|  |
| --- |
| 密级：受控 |
|  |
|  |



**YANG Tools使用指南**

|  |  |
| --- | --- |
| **版本** | V0.1 |

修订记录

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 日期 | 修订版本 | 修改者 | 操作 | 修改描述 |
| 2013/10/25 | V0.1 | 黄东远 | 创建 | 创建并编写文档 |

目录

[1 YANG Tools 4](#_Toc386961461)

[1.1 YANG Schema和YANG Model 4](#_Toc386961462)

[1.1.1 YANG schema 4](#_Toc386961463)

[1.1.2 支持控制器的YANG修改和扩展 4](#_Toc386961464)

[1.1.3 YANG Schema的Java模型 5](#_Toc386961465)

[1.1.4 未知Schema节点和扩展处理 6](#_Toc386961466)

[1.2 YANG Data(绑定独立数据) 8](#_Toc386961467)

[1.2.1 节点 8](#_Toc386961468)

[1.2.2 节点变更 9](#_Toc386961469)

[1.3 YANG to Java mapping 10](#_Toc386961470)

[1.3.1 包名 10](#_Toc386961471)

[1.3.2 类和接口名称 12](#_Toc386961472)

[1.3.3 Getters and setters name 12](#_Toc386961473)

[1.3.4 Module 13](#_Toc386961474)

[1.3.5 Typedef 14](#_Toc386961475)

[1.3.6 Container 20](#_Toc386961476)

[1.3.7 leaf 20](#_Toc386961477)

[1.3.8 Leaf-list 21](#_Toc386961478)

[1.3.9 List 21](#_Toc386961479)

[1.3.10 Choice and case 24](#_Toc386961480)

[1.3.11 Grouping and uses 26](#_Toc386961481)

[1.3.12 Rpc 26](#_Toc386961482)

[1.3.13 Notification 27](#_Toc386961483)

[1.3.14 Augment 28](#_Toc386961484)

[1.3.15 Identity 28](#_Toc386961485)

[1.4 YANG Maven Tools使用指南 29](#_Toc386961486)

[1.4.1 先决条件 29](#_Toc386961487)

[1.4.2 工程配置 29](#_Toc386961488)

[1.4.3 配置yang-maven-plugin 30](#_Toc386961489)

[1.4.4 配置依赖 31](#_Toc386961490)

[1.4.5 构建 32](#_Toc386961491)

[1.4.6 Eclipse集成(m2e) 32](#_Toc386961492)

[1.4.7 设计和编码指南 34](#_Toc386961493)

[1.5 代码生成器DEMO 35](#_Toc386961494)

# YANG Tools

YANG Tools是OpenDaylight的一个基础项目，它的目标是为NETCONFIG和YANG的JAVA工程和应用提供工具和类库支持，例如MD-SAL、Netconf/OFConfig plugins，它被定位于与Opendaylight控制器无关的，可以与被其它工程和人员重复使用。

它有以下几个子项目：

（1）Concepts：定义了项目无关的基础概念和帮助类，可以在YANG Tools项目外使用。

（2）YANG：与YANG相关的工程

----yang-data：绑定独立的DOM节点API；

----yang-model：YANG Schema的Java表示；

----yang-parser：YANG文件解析器，将YANG文件解析成Java YANG Model；

----yang-binding：YANG绑定基础API

----yang-maven-plugin：maven插件，可在其它工程中配置，将YANG文件生成Java绑定(DTOs、RPCs、Notifications等)。

（3）Code Generator：绑定生成器，用于将YANG Model生成Java绑定(DTOs、RPCs、Notifications等)

（4）RestConf：用于生成Rest API。

## YANG Schema和YANG Model

YANG Schema是描述模块中的数据、过程、消息通知的结构。

### YANG schema

Modules – 一组暴露的功能，特征、类型、RPCs和数据结构

Features – 一种将Schema中一部分标记成条件化的机制

Types – 数据类型，在本模块内，或者导入该模块的其它模块内使用

Data Structures – 定义数据树的数据节点

RPCs – 模块操作

Notifications – 消息通知

Relationships between data – 数据之间关系

Validation constraints – 校验约束

Extensions of other modules and their functionality (augmentation) – 扩展其它模块和功能

YANG Extensions – YANG扩展

### 支持控制器的YANG修改和扩展

**处理多模块版本**

在文档引入一个隔离层，基于模块版本来隔离数据、RPCs和消息，从而达到支持多版本的客户端，避免数据在不同版本间被偶然串改的的风险。

由于多个不同版本的模块会同时存在于系统中，框架必须支持组件在各个不同版本间转化数据。

**QNames**

一个正常的XML QName由本地元素名称、XML命名空间和模块版本组成。

在YANG上下文中，QName是定义的节点、类型、过程或消息的全名。它由XML命名空间，YANG模型版本和本地名称组成。防止不同schema之间的本地节点的名称冲突。

QName = (XMLNamespace,Revision,LocalName)

