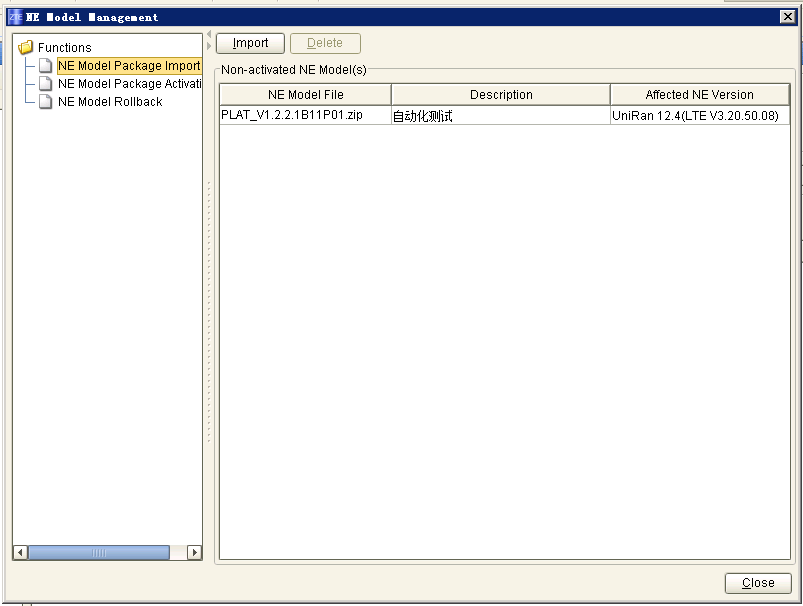
**MO模型库包设计规范**

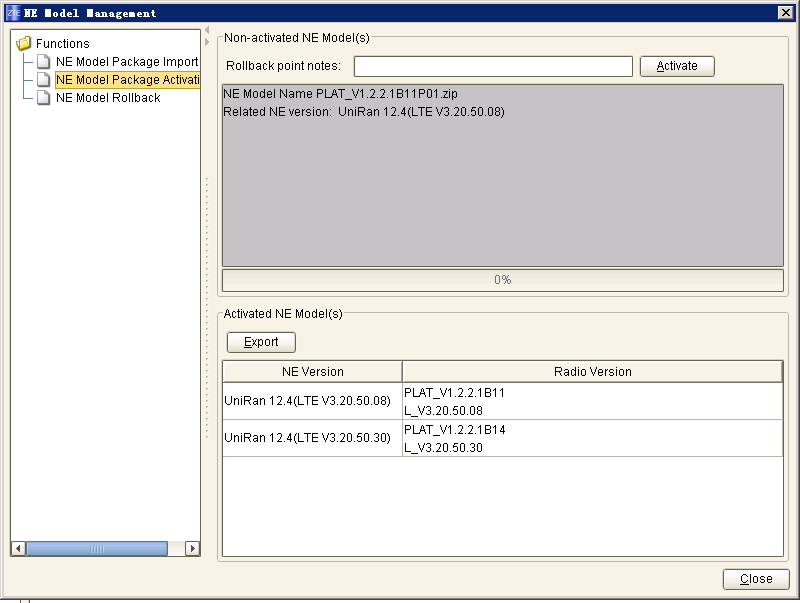
# 模型库设计

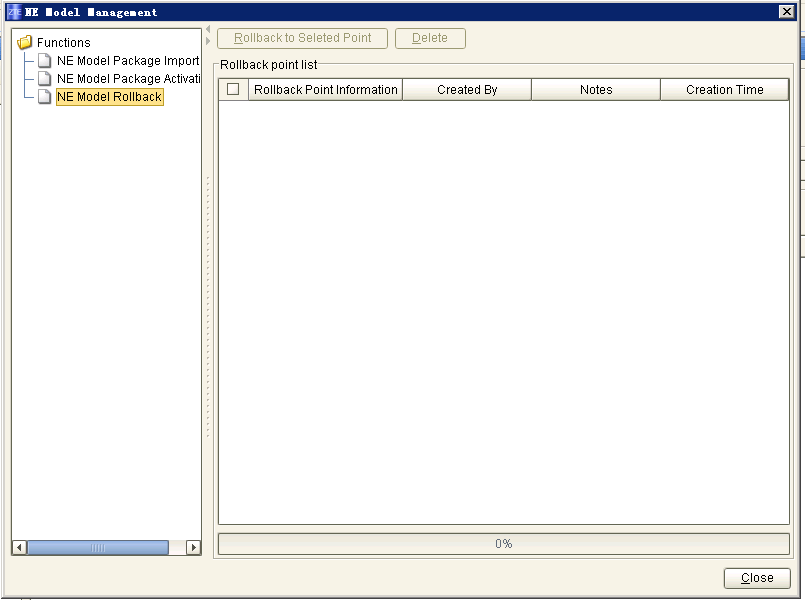
本文编写的目的是为了各个产品在制作MO模型库包时，按照本文的要求进行编写。

当前网管：网管功能和网元模型逻辑混在一起，升级网元版本时，也要升级网管版本。

扁平化网管：网管功能和网元模型逻辑解耦分离，升级网元版本可以通过导入模型包实现，网管不用升级，对大部分场景，网管进程也无须重启（是否重启网管进程，由各领域对新网元模型包的处理能力决定）。







## 网元模型库管理能力

1. 提供了网元模型包的定义规范。按照该规范实现网元模型和网管功能的解耦。
2. 在网管上实现网元模型包的导入，加载，回退，客户端推送分发。支持热加载模式，也支持按要求重启分进程的需求。

