc402382993)

[3. 关于Tokit库的一些补充 21](#_Toc402382994)

[3.1 问：一般的做法不是已经很方便了吗，为什么还要有Tokit 21](#_Toc402382995)

[3.2问：如果希望对Tokit生成的文件进行调整，该怎么做？ 21](#_Toc402382996)

[3.3问： Tokit生成的c++代码载入xml的速度快吗？ 22](#_Toc402382997)

# 1. Tokit库简介

## 1.1简介

Tokit是一个小工具库，服务于网游配置数据场景，用于方便策划在excel中编辑配置数据、程序人员根据excel中的数据编写对应代码，旨在方便程序和策划的对接，改善数据制作流程以实现快速开发。目前仅支持c++。

Tokit的发布地址是：<https://github.com/tokitgit/tokit>

## 1.2 Tokit可以做到什么

tokit支持只通过一个excel就生成对应的c++文件和xml数据文件，并支持生成xsd文件。其中，c++文件将包含自动载入xml的方法并生成出必要的接口，可以减小程序人员方面的工作量。

tokit目前仅支持生成c++文件。

## 1.3 Tokit的由来

12年的时候，我还在一家端游研发公司工作做服务器研发，其时，因项目需要，由我负责对原有的xml配置数据导入导出流程进行整理，并被明确要求通过对象序列化的技术进行配置数据载入速度优化。

当时服务端(c++)采用的是CodeSynthesis XSD库，虽然成功完成了优化，但基于这个库流程的繁琐和复杂仍给我留下深刻印象，也因此，我开始留意其他项目采用的配置数据导出流程。

后来，公司内开新的页游项目，我又根据主管以前项目的经验，抛弃了CodeSynthesis XSD库，基于《游戏编程精粹4》中《使用XMl而不牺牲速度》文章提供的xdstoolkit 1.03库重新构建了一整套新的流程。

由于天生为速度而生，所以，虽然xdstoolkit并不容易理解，且与流程的契合仍然不足，在编译速度和处理速度还是给我留下了深刻的印象。

也因此，尽管后来去了新公司，并大概了解到了更多项目通行的导入导出配置数据的做法，如excel转csv、excel中使用vba宏导出xml、直接存数据库，或者，我仍然觉得xdstoolkit的优势实在巨大。于是，尽管已经拿不到我自己写的源码和齐备的文档，我还是决定重写一套。

然而，时过境迁，在重写的过程中，也慢慢意识到快速开发才是关键，对于服务端来说配置数据的载入速度如何似乎并不显得如何重要，我也便渐渐萌发了写一套单独的，仅为了快速开发的c++小工具的想法。

这个工具将专门服务于网游的配置数据过程，旨在方便策划人员配置数据以及c++程序方面与数据的载入、对接，并尽可能的方便使用。

工具的名称从xsdtoolkit中截取，取to和kit的结合。

这也就是tokit库的由来。

## 1.4如何使用Tokit

### 1.4.1使用Tokit命令

使用tokit要求有一个符合tokit格式的excel文件，使用方式如下

|  |
| --- |
| **tokit.exe** *<excel文件>*  [-xsd*<放置xsd文件的目录>*] **|**  [-c++ *<h模板> <cpp模板> <放置c++文件的目录>*] **|**  [-saveasxml  *<放置xml文件的目录>*]  选项：  -xsd  根据excel生成xsd文件  -c++  根据excel生成c++文件  -saveasxml  将excel文件导出成为xml数据文件  注意:  **其中的excel文件必须符合tokit要求的格式，否则将提示出错** |

比如

|  |
| --- |
| 假设有一个符合tokit格式的excel文件《装备.xlsx》，里面有一个工作表item，则  **tokit.exe** *../装备.xlsx* -xsd*./xsd/*「将在xsd目录下生成item.xsd文件」  **tokit.exe** *../装备.xlsx* -c++*./template/c++\_template.h ./template/c++\_template.cpp ./c++/*「将在c++目录下生成item.h和item.cpp文件」  **tokit.exe** *../装备.xlsx* -saveasxml*./xml/*「将在xml目录下生成item.xml数据文件」  这些参数可结合起来：  **tokit.exe** *../装备.xlsx* -xsd*./xsd/* -c++*./template/c++\_template.h ./template/c++\_template.cpp ./c++/*  -saveasxml*./xml/*「将生成item.xsd、item.h、item.cpp和item.xml文件」 |

