## 标准SVN目录结构

Trunk ： 主干目录，此目录下的文件为基准文件  
Branches ： 用于开发的分支目录  
Tags ： 用于发布的版本目录



***这是一个标准的布局，trunk为主开发目录，branches为分支开发目录，tags为tag存档目录（不允许修改）***  
  
  
）

## SVN的工作机制在某种程度上就像一颗正在生长的树

一颗有树干和许多分支的树  
    \* 分支从树干生长出来，并且细的分支从相对较粗的树干中长出  
    \* 一棵树可以只有树干没有分支（但是这种情况不会持续很久，随着树的成长，肯定会有分支啦，^^）  
    \* 一颗没有树干但是有很多分支的树看起来更像是地板上的一捆树枝  
    \* 如果树干患病了，最终分支也会受到影响，然后整棵树就会死亡  
    \* 如果分支患病了，你可以剪掉它，然后其他分支还会生长出来的哦！  
    \* 如果分支生长太快了，对于树干它可能会非常沉重，最后整棵树会垮塌掉  
    \* 当你感觉你的树、树干或者是分支看起来很漂亮的时候，你可以给它照张相，这样就就可以记得它在那时是多么的赞。

### Trunk

　Trunk是放置稳定代码的主要环境，就好像一个汽车工厂，负责将成品的汽车零件组装在一起。

　　以下内容将告诉你如何使用SVN trunk：

* 除非你必须处理一些容易且能迅速解决的BUG，或者你必须添加一些无关逻辑的文件（比如媒体文件：图像，视频，CSS等等），否则永远不要在trunk直接做开发
* 不要因为特殊的需求而去对先前的版本做太大的改变，如何相关的情况都意味着需要建立一个branch（如下所述）
* 不要提交一些可能破坏trunk的内容，例如从branch合并
* 如果你在某些时候偶然间破坏了trunk，bring some cake the next day (”with great responsibilities come… huge cakes”)

### Branches

一个branch就是从一个SVN仓库中的子树所作的一份普通拷贝。通常情况它的工作类似与UNIX系统上的符号链接，但是你一旦在一个SVN branch里修改了一些文件，并且这些被修改的文件从拷贝过来的源文件独立发展，就不能这么认为了。当一个branch完成了，并且认为它足够稳定的时候，它必须合并回它原来的拷贝的地方，也就是说：如果原来是从trunk中拷贝的，就应该回到trunk去，或者合并回它原来拷贝的父级branch。

　　以下内容将告诉你如何使用SVN branches：

* 如果你需要修改你的应用程序，或者为它开发一个新的特性，请从trunk中创建一个新的branch，然后基于这个新的分支进行开发
* 除非是因为必须从一个branch中创建一个新的子branch，否则新的branch必须从trunk创建
* 当你创建了一个新branch，你应当立即切换过去。如果你没有这么做，那你为什么要在最初的地方创建这个分支呢？

### Tags

从表面上看，SVN branches和SVN tags没有什么差别，但是从概念上来说，它们有许多差别。其实一个SVN tags就是上文所述的“为这棵树照张相”：一个trunk或者一个branch修订版的命名快照。

　　以下内容将告诉你如何使用SVN tags：

* 作为一个开发者，永远不要切换至、取出，或者向一个SVN tag提交任何内容：一个tag好比某种“照片”，并不是实实在在的东西，tags只可读，不可写。
* 在特殊或者需要特别注意的环境中，如：生产环境（production）、？（staging）、测试环境（testing）等等，只能从一个修复过的（fixed）tag中checkout和update，永远不要commit至一个tag。
* 对于上述提及到的环境，可以创建如下的tags：“production”，“staging”，“testing”等等。你也可以根据软件版本、项目的成熟程度来命名tag：“1.0.3”，“stable”，“latest”等等。
* 当trunk已经稳定，并且可以对外发布，也要相应地重新创建tags，然后再更新相关的环境（production, staging, etc）

