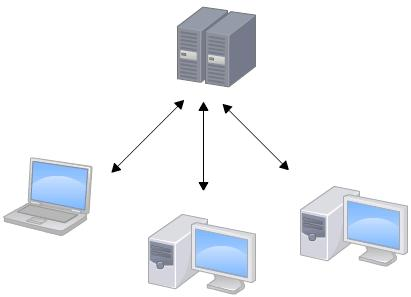
### ****一、 集中式vs分布式****

**1. Subversion属于集中式的版本控制系统**  
集中式的版本控制系统都有一个单一的集中管理的服务器，保存所有文件的修订版本，而协同工作的人们都通过客户端连到这台服务器，取出最新的文件或者提交更新。



**Subversion的特点概括起来主要由以下几条：**

* 每个版本库有唯一的URL（官方地址），每个用户都从这个地址获取代码和数据；
* 获取代码的更新，也只能连接到这个唯一的版本库，同步以取得最新数据；
* 提交必须有网络连接（非本地版本库）；
* 提交需要授权，如果没有写权限，提交会失败；
* 提交并非每次都能够成功。如果有其他人先于你提交，会提示“改动基于过时的版本，先更新再提交”… 诸如此类；
* 冲突解决是一个提交速度的竞赛：手快者，先提交，平安无事；手慢者，后提交，可能遇到麻烦的冲突解决。

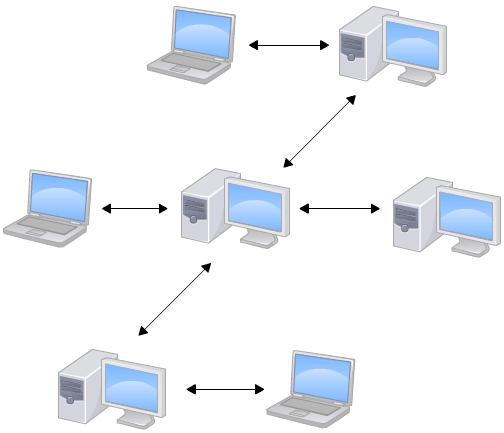
好处：每个人都可以一定程度上看到项目中的其他人正在做些什么。而管理员也可以轻松掌控每个开发者的权限。

缺点：中央服务器的单点故障。

若是宕机一小时，那么在这一小时内，谁都无法提交更新、还原、对比等，也就无法协同工作。如果中央服务器的磁盘发生故障，并且没做过备份或者备份得不够及时的话，还会有丢失数据的风险。最坏的情况是彻底丢失整个项目的所有历史更改记录，被客户端提取出来的某些快照数据除外，但这样的话依然是个问题，你不能保证所有的数据都已经有人提取出来。

Subversion原理上只关心文件内容的具体差异。每次记录有哪些文件作了更新，以及都更新了哪些行的什么内容。

**2. Git属于分布式的版本控制系统**

****

Git记录版本历史只关心文件数据的整体是否发生变化。Git 不保存文件内容前后变化的差异数据。

实际上，Git 更像是把变化的文件作快照后，记录在一个微型的文件系统中。每次提交更新时，它会纵览一遍所有文件的指纹信息并对文件作一快照，然后保存一个指向这次快照的索引。为提高性能，若文件没有变化，Git 不会再次保存，而只对上次保存的快照作一连接。

在分布式版本控制系统中，客户端并不只提取最新版本的文件快照，而是把原始的代码仓库完整地镜像下来。这么一来，任何一处协同工作用的服务器发生故障，事后都可以用任何一个镜像出来的本地仓库恢复。这类系统都可以指定和若干不同的远端代码仓库进行交互。籍此，你就可以在同一个项目中，分别和不同工作小组的人相互协作。你可以根据需要设定不同的协作流程。