## 模型库文件定义

模型库是一个zip包，通常它包含一个网元版本的和网元业务逻辑相关的内容。模型包有2种形式：

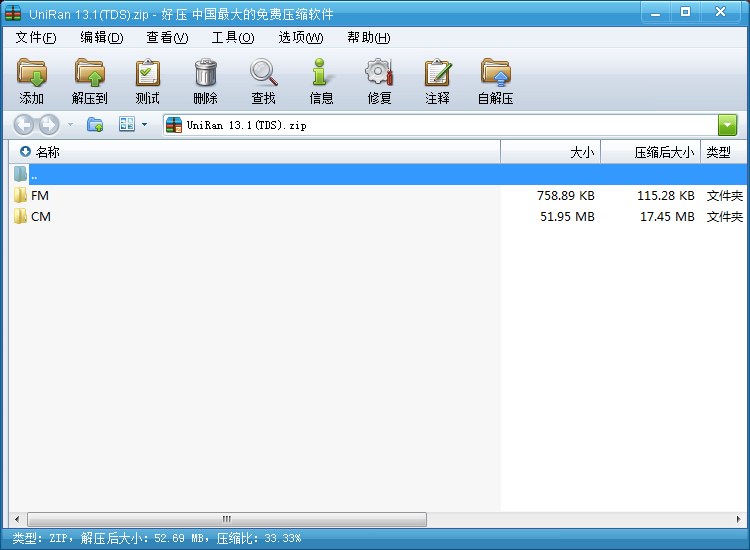
1. 一个网元版本全新的模型包。
2. 一个网元版本的补丁模型包。比如某个网元版本的groovy约束有bug，因此出一个模型的补丁包来解决这个问题。还有一种就是针对新增一个物理设备，比如新增rru规格，它也可以算作一种模型补丁包。
3. 网元模型包不光是配置模型，还应该包括告警（告警码），动态，诊断，性能（计数器）等各个领域与网元模型相关的内容。
4. 网元模型包的发布和网元版本是绑定的，它的发布时间和网元版本发布时间保持一致。

配置的模型库有一些特别的地方，如下：

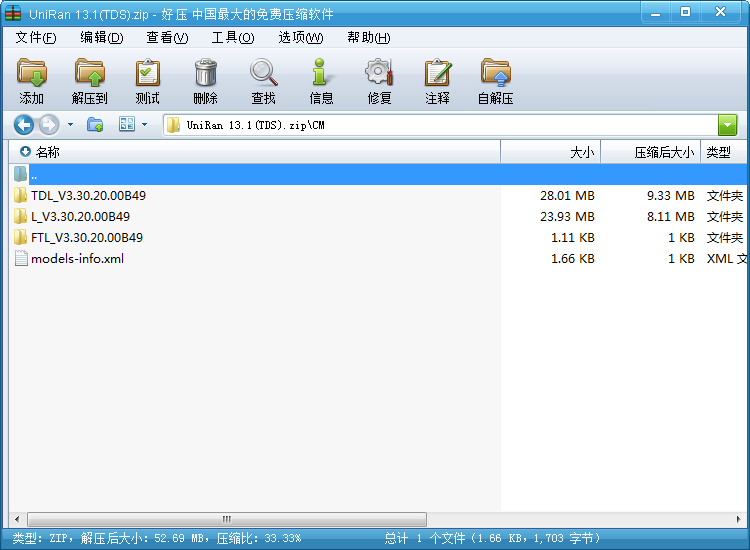
* 不同的网元版本，模型定义都不同。即管理多少个网元版本，就必须有多少个网元版本的模型定义。而其他模块，可能只需要定义一套最新的模型库，向下兼容老网元版本即可。
* 配置模型不光定义网元模型，还定义了网元下的各个制式的模型。各个制式模型可以像搭积木一样组合成网元模型。比如一个层三平台的制式模型，可以用于多个网元版本。

### 模型包文件结构

模型包文件是一个zip文件，通过网管的导入并激活到系统后形成模型库的工作目录。模型包文件先按业务领域划分，比如cm，ddm，fm，然后每个领域内再存放领域内部的模型文件。下面是模型包压缩文件内部第一层的目录示范：