其中，

XMLNamespace – YANG模块指定的命名空间。

Revision –YANG module修订时间

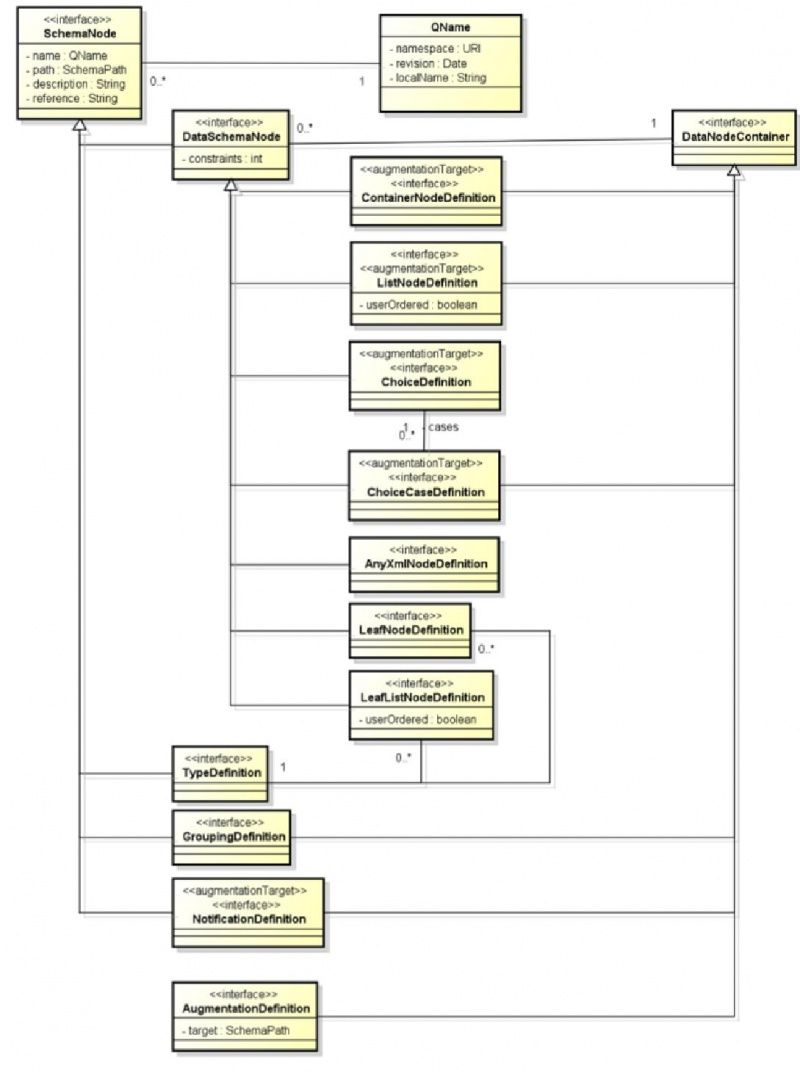
LocalName – YANG module所定义节点的标识符

**RPCs**

在正常的NETCONF/YANG使用中，RPCs用来对NETCONF服务器和客户端进行功能与API建模。而在SAL中，RPCs用来对提供者提供的和消费者消费的功能进行建模。

### YANG Schema的Java模型

下图展示了YANG Schema Java模型。



YANG Parser，用来产生Java模型对象。

### 未知Schema节点和扩展处理

YANG规范允许出现未在规范出现的schema节点。通常这样的节点定义在模块的一个或多个的扩展语句中，这些没有进一步语义的节点是按计算机可以理解的格式定义的。为了处理这样的节点，新增未知的节点类型。一个未知的节点类型可以是任意schema节点的儿子。通常用来存储扩展信息，提供给能理解其语义的组件。

#### 扩展节点类继承

**ExtensionDefinition** – YANG extension的schema节点。它的QName基于YANG语句。

例如yang-ext.yang中定义如下片段，即表示一个ExtensionDefinition

extension "context-reference" {

argument "context-type";

}

UnknownSchemaNode– 使用YANG扩展定义的语句，这个schema节点有两个QName：

name – 节点所在YANG module的名称

nodeTypeQName –ExtensionDefinition的Qname

例如下面这个例子，

grouping node-flow {

leaf node {

ext:context-reference "inv:node-context";

type inv:node-ref;

}

leaf flow-table {

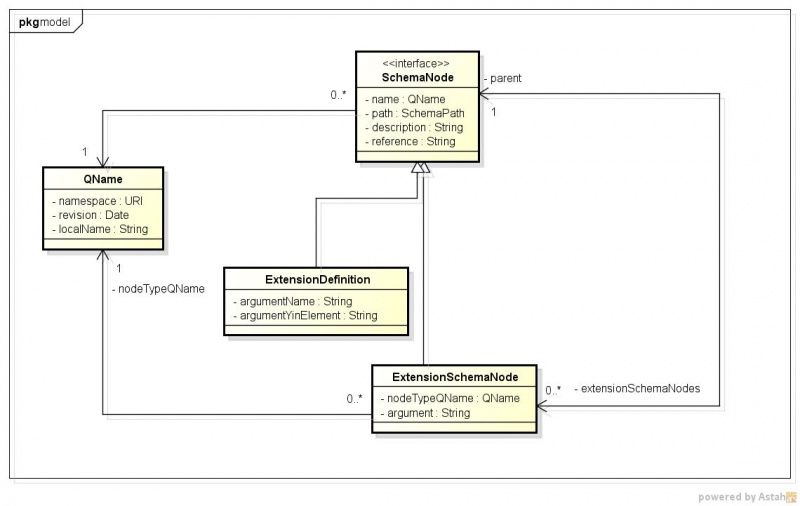
type flow-table-ref;

}

uses types:flow;

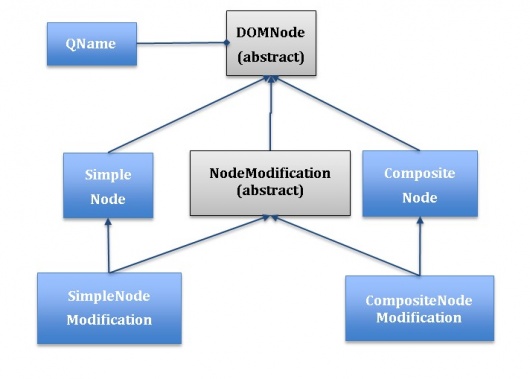
}

ext:context-reference如上例定义，则ext:context-reference "inv:node-context"语句表示一个UnknownSchemaNode节点。



## YANG Data(绑定独立数据)

在SAL控制器上下文中，数据映射成一棵类似于XML的树，并作一些修改以支持多版本的数据模型和控制器组件。最大的区别就是使用一个全新的QName定义，该定义允许有不同的树分枝，这些分枝基于数据schema版本。下图表示节点分类及关系。



**QName** – 节点类型标识符，类似于XML的QName

**DOMNode** – 表示数据树中的节点，它定义了基本的节点参数，例如QName。

有以下两类DOMNode：:

**SimpleNode** – 简单节点，表示一个节点只包含一个简单值(e.g.整数，字符串等)的节点

**CompositeNode** – 复合节点，表示一个节点包含一个或多个其它节点，但不包含简单值。.

**NodeModification** – 表示对一个节点的变更，也有两类NodeModification：

**SimpleNodeModification** – 表示对简单节点的变更，例如替换或者删除一个节点的值。

**CompositeNodeModification** – 表示对复合节点的变更，例如替换复合节点的内容。

没有使用YANG对binding-independent数据格式建模，是因为YANG anyxml语句的存在(比数据DOM树需要更多的方法)以及对它的语句分析，阻止我们使用直接的YANG对数据节点类型进行映射。