另外，因为Git在本地磁盘上就保存着所有有关当前项目的历史更新，并且Git中的绝大多数操作都只需要访问本地文件和资源，不用连网，所以处理起来速度飞快。用SVN的话，没有网络或者断开VPN你就无法做任何事情。但用Git的话，就算你在飞机或者火车上，都可以非常愉快地频繁提交更新，等到了有网络的时候再上传到远程的镜像仓库。换作其他版本控制系统，这么做几乎不可能，抑或是非常麻烦。

**Git具有以下特点：**

Git中每个克隆(clone)的版本库都是平等的。你可以从任何一个版本库的克隆来创建属于你自己的版本库，同时你的版本库也可以作为源提供给他人，只要你愿意。

Git的每一次提取操作，实际上都是一次对代码仓库的完整备份。

提交完全在本地完成，无须别人给你授权，你的版本库你作主，并且提交总是会成功。

甚至基于旧版本的改动也可以成功提交，提交会基于旧的版本创建一个新的分支。

Git的提交不会被打断，直到你的工作完全满意了，PUSH给他人或者他人PULL你的版本库，合并会发生在PULL和PUSH过程中，不能自动解决的冲突会提示您手工完成。

冲突解决不再像是SVN一样的提交竞赛，而是在需要的时候才进行合并和冲突解决。

Git 也可以模拟集中式的工作模式

Git版本库统一放在服务器中

可以为 Git 版本库进行授权：谁能创建版本库，谁能向版本库PUSH，谁能够读取（克隆）版本库

团队的成员先将服务器的版本库克隆到本地；并经常的从服务器的版本库拉（PULL）最新的更新；

团队的成员将自己的改动推（PUSH）到服务器的版本库中，当其他人和版本库同步（PULL）时，会自动获取改变

Git 的集中式工作模式非常灵活

你完全可以在脱离Git服务器所在网络的情况下，如移动办公／出差时，照常使用代码库

你只需要在能够接入Git服务器所在网络时，PULL和PUSH即可完成和服务器同步以及提交

Git提供 rebase 命令，可以让你的改动看起来是基于最新的代码实现的改动

Git 有更多的工作模式可以选择，远非 Subversion可比

**二、 版本库与工作区**

Subversion的工作区和版本库是截然分开的，而Git的工作区和版本库是如影随形的。

**1. SVN的版本库和工作区是分离的**  
• Subversion 的工作区和版本库物理上分开：Subversion的版本库和工作区是存储在不同路径下，一般是在不同的主机中，Subversion的企业级部署中，版本库在服务器上，只能通过 https, http, svn 等协议访问，而不能直接被用户接触到。  
• Subversion的工作区是一份版本库在某个历史状态下的快照，如：版本库最新的数据检出到工作区。  
• Subversion的工作区中每一个目录下都包含一个名为 .svn 的控制目录（隐藏的目录），该目录的作用是：  
① 标识工作区和版本库的对应关系。  
② 包含一份该子目录下检出文件的原始拷贝。当文件改动的差异比较或者本地改动的回退时，可以直接参考原始拷贝而无须通过网络访问远程版本库。  
• Subversion 的 .svn 控制目录会引入很多麻烦：  
① .svn 下的文件原始考本，会导致在目录下按照文件内容搜索时，多出一倍的搜索时间和搜索结果。  
② .svn 很容易在集成时，引入产品中，尤其是 Web 应用，将 .svn 目录带入Web服务器会导致安全隐患。因为一个不允许目录浏览的Web目录，可以通过 .svn/entries 文件查看到该目录下可能存在的文件。

**2 .Git 的版本库和工作区如影随形**

• Git 的版本库和工作区在同一个目录下，工作区的根目录有一个.git的子目录，这个名为 .git的目录就是版本库本身，它是Git 用来保存元数据和对象数据库的地方。该目录非常重要，每次克隆镜像仓库的时候，实际拷贝的就是这个目录里面的数据。所以千万要小心删除这个文件。  
• 工作区中其他文件为工作区文件，可能是从 .git 中检出的，或者是要检入的，或者是运行产生的临时文件等。