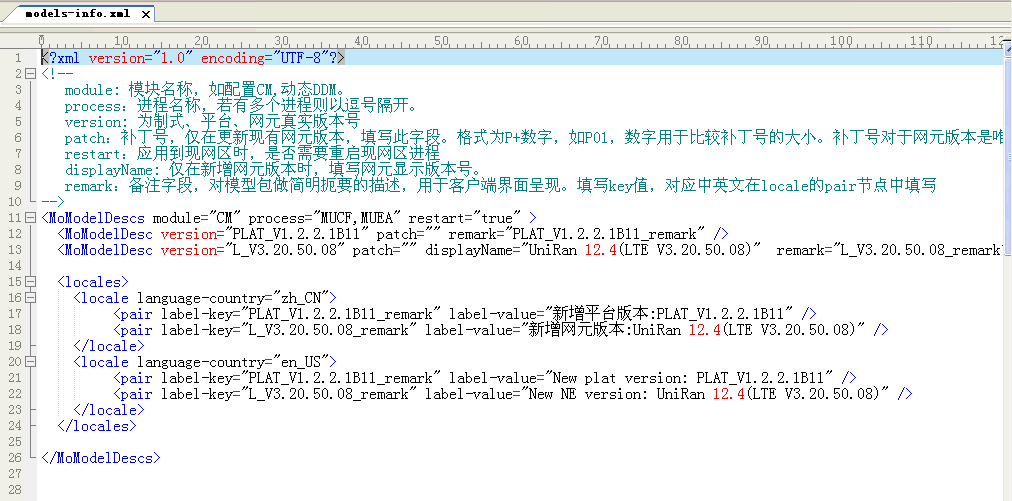


下面以配置领域的模型包结构为例（其他模块类似）如下图所示：

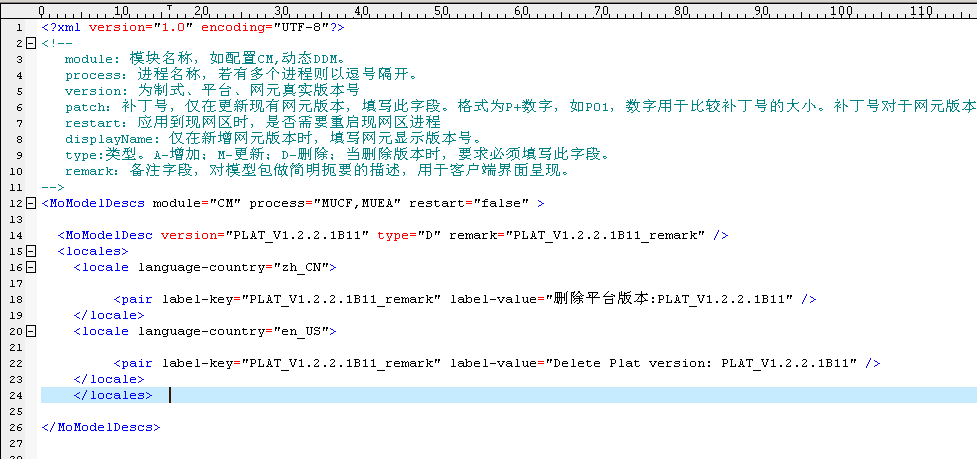


每个领域的模型包中必须有一个models-info.xml文件描述模型信息，models-info.xml文件如下图所示，它体现了网元模型类型及相关网元版本信息：

新增版本示例：



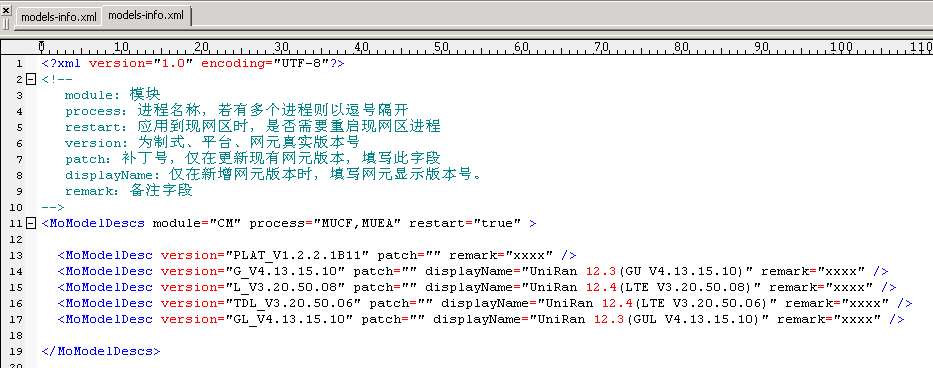
删除版本示例：



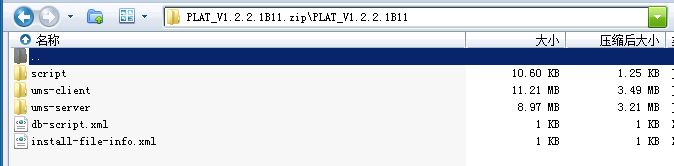
* module: 模块名称，如配置CM,动态DDM。模型库系统按此名称创建模型库工作目录的第一级子目录。
* process：进程名称，若有多个进程则以逗号隔开。
* restart：应用到现网区时，是否需要重启现网区进程，目前取决于模型包中是否有对jar包进行了更新，如果有则需要填写为true。
* version: 为制式、平台、网元真实版本号，要求与同目录下文件夹同名。
* patch：补丁号，仅在更新现有网元版本，填写此字段。格式为P+数字，如P01，数字用于比较补丁号的大小。补丁号对于网元版本是唯一的，跟着version走（比如配置模块申请了版本A的补丁号为PO1，那么动态模块要出版本A的补丁，只能是PO2了）,打补丁规则是从小到大，不允许打完大补丁号的补丁后再打小补丁号的补丁。
* type:类型。D-删除；**当删除版本时，要求必须填写此字段。**
* displayName: 仅在新增网元版本时，填写网元显示版本号，用于客户端界面呈现，配置模块使用，其他模块不关注。
* remark：备注字段，对模型包做简明扼要的描述，用于客户端界面呈现。