### 节点

每个节点都必须有一个QName，一个Container节点可以存在多个相同QName的节点。一共有两种节点，简单节点和复合节点。

#### 简单节点

简单节点是不包含其它节点包含值的叶子节点。在XML中，简单节点就是只包含文本数据(CDATA或PCDATA)的元素。简单节点由下列YANG数据样式组成：

leaf — 简单节点可以表示各种类型的YANG leaf，除了空类型(类似于XML中的空元素)

item of a leaf-list — leaf-list中的项

#### 复合节点

复合节点表示数据树上的一个分枝，它包含嵌套的复合节点或叶子节点。它相当于XML中不直接包含文本数据(CDATA、PCDATA)的元素。它可以由下列YANG数据样式表示。

container：复合节点可以表示YANG container，包括该container下的所有子样式节点。

item in the list：复合节点可以表示YANG list中的一项，包括该项中的所有子样式节点。

anyxml：任意XML

leaf with empty type：空叶子

### 节点变更

在数据变化的情况下，节点变更作为附加的元数据，附加到数据树。修改类型基于NETCONF的修改配置的RPC调用。为了修改配置或状态数据，必须要创建一棵表示数据变更的树，并将变更应用到目标树。

#### 简单节点变更

简单节点变更有以下几种：

replace — 使用节点变更标识的数据替换目标数据树上所有相关的数据，如果目标树数据不存在则创建它。

create — 使用节点变更标识的数据，当且仅当目标数据树不存在相应数据时，新增数据到目标树。如果数据已存在，返回”data-exists”的错误。

delete — 从目标数据树上删除节点变更标识的数据，当且仅当数据存在时，执行删除；否则，返回”data-missing”的错误。

remove — 从目标数据树上删除节点变更标识的节点属性，当对应属性存在时，执行删除；否则，忽视修改。

#### 复合节点变更

复合节点变更有以下几种：

merge — 将节点变更标识的数据合并到数据树上的对应级别中。

replace — 使用节点变更标识的数据替换目标数据树上所有相关的数据，如果目标树数据不存在则创建它

create — 使用节点变更标识的数据，当且仅当目标数据树不存在相应数据时，新增数据到目标树。如果数据已存在，返回”data-exists”的错误。

delete — 从目标数据树上删除节点变更标识的数据，当且仅当数据存在时，执行删除；否则，返回”data-missing”的错误。

remove — 从目标数据树上删除节点变更标识的节点属性，当对应属性存在时，执行删除；否则，忽视修改。

## YANG to Java mapping

在本节中，将介绍YANG schema到JAVA的映射规则。

### 包名

包名由以下几部分组成：

（1）opendaylight前缀：每一个包名开始于：**org.opendaylight.yang.gen.v**，它硬编码在**BindingGeneratorUtil.moduleNamespaceToPackageName()**

（2）YANG version

YANG version 由**module**的**yang-version**子句中的参数指定

（3）namespace

等于**module** 的**namespace**子句的参数，将参数中的字符**： / : - @ $ # ' \* + , ; =** 替换成点号”.”。

（4）revision

revision由缩写rev，跟上**module** 的**revision**子句的参数，再去掉字符”-“和年份的前两位数字零(e.g. rev130709)。

在包名生成后，检查一下是否包含JAVA关键字或数字。如果有的话，在其前面加上字符”\_”。出现下面列出的关键字，需要在前面加上”\_”。

abstract, assert, boolean, break, byte, case, catch, char, class, const, continue, default, double, do, else, enum, extends, false, final, finally, float, for, goto, if, implements, import, instanceof, int, interface, long, native, new, null, package, private, protected, public, return, short, static, strictfp, super, switch, synchronized, this, throw, throws, transient, true, try, void, volatile, while

如下所示的YNAG model：

module module {

namespace "urn:2:case#module"

prefix "sbd";

organization "OPEN DAYLIGHT";

contact "http://www.whatever.com/";

revision 2013-07-09 {

}

}

对应的包名将是org.opendaylight.yang.gen.v1.urn:2:case#module.rev130709，然后替换数字和JAVA关键字：

org.opendaylight.yang.gen.v1.urn.\_2.\_case.module.rev130709

#### 其它包名

在一些情况下，需要生成一些额外的包名。这种情况发生在那些上级节点上，这些上级节点包含下级YANG元素。如下表所示：

|  |  |
| --- | --- |
| 上级元素 | 下级元素 |
| list | list, container, choice |
| container | list, container, choice |
| choice | leaf, list, leaf-list, container, case |
| case | list, container, choice |
| rpc.input, rpc.output | list, container, (不支持choice) |
| notification | list, container, (不支持choice) |
| augment | list, container, choice, case |

在这些情况下，下级元素将不只映射成上级元素接口的JAVA getter方法，而且生成由上级元素的包名加上上级元素名称组成的包名。

在下面的YANG model的例子中，假设container元素conf是module的直接子元素。

container **cont** {

container **cont-inner** {

   }

   list **outter-list** {

list list-in-list {

}

   }

}

Container元素***conf***是下级子元素***conf-inner***和***outter-list***的上级元素。JAVA代码将生成如下结构：

----org.opendaylight.yang.gen.v1.urn.module.rev130709：package包含module的所有下级元素

--------Cont.java

----org.opendaylight.yang.gen.v1.urn.module.rev130709.cont：package包含conf的下级元素。

--------ContInner.java

--------OutterList.java

再看另外一个YANG model：

container cont {

   container cont-inner {

}

list **outter-list** {

list **list-in-list** {

   }

  }

}

在这个例子中，list元素***outter-list***是下级元素**list-in-list**的上级元素。JAVA代码将生成如下结构：

----org.opendaylight.yang.gen.v1.urn.module.rev130709.**cont.outter.list**：*package*包含outter-list的下级元素

----ListInList.java

值得注意的是，上面的package \*.**cont**.**outter-list**最终替换成了\*.**cont**.**outter.list**，因为依据包名规范”-“要被替换成”.”。