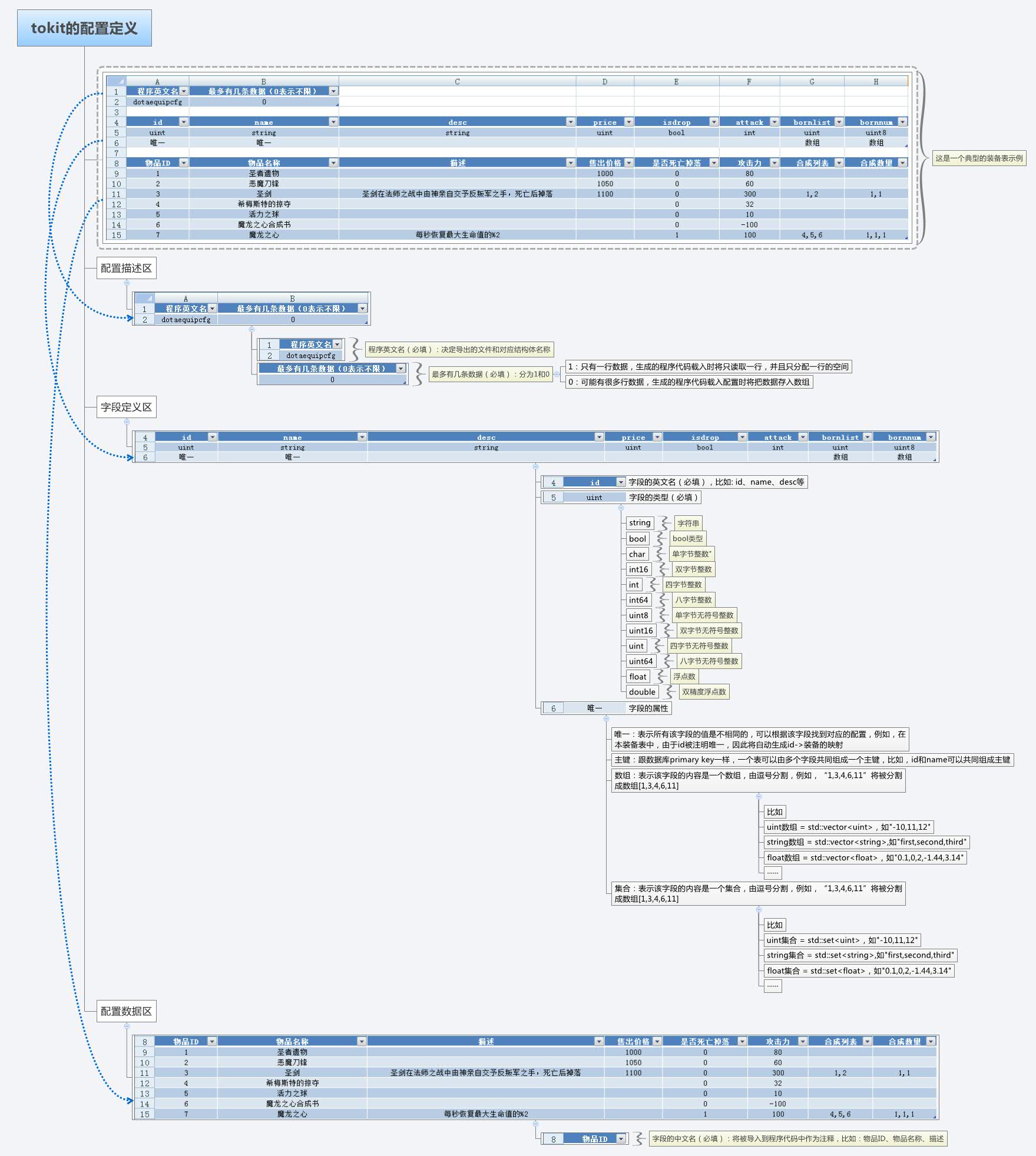
### 1.4.2制作Tokit所需要的excel

tokit需要的的excel格式如下表：

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  |  | | --- | --- | --- | --- | | **程序英文名** | **最多有几条数据（0表示不限）** |  | 文件定义区 | | 必填，例如:dotaequipcfg，决定生成的程序文件名和类名 | 0 |  |  | |  |  |  |  | | **字段1的英文名，例如:id** | **字段2的英文名** | **字段3的英文名** | **…**  字段定义区 | | 字段类型，必填，例如:uint | … | … | … | | 字段属性，选填，例如:唯一 |  |  |  | |  |  |  | 配置数据区 | | **字段1的描述，例如：装备Id，用于标识装备** | **字段2的描述** | **字段3的描述** | **…** | | 数据区域…… | … | … | … | | 数据区域…… | … | … | … | | … | … | … | … | |

其中，excel分为3个区域，文件定义区、字段定义区、配置数据区，这3块区域的起始行都必须固定，否则无法被tokit识别。

我们以一个典型的装备表为例，装备表的英文名称为dotaequipcfg，详细各个单元格的含义可参见下图（可放大查看）。



### 1.4.3在项目中接入tokit

c++项目想要和tokit对接，必须先包含tokit需要的文件

1.<type.h>、<tokit\_util.h>、<tokit\_util.cpp>

2.rapidxml第三方开源库（用于载入xml数据）

项目跟tokit对接后，载入数据的工作很简单，包含所需文件后，添加一行语句即可：

|  |
| --- |
| xxxxx::instance().load(); |

详细可参照example\c++\c++\_example项目

## 1.5示例

我们以最普遍的场景<装备表>、<英雄表>为例，除了一般的装备数据和英雄数据以外，现假设有需求：其中的一些装备可由其他装备合成而来。

于是策划制订了如下的excel表格（装备名和英雄名直接取自dota）。

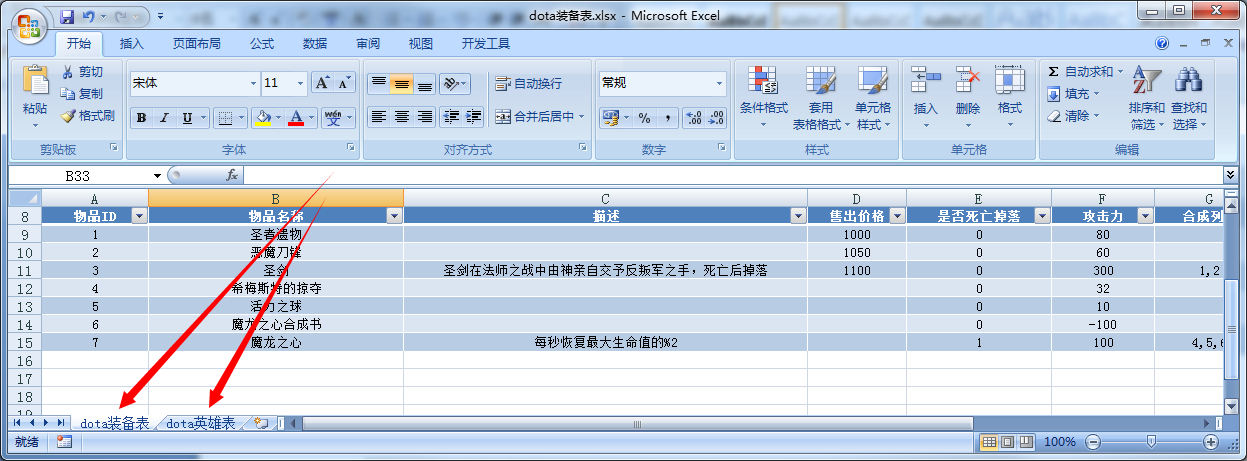
|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **物品ID** | **物品名称** | **描述** | **售出价格** | **是否死亡掉落** | **攻击力** | **合成列表** | **合成数量** |
| 1 | 圣者遗物 |  | 1000 | 0 | 80 |  |  |
| 2 | 恶魔刀锋 |  | 1050 | 0 | 60 |  |  |
| 3 | 圣剑 | 圣剑在法师之战中由神亲自交予反叛军之手，死亡后掉落 | 1100 | 0 | 300 | 1,2 | 1,1 |
| 4 | 希梅斯特的掠夺 |  |  | 0 | 32 |  |  |
| 5 | 活力之球 |  |  | 0 | 10 |  |  |
| 6 | 魔龙之心合成书 |  |  | 0 | -100 |  |  |
| 7 | 魔龙之心 | 每秒恢复最大生命值的%2 |  | 1 | 100 | 4,5,6 | 1,1,1 |

**dota装备表**（<合成列表>字段表示合成该装备所需的装备Id列表，由逗号分割）

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **英雄ID** | **英雄名称** | **力量** | **敏捷** | **智力** |
| 1 | 小黑 | 10.5 | 20.00001 | 301.1111 |
| 2 | 风行 | 10.5 | 20.00001 | 301.1111 |
| 3 | 凤凰 | 10.5 | 20.00001 | 301.1111 |
| 4 | 斧王 | 10.5 | 20.00001 | 301.1111 |
| 5 | 宙斯 | 10.5 | 20.00001 | 301.1111 |
| 6 | 虚空 | 10.5 | 20.00001 | 301.1111 |
| 7 | 猴子 | 10.5 | 20.00001 | 301.1111 |