**框架支持一个zip包中包含多个网元版本模型，如果存在多个网元版本模型，则在models-info.xml中填写对应的模型信息：**



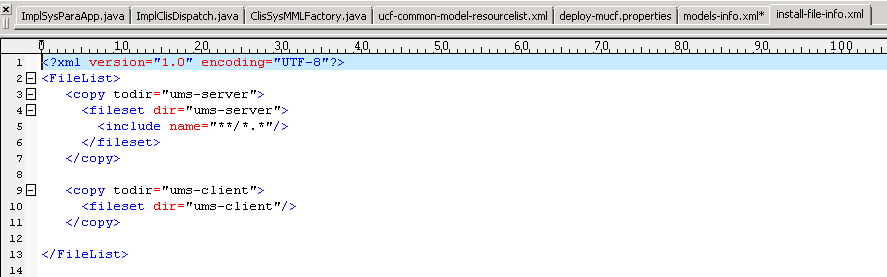


每个制式（或平台）版本模型包的结构如下图所示：

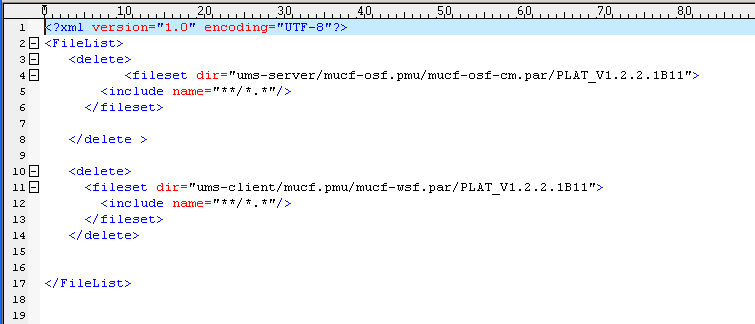


一个是客户端的业务逻辑，一个是服务器端业务逻辑，一个是安装信息文件。安装信息文件描述模型的安装信息(包括文件拷贝、删除）

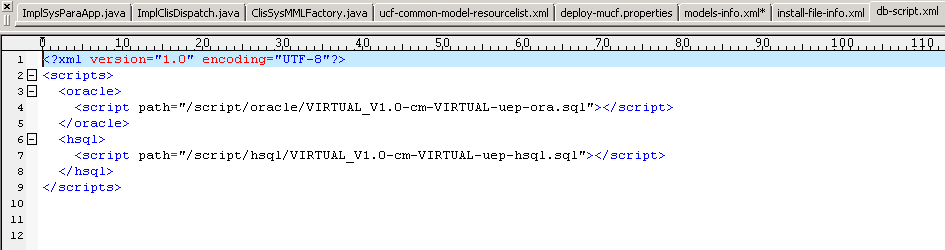
新增版本示例：



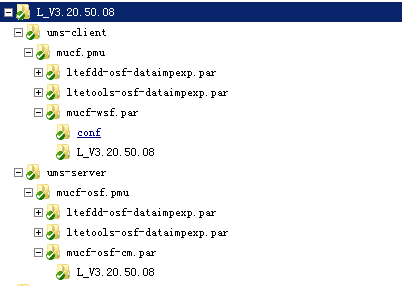
删除版本示例：



通常，MO模型包是不需要包含数据库脚本的（系统会自动根据mo模型文件维护数据库表结构及字段），除非产品有特殊需求，可指定执行产品定制的数据库脚本，脚本文件放置在script目录下，并编写脚本执行描述文件db-script.xml：



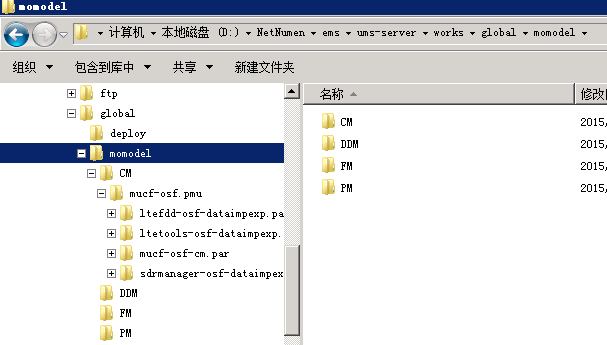
客户端与服务端的结构目录如下：



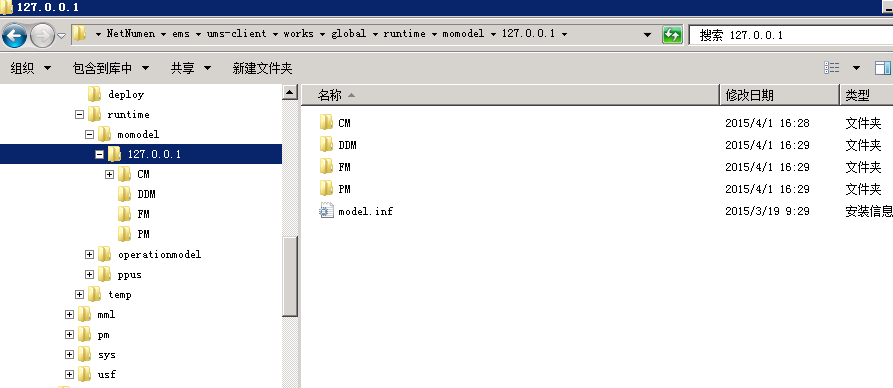
### 模型库的工作目录

模型包文件激活生效后，在网管中形成模型库的工作目录，它先按业务领域划分，比如cm，ddm，fm，然后各领域模块自行决定目录下的文件内容。模型库的工作目录参见“图A 模型库工作目录”。

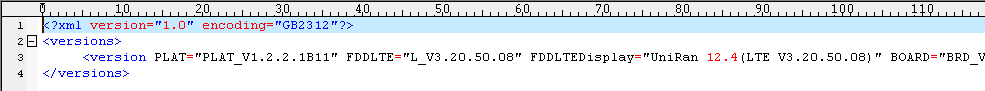
激活生效后，对应目录结果为：



图A 模型库工作目录



配置模块的deploy-\*-version.xml文件放置在ums-client和ums-server目录下，命令方式为deploy-网元版本-version.xml，为方便系统删除网元版本，文件内容只能填写一个网元版本记录：



系统也支持多个模块的模型包zip包压缩成一个大的zip包，作为一个整体导入系统，系统会自动按模块进行分类激活。举例来说告警可以有一个告警模型的zip包，配置可以有一个配置的模型zip包，它们打成一个大的zip包。

模块之间的模型包不能存在依赖关系。

#### 模型文件的目录要求

1. 所有的模型文件都必须存放在一个momodel总目录下，如果不放在这个总目录下会导致无法分发到对应的进程中。所以各领域进程访问时模型文件时需要考虑这点。
2. 模型目录下的文件总尺寸不能过大，用户应该只将和网元模型相关的文件放在该目录下。文件夹尺寸过大的话，会导致导入，激活，回退效率降低，并浪费磁盘空间。
3. 模型按照进程为单位进行分发，所以一个进程只能有一个模块号。比如告警和诊断是同一个进程，那么它们必须共用一个模块号。