### 类和接口名称

一些YANG元素会被映射成JAVA类和接口。YANG元素的名称可能包含一些JAVA命名规范不允许的字符。

（1）去除元素名称前后的空白字符；

（2）删除空格，减号-，下划线\_等字符，并将接下来的字符转化成大写字母；

（3）首字符大写。

例如，

example-name without\_capitalization

被映射成：

ExampleNameWithoutCapitalization

### Getters and setters name

**在某些情况下，YANG元素会生成getter或setter方法。这些方法由类**

**MethodSignatureBuilder创建。**

**getter处理如下：**

**（1）YANG元素的名称转化成JAVA类名称格式；**

**（2）加上get前缀；**

**（3）返回类型为YANG元素的type子句规定的类型。**

**setter处理如下：**

**（1）YANG元素的名称转化成JAVA类名称格式；**

**（2）加上set前缀；**

**（3）input的参数名称设置成元素名称，并转化成JAVA参数格式；**

**（4）返回类型为void。**

### Module

YANG module会映射成三个JAVA类(不一定有三个，可能一个都没有，取否决是有数据节点、RPC节点和Notification节点)。三个类分别在不同的的JAVA文件中。JAVA文件的名称由以下部分组成：

<YANG\_module\_name><Sufix>.java

其中，<sufix>可以是**Data**或**Service**或Listener

|  |  |
| --- | --- |
| YANG | JAVA |
| module module {    namespace "urn:module";  prefix "sbd";    organization "OPEN DAYLIGHT";  contact "http://www.whatever.com/";    revision 2013-07-09 {  }  } | ModuleData.java  **package** org.opendaylight.yang.gen.v1.urn.module.rev130709;  **public** **interface** ModuleData {  } |
| ModuleService.java  **package** org.opendaylight.yang.gen.v1.urn.module.rev130709;  **public** **interface** ModuleService {  }  ModuleListener.java  **package** org.opendaylight.yang.gen.v1.urn.module.rev130709;  **public** **interface** ModuleListener {  } |

#### Data Interface

数据接口的映射类似于container，但只包含module内的顶级节点

#### Service Interface

服务接口用来描述module内定义的RPC合约。RPC合约使用rpc语句定义。

#### Listener Interface

Listener接口用来描述module内定义的Notification。例如YANG模块module定义一个notification如下：

notification notif {

}

它将生成如下监听接口：

public interface ModuleListener extends NotificationListener {

void onNotif(Notif notification);

}

### Typedef

YANG的typedef语句映射成JAVA类。typedef一般包含如下子句：

|  |  |
| --- | --- |
| **子句** | **如何映射成JAVA** |
| type | 映射成类属性 |
| descripton | 不映射 |
| units | 不映射 |
| default | 不映射 |

#### 合法的参数类型

一些简单的参数类型的映射如下：

|  |  |
| --- | --- |
| 参数类型 | 映射成JAVA |
| boolean | Boolean |
| empty | Boolean |
| int8 | Byte |
| int16 | Short |
| int32 | Integer |
| int64 | Long |
| string | String or Class(如果指定了pattern子句) |
| decimal64 | Double |
| uint8 | Short |
| uint16 | Integer |
| uint32 | Long |
| int64 | BigInteger |
| binary | byte[] |

复杂参数类型映射如下：

|  |  |
| --- | --- |
| 参数类型 | 映射成JAVA |
| enumeration | enum |
| bits | class |
| leafref | ?? |
| identityref | ?? |
| union | class |
| instance-identifier | class |

##### Enumeration的enum子句

YANG的枚举类型包含一些enum子句。Enumeration映射成JAVA enum类型，每一个YANG的enu子句映射成JAVA enum预定义值。Enum子句有如下子句：

|  |  |
| --- | --- |
| Enum's substatement | mapping to JAVA |
| description | 不映射 |
| value | 映射成JAVA enum预定义值的参数 |

一个YANG枚举类型映射成JAVA的例子如下：

|  |  |
| --- | --- |
| YANG | JAVA |
| typedef typedef-enumeration {  type enumeration {  **enum** enum1 {  description "enum1 description";  value 18;  }  **enum** enum2 {  value 16;  }  **enum** enum3 {  }  }  } | **public** **enum** TypedefEnumeration {  Enum1(18),  Enum2(16),  Enum3(19);  **int** value;    **private** TypedefEnumeration(**int** value) {  **this**.value = value;  }  } |

##### bits的bit子句

YANG的bits类型包含一些bit子句。YANG bits映射成JAVA类，每一个bit子句映射成JAVA类的boolean型属性，并且附加hash、toString、equals等重载方法的实现。

|  |  |  |
| --- | --- | --- |
| YANG | JAVA | JAVA overriden Object methods |
| typedef typedef-bits {  type bits {  bit first-bit {  description "first-bit description";  position 15;  }  bit second-bit;  }  } | **public** **class** TypedefBits {    **private** Boolean firstBit;  **private** Boolean secondBit;    **public** TypedefBits() {  **super**();  }    **public** Boolean getFirstBit() {  **return** firstBit;  }    **public** **void** setFirstBit(Boolean firstBit) {  **this**.firstBit = firstBit;  }    **public** Boolean getSecondBit() {  **return** secondBit;  }    **public** **void** setSecondBit(Boolean secondBit) {  **this**.secondBit = secondBit;  }  } | @Override  **public** **int** hashCode() {  **final** **int** prime = 31;  **int** result = 1;  result = prime \* result +  ((firstBit == **null**) ? 0 : firstBit.hashCode());  result = prime \* result +  ((secondBit == **null**) ? 0 : secondBit.hashCode());  **return** result;  }    @Override  **public** **boolean** equals(Object obj) {  **if** (**this** == obj) {  **return** **true**;  }  **if** (obj == **null**) {  **return** **false**;  }  **if** (getClass() != obj.getClass()) {  **return** **false**;  }  TypedefBits other = (TypedefBits) obj;  **if** (firstBit == **null**) {  **if** (other.firstBit != **null**) {  **return** **false**;  }  } **else** **if**(!firstBit.equals(other.firstBit)) {  **return** **false**;  }  **if** (secondBit == **null**) {  **if** (other.secondBit != **null**) {  **return** **false**;  }  } **else** **if**(!secondBit.equals(other.secondBit)) {  **return** **false**;  }  **return** **true**;  }    @Override  **public** String toString() {  StringBuilder builder = **new** StringBuilder();  builder.append("TypedefBits [firstBit=");  builder.append(firstBit);  builder.append(", secondBit=");  builder.append(secondBit);  builder.append("]");  **return** builder.toString();  } |