**dota英雄表**

两个表分别存放在一个excel的2个工作表中



这时候，策划的工作已经完成了。

一般情况下，程序人员要想使用里面的数据，得开始准备写一些载入代码以及必要的查找装备和伙伴接口。

但使用tokit可以减少大部分的工作量，因为很多重复的代码都可以由tokit自动生成出来，免去重复劳动。

不过此时的格式是无法被tokit识别的，因为缺少每个字段的类型信息，所以需要程序人员再定义好各个字段的类型，并配置好生成的c++类的名称。

→从这一步开始接入tokit。

程序人员补上其余信息，使装备表和英雄表变成如下。

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **程序英文名** | **最多有几条数据（0表示不限）** |  |  |  |  |  |  |
| dotaequipcfg | 0 |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
| **id** | **name** | **desc** | **price** | **isdrop** | **attack** | **bornlist** | **bornnum** |
| uint | string | string | uint | bool | int | uint | uint8 |
| 唯一 | 唯一 |  |  |  |  | 数组 | 数组 |
|  |  |  |  |  |  |  |  |
| **物品ID** | **物品名称** | **描述** | **售出价格** | **是否死亡掉落** | **攻击力** | **合成列表** | **合成数量** |
| 1 | 圣者遗物 |  | 1000 | 0 | 80 |  |  |
| 2 | 恶魔刀锋 |  | 1050 | 0 | 60 |  |  |
| 3 | 圣剑 | 圣剑在法师之战中由神亲自交予反叛军之手，死亡后掉落 | 1100 | 0 | 300 | 1,2 | 1,1 |
| 4 | 希梅斯特的掠夺 |  |  | 0 | 32 |  |  |
| 5 | 活力之球 |  |  | 0 | 10 |  |  |
| 6 | 魔龙之心合成书 |  |  | 0 | -100 |  |  |
| 7 | 魔龙之心 | 每秒恢复最大生命值的%2 |  | 1 | 100 | 4,5,6 | 1,1,1 |

**dota装备表**（相比上面的装备表，添加了：装备表的英文名dotaequipcfg、各个字段的英文名和程序类型等）

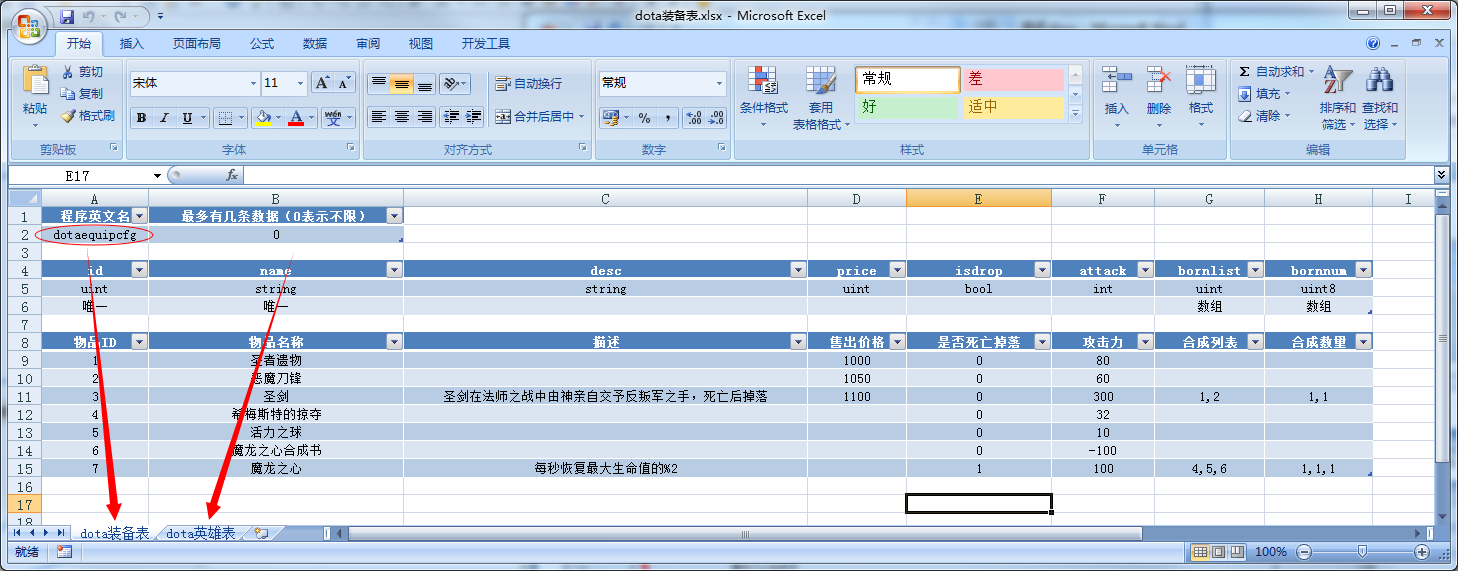
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **程序英文名** | **最多有几条数据（0表示不限）** |  |  |  |
| dotaherocfg | 0 |  |  |  |
|  |  |  |  |  |
| **id** | **name** | **strength** | **agile** | **intelligense** |
| uint | string | double | double | double |
| 唯一 | 唯一 |  |  |  |
|  |  |  |  |  |
| **英雄ID** | **英雄名称** | **力量** | **敏捷** | **智力** |
| 1 | 小黑 | 10.5 | 20.00001 | 301.1111 |
| 2 | 风行 | 10.5 | 20.00001 | 301.1111 |
| 3 | 凤凰 | 10.5 | 20.00001 | 301.1111 |
| 4 | 斧王 | 10.5 | 20.00001 | 301.1111 |
| 5 | 宙斯 | 10.5 | 20.00001 | 301.1111 |
| 6 | 虚空 | 10.5 | 20.00001 | 301.1111 |
| 7 | 猴子 | 10.5 | 20.00001 | 301.1111 |

**dota英雄表**

可以注意到，<装备表>和<英雄表>的id和name都填上了<唯一>，表示所有装备和英雄的id和名称都是唯一的，因此tokit将认为，可根据id和name找到对应的装备和伙伴。

并且，<合成列表>填上了<数组>，并且里面的值都由“，”分割开来，表示一个装备可能由多个装备合成而来。tokit将自动将<合成列表>识别成数组，并在载入的时候将里面的值按“，”进行分割。