### 一般工作流步骤

　假设你必须添加了一个特性至一个项目，且这个项目是受版本控制的，你差不多需要完成如下几个步骤：

1. 使用SVN checkout或者SVN switch从这个项目的trunk获得一个新的工作拷贝（branch）
2. 使用SVN切换至新的branch
3. 完成新特性的开发（当然，要做足够的测试，包括在开始编码前）
4. 一旦这个特性完成并且稳定（已提交），并经过你的同事们确认，切换至trunk
5. 合并你的分支至你的工作拷贝（trunk），并且解决一系列的冲突
6. 重新检查合并后的代码
7. 如果可能的话，麻烦你的同事对你所编写、更改的代码进行一次复查（review）
8. 提交合并后的工作拷贝至trunk
9. 如果某些部署需要特殊的环境（生成环境等等），请更新相关的tag至你刚刚提交到trunk的修订版本
10. 使用SVN update部署至相关环境

## 常用两种开发模式

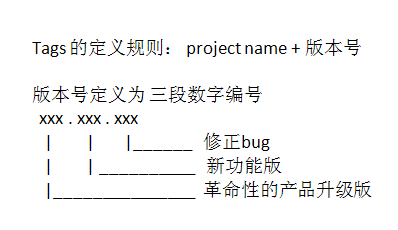
### （一）Trunk作为主要的开发目录

一般的，我们的所有的开发都是基于trunk进行开发，当一个版本/release开发告一段落（开发、测试、文档、制作安装程序、打包等）结束后，代码处于冻结状态（人为规定，可以通过hook来进行管理）。此时应该基于当前冻结的代码库，打tag。当下一个版本/阶段的开发任务开始，继续在trunk进行开发。  
此时，如果发现了上一个已发行版本（Released Version）有一些bug，或者一些很急迫的功能要求，而正在开发的版本（Developing Version）无法满足时间要求，这时候就需要在上一个版本上进行修改了。应该基于发行版对应的tag，做相应的分支（branch）进行开发。  
例如，刚刚发布1.0，正在开发2.0，此时要在1.0的基础上进行bug修正。  
按照时间的顺序  
  
1.0开发完毕，代码冻结  
基于已经冻结的trunk，为release1.0打tag  
此时的目录结构为  
svn://proj/  
             +trunk/ (freeze)  
             +branches/  
             +tags/  
                     +tag\_release\_1.0　(copy from trunk)  
2.0开始开发，trunk此时为2.0的开发版  
发现1.0有bug，需要修改，基于1.0的tag做branch  
此时的目录结构为  
svn://proj/  
             +trunk/ ( dev 2.0 )  
             +branches/  
                           +dev\_1.0\_bugfix (copy from tag/release\_1.0)  
             +tags/  
                     +release\_1.0　(copy from trunk)  
在1.0 bugfix branch进行1.0 bugfix开发，在trunk进行2.0开发  
在1.0 bugfix 完成之后，基于dev\_1.0\_bugfix的branch做release等  
根据需要选择性的把dev\_1.0\_bugfix这个分支merge回trunk（什么时候进行这步操作，要根据具体情况）  
这是一种很标准的开发模式，很多的公司都是采用这种模式进行开发的。trunk永远是开发的主要目录

### （二）在每一个release的branch中进行各自的开发，trunk只做发布使用

这种开发模式当中，trunk是不承担具体开发任务的，一个版本/阶段的开发任务在开始的时候，根据已经release的版本做新的开发分支，并且基于这个分支进行开发。还是举上面的例子，这里面的时序关系是。  
  
1.0开发，做dev1.0的branch  
此时的目录结构  
svn://proj/  
             +trunk/ (不担负开发任务 )  
             +branches/  
                           +dev\_1.0 (copy from trunk)  
             +tags/  
1.0开发完成，merge dev1.0到trunk  
此时的目录结构  
svn://proj/  
             +trunk/ (merge from branch dev\_1.0)  
             +branches/  
                           +dev\_1.0 (开发任务结束，freeze)  
             +tags/  
根据trunk做1.0的tag  
此时的目录结构  
svn://proj/  
             +trunk/ (merge from branch dev\_1.0)  
             +branches/  
                           +dev\_1.0 (开发任务结束，freeze)  
             +tags/  
                     +tag\_release\_1.0 (copy from trunk)  
1.0开发，做dev2.0分支  
此时的目录结构  
svn://proj/  
             +trunk/  
             +branches/  
                           +dev\_1.0 (开发任务结束，freeze)  
                           +dev\_2.0 （进行2.0开发）  
             +tags/  
                     +tag\_release\_1.0 (copy from trunk)  
1.0有bug，直接在dev1.0的分支上修复