##### Union的type子句

如果typedef中的type为union，它包含type子句。Union typedef映射成JAVA类，它的type子句映射成私有类属性。对每个YANG union的子类型，它从一个只带一个类属性参数的JAVA构造器生成。

|  |  |  |
| --- | --- | --- |
| YANG | JAVA | JAVA overriden Object methods |
| typedef typedef-union {  type union {  type int32;  type string;  }  } | **public** **class** TypedefUnion {  **private** Integer int32;  **private** String string;  **public** TypedefUnion(Integer int32) {  **super**();  **this**.int32 = int32;  }    **public** TypedefUnion(String string) {  **super**();  **this**.string = string;  }    **public** Integer getInt32() {  **return** int32;  }    **public** String getString() {  **return** string;  }  } | @Override  **public** **int** hashCode() {  **final** **int** prime = 31;  **int** result = 1;  result = prime \* result + ((int32 == **null**) ? 0 : int32.hashCode());  result = prime \* result + ((string == **null**) ? 0 : string.hashCode());  **return** result;  }    @Override  **public** **boolean** equals(Object obj) {  **if** (**this** == obj) {  **return** **true**;  }  **if** (obj == **null**) {  **return** **false**;  }  **if** (getClass() != obj.getClass()) {  **return** **false**;  }  TypedefUnion other = (TypedefUnion) obj;  **if** (int32 == **null**) {  **if** (other.int32 != **null**) {  **return** **false**;  }  } **else** **if**(!int32.equals(other.int32)) {  **return** **false**;  }  **if** (string == **null**) {  **if** (other.string != **null**) {  **return** **false**;  }  } **else** **if**(!string.equals(other.string)) {  **return** **false**;  }  **return** **true**;  }    @Override  **public** String toString() {  StringBuilder builder = **new** StringBuilder();  builder.append("TypedefUnion [int32=");  builder.append(int32);  builder.append(", string=");  builder.append(string);  builder.append("]");  **return** builder.toString();  } |

#### String映射

YANG String的映射与type子句指定的length和pattern有关，描述如下：

|  |  |
| --- | --- |
| Type子句 | 映射成JAVA |
| length | 不映射 |
| pattern | list of string constants = list of patterns  list of Pattern objects  static initialization block where list of Patterns is initialized from list of string of constants |

YANG String映射举例：

|  |  |  |
| --- | --- | --- |
| **YANG** | **JAVA** | **JAVA overriden Object methods** |
| typedef typedef-string {  type string {  length 44;  pattern "[a][.]\*"  }  } | **public** **class** TypedefString {    **private** **static** **final** List<Pattern> patterns = **new** ArrayList<Pattern>();  **public** **static** **final** List<String> PATTERN\_CONSTANTS = Arrays.asList("[a][.]\*");    **static** {  **for** (String regEx : PATTERN\_CONSTANTS) {  patterns.add(Pattern.compile(regEx));  }  }    **private** String typedefString;    **public** TypedefString(String typedefString) {  **super**();  **this**.typedefString = typedefString;  }    **public** String getTypedefString() {  **return** typedefString;  }  } | @Override  **public** **int** hashCode() {  **final** **int** prime = 31;  **int** result = 1;  result = prime \* result + ((typedefString == **null**) ? 0 : typedefString.hashCode());  **return** result;  }    @Override  **public** **boolean** equals(Object obj) {  **if** (**this** == obj) {  **return** **true**;  }  **if** (obj == **null**) {  **return** **false**;  }  **if** (getClass() != obj.getClass()) {  **return** **false**;  }  TypedefString other = (TypedefString) obj;  **if** (typedefString == **null**) {  **if** (other.typedefString != **null**) {  **return** **false**;  }  } **else** **if**(!typedefString.equals(other.typedefString)) {  **return** **false**;  }  **return** **true**;  }    @Override  **public** String toString() {  StringBuilder builder = **new** StringBuilder();  builder.append("TypedefString [typedefString=");  builder.append(typedefString);  builder.append("]");  **return** builder.toString();  } |

### Container

YANG Container映射成一个继承接口DataObject和Augmentable<***container\_interface***>的JAVA接口，其中，***container\_interface***是container映射得到的接口名称。

一个映射例子如下所示：

|  |  |
| --- | --- |
| YANG | JAVA |
| container cont {  } | **public** **interface** Cont **extends** DataObject, Augmentable<Cont> {  } |  |

### leaf

每个leaf都至少包含一个type子句，leaf映射成上级元素的getter方法，这个方法返回类型与type子句的类型一致。

一个映射例子如下所示：

|  |  |
| --- | --- |
| **YANG** | **JAVA** |
| module module {  namespace "urn:module";  prefix "sbd";  organization "OPEN DAYLIGHT";  contact "http://www.whatever.com/";  revision 2013-07-09 {  }  leaf lf {  type string;  }  } | **package** org.opendaylight.yang.gen.v1.urn.module.rev130709;  **public** **interface** ModuleData {  String getLf();  } |

一个caontainer级别的leaf映射例子如下：

|  |  |
| --- | --- |
| YANG | JAVA |
| container cont {  leaf lf {  type string;  }  } | **public** **interface** Cont **extends** DataObject, Augmentable<Cont> {  String getLf();  } |

### Leaf-list

一个leaf-list包含一个type子句，leaf-list映射成上级元素的getter方法，这个方法返回List<***leaf-list-type***>，其中***leaf-list-type***与type子句的类型一致。

leaf-list映射示例：

|  |  |
| --- | --- |
| **YANG** | **JAVA** |
| container cont {  leaf-list lf-lst {  type typedef-union;  }  } | **public** **interface** Cont **extends** DataObject, Augmentable<Cont> {  List<TypedefUnion> getLfLst();  } |