### 虚拟网元版本技术

模型库管理的均是一个个网元版本（或制式版本）。比如对配置模型包而言，大部分网元逻辑都放在制式版本目录中，少量网元逻辑放在网元版本目录中。但是配置中还有一类变化比较频繁的业务逻辑，比如模版算法包，既不属于制式版本，也不属于网元版本。此时我们可以建立一个虚拟网元版本，将这些算法包放在这个虚拟网元版本目录下。这样带来很多好处：

1. 可以借助模型库的导入，分发，加载，回退功能以及客户端分发功能，实现算法包的更新需求。
2. 借助模型网元版本号，补丁号技术，实现新旧算法包的替换覆盖管理。

这实际上借助模型包管理功能实现了网管功能的更新。

还有一类配置文件也可以使用虚拟网元版本技术进行网管分发，比如扁平化功能的开关项。在只读模式时，某些菜单项和服务需要关闭，而在全模式下，这些菜单项和服务需要打开。我们可以把这些开关项做成配置文件，放在一个虚拟网元版本下。这样通过导入模型包就可以实现开关配置文件的添加和更新。特别是对客户端也可以实现内容的分发和推送，从而避免为每个客户端打补丁。

对于告警领域，它的模型包只有一套最新的版本，向下兼容所有网元版本，因此告警的模型包版本也可以是一个虚拟网元版本号，只有虚拟网元版本号比较新，或者补丁号比较大，就可以实现新包覆盖旧模型的文件。

## 模型包的特征和分类

### 模型包版本标识

模型库按网元版本号来区分，不同网元版本号代表不同的网元版本。同一个网元版本号，可以打多次补丁，补丁用网元版本号后面的patchVersion来标识。补丁号不作为关键字，只用于描述当前的补丁信息。因此给网元模型版本打补丁并不涉及升级网元版本。

网元版本号和补丁号均由版本经理统一管理，对于补丁号，各个领域模块制作补丁前，均需要向版本经理申请补丁编号，然后各自制作相关补丁。最终生成的补丁包中，各领域模块中可能会包含多个补丁号，系统以最新的补丁号作为网元的最终补丁号显示在界面上。版本经理必须确保补丁包中应该包括最新补丁号之前的所有补丁。系统在确定打入一个补丁包后，就不再允许旧的补丁包（即补丁号比系统当前的小）打入，避免错误覆盖。

对于配置管理模块，它还可以通过制式版本号+制式的补丁号来更加详细的标识模型库版本。制式补丁号同样不允许旧的补丁覆盖新的补丁。

模型库打补丁不支持像网管补丁那样的紧急补丁（即通知单）。因此模型库补丁最好是提前规划好，按计划提供。

### 模型包补丁号规则

同版本的模型包，如果版本号补丁号都相同，则不允许在网管中导入生效，为了解决这个问题，可以利用更新模型包的补丁号来解决，当导入的模型包，补丁号更大时，系统允许导入。

整个网管开发是根据统一的迭代号来规划的，每个迭代都可以发布一个网元模型包，为了便于测试和使用，可以在补丁号中增加迭代号信息，这样每次发布的模型包都是更大的补丁号，因此可以直接导入生效。比如S59迭代，则模型包的补丁号为patch=5900。后面2个0是为了方便出真正的补丁的编号。

### 模型包和网管补丁的关系

模型包和网管补丁分离，也就是说模型包的内容不在网管补丁中体现，它和网管补丁的内容是正交的。这样后续模型包更新和网管补丁更新就不会冲突。

### 模型包详细分类（暂时不考虑）

按照网管版本和补丁的思路，模型包也分成完整模型包和模型补丁包。还有一类是新增rru规格，微站规格等，还有紧急补丁包。

完整模型包：包括一个网元版本的所有模型内容。比如新增一个网元版本。

模型补丁包：针对一个现有网元版本做的修正，比如bug修改。模型补丁包不允许修改数据库模型。

紧急补丁包：在模型补丁包来不及发布的情况下，可以制作紧急补丁包（类似网管的通知单），发布到现场。紧急补丁包直接修改主模型包中的文件，但是可以不修改补丁号。紧急补丁包不允许修改数据库模型。

新rru规格或微站模型包：在模型补丁包来不及发布的情况下，现场急需支持某种rru或微站规格，可以做rru规格包或微站模型包发布到现场。rru规格包或微站模型包存放在专有目录中，和模型的主目录分开。利用系统内存合并的功能来实现模型包的应用。rru规格包或微站模型包可以不修改补丁号。

紧急补丁包和rru规格包等虽然不记录补丁号，但是可以通过模型管理操作日志来记录用户操作的信息，从而获取紧急补丁包的操作过程信息。

当现场打过紧急补丁包后，下一次再打模型补丁包或紧急补丁包，需要由项目自行保证之前的紧急补丁包内容不会被覆盖。通常项目可以在发布的模型补丁包中包含以前已经打过的紧急补丁包内容。这点需要项目特别关注。

## 模型库的使用

扁平化系统中需要由一个统一模型库管理的模块，用于管理导入的模型库。对模型库的管理操作主要是增加，删除，修改，查看和应用。其中增删改查操作可以做成统一的模块，但是应用模型库和具体模块实现有关，需要各模块具体分析。

由于扁平化网管系统各业务模块会分进程管理，因此模型库管理是一个单独的进程（目前放在UCA进程中，可以方便控制其他进程重启），如下图所示：



模型管理器导入一个网元的模型包后，模型包内容存放在一个专有的目录中。然后依次通知各业务领域模块发起激活操作，各业务模块读取网管模型信息，将模型信息重新加载到内存中，从而完成模型的生效。