## 新版本的定义规则



**Branch的定义规则：projectname + bugName/requirement +日期（开发人员注意的）**

## 常用版本号的意义：

Alpha：是内部测试版,一般不向外部发布,会有很多Bug.一般只有测试人员使用。  
  
Beta：也是测试版，这个阶段的版本会一直加入新的功能。在Alpha版之后推出。  
  
RC：(Release　Candidate) 顾名思义么 ! 用在软件上就是候选版本。系统平台上就是发行候选版本。RC版不会再加入新的功能了，主要着重于除错。  
  
GA :General Availability,正式发布的版本，在国外都是用GA 来说明release版本的。  
  
RTM：(Release to Manufacture)是给工厂大量压片的版本，内容跟正式版是一样的，不过RTM版也有出限制、评估版的。但是和正式版本的主要程序代码都是一样的。  
  
OEM：是给计算机厂商随着计算机贩卖的，也就是随机版。只能随机器出货，不能零售。只能全新安装，不能从旧有操作系统升级。包装不像零售版精美，通常只有一面CD和说明书(授权书)。   
  
RVL：号称是正式版，其实RVL根本不是版本的名称。它是中文版/英文版文档破解出来的。   
  
EVAL：而流通在网络上的EVAL版，与“评估版”类似，功能上和零售版没有区别。   
  
RTL：Retail(零售版)是真正的正式版，正式上架零售版。在安装盘的i386文件夹里有一个eula.txt，最后有一行EULAID，就是你的 版本。比如简体中文正式版是EULAID:WX.4\_PRO\_RTL\_CN，繁体中文正式版是WX.4\_PRO\_RTL\_TW。其中：如果是WX.开头是 正式版，WB.开头是测试版。\_PRE，代表家庭版；\_PRO，代表专业版。  
  
α、β、λ常用来表示软件测试 过 程中的三个阶段，α是第一阶段，一般只供内部测试使用；β是第二个阶段，已经消除了软件中大部分的不完善之处，但仍有可能还存在缺陷和漏洞，一般只提供给 特定的用户群来测试使用；λ是第三个阶段，此时产品已经相当成熟，只需在个别地方再做进一步的优化处理即可上市发行。

## 首金版本控制规则

### POM文件统一更改（添加jar包的时候，优先保持和整体框架的一致）

### 每次大的发布之后，打包tag，然后部署到线上测试环境。

### 开发环境打的包将全部是snapshot版本，阿里测试上打的包全是release版本。开发环境打包路径是直接从trunk中打包，阿里测试环境中的发布路径是从tag中获取。每次打包都需要修改。

## 新人开发环境的搭建：

### 从trunk下面直接checkout源码：

内网：<https://192.168.20.2:10096/svn/sjwwd/sjpt/SMF/trunk>

外网：<https://1.202.226.243:10096/svn/sjwwd/sjpt/SMF/trunk>

### 然后clean，install即可直接运行（SMF、CnpaySmartSchedule父节点， maven常用使用是否有疑问）。

### 通常情况下大家开发的时候都默认连接开发环境，轻易不连接阿里测试环境数据库（即pom中默认配置环境）。

## 测试驱动开发

<http://www.infoq.com/cn/presentations/zxq-tdd/>

<http://www.ibm.com/developerworks/cn/linux/l-tdd/>

<http://baike.baidu.com/link?url=epmWhC3wuQdtAuaOoFrNL45l20qMZDCiK8YYRvey5Cub2fwyeJ4dDf4CSa4iSvkJFFwZBGw60ZYODLTtC49Sja>

http://www.csdn.net/article/2013-03-18/2814535-unit-tests

### Tomcat源码参考（测试目录和开发目录一致，只是测试类名多了一个Test）。

### 单元测试常用框架及规范（Junit，EasyMock）

#### 测试类命名是java文件名称+Test

@RunWith(SpringJUnit4ClassRunner.**class**)

"transactionManager", defaultRollback = **true**)

**public** **class** CmbcServiceImplTest {}

#### 测试方法名是：test +　待测试方法名

@Test

public void testPayment() throws Exception {

P2pT000016 vo = new P2pT000016();

vo.setBusinCode("102");

vo.setPayFundAcc("9595100003086016");// 付款人账户

vo.setPrdCode("JKSQ201903290004DK");

vo.setAmt("20.3");

vo.setCurrType("156");

vo.setSubCode("007");

vo.setSummary("执行多余的逾期管理费打给平台 操作");