• 版本库可以脱离工作区而存在，成为 bare（赤裸）版本库。可以用 –bare 参数来创建。但是工作区不能脱离版本库而存在，即工作区的根目录下必须有一个名为 .git 的版本库克隆文件。  
• Git 的版本库因为就在工作区中，能直接被用户接触到。  
① 用户可以编辑 .git/config 文件，修改配置，增添新的源  
② 用户可以编辑 .git/info/exclude 文件，创建本地忽略…

• Git 的工作区中只在工作区的根目录下有一个 .git 目录，此外再无任何控制目录。Git 工作区下唯一的 .git 目录是版本库，并非 .svn 的等价物，如果删除了 .git 目录，而又没有该版本库的其他镜像（克隆）的话，你破坏了整个历史，版本库也永远的失去了。  
• Git 在本地的 .git 版本库，提供了完全的改动历史。除了和其他人数据交换外，任何版本库相关的操作都在本地完成，更多的本地操作，避免了冗长的网络延迟，大大节省了时间。例如：查看 log，切换到任何历史版本等操作都无须连接网络。

• Git如何保证安全：本地创建一个Git库，因为工作区和库是在同一个目录中，如果工作区删除了，或者所在的磁盘分区格式化了，数据不是全都没有了么？其实我们可以这样做：  
① 在一个磁盘分区中创建版本库（最好是用 –bare 参数创建），然后在另外的磁盘分区中克隆一个新的作为工作区。在工作区的提交要不时的PUSH到另外分区的版本库，这样就实现了本地的数据镜像。你甚至可以在本地创建更多的版本库镜像，安全性要比Subversion的一个库加上一个工作区安全。  
② 另一个办法：把你的版本库共享给他人，当他人克隆了你的版本库时，你就拥有了一个异地备份。

**三、 全局版本号和全球版本号**

SVN的全局版本号和CVS的每个文件都独立维护一套版本号相比，是一个非常大的进步。在看似简单的全局版本号的背后，是Subversion提供对于事物处理的支持，每一个事物处理（即一次提交）都具有整个版本库全局唯一的版本号。

Git的版本号则更进一步，版本号是全球唯一的。Git 对于每一次提交，通过对文件的内容或目录的结构计算出一个SHA-1 哈希值，得到一个40位的十六进制字符串，Git将此字符串作为版本号。

**1. SVN与Git版本号比较**

• 所有保存在Git 数据库中的数据都是用此40位的哈希值作索引的，而不是靠文件名。  
• 使用哈希值作版本号的好处就是对于一个分布式的版本控制系统，每个人每次提交后形成的版本号都不会出现重复。另一好处是保证数据的完整性，因为哈希值是根据内容或目录结构计算出来的，所以我们还可以据此来判断数据内容是否被篡改。

• SVN 的版本号是连续的，可以预判下一个版本号，而 Git 的版本号则不是。

因为 subversion 是集中式版本控制，很容易实现版本号的连续性。Git 是分布式的版本控制系统，而且 Git 采用 40 位长的哈希值作为版本号，每个人的提交都是各自独立完成的，没有先后之分（即使提交有先后之分，也由于PUSH/PULL的方向和时机而不同）。Git 的版本号虽然不连续，但是是有线索的，即每一个版本都有对应的父版本（一个或者两个），进而可以形成一个复杂的提交链

• Git 的版本号简化：Git 可以使用从左面开始任意长度的字串作为简化版本号，只要该简化的版本号不产生歧义。一般采用7位的短版本号（只要不会出现重复的，你也可以使用更短的版本号）。

**四、 部分检出**

Subversion可以将整个库检出到工作区，也可以将某个目录检出到工作区。对于要使用一个庞大、臃肿的版本库的用户来说，部分检出是非常方便和实际的。  
但是Git只能全部检出，不支持按照目录进行的部分检出。