其中，YANG typedef-union和JAVA TypedefUnion与1.3.8.1.3一样。

### List

YANG list元素映射成JAVA接口。上级元素中，它被映射成一个返回list类型的getter方法。

list子句映射成JAVA:

|  |  |
| --- | --- |
| Substatement | Mapping to JAVA |
| key | class |

这里的key会映射成一个名称为**listJavaName**Key的JAVA类，并实现dentifier<**ListJavaName**>接口，其中**ListJavaName**是list元素映射成的java接口名称。而**ListJavaName**接口要继承Identifiable<**ListJavaName**Key>，并在接口内定义如下方法：**ListJavaName**Key getKey();

在如下list映射的例子中，**outter-list**映射成JAVA接口**OutterList**，以及**Cont接口中**的返回类型为List<**OutterList**>的getter方法。

|  |  |  |
| --- | --- | --- |
| **YANG** | **JAVA** | **JAVA overriden Object methods** |
| container cont {  list outter-list {  key "leaf-in-list"  leaf leaf-in-list {  type uint64;  }  leaf-list leaf-list-in-list {  type string;  }  list list-in-list {  leaf-list inner-leaf-list {  type int16;  }  }  }  } | ListInList.java  **package** org.opendaylight.yang.gen.v1.urn.module.rev130709.cont.outter.list;  **import** org.opendaylight.yangtools.yang.binding.DataObject;  **import** org.opendaylight.yangtools.yang.binding.Augmentable;  **import** java.util.List;    **public** **interface** ListInList **extends** DataObject, Augmentable<ListInList> {    List<Short> getInnerLeafList();  }  OutterListKey.java  **package** org.opendaylight.yang.gen.v1.urn.module.rev130709.cont;    **import** org.opendaylight.yang.gen.v1.urn.module.rev130709.cont.OutterListKey;  **import** java.math.BigInteger;    **public** **class** OutterListKey **implements** Identifier<OutterListKey> {    **private** BigInteger LeafInList;    **public** OutterListKey(BigInteger LeafInList) {  **super**();  **this**.LeafInList = LeafInList;  }    **public** BigInteger getLeafInList() {  **return** LeafInList;  }  }  OutterList.java  **package** org.opendaylight.yang.gen.v1.urn.module.rev130709.cont;    **import** org.opendaylight.yangtools.yang.binding.DataObject;  **import** org.opendaylight.yangtools.yang.binding.Augmentable;  **import** java.util.List;  **import** org.opendaylight.yang.gen.v1.urn.module.rev130709.cont.outter.list.ListInList;    **public** **interface** OutterList **extends** DataObject, Augmentable<OutterList>,Identifiable<OutterListKey> {    List<String> getLeafListInList();    List<ListInList> getListInList();    */\**  *Returns Primary Key of Yang List Type*  *\*/*  OutterListKey getKey();    }  Cont.java  **package** org.opendaylight.yang.gen.v1.urn.module.rev130709;      **import** org.opendaylight.yangtools.yang.binding.DataObject;  **import** org.opendaylight.yangtools.yang.binding.Augmentable;  **import** java.util.List;  **import** org.opendaylight.yang.gen.v1.urn.module.rev130709.cont.OutterList;    **public** **interface** Cont **extends** DataObject, Augmentable<Cont> {    List<OutterList> getOutterList();    } | for OutterListKey.java  @Override  **public** **int** hashCode() {  **final** **int** prime = 31;  **int** result = 1;  result = prime \* result + ((LeafInList == **null**) ? 0 : LeafInList.hashCode());  **return** result;  }    @Override  **public** **boolean** equals(Object obj) {  **if** (**this** == obj) {  **return** **true**;  }  **if** (obj == **null**) {  **return** **false**;  }  **if** (getClass() != obj.getClass()) {  **return** **false**;  }  OutterListKey other = (OutterListKey) obj;  **if** (LeafInList == **null**) {  **if** (other.LeafInList != **null**) {  **return** **false**;  }  } **else** **if**(!LeafInList.equals(other.LeafInList)) {  **return** **false**;  }  **return** **true**;  }    @Override  **public** String toString() {  StringBuilder builder = **new** StringBuilder();  builder.append("OutterListKey [LeafInList=");  builder.append(LeafInList);  builder.append("]");  **return** builder.toString();  } |

### Choice and case

choice元素映射同list元素类似，也就是映射成JAVA接口(假定是MarkerInterface)，和上级元素的返回类型为MarkerInterface的getter方法

case子句映射成继承MarkerInterface的JAVA接口。

一个映射示例如下所示：

|  |  |
| --- | --- |
| **YANG** | **JAVA** |
| container cont {  choice choice-test {  **case** case1 {  }  **case** case2 {  }  }  } | Case1.java  **package** org.opendaylight.yang.gen.v1.urn.module.rev130709.cont.choice.test;    **import** org.opendaylight.yangtools.yang.binding.DataObject;  **import** org.opendaylight.yangtools.yang.binding.Augmentable;  **import** org.opendaylight.yang.gen.v1.urn.module.rev130709.cont.ChoiceTest;    **public** **interface** Case1 **extends** DataObject, Augmentable<Case1>, ChoiceTest {  }  Case2.java  **package** org.opendaylight.yang.gen.v1.urn.module.rev130709.cont.choice.test;    **import** org.opendaylight.yangtools.yang.binding.DataObject;  **import** org.opendaylight.yangtools.yang.binding.Augmentable;  **import** org.opendaylight.yang.gen.v1.urn.module.rev130709.cont.ChoiceTest;    **public** **interface** Case2 **extends** DataObject, Augmentable<Case2>, ChoiceTest {  }  ChoiceTest.java  **package** org.opendaylight.yang.gen.v1.urn.module.rev130709.cont;    **import** org.opendaylight.yangtools.yang.binding.DataObject;    **public** **interface** ChoiceTest **extends** DataObject {  }  Cont.java  **package** org.opendaylight.yang.gen.v1.urn.module.rev130709;    **import** org.opendaylight.yangtools.yang.binding.DataObject;  **import** org.opendaylight.yangtools.yang.binding.Augmentable;  **import** org.opendaylight.yang.gen.v1.urn.module.rev130709.cont.ChoiceTest;    **public** **interface** Cont **extends** DataObject, Augmentable<Cont> {    ChoiceTest getChoiceTest();    } |