激活模型时，可能会要求业务进程重启，是否重启标识定义在模型包中。

1. 若模型包标记为需要重启，则模型管理器会先关闭业务进程，然后再拷贝模型文件，启动业务进程，调用业务进程接口进行模型库的加载。
2. 若模型包标记为不需要重启，则模型管理器直接拷贝模型文件，然后调用业务进程接口进行模型库的重新加载。

对各业务进程而言，需要实现如下几种状态，如下图所示：



1. 未启动：指业务进程未启动。
2. 空跑：指业务进程启动了，但是没有加载模型，处于空跑状态。
3. 加载模型运行：指业务进程启动了，并且加载了模型，处于正常运行状态。

定义空跑状态的意义在于将模型管理和业务模块解耦。在空跑状态下，模型管理进程可以对业务进程进行拷贝，修改，删除模型文件，等这些工作全部完成后，再通知业务模块加载模型进入正常运行状态。即模型库文件的升级，更新均由模型管理器来完成。业务进程不需要关心模型库文件的变化。业务进程只需要实现加载模型的过程就可以了。

## 各领域模块的模型定义

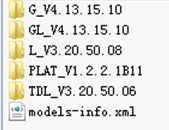
### 配置模型信息分类

一个网元由一个或多个制式组成，配置模型信息按照制式进行分类，即大部分模型信息都是存储在以制式版本号为目录的文件夹中，具体描述如下：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 模块 | 内容 | 内容形式 | 位置 | 备注 |
| 配置管理 |  |  |  |  |
|  | MOM模型 | XML文件 | 服务端,客户端 |  |
|  | 物理设备模型 | 对象级建模xml | 服务端,客户端 |  |
|  |  | 设备包容关系xml | 服务端,客户端 |  |
|  | 自动创建逻辑 约束检查逻辑  监听器 | groovy/Lua脚本，  监听器是java代码，mao后改成Lua脚本。 | 服务端,客户端 | 效率和正确性如何保证？ |
|  | 自定义默认值 | 对象级建模xml | 服务端,客户端 | 非网元模型 |
|  | NE升级脚本 | sql脚本，后续用Lua | 客户端 |  |
|  | 模版定义和算法 | 模版描述xml文件，java算法类 | 客户端,服务端 | 非网元模型 |
|  | 客户端模型定制包 | java类 | 客户端 | 非网元模型 |
|  | 产品可能有需要增加的模型信息，比如单站升级 | 产品分析 | ？ | 非网元模型 |

### 配置模型包目录设定建议

当前配置模型包放在momodel\CM目录下，子目录有：



上面每个目录都代表一个网元版本或制式版本，但是配置系统中有一些内容并不适合放在上述这些目录中，因为可能存在多个网元版本共用的jar包，比如监听器类，客户端二次开发类。同路径同名的类在模型包中是不允许同时存在的。

框架建议在配置模型包目录下再增加每个制式的func目录用于存放上述类，目录如下：

LFUNC\_V1.0

TDSFUNC\_V1.0

TDDLTEFUNC\_V1.0

UMTSFUNC\_V1.0

PlatFunc 存放层三平台的客户端定制java类，模版算法包

MUCF\_COMMON 目前只存放了界面菜单开关的定义。

…（一个制式一个目录）

在原有的网元和制式目录下，存放如下文件：

1. MOM模型文件
2. 约束脚本（groovy，Lua）
3. 物理设备模型文件
4. Master模型文件
5. 升级脚本

在每个制式的func目录下，存放如下文件

1. 模版定义和算法
2. 监听器代码
3. 客户端二次定制代码

这些func目录实际是一个虚拟网元，因此当func目录下文件需要更新或出补丁时，可以对该虚拟网元出补丁版本来解决。这样也可以实现新旧补丁版本的次序控制。

## 模型初始化默认激活

### 概述

要实现模型包在ems安装时自动激活对应的制式模型，需要满足2个条件：

1、各个制式的产品提供制式标识文件。用于表示当前网管支持哪些制式。

2、对每个网元模型，包括虚拟网元模型，标识在什么制式下需要被激活。

模型包在ems安装时自动激活原理：

每次安装或增量安装ems时，uep都会把所有模型包拷贝到特定目录。Ems第1次启动后，模型管理进程首先检测特定目录中是否有模型文件，如果有就按照安装的制式信息进行激活，（已激活的模型包会跳过）。最终删除特定目录中的模型文件。下次ems重启时就不会再次激活了。

这样网管在大版本发布时，内部就已经提供了大版本对应的网元模型文件。升级网管时自动增加新的网元模型。而网管中网元模型管理的功能，则主要应对大版本之间的一些模型补丁，比如新增一款rru的热补丁。