**1. SVN的部分检出**

• 在SVN中，从仓库checkout的一个工作树，每个子目录下都维护着自己的.svn目录，记录着该目录中文件的修改情况以及和服务器端仓库的对应关系。所以SVN可以checkout部分路径下的内容（部分检出），而不用checkout整个版本库或分支。

• Subversion 有一条命令：svn export ，可以将 subversion 版本库的一个目录下所有内容导出到指定的目录下。Subversion 需要 svn export 命令是因为该命令可以导出一个干净的目录，即不包含 .svn 目录（包含配置文件和文件原始拷贝）。

**2. Git的检出**

• Git 没有部分检出，这并不是说只有将整个库克隆下来才能查看文件。有很多 git 工具，提供直接浏览git库的功能，例如 gitweb, trac 的 git 版本库浏览, redmine 的 git 版本库浏览。

• Git-submodule 可以实现版本库的模块化：Git 通过子模块处理这个问题。  
子模块允许你将一个Git 仓库当作另外一个Git仓库的子目录。这允许你克隆另外一个仓库到你的项目中并且保持你的提交相对独立。

• Git 为什么没有实现 svn export 的功能？由于git的本地仓库信息完全维护在project根目录的.git目录下，（不像svn一样，每个子目录下都有单独的.svn目录）。所以，只要clone，checkout然后删除.git目录就可以了。

**五、 更新和提交**

**1．更新操作**

在SVN中，因为只有一个中心仓库，所以所谓的远程更新，也就是svn update ,通过此命令来使工作区和版本库保持同步。  
对于git来说，别人的改动是存在于远程仓库上的，所以git checkout命令尽管在某些功能上和svn中的update类似（例如取仓库特定版本的内容），但是在远程更新这一点上，还是不同的，不属于git checkout的功能涵盖范围。 Git使用git fetch和git pull来完成远程更新任务，fetch操作只是将远程数据库的object拷贝到本地，然后更新remotes head的refs，git pull 的操作则是在git fetch的基础上对当前分支外加merge操作。

**2．SVN中的commit命令**

对于SVN来说，由于是中心式的仓库管理形式，所以并不存在特殊的远程提交的概念，所有的commit操作都可以认为是对远程仓库的更新动作。在工作区中对文件进行添加、修改、删除操作要同步到版本库，必须使用 commit命令。  
• add 命令，是将未标记为版本控制状态的文件标记为添加状态，并在下次提交时入库。  
• delete命令，是通过SVN来删除文件，并在下次提交后有效。  
• Subversion 有提交列表功能，即将某些文件加入一个修改列表，提交可以只提交处于该列表的文件。

**3．Git中的暂存区域（stage）**

Git 管理项目时，文件在三个工作区域中流转：Git 的本地数据目录，工作目录以及暂存区域。暂存区域（stage）是介于 workcopy 和 版本库  HEAD 版本的一种中间状态。所谓的暂存区域只不过是个简单的文件，一般都放在git 目录中。有时候人们会把这个文件叫做索引文件，不过标准说法还是叫暂存区域。

要将一个文件纳入版本管理的范畴，首先是要用git  add将文件纳入stage的监控范围，只有更新到stage中的内容才会在commit的时候被提交。另外，文件本身的改动并不会自动更新到stage中，每次的任何修改都必须重新更新到stage中去才会被提交。对于工作区直接删除的文件，需要用 git rm 命令进行标记，在下次提交时，在版本库中删除。

• 工作区的文件改动（新增文件，修改文件，删除文件），必须用 git add 或者 git rm 命令标识，使得改动进入 stage  
• 提交只对加入 stage 的改动进行提交  
• 如果一个文件改动加入 stage 后再次改动，则后续改动不改变 stage。即该文件的改动有两个状态，一个是标记到 stage 中并将在下次提交时入库的改动，另外的后续改动则不被提交，除非再次使用 git add 命令将改动加入到 stage 中。  
• Git的stag让你在提交的时候清楚的知道git将要提交哪些改动。除非提交的时候使用 -a 参数（不建议使用）。