### Grouping and uses

grouping映射成一个继承DataObject的JAVA接口。而对于使用use子句的那些元素，映射成继承grouping生成接口的JAVA接口。

一个映射示例如下：

|  |  |
| --- | --- |
| **YANG** | **JAVA** |
| grouping grp {    }    container cont {  uses grp;  } | Cont.java  **package** org.opendaylight.yang.gen.v1.urn.module.rev130709;    **import** org.opendaylight.yangtools.yang.binding.DataObject;  **import** org.opendaylight.yangtools.yang.binding.Augmentable;    **public** **interface** Cont **extends** DataObject, Augmentable<Cont>, Grp {  }  Grp.java  **package** org.opendaylight.yang.gen.v1.urn.module.rev130709;    **import** org.opendaylight.yangtools.yang.binding.DataObject;    **public** **interface** Grp **extends** DataObject {  } |

### Rpc

RPC映射成类ModuleService的JAVA方法。

RPC子句映射规则如下：

|  |  |
| --- | --- |
| **Rpc子句** | **mapped to JAVA** |
| input | interface |
| output | interface |

RPC映射示例如下：

|  |  |
| --- | --- |
| **YANG** | **JAVA** |
| rpc rpc-test1 {  output {  leaf lf-output {  type string;  }  }  input {  leaf lf-input {  type string;  }  }  } | ModuleService.java  **package** org.opendaylight.yang.gen.v1.urn.module.rev130709;    **import** java.util.concurrent.Future;  **import** org.opendaylight.yangtools.yang.common.RpcResult;    **public** **interface** ModuleService {    Future<RpcResult<RpcTest1Output>> rpcTest1(RpcTest1Input input);    }-  RpcTest1Input.java  **package** org.opendaylight.yang.gen.v1.urn.module.rev130709;    **public** **interface** RpcTest1Input {    String getLfInput();    }  RpcTest1Output.java  **package** org.opendaylight.yang.gen.v1.urn.module.rev130709;    **public** **interface** RpcTest1Output {    String getLfOutput();    } |

### Notification

Notification映射成JAVA接口，继续Notification interface。

一个notification映射示例如下所示:

|  |  |
| --- | --- |
| **YANG** | **JAVA** |
| notification notif {  } | ModuleListener.java  **package** org.opendaylight.yang.gen.v1.urn.module.rev130709;    **import** java.util.concurrent.Future;  **import** org.opendaylight.yangtools.yang.common.RpcResult;    **public** **interface** ModuleListener {  void onNotif(Notif notification);  }  Notif.java  **package** org.opendaylight.yang.gen.v1.urn.module.rev130709;    **import** org.opendaylight.yangtools.yang.binding.DataObject;  **import** org.opendaylight.yangtools.yang.binding.Augmentable;  **import** org.opendaylight.yangtools.yang.binding.Notification;    **public** **interface** Notif **extends** DataObject, Augmentable<Notif>, Notification {  } |

### Augment

Augment映射成JAVA接口，这个接口名称开头是被扩大接口的名称。后缀是扩大接口的顺序，并且继承 Augmentation<>接口，泛型参数为被扩大的接口。

下面是一个augment映射示例，在例子中，被扩大的接口是Cont，所以参数化类型是

*Augmentation<Cont>.*

|  |  |
| --- | --- |
| **YANG** | **JAVA** |
| container cont {  }    augment "/cont" {  } | Cont.java  **package** org.opendaylight.yang.gen.v1.urn.module.rev130709;    **import** org.opendaylight.yangtools.yang.binding.DataObject;  **import** org.opendaylight.yangtools.yang.binding.Augmentable;    **public** **interface** Cont **extends** DataObject, Augmentable<Cont> {    }  Cont1.java  **package** org.opendaylight.yang.gen.v1.urn.module.rev130709;    **import** org.opendaylight.yangtools.yang.binding.DataObject;  **import** org.opendaylight.yangtools.yang.binding.Augmentation;    **public** **interface** Cont1 **extends** DataObject, Augmentation<Cont> {    } |

### Identity

identity语句是为了定义一个新的全局唯一地、抽象的和没有类型的实体。YANG的base子句，取一个string型参数表示已存在identity的名称。因此identity语句映射成JAVA抽象类，而base子句映射成继承base指定的抽象类(没有指定表示继承BaseIdentity)。identity名称转化成类名。

一个identity映射示例如下：

|  |  |
| --- | --- |
| **YANG** | **JAVA** |
| identity toast-type {  } | ToastType.java  **public** **abstract** **class** ToastType **extends** BaseIdentity {  **protected** ToastType() {  **super**();  }  } |
| identity white-bread {  base toast-type;  } | WhiteBread.java  **public** **abstract** **class** WhiteBread **extends** ToastType {  **protected** WhiteBread() {  **super**();  }  } |

## YANG Maven Tools使用指南

本节说明如何在自己的项目中使用YANG Tools。

### 先决条件

Maven 3

### 工程配置

建议从已存在的YANG Model工程拷贝pom.xml进行修改。

#### Repositories

按如下方式配置plugin repositories：

<pluginRepositories>

<pluginRepository>

<id>opendaylight-release</id>

<name>opendaylight-release</name>

<url>http://nexus.opendaylight.org/content/repositories/opendaylight.release/</url>

</pluginRepository>

<pluginRepository>

<id>opendaylight-snapshot</id>

<name>opendaylight-snapshot</name>

<url>http://nexus.opendaylight.org/content/repositories/opendaylight.snapshot/</url>

</pluginRepository>

</pluginRepositories>

接下来配置需要的依赖：

<repositories>

<repository>

<id>opendaylight-release</id>

<name>opendaylight-release</name> <url>http://nexus.opendaylight.org/content/repositories/opendaylight.release/</url>

</repository>

<repository>

<id>opendaylight-snapshot</id>

<name>opendaylight-snapshot</name>

<url>http://nexus.opendaylight.org/content/repositories/opendaylight.snapshot/</url>