### 标识文件

这个文件是由产品提供，放在各个制式的安装包中，在安装ems的时候放置到指定的目录文件夹下面。这个文件可以没有内容，仅仅用文件名进行标识。文件名规范：

UMTS.tag

GSM.tag

TDS.tag

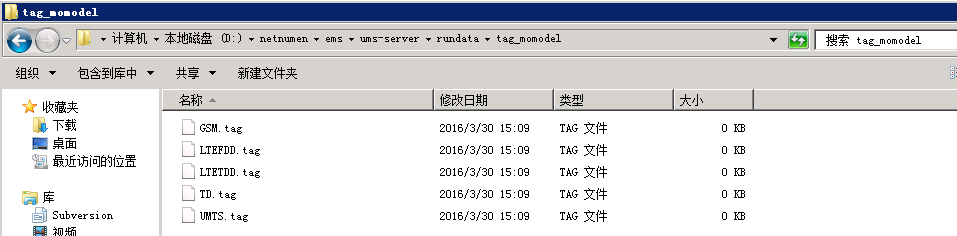
CDMA.tag

LTEFDD.tag

LTETDD.tag

文件放置路径：ums-server\rundata\tag\_momodel\

样式截图：



tag文件的目的是标记这个ems安装并支持了哪些制式。tag文件应该直接放在制式主安装包的zip文件中。

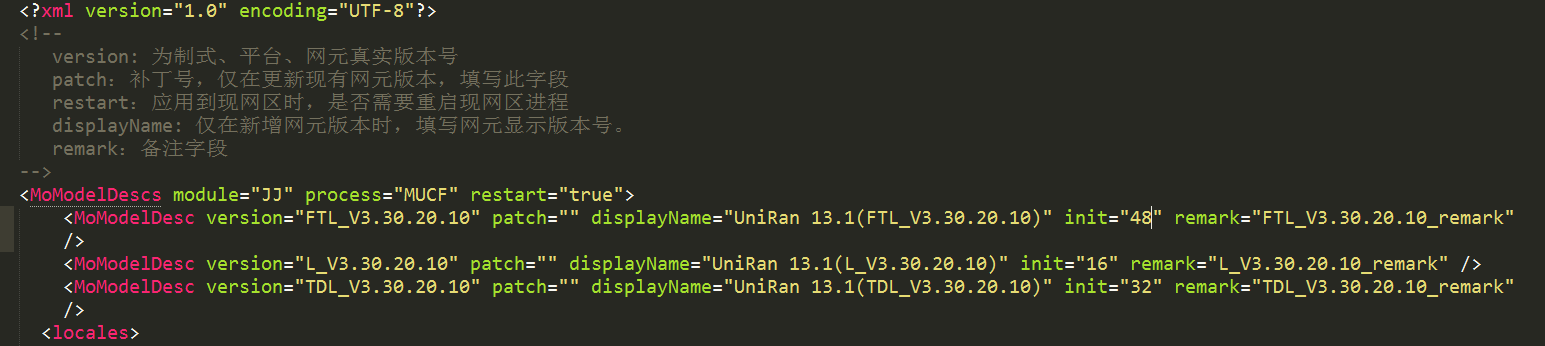
举例来说：LTEFDD.tag这个文件应该放在fddV3-V12.15.10B56.zip文件中，目录就是ums-server\rundata\tag\_momodel。 ems在安装时就会自动将这个tag文件拷贝到对应的目录位置。

如果针对ltefdd制式，产品提供了多个zip安装包（比如还有一些ltefdd工具安装包，北向安装包），那么tag文件只需要放在最主要的安装包中就可以了。因为要支持某个制式，主安装包是必选的，主安装包选中安装了，就认为ems支持了这个制式。

### 模型包内部修改

对模型包内部的修改，主要作用是将模型包内部的具体产品模型文件与安装了的产品相关联起来，以此来进行默认激活模型。需要产品在\*-models-info.xml文件中添加init属性。

建议配置文件修改截图：



Init中填写制式对应的数字值（1->UMTS，2->GSM，4->TDS，8->CDMA，16->LTEFDD，32->LTETDD），累加表示多模，逗号表示或的意思。

Init=”1” 表示当安装了umts制式时激活此模型包

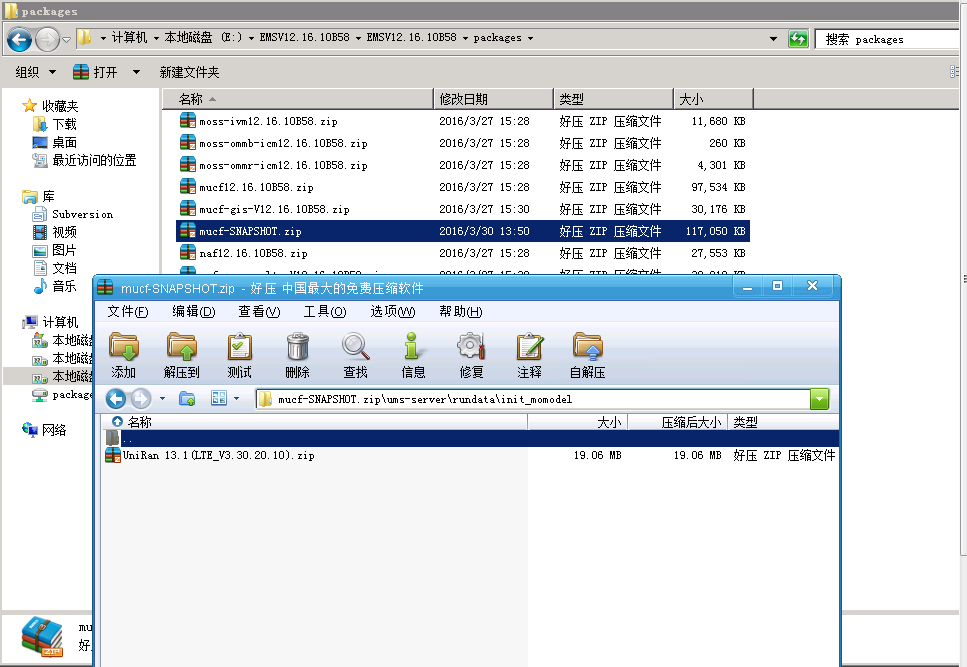
Init=”3” 表示当安装了umts**和**gsm制式时激活此模型包(必须同时安装gsm和umts，一般是多模网元模型包)

Init=”1,2” 表示当安装了umts**或**gsm制式时激活此模型包

如果没有init字段，则表示无论安装什么制式都会激活该模型包

### 初始化模型包的放置路径

初始化模型包可以作为一些安装包放在ems的package中。即xxx网元模型安装包.zip，在zip文件中放置目前的网元模型包，目录为：\ums-server\rundata\init\_momodel\xxx网元模型包.zip。可以参照如下截图：