我们可以从文件所处的位置来判断其状态：如果是git目录中保存着的特定版本文件，就属于已提交状态；如果作了修改并已放入暂存区域，就属于已暂存状态；如果自上次取出后，作了修改但还没有放到暂存区域，就是已修改状态，如果取出后未进行修改则是未修改状态。

在git中，因为有本地仓库和remote仓库之分，所以也就区别于commit 操作，存在额外的push命令，用于将本地仓库的数据更新到远程仓库中去。git push 可以选择需要提交的、更新的分支以及制定该分支在远程仓库上的名字。

**六、 分支和里程碑的实现**

几乎每一种版本控制系统都以某种形式支持分支。使用分支意味着你可以从开发主线上分离开来，然后在不影响主线的同时继续工作。在很多版本控制系统中，这是个昂贵的过程，常常需要创建一个源代码目录的完整副本，对大型项目来说会花费很长时间。

轻量级分支/里程碑的含义是，创建分支/里程碑的复杂度是o(1)，不会因为版本库的愈加庞大而变得缓慢。在CVS中，创建分支的复杂度是o(n)的，导致大的版本库的的分支创建非常缓慢。

**1．Subversion的分支/里程碑**

Subversion轻量级分支和里程碑的实现是通过svn cp命令，即带历史的拷贝就是创建快速创建分支和里程碑的秘籍。Subversion的版本库有特殊的设计，当你复制一个目录，你不需要担心版本库会变得十分巨大—Subversion并不是拷贝所有的数据，相反，它只是建立了一个已存在目录树的入口。这种“廉价的拷贝”就是创建分支/里程碑是轻量级的原因。  
由于Svn的分支和标签是来自目录拷贝，约定俗成是拷贝在 branches/和tags/目录下。所谓分支，tag等概念都只是仓库中不同路径上的一个对象或索引而已，和普通的路径并没有什么本质的区别，谁也不能阻止在一个提交中同时修改不同分支中的数据。  
里程碑是对某个历史提交所起的一个别名，作为历史的标记，是不应该被更改的。svn的里程碑要建立到 tags/目录下，要求不要在tags/下的里程碑目录下进行提交。但是谁也阻止不了对未进行权限控制的里程碑的篡改。

**2．Git 的轻量级分支和里程碑**

Git中的分支实际上仅是一个包含所指对象校验和（40个字符长度SHA-1 哈希值）的文件，所以创建和销毁一个分支就变得非常廉价。说白了，新建一个分支就是向一个文件写入41个字节（版本号外加一个换行符）那么简单，自然速度就很快了。 Git的实现与项目复杂度无关，它永远可以在几毫秒的时间内完成分支的创建和切换。这和大多数版本控制系统形成了鲜明对比。

Git的分支是完全隔离的，而Subversion则没有。分支本来就应该是相对独立的命名空间，一个提交一般只能发生在一个分支中。在Git中，其内部的对象层级依赖关系或许和SVN类似，但是其工作树的视图表现形式和SVN完全不同。工作树永远是一个完整的分支，不同的分支由不同的head索引去构建，你不可能在工作树中同时获得多个分支的内容。

Git使用的标签有两种类型：轻量级的（lightweight）和含附注的（annotated）。① 轻量级标签就像是个不会变化的分支，实际上它就是个指向特定提交对象的引用。② 而含附注标签，实际上是存储在仓库中的一个独立对象，它有自身的校验和信息，包含着标签的名字，电子邮件地址和日期，以及标签说明，标签本身也允许使用GNU Privacy Guard (GPG) 来签署或验证。

Git的里程碑是只读的，Git完全遵守历史不可更改这一时空法则。用户不能向git的里程碑中提交，否则里程碑就不是标记，而成了一个分支。当然Git允许用户删除里程碑再重新创建指定到不同历史提交。