修改后的excel内容如下所示



由于装备表排在第一个，名为**dotaequipcfg**，于是tokit生成了对应的c++文件<dotaequipcfg.h>和<dotaequipcfg.cpp>，里面包含了dotaequipcfgmgr类及其实现，并定义了2个结构体：dotaequipcfg\_t和dotaherocfg\_t。如下：

|  |
| --- |
| // dota装备表  struct dotaequipcfg\_t{  typedef uint32 id\_t;  typedef std::string name\_t;  typedef std::vector<uint32> bornlistvec\_t;  typedef std::vector<uint8> bornnumvec\_t;  dotaequipcfg\_t();  id\_t id; // 物品ID<<<唯一>>>  name\_t name; // 物品名称<<<唯一>>>  std::string desc; // 描述  uint32 price; // 售出价格  bool isdrop; // 是否死亡掉落  int32 attack; // 攻击力  bornlistvec\_t bornlist; // 合成列表<<<数组>>>  bornnumvec\_t bornnum; // 合成数量<<<数组>>>  };  // dota英雄表  struct dotaherocfg\_t{  typedef uint32 id\_t;  typedef std::string name\_t;  dotaherocfg\_t();  id\_t id; // 英雄ID<<<唯一>>>  name\_t name; // 英雄名称<<<唯一>>>  double strength; // 力量  double agile; // 敏捷  double intelligense; // 智力  }; |

可以看到，<合成列表>和<合成数量>均已被自动识别为整型数组bornlistvec\_t和bornnumvec\_t，程序载入的时候将自动分割“，”并存入数组。

并且，dotaequipcfgmgr类中还定义了几个map映射，表示可由对应的id和name找到装备或者英雄

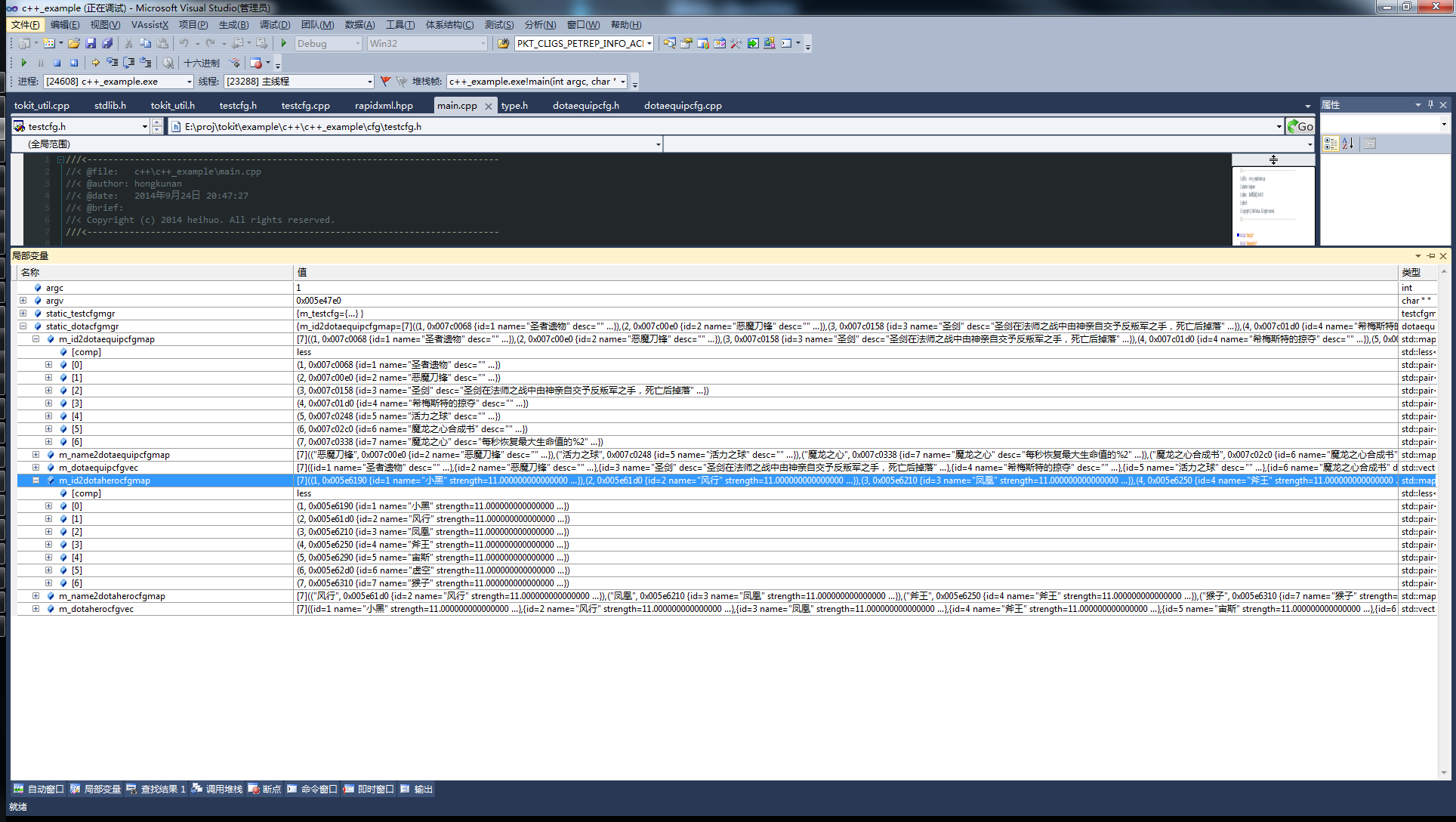
如下。

|  |
| --- |
| // dota装备表  typedef std::map<dotaequipcfg\_t::id\_t, dotaequipcfg\_t\*> id2dotaequipcfgmap; // 物品ID -> dotaequipcfg\_t  typedef std::map<dotaequipcfg\_t::name\_t, dotaequipcfg\_t\*> name2dotaequipcfgmap; // 物品名称 -> dotaequipcfg\_t  typedef std::vector<dotaequipcfg\_t> dotaequipcfgvec;  // dota英雄表  typedef std::map<dotaherocfg\_t::id\_t, dotaherocfg\_t\*> id2dotaherocfgmap; // 英雄ID -> dotaherocfg\_t  typedef std::map<dotaherocfg\_t::name\_t, dotaherocfg\_t\*> name2dotaherocfgmap; // 英雄名称 -> dotaherocfg\_t  typedef std::vector<dotaherocfg\_t> dotaherocfgvec; |