TransInput ti = new TransInput("E012016010400527", "9595100003086016", vo.getTcode(), "1181550");

ti.setReserve2("JY201604011903380020trade11");

vo.setTransInput(ti);

Object sendCmbc = httpUtils.sendCmbc(vo);

System.out.println(sendCmbc);

}

### 那些类及方法需要写单元测试

### 私有方法写单元测试(TOPIC)

#### 测试中的bad smell

● 一个方法中有太多test case——被测试的方法做了太多事情。

● 太多的setup/teardown——表示被测试类的耦合性太高。

● 改变一个地方，多处测试受影响——也许是测试的设计问题，也许是实现代码中有过多依赖。

● 测试上下文中有太多依赖——设计中的耦合性太高。

● 测试运行速度缓慢——表示你的单元测试也许在使用外部系统，例如网络、数据库、文件系统等等。通常也意味着被测试类有过多的职责。

为了测试的目的，把成员变量或者方法的访问权限变成protected或者public——可能是因为测试代码跟被测试的代码耦合太高，也可能是本来私有的东西有太多行为，这种情况下应该考虑把它抽出来作为独立的对象。

#### 怎么为私有方法写单元测试？这个难题由来已久了。

有人会选择跳过私有方法，有人会选择去掉private限定符；跳过私有方法自然不是良策，但提升访问权限也会破坏封装，在Naresh Jain看来也算得上是bad smell，那怎么解决才好？

#### 应该为私有方法添加测试么？

在成功的用了TDD或者测试驱动重构（Test-Driven Refactoring）以后，你的代码中就不会出现针对私有方法的测试。

如果你用TDD编写全新的代码，在没有测试之前是没有功能的。私有方法是到了重构那一步的最后才会出现。把代码转移到私有方法中的这个过程，已经被先前写过的测试覆盖到了。所以，如果你成功了用了TDD，代码中就不会出现针对私有方法的测试。

如果你在改善遗留代码，你就该使用测试驱动重构。这样的话，可能会临时针对私有方法写一些测试。但是，随着测试覆盖率的增加，那些public方法的测试会覆盖到所有的路径，也包括了私有方法的调用。所以，你也不再需要测试私有方法。

#### 如果真有需要测试私有方法，则可以使用反射

@SuppressWarnings("deprecation")

@Test

public void test1() throws Exception {

CmbcServiceImpl serviceImpl = new CmbcServiceImpl();

Class class1 = serviceImpl.getClass();

try {

Method getDateWeeHours = class1.getDeclaredMethod("getDateWeeHours");

getDateWeeHours.setAccessible(true);//设为可见

Date result = (Date)getDateWeeHours.invoke(serviceImpl);

// Assert.assertEquals(expect, result);

System.out.println(result);

} catch (Exception e) {

e.printStackTrace();

}

}

### 测试是针对函数写测试用例吗？如果定义单元测试粒度。

### 测试覆盖率（TOPIC）

覆盖率是度量测试完整性的一个手段，是测试有效性的一个度量； 测试覆盖是由测试需求和[测试用例](http://baike.baidu.com/view/106882.htm)的覆盖或已执行代码的覆盖表示的。

简而言之，测试覆盖是就需求（基于需求的）或代码的设计/实施标准（基于代码的）而言的完全程度的任意评测，如[用例](http://baike.baidu.com/view/706238.htm)的核实（基于需求的）或所有代码行的执行（基于代码的）

# [EasyMock的使用](http://www.cnblogs.com/zhongjinbin/archive/2012/01/31/2333747.html)

### 1. EasyMock介绍

Mock 方法是单元测试中常见的一种技术，它的主要作用是模拟一些在应用中不容易构造或者比较复杂的对象，从而把测试与测试边界以外的对象隔离开。同时也可以当调用别人的模块，而该模块又没有实现时（只提供接口），我们可以在独立的环境中测试自己的模块逻辑。

### 2.使用前的准备

*下载所需的jar包：easymock-3.0.jar（或以上版本），junit-4.4.jar，cglib-nodep-2.1\_3.jar*

### 3.使用方法较简单。主要有以下步骤：

*\*•使用 EasyMock 生成 Mock 对象；  
 \*•设定 Mock 对象的预期行为和输出；  
 \*•将 Mock 对象切换到 Replay 状态；  
 \*•调用 Mock 对象方法进行单元测试；  
 \*•对 Mock 对象的行为进行验证。*