**3．多分支间的切换**

SVN中提供了一个功能switch，使用switch可以在同一个工作树上，在不同的分支中进行切换。  
Git在分支中进行切换使用的命令是checkout。

**七、 分支与合并**

Git 和 Svn 的分支实现机制完全的不同，这也直接导致了 SVN 在分支合并中困难重重。尽管在 SVN 1.5 之后，通过 svn:mergeinfo 属性引入了合并追踪机制，但是在特定情况下，合并仍会出现很多困难。

**1． SVN的分支合并**

当你在一个分支上工作数周或几个月之后，主干的修改也同时在进行着，两条线的开发会区别巨大，当你想合并分支回主干，可能因为太多冲突，已经无法轻易合并你的分支和主干的修改。

另一个问题，Subversion不会记录任何合并操作，当你提交本地修改，版本库并不能判断出你是通过svn merge还是手工修改得到这些文件。所以你必须手工记录这些信息（说明合并的特定版本号或是版本号的范围）。

要解决以上的问题只有通过有规律的将主干合并到分支来避免，制定这样一个政策：每周将上周的修改合并到分支，注意这样做时需要小心，你必须手工记录合并的过程，以避免重复的合并，你需要小心的撰写合并的日志信息，精确的描述合并包括的范围。这样做看起来有点像是胁迫。

SVN 的版本号是连续的版本号。每一次新的提交都会版本号+1 ，而无论这个提交是在哪个分支中进行的。SVN一个提交可以同时修改不同分支的不同文件，因为提交命令可以在 /trunk, /branches, /tags 的上一级目录执行。

• SVN 的提交是单线索的，每一个提交（最原始的提交0除外）都只有一个父节点（版本号小一个的提交节点）  
• SVN 的提交链只有一条，仅从版本号和提交说明，我们无法获得分支图  
• SVN 的分支图在某些工具（如乌龟SVN）可以提供，那是需要对提交内容进行检查，对目录拷贝动作视为分支，对 svn:mergeinfo 的改动视为合并，但这会由于目录管理的灵活性，导致千奇百怪的分支图表。

**2．Git的分支合并**

在 git 版本库中创建分支的成本几乎为零，所以，不必吝啬多创建几个分支。当第一次执行git-init时，系统就会创建一个名为”master”的分支。 而其它分支则通过手工创建。下面列举一些常见的分支策略。

① 创建一个属于自己的个人工作分支，以避免对主分支 master 造成太多的干扰，也方便与他人交流协作。   
② 当进行高风险的工作时，创建一个试验性的分支，扔掉一个烂摊子总比收拾一个烂摊子好得多。   
③ 合并别人修改的时候，最好创建一个临时的分支用来合并，合并完成后再“fatch”到自己的分支。

Git分支相关的操作命令

**八、 撤消操作**

**1．提交的撤销**

在Subversion中一旦完成向服务器的数据提交，你就没有办法再从客户端追回，只能在后续的提交中修正（回退或者修改）等。因为Subversion作为集中式的版本控制，不能允许个人对已提交的数据进行篡改。Subversion具有一个非常重要的特性就是它的信息从不丢失，即使当你删除了文件或目录，它也许从最新版本中消失了 ，但这个对象依然存在于历史的早期版本中。  
Git则不同，Git是分布式版本控制系统，代码库是属于个人，允许任意修改。Git通过对提交建立数字摘要来保证提交的唯一性和不可更改性，通过版本库在多人之间的多份拷贝来保障数据的安全性。Git可以丢弃最新的一个或几个提交，使用 git reset –hard命令可以永远丢弃最新的一个或者几个提交。