项目跟tokit生成的c++文件对接后，载入数据的工作很简单，添加一行语句：

|  |
| --- |
| dotaequipcfgmgr::instance().load(); |

可以在vs里面调试跟踪到，调用load()之后，xml数据已被载入到内存当中



同时，载入成功后可根据tokit自动生成的接口进行调用：

|  |
| --- |
| dotaequipcfg\_t \*dotaequipcfg = dotaequipcfgmgr::instance().get\_dotaequipcfg\_by\_id(1);  if (NULL == dotaequipcfg){  return;  }  dotaherocfg\_t \*dotaherocfg = dotaequipcfgmgr::instance().get\_dotaherocfg\_by\_name("小黑");  if (NULL == dotaherocfg){  return;  }  const dotaequipcfgmgr::id2dotaequipcfgmap &dotaequipcfgmap = dotaequipcfgmgr::instance().get\_id2dotaequipcfgmap(); |

更加详细的用法可参照example\c++\c++\_example项目或自己查看tokit生成的c++文件接口。

# 2. 主流技术简介

配置数据需要的是一整套流程，为了说明为什么会专门写这个库，有必要先介绍一下我所了解的一般通行的做法。

## 2.1 使用xml文件

xml作为项目配置数据的存储格式，具有强大的表达能力，也方便在xml文件内调整数据进行测试，优点主要集中在设计和开发阶段。基于xml的工具以及c++与xml结合的方式比较多样，所以关于xml文件的部分会重点说明。

### Excel转xml

一般，xml文件都是由excel导出而来，以方便策划编写数据，excel转xml的方式一般分以下几种

#### 1.1 excel中直接使用vba宏转成xml

我以<http://www.codeproject.com/Articles/6950/Export-Excel-to-XML-in-VBA>为例，这个页面讲解了如何在excel中导出xml文件并附带了源代码。

下载示例后可以看到，excel中的内容如下

|  |  |  |  |
| --- | --- | --- | --- |
| **/student/id** | **/student/name** | **/student/age** | **/student/mark** |
| 1 | Raymond | 11 | 0 |
| 2 | Moon | 11 | 100 |
| 3 | Billy | 11 | 100 |
| 4 | Pan | 12 | 80 |
| 5 | Queenie | 10 | 90 |

点击按钮后将生成一个xml文件，内容如下

|  |
| --- |
| <?xml version="1.0" ?>  <data>  <student>    <id>1</id>    <name>Raymond</name>    <age>11</age>    <mark>0</mark>    </student>  <student>    <id>2</id>    <name>Moon</name>    <age>11</age>    <mark>100</mark>    </student>  <student>    <id>3</id>    <name>Billy</name>    <age>11</age>    <mark>100</mark>    </student>  <student>    <id>4</id>    <name>Pan</name>    <age>12</age>    <mark>80</mark>    </student>  <student>    <id>5</id>    <name>Queenie</name>    <age>10</age>    <mark>90</mark>    </student>  </data> |

#### 1.2 excel中直接使用xml映射源导出xml

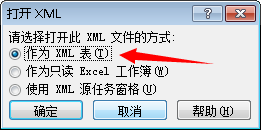
##### 1.2.1通过xml添加xml映射源

详见<http://jingyan.baidu.com/article/20b68a8851721e796cec6290.html>（Excel导出至XML格式）和<http://zzfei.com/excel-to-xml/>（从EXCEL导出数据到XML）。

以下面这个装备表<item.xml>为例，现在只有3条数据。

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8" standalone="yes" ?>  <equipcfgs>  <equipcfg>  <id>1</id>  <name>单手剑</name>  <lvl>1</lvl>  <price>100</price>  </equipcfg>  <equipcfg>  <id>2</id>  <name>单手剑2</name>  <lvl>2</lvl>  <price>200</price>  </equipcfg>  <equipcfg>  <id>3</id>  <name>单手剑3</name>  <lvl>3</lvl>  <price>300</price>  </equipcfg>  </equipcfgs> |

添加xml映射的方式有2种，一种是按照上面网址里的教程所说，另外一种相对来说更为方便，直接将xml拖动到excel中，在弹出的选择框中选择第一项：作为XML表，点击确定之后，excel将自动识别xml的格式并转换为excel的内容。这2种方式的效果是一样的。



将出现如下3行数据

|  |  |  |  |
| --- | --- | --- | --- |
| **id** | **name** | **lvl** | **price** |
| 1 | 单手剑 | 1 | 100 |
| 2 | 单手剑2 | 2 | 200 |
| 3 | 单手剑3 | 3 | 300 |

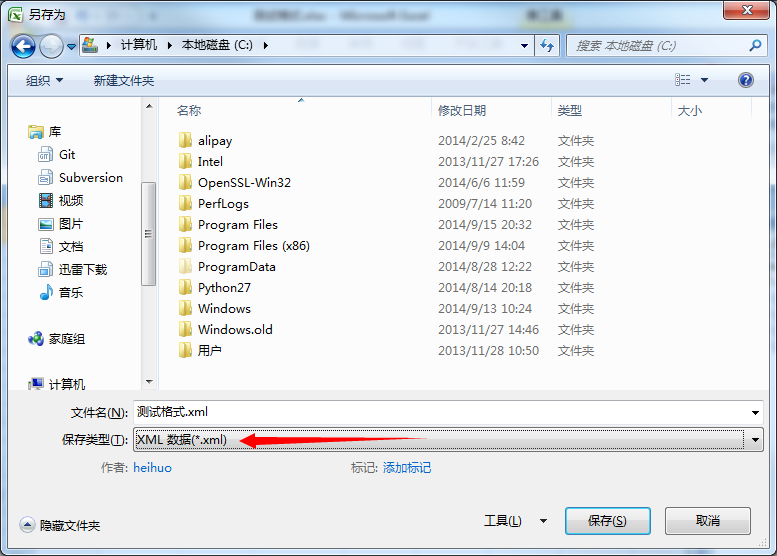
这时候可以查看xml映射框，可以看出excel已经识别出了xml的节点结构信息。如下图：



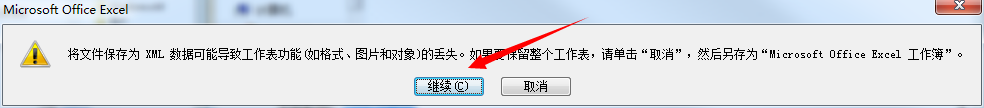
并在excel多添加2行数据

|  |  |  |  |
| --- | --- | --- | --- |
| 4 | 单手剑4 | 4 | 400 |
| 5 | 单手剑5 | 5 | 500 |