</repository>

</repositories>

### 配置yang-maven-plugin

增加yang-maven-plugin到pom.xml：

<build>

<plugins>

<plugin>

<groupId>org.opendaylight.yangtools</groupId>

<artifactId>yang-maven-plugin</artifactId>

<version>0.6.0-SNAPSHOT</version>

<executions>

<execution>

<goals>

<goal>generate-sources</goal>

</goals>

<configuration>

<yangFilesRootDir>src/main/yang</yangFilesRootDir>

<codeGenerators>

<generator>

<codeGeneratorClass>

org.opendaylight.yangtools.maven.sal.api.gen.plugin.CodeGeneratorImpl

</codeGeneratorClass>

<outputBaseDir>

target/generated-sources/sal

</outputBaseDir>

</generator>

</codeGenerators>

<inspectDependencies>true</inspectDependencies>

</configuration>

</execution>

</executions>

<dependencies>

<dependency>

<groupId>org.opendaylight.yangtools</groupId>

<artifactId>maven-sal-api-gen-plugin</artifactId>

<version>0.6.0-SNAPSHOT</version>

<type>jar</type>

</dependency>

</dependencies>

</plugin>

</plugins>

</build>

### 配置依赖

YANG Tools生成的APIs需要参考org.opendaylight.yangtools:yang-binding和

org.opendaylight.yangtools:yang-common两个工程。

将这两个工程添加到你的工程依赖。

<dependencies>

<dependency>

<groupId>org.opendaylight.yangtools</groupId>

<artifactId>yang-binding</artifactId>

<version>0.5.7-SNAPSHOT</version>

</dependency>

<dependency>

<groupId>org.opendaylight.yangtools</groupId>

<artifactId>yang-common</artifactId>

<version>0.5.7-SNAPSHOT</version>

</dependency>

</dependencies>

### 构建

代码在generate-sources阶段生成。生成代码运行如下命令：

mvn generate-sources

### Eclipse集成(m2e)

#### m2e警告

如果在eclipse中使用，需要将下列片段增加到你的pom.xml中，以越过m2e的警告。

<build>

<pluginManagement>

<plugins>

<plugin>

<groupId>org.eclipse.m2e</groupId>

<artifactId>lifecycle-mapping</artifactId>

<version>1.0.0</version>

<configuration>

<lifecycleMappingMetadata>

<pluginExecutions>

<pluginExecution>

<pluginExecutionFilter>

<groupId>

org.opendaylight.controller

</groupId>

<artifactId>

yang-maven-plugin

</artifactId>

<versionRange>

[0.5,)

</versionRange>

<goals>

<goal>

generate-sources

</goal>

</goals>

</pluginExecutionFilter>

<action>

<ignore></ignore>

</action>

</pluginExecution>

</pluginExecutions>

</lifecycleMappingMetadata>

</configuration>

</plugin>

</plugins>

</pluginManagement>

</build>

#### 源文件发现

如果生成的文件没有引入到eclipse，需要在增加mojo-build-helper-plugin插件。

<plugin>

<groupId>org.codehaus.mojo</groupId>

<artifactId>build-helper-maven-plugin</artifactId>

<version>1.7</version>

<executions>

<execution>

<phase>generate-sources</phase>

<goals>

<goal>add-source</goal>

</goals>

<configuration>

<sources>

<source>target/generated-sources/sal</source>

</sources>

</configuration>

</execution>

</executions>

</plugin>

### 设计和编码指南

#### 类和接口名

（1）接口名称不要使用I前缀，可以使用平常、简单和相对短的接口名称。接口使用人员不需要关心他是使用抽象类或接口，他不需要使用构造器创建对象，因此，使用User而不是IUser。

（2）接口名称尽可能使用后缀，表示一些特定的语义。比如Readable比Read、Reading要好得多。

（3）实现接口的类名，要和接口要一定的相关性。例如，实现User接口可以是UserImpl、UserMock，UserDbImpl或者DbBasedUser(尽管最后一个不太影响)

（4）保持接口、抽象类和其它实体在不同的JAVA包中。

#### getter和setter方法

（1）getter和setter方法应该只针对简单属性。调用getter和setter不应该失败，且不因网络环境或其它条件而阻塞。

（2）getter和setter方法不应该抛出检查异常，并且尽可能不抛出任何异常。实际上他们也可能会抛出运行时异常，但运行时异常发生时，系统应该已经出现严重错误。

（3）当你的访问或设置方法可能会失败，给他们取不同的名字，例如，使用retrieveFoo替换getFoo，registerBar替换setBar。

#### 良好习惯

基于接口而不是基于实现类进行编码。例如定义本地变量Map map，而不是HashMap map。

#### 源代码结构和构建

##### Package Names

（1）使用org.opendaylight.yang作为包名前缀prefix；

（2）前缀后面紧跟组件名称，例如，org.opendaylight.yang.parser；

（3）复杂组件可以有多个子包，例如，org.opendaylight.yang.parser.util；

（4）包名的最后一部分，用来区分组件的公共和私有部分。

公共接口以及它的实现分别放在单独的包里。例如，包org.opendaylight.yang.service包含公共服务定义，而包org.opendaylight.yang.service.mock则是对服务的实现。

一些建议的取值如下表所示：

|  |  |
| --- | --- |
| **api** | 对外的公共接口 |
| **spi** | 计划提供功能的公共接口 |
| **impl** | 主要组件实现，如果不只有一个使用实现，不要使用后缀，而使用其它描述实现的名称。 |
| **util** | 工具函数库 |
| **mock** | 用于测试目的的模拟实现 |
| **test** | 组件单元测试 |

## 代码生成器DEMO

这里以code-generator项目里的一个例子modeling-sample，来说明一个YANG文件，如何通过YANG Tools生成JAVA接口。一般有如下步骤：

（1）获取最新版本的yangtools.git，参考下面链接中文档；

[OpenDaylight\_Controller:Pulling,\_Hacking,\_and\_Pushing\_the\_Code\_from\_the\_CLI](https://wiki.opendaylight.org/view/OpenDaylight_Controller:Pulling,_Hacking,_and_Pushing_the_Code_from_the_CLI)

（2）编译所有的包

mvn clean install -DskipTests

（3）进入到YANG文件所在目录：

cd code-generator\samples\modeling-sample

其中，文件夹中存在预告配置的样例toaster.yang(查看pom.xml)。

pom.xml包含工程配置和code generation插件；

src/main/yang/toaster.yang 包含一个YANG模型toaster

（4）构建工程：

mvn clean install

（5）生成的JAVA接口源代码在如下目录：

target/generated-sources/sal folder