**2．提交说明的修改**

提交后如果对提交说明不满意，如何实现对提交说明的修改：  
⑴ Git可以使用命令git commit –amend修改提交说明。  
• Git可以修改最后一次提交说明，并不是说不能修改历史版本的提交说明，只是修改最后一个版本提交说明拥有最简单的命令；  
• Git修改提交说明，会改变提交的commit-id。即修改提交说明后，将产生一个新的提交；  
• Git可以通过git reset –hard ，git commit –amend，git rebase onto 等命令来实现对历史提交的修改；  
• 使用stg工具可以更为简单的修改历史提交的提交说明，包括提交内容；  
⑵ Subversion也可以修改提交说明，是通过修改提交的svn:log版本属性实现的：  
• 不但可以修改最后一次提交的说明，并且可以修改历史提交的提交说明；  
• Subversion修改提交说明是不可逆的操作，可能会造成说明被恶意修改；  
• Subversion缺省关闭修改提交说明的功能。管理员在设置了提交说明更改的邮件通知后，才可以打开该功能。

**3．修改和重构历史提交**

Git可以修改和重构历史提交：使用Git本身的reset以及 rebase 命令可以修改或者重整/重构历史提交，非常灵活。使用强大的 stg 可以使得历史提交的重构更为简洁，如果您对 stg 或者 Hg/MQ 熟悉的话。  
Subversion 修改历史提交，只能由管理员完成。  
Subversion 是集中式版本控制系统，从客户端一旦完成提交，就没有办法从客户端撤销提交。但是管理员可以在服务器端完成提交的撤销和修改，但是操作过程和代价较大。

**九、 权限管理**

Subversion通过对文件目录授权来实现权限管理，子目录默认继承父目录的权限。但是也有缺憾，即权限不能在分支中继承，不能对单个文件授权。例如为 /trunk及其子目录的授权，不能继承到分支或者标签中相应的目录下。

Git 的授权做不到Subversion那样精细。Git的授权模型只能实现非零即壹式的授权，要么拥有全部的写权限，要么没有写权限，要么拥有整个版本库的读权限，要么禁用。

从技术上将，Git可能永远也做不到类似SVN的路径授权（读授权）：

• 如果允许按照路径授权，则各个克隆的关系将不再是平等的关系，有的内容多，有的内容少，分布式的理念被破坏

• 如果只有部分路径可读，则克隆出来的提交和原始提交的提交ID可能不同。因为提交ID是和提交内容有关的，克隆中提交的部分内容被丢弃，势必提交的ID也要重新计算

• 允许全部代码可读，只允许部分代码可写，在版本控制的管理下，是没有多大实际意义的，而且导致了提交的逻辑上的不完整。

那么有什么办法来解决授权的问题？

1. 公司内部代码开放。即代码在公司内部，对项目组成员一视同仁的开放。

2. 公司对代码库进行合理分解，对每个代码库分别授权。即某个代码库对团队成员完全开放，对其它团队完全封闭。

3. 公司使用Subversion做集中式的版本控制，个人和/或团队使用 Git-svn。这样在无法改变公司版本控制策略时，程序员可以采用的变通之法。

4. Git服务器的部署实际上可以使用钩子对分支和路径进行写授权，即可以控制谁能够创建分支，能够写特定文件。

**十、优缺点比较**

**1．SVN优缺点**

优点：

1、 管理方便，逻辑明确，符合一般人思维习惯。

2、 易于管理，集中式服务器更能保证安全性。

3、 代码一致性非常高。

4、 适合开发人数不多的项目开发。

缺点：

1、 服务器压力太大，数据库容量暴增。

2、 如果不能连接到服务器上，基本上不可以工作，看上面第二步，如果服务器不能连接上，就不能提交，还原，对比等等。

3、 不适合开源开发（开发人数非常非常多，但是Google app engine就是用svn的）。但是一般集中式管理的有非常明确的权限管理机制（例如分支访问限制），可以实现分层管理，从而很好的解决开发人数众多的问题。

**2．Git优缺点**  
优点：

1、适合分布式开发，强调个体。

2、公共服务器压力和数据量都不会太大。

3、速度快、灵活。

4、任意两个开发者之间可以很容易的解决冲突。

5、离线工作。

缺点：

1、学习周期相对而言比较长。

2、不符合常规思维。

3、代码保密性差，一旦开发者把整个库克隆下来就可以完全公开所有代码和版本信息。