添加后点击开始 - 另存为，在出现的保存文件框中将保存类型改为xml数据类型



此时点击保存将弹出



最后将成功导出xml数据，此时xml数据由原先的3条变成了现在的5条。

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8" standalone="yes" ?>  <equipcfgs>  <equipcfg>  <id>1</id>  <name>单手剑</name>  <lvl>1</lvl>  <price>100</price>  </equipcfg>  <equipcfg>  <id>2</id>  <name>单手剑2</name>  <lvl>2</lvl>  <price>200</price>  </equipcfg>  <equipcfg>  <id>3</id>  <name>单手剑3</name>  <lvl>3</lvl>  <price>300</price>  </equipcfg>  <equipcfg>  <id>4</id>  <name>单手剑4</name>  <lvl>4</lvl>  <price>400</price>  </equipcfg>  <equipcfg>  <id>5</id>  <name>单手剑5</name>  <lvl>5</lvl>  <price>500</price>  </equipcfg>  </equipcfgs> |

##### 1.2.2通过xsd添加xml映射源

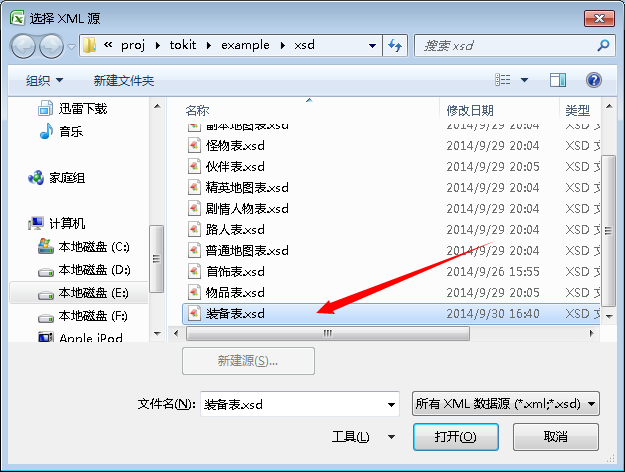
具体步骤与上面类似，点击xml映射按钮后，弹出添加xml映射框。



|  |
| --- |
| <?xml version="1.0" encoding="UTF-8" ?>  <xs:schema xmlns:xs="**http://www.w3.org/2001/XMLSchema**">  <xs:element name="**equipcfgs**">  <xs:complexType>  <xs:sequence minOccurs="**0**" maxOccurs="**unbounded**">  <xs:element name="**equipcfg**">  <xs:complexType>  <xs:sequence>  <xs:element name="**id**" type="**xs:unsignedInt**" />  <xs:element name="**name**" type="**xs:string**" />  <xs:element name="**lvl**" type="**xs:unsignedInt**" />  <xs:element name="**price**" type="**xs:unsignedInt**" />  </xs:sequence>  </xs:complexType>  </xs:element>  </xs:sequence>  </xs:complexType>  </xs:element>  </xs:schema> |

<装备表.xsd>

以上方的<装备表.xsd>为例，点击添加后，选中<装备表.xsd>然后点击打开。



成功打开后再点击确定，关闭xml映射框，将得到下面的图示结果。

****

拖动节点到excel中时，则有

|  |  |  |  |
| --- | --- | --- | --- |
| **id** | **name** | **lvl** | **price** |
|  |  |  |  |

编辑数据后再另存为xml将成功导出。

### c++载入xml

#### 2.1 硬编码载入xml

纯粹手动编写读取和解析代码，这种方式在配置表结构不常变动以及配置表量小的情况下是最方便的。但应付比较大一点的文件量则显得工作繁琐。

#### 2.2 xml数据绑定

关于xml数据绑定与正常的手动编写dom解析代码，两者差别可见下面的对比

|  |  |
| --- | --- |
| // 手动编写dom解析代码  DOMElement\* c = ...  DOMNodeList\* l;  l = c->getElementsByTagName ("name");  DOMNode\* name = l->item (0);  l = c->getElementsByTagName ("email");  DOMNode\* email = l->item (0);  l = c->getElementsByTagName ("phone");  DOMNode\* phone = l->item (0);  cout << name->getTextContent () << ", "  << email->getTextContent () << ", "  << phone->getTextContent () << endl; | // xml数据绑定  Contact c = ...  cout << c.name () << ", "  << c.email () << ", "  << c.phone () << endl; |

本表格直接取自codesynthesis官网页面<http://www.codesynthesis.com/products/xsd/>

xml与对象的绑定关系可见下表：

|  |  |
| --- | --- |
| <contact>  <name>John Doe</name>  <email>j@doe.com</email>  <phone>555 12345</phone>  </contact> | auto\_ptr<Contact> c = contact ("c.xml");  cout << c->name () << ", "  << c->email () << ", "  << c->phone () << endl; |

本表格直接取自codesynthesis官网页面<http://www.codesynthesis.com/products/xsd/>

具体原理可参见<http://www.cnblogs.com/mywolrd/archive/2012/04/24/2468908.html>，里面的《更好的解决方案》一小栏大概讲解了xml与c++对象绑定的过程。

支持xml数据对象绑定的c++库有CodeSynthesis XSD（跨平台），xdstoolkit，以及gsoap等。

CodeSynthesis XSD见<http://www.codesynthesis.com/products/xsd/>

xdstoolkit来源于《游戏编程精粹4》中的《使用XMl而不牺牲速度》，作者Mark T. Price，参考资料可见<http://blog.csdn.net/mikefeng/article/details/1327330>

gsoap则参见<http://www.cnblogs.com/diylab/archive/2009/02/13/1390287.html>

由于要对c++和xml进行绑定，所以需要提供结构的定义信息，一般是要求xsd文件，比如CodeSynthesis XSD根据xsd文件，可以生成对应的c++文件，而有的则根据使用场景自行定义，如xdstoolkit要求提供的是c++结构体文件，通过对文件进行语法解析，再生成对应的xsd文件 。

以codesynthesis xsd中的范例<hello.xsd>为例，

|  |
| --- |
| <?xml version="1.0" ?>  <xsd:schema xmlns:xsd="**http://www.w3.org/2001/XMLSchema**">  <xsd:complexType name="**hello\_t**">  <xsd:sequence>  <xsd:element name="**greeting**" type="**xsd:string**" />  <xsd:element name="**name**" type="**xsd:string**" maxOccurs="**unbounded**" />  </xsd:sequence>  </xsd:complexType>  <xsd:element name="**hello**" type="**hello\_t**" />  </xsd:schema> |