网元模型包继续使用原来的模型包zip文件，其中LTE模型包提供plat+lte的网元模型，TL模型包提供TDS+TL双模的网元模型。

安装包的配置信息文件zipPackageName.xml，需要注意如下事项：

1、必须设定不可见，但是默认安装。id必须固定，后续不能修改。

**<zip id="ums.install.production.ne\_model.lte" visiable="false" defaultInstall="true">** 即不可见，但是默认安装。Id必须固定，后续不能再修改。

**对lte的模型安装包，id为"ums.install.production.ne\_model.lte"**

**对TL的模型安装包，id为"ums.install.production.ne\_model.tds"**

**对GU的模型安装包，id为"ums.install.production.ne\_model.gsm\_umts"**

**对cdma的模型安装包，id为"ums.install.production.ne\_model.cdma"**

2、在扁平化切换开关没有做好之前，先把zipPackageName.xml改成zipPackageName.xml.bak

以lte的模型包为例：

**<?xml version="1.0" encoding="GBK"?>**

**<zip id="ums.install.production.ne\_model.lte" visiable="false" defaultInstall="true">**

**<!-- 该zip文件的描述 -->**

**<description>**

**<locale en="lte Model Package" />**

**<locale zh="LTE模型包" />**

**</description>**

**<!-- 该zip文件所属的ppu，若一个ppu对应多个zip文件，那么这儿定义的描述最好为一致的，如果不一致则采取最先一个zip包的描述 -->**

**<owner-ppu id="MCM">**

**<description>**

**<locale en="SDR Base Station Configuration Management" />**

**<locale zh="SDR基站配置管理" />**

**</description>**

**</owner-ppu>**

**<!-- zip文件依赖的其他的zip文件，可依赖多个zip文件，该依赖决定了后面的脚本执行顺序。 -->**

**<depend>**

**<zip id="ums.install.production.uca" />**

**</depend>**

**<internaldbs>**

**</internaldbs>**

**</zip>**

## 模型库激活各模块需要实现的接口

各模块需要分别实现服务端和客户端接口.

### 服务端接口

**1.模型加载接口（必须实现）：**

**package** com.zte.ums.mucf.osf.momodel.impl;

**import** com.zte.ums.mucf.common.momodel.IModelLoader;

/\*\*

\*

\* 服务端网元模型加载接口

\* **@author** Administrator

\*

\*/

**public** **interface** IServerModelLoader **extends** IModelLoader{

/\*\*

\* 网元模型加载

\*

\* **@param** modelPath：网元模型所在路径

\* **@param** reload 是否重新加载：true重新加载或是初始化业务相关文件、数据库配置、jar包等；false将不会。

\* **@throws** Exception

\*/

**public** **void** load(String modelPath,**boolean** reload)**throws** Exception;

/\*\*

\* 获取模块名称，此名称必须与模型包中描述的模块名称一致

\* **@return** 模块名称,在com.zte.ums.mucf.common.momodel.IMoModelActivate类中有常量定义

\*/

**public** String getModule();

/\*\*

\* 配置模型已更新更新，对配置模型有依赖的模块，需要在此方法中更新配置模型

\*/

**public** **void** cmModelChanged();

}

1. 模型检查接口（选择实现）：

/\*\*

\*此接口主要用于模型包检查使用，如果对即将导入的模型包不做校验检查的话，可以不实现此接口

\* **@date** 2015年12月31日

\* **@className** ModelChecker

\*/

**public** **interface** IModelChecker **extends** Serializable {

/\*\*

\* 模型检查接口

\* **@date** 2015年12月31日

\* **@return** String：null或者""表示检查通过；其他情况为检查不通过的结果

\* **@throws** Exception 可以抛出异常

\*/

**public** String check(Map<String,Object> extend) **throws** Exception;

}

**3.各模块实现上述接口，并且在模块启动代码中向模型管理框架注册此接口：**

//以配置模块为例（没有添加模型包检查功能）：

CmServerModelLoader modelLoader = **new** CmServerModelLoader();

ModelReloadManager manager = **new** ModelReloadManager(IMoModelActivate.***MODULE\_NAME\_CM***,modelLoader);

manager.init();

//以配置模块为例（添加模型包检查功能）：

CmServerModelLoader modelLoader = **new** CmServerModelLoader();

CmModelChecker check = new CmModelChecker();//注册一个检查器

ModelReloadManager manager = **new** ModelReloadManager(IMoModelActivate.***MODULE\_NAME\_CM***,modelLoader,check);

manager.init();

### 客户端接口

1. 加载网元模型接口：

**package** com.zte.ums.mucf.common.momodel;

/\*\*

\* 网元模型加载接口

\* **@author** Administrator

\*

\*/

**public** **interface** IModelLoader {

/\*\*

\* 网元模型加载

\*

\* **@param** modelPath 网元模型所在路径

\* **@param** reload 是否重新加载 false 初始化 true 重新加载

\* **@throws** Exception

\*/

**public** **void** load(String modelPath,**boolean** reload)**throws** Exception;

/\*\*

\* 获取模块名称

\* **@return** 模块名称

\*/

**public** String getModule();

}

1. **在客户端par包下配置一个\*-model-loader-config.xml文件，将网元模型加载接口实现类注册到模型管理框架中，配置文件示例如下：**

<?xml version="1.0" encoding="UTF-8"?>

<ModelLoader>

<Loader class="com.zte.ums.mucf.wsf.momodel.cm.CmClntModelLoader" />

</ModelLoader>

1. **各模块在打开分视图时，先调用以下代码：**

com.zte.ums.mucf.wsf.momodel.main.ClntModelManager.checkAndAtivateMoModel(ClientViewContext);