<hello.xsd>

通过codesynthesis xsd的工具将生成c++文件<hello.hxx>和<hello.cxx>

|  |
| --- |
| class hello\_t: public ::xml\_schema::type  {  public:  // greeting  //  typedef ::xml\_schema::string greeting\_type;  typedef ::xsd::cxx::tree::traits< greeting\_type, char > greeting\_traits;  const greeting\_type&  greeting () const;  greeting\_type&  greeting ();  void  greeting (const greeting\_type& x);  void  greeting (::std::auto\_ptr< greeting\_type > p);  // name  //  typedef ::xml\_schema::string name\_type;  typedef ::xsd::cxx::tree::sequence< name\_type > name\_sequence;  typedef name\_sequence::iterator name\_iterator;  typedef name\_sequence::const\_iterator name\_const\_iterator;  typedef ::xsd::cxx::tree::traits< name\_type, char > name\_traits;  const name\_sequence&  name () const;  name\_sequence&  name ();  void  name (const name\_sequence& s);  // Constructors.  //  hello\_t (const greeting\_type&);  hello\_t (const ::xercesc::DOMElement& e,  ::xml\_schema::flags f = 0,  ::xml\_schema::container\* c = 0);  hello\_t (const hello\_t& x,  ::xml\_schema::flags f = 0,  ::xml\_schema::container\* c = 0);  virtual hello\_t\*  \_clone (::xml\_schema::flags f = 0,  ::xml\_schema::container\* c = 0) const;  hello\_t&  operator= (const hello\_t& x);  virtual  ~hello\_t ();  // Implementation.  //  protected:  void  parse (::xsd::cxx::xml::dom::parser< char >&,  ::xml\_schema::flags);  protected:  ::xsd::cxx::tree::one< greeting\_type > greeting\_;  name\_sequence name\_;  }; |

<hello.hxx>中的代码片段

与之对应的xml格式是

|  |
| --- |
| <?xml version="1.0" ?>  <!--  file : examples/cxx/tree/hello/hello.xml  copyright : not copyrighted - public domain  -->  <hello xmlns:xsi="**http://www.w3.org/2001/XMLSchema-instance**" xsi:noNamespaceSchemaLocation="**hello.xsd**">  <greeting>Hello</greeting>  <name>sun</name>  <name>moon</name>  <name>world</name>  </hello> |

而在xdstoolkit中，则是根据c++头文件生成对应的xsd文件，以下面的<itemprop.h>为例。

|  |
| --- |
| #ifndef \_ItemProp\_h\_  #define \_ItemProp\_h\_  #include <map>  struct ItemProp  {  unsigned char Id;  char Name[128];  unsigned char Quality;  unsigned char Price;  };  extern struct ItemProp \*g\_ItemProps;  #endif // \_ItemProp\_h\_ |

<itemprop.h>文件

与之对应的xml数据格式如下，每个ItemProp结构体将对应一个entry的xml节点

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8" ?>  <ItemProps xmlns:xsi="**http://www.w3.org/2001/XMLSchema-instance**">  <ItemPro>  <g\_ItemProps>  <entry>  <Id>0</Id>  <Name>物品0</Name>  <Quality>0</Quality>  <Price>0</Price>  </entry>  <entry>  <Id>255</Id>  <Name>物品255</Name>  <Quality>255</Quality>  <Price>255</Price>  </entry>  <entry>  <Id>1</Id>  <Name>物品1</Name>  <Quality>1</Quality>  <Price>1</Price>  </entry>  <entry>  <Id>254</Id>  <Name>物品254</Name>  <Quality>254</Quality>  <Price>254</Price>  </entry>  </g\_ItemProps>  </ItemPro>  </ItemProps> |

#### 2.3 加速载入xml

关于加速载入xml的思路，如果是采用对象序列化的技术，可以预先将xml读出并存入对应的对象，序列化成二进制串存入文件中，在读取的时候再进行反序列化操作即可，由于省去了xml的读取、解析和转换过程，所以反序列化式读取的耗时一般低于纯文本xml的载入。比如CodeSynthesis XSD就支持序列化。

另外一种思路是自行转换成相应的格式，也是xdstoolkit所做的，先将xml数据转成自定义格式xds，直接用xds文件作为已发布版本的配置数据。如果有跟踪过xdstoolkit的源码，可以发现，经过转换成xds，载入过程基本等价于将xds文件内容直接映射到内存中，速度是非常快的。以一般的游戏配置数据而言，读取时间一般在0.001毫秒级别，比序列化式加速要快上至少2个数量级。

## 2.2. excel转csv文件

csv格式的优势十分明显，相较于xml，更易于从excel转换而来，且格式清晰简洁无冗余，载入速度更快，缺点是当csv文件内行数较多时，不易直接查找到某个值所对应的字段。

导出csv时在excel中的数据区域外最好不要有多余的字符，否则会导致误导出非数据区的文本内容

可以按如下方式使用csv。

|  |  |  |  |
| --- | --- | --- | --- |
| **装备id** | **装备名称** | **装备等级** | **价格（铜钱）** |
| **id** | **name** | **lvl** | **price** |
| 1 | 单手剑 | 1 | 100 |
| 2 | 单手剑2 | 2 | 200 |
| 3 | 单手剑3 | 3 | 300 |
| 4 | 单手剑4 | 4 | 400 |
| 5 | 单手剑5 | 5 | 500 |

excel另存为csv后，导出结果如**下**

|  |
| --- |
| **装备id,装备名称,装备等级,价格（铜钱）**  id,name,lvl,price  1,单手剑,1,100  2,单手剑2,2,200  3,单手剑3,3,300  4,单手剑4,4,400  5,单手剑5,5,500 |

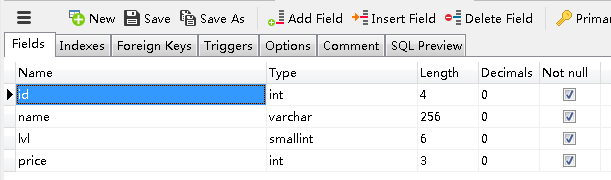
采用csv格式对策划来说最为友好，但对程序方面的对接就不利了，因为程序方面必须确保字段的加载顺序和csv的字段顺序严格一致，否则数据量大了之后易出疏漏，在这方面最好有自动机制加以保证。

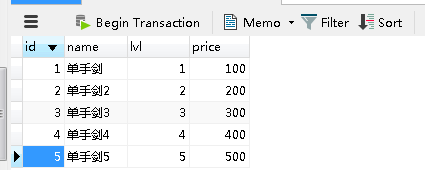
## 2.3. 直接存在数据库中

直接将配置存在数据库中，缺点十分明显，由于策划一般是先在excel中编辑数据再导出，所以必须有一个从excel导出数据到数据库的过程，这个过程一般需要借助数据库gui工具的使用或者带格式数据的导入导出功能，比如，我现在（2014年10月）的项目采用mysql数据库，使用的navicat premium图形化数据库工具就支持在excel表中直接鼠标选中区域复制再手动粘贴到数据库（经亲身体验，非常低效），不管哪种方式，都必须先在数据库中建立起和策划制作的配置数据字段严格一致的表。此外，一个弊端是，无法校验数据的正确性。就我所知的mysql GUI工具，似乎没有哪款产品能有效地监测数据变动的，经常容易出现误改数据又发现不了的情况。存在数据库中的配置数据在发布前如果未经过diff工具的检测，仅靠人工操作，那么数据的正确性是很值得怀疑的。相应的解决办法是diff旧配置数据的sql文件。

但数据存储在数据库中有一个巨大的好处是配置统一存放，当有多台服务器共用同一套配置时，更新配置会显得非常方面。

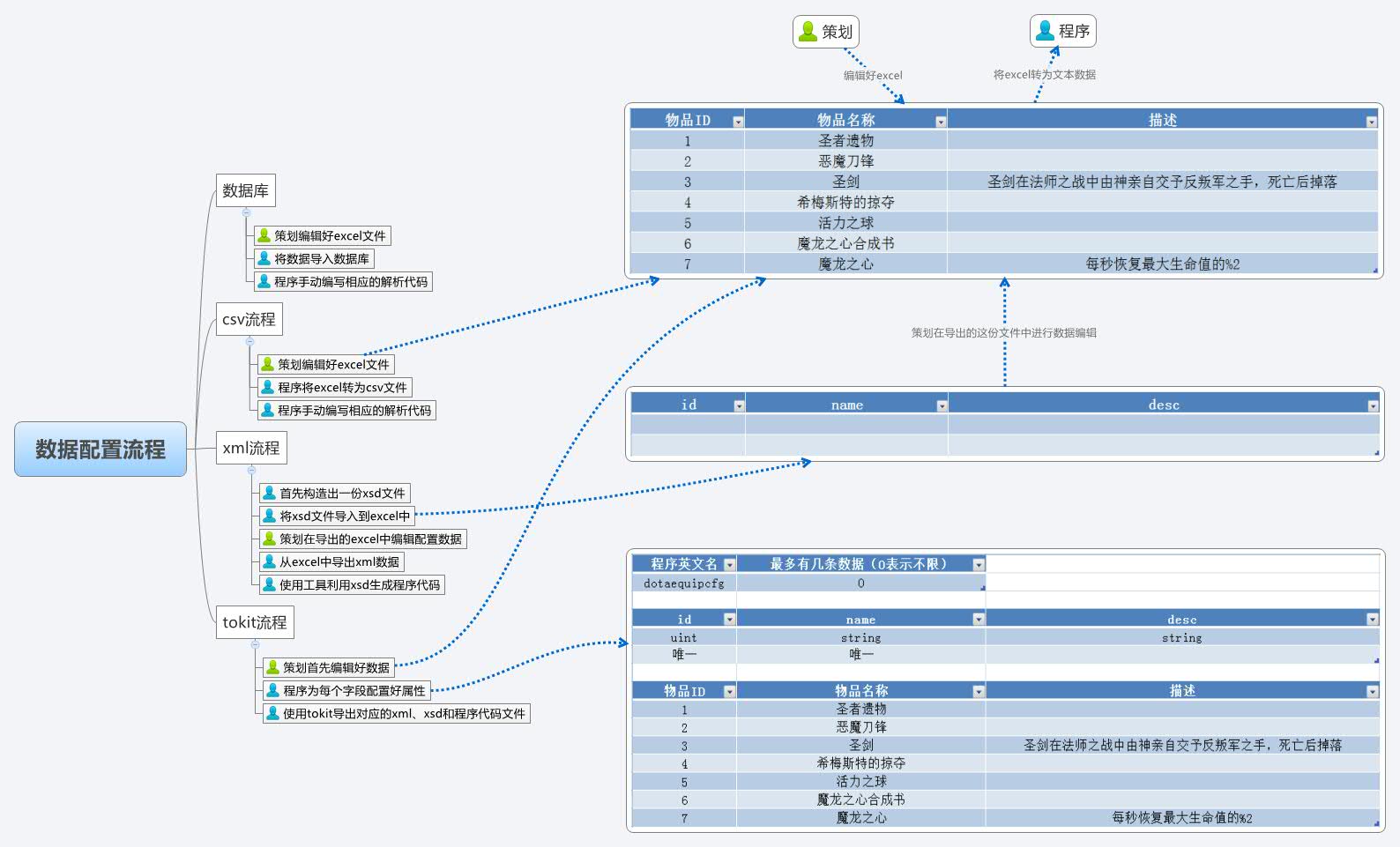
比如

****

****

## 2.4 流程小结

上述的方式各有优劣，但由于大部分的策划都是用excel配置数据，其实都绕不过excel。具体粗略的分别可由下图来表示。



# 3. 关于Tokit库的一些补充

## 3.1 问：一般的做法不是已经很方便了吗，为什么还要有Tokit

因为还不够方便。

以我所了解的技术而言：

vba-->可以导出合适的数据格式，但用于自动生成代码则不是那么容易了。

xml数据对象绑定-->只做到了绑定，类代码自动创建，但接口还是要程序员自行编写。

数据库、csv-->不仅要自行实现类，解析代码也要手写

而Tokit希望做到的是：尽量少写代码

## 3.2问：如果希望对Tokit生成的文件进行调整，该怎么做？

tokit在生成c++代码时，要求提供2个模板文件，这2个文件放在example\tool\template下，名为<c++\_template.h>和<c++\_template.cpp>。

而这2个模板文件就是为了方便调整所生成的代码的。

所以，如果希望稍微变动生成的c++文件，可以适当的改动2个文件的内容。

注意到，文件中有大量的%xxx%文本，tokit将自动把对应的文本替换成实际的内容。

其中%cfg%可能会是最经常用到的，含义是当前文件的英文名。

比如，现有一个装备表，名为item，则tokit将自动把模板文件中的%cfg%替换为item。

如果需要更多的定制，就需要改动tokit的源码了。

## 3.3问： Tokit生成的c++代码载入xml的速度快吗？

是的，很快。

这要归功于tokit使用了rapidxml开源库作为xml解析器。

rapidxml设计的初衷是高速读取xml文件，并且也真的做到了，据说比tinyxml快30倍起（我知道很多人都在用tinyxml）。

具体数据可查看rapidxml的官方手册<http://rapidxml.sourceforge.net/manual.html>中的4. Performance小节。

下面表格就是取自4. Performance小节：

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **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 |

*(\*) All results are in CPU cycles per character of source text（中文翻译：表格中的数值表示源文件中每个字符耗费的cpu周期）*

从表格中可以看到，rapidxml : strlen() < 10 : 1。表明rapidxml解析一个xml文件的时间小于strlen一个xml文件时间的10倍。

这是非常快的。