第 1 篇：初识 Git

Git 是一款分布式版本控制系统，有别于像 CVS 和 SVN 那样的集中式版本控制系统，Git 可以让研发团队更加高效地协同工作，从而提高生产率。使用 Git，开发人员的工作不会因为频繁地遭遇提交冲突而中断，管理人员也无须为数据的备份而担心。有着像 Linux 这样的庞大项目的考验，Git 可以胜任任何规模的团队，即便这个团队分布于世界各地。

Git 是开源社区送给每一个人的宝贝，用好它可以实现个人的知识积累、保护好自己的数据，以及和他人分享自己的成果。这在其他的很多版本控制系统中是不可想象的。试问你会仅仅为个人的版本控制而花费高昂的费用去购买商业版本控制工具么？你会去使用必须搭建额外的服务器才能使用的版本控制系统么？你会把“鸡蛋”放在具有单点故障、服务器软硬件有可能崩溃的唯一的“篮子”里么？如果你不会，那么选择 Git，一定是最明智的选择。

本篇我们首先用一章的内容回顾一下版本控制的历史，并以此向版本控制的前辈 CVS 和 SVN 致敬。在第 2 章通过一些典型的版本控制的实例向您展示 Git 独特的魅力，让您爱上 Git。在本篇的最后一章会介绍 Git 在 Linux、Mac OS X 及 Windows 下的安装，这是我们下一步研究 Git 的基础。

还有在这里有必要纠正一下 Git 的发音。一种错误是按照单个字母来发音，另外一种更为普遍的错误是把整个单词读作“技特”，实际上 Git 中字母 G 的发音应该是和下列单词中的 G 类似：GOD GIVES GREAT GIFT。因此 Git 正确的发音应该听起来像是“歌易特”。在我的一再坚持下，编辑同意本书的英文名（如果有的话）为《Got Git》，当面对这样的书名时您还会把 Git 读错么？

第1章　版本控制的前世和今生

除了茫然未知的宇宙，几乎任何事物都是从无到有，从简陋到完善。随着时间车轮的滚滚向前，历史被抛在身后逐渐远去，如同我们的现代社会，世界大同，到处都是忙碌和喧嚣，再也看不到已经远去的刀耕火种、男耕女织的慢生活岁月。

版本控制系统是一个另类。虽然其历史并不短暂，也有几十年，但是他的演进进程却一直在社会的各个角落重复着，而且惊人的相似。有的人从未使用甚至从未听说过版本控制系统，他和他的团队就像停留在黑暗的史前时代，任由数据自生自灭。有的人使用着有几十年历史的 CVS 或其改良版 Subversion，让时间空耗在网络连接的等待中。再有就是以 Git 为代表的分布式版本控制系统，已经风靡整个开源社区，正等待你的靠近。

1.1 黑暗的史前时代

人们谈及远古，总爱以黑暗形容。黑暗实际上指的是秩序和工具的匮乏，而不是自然，如以自然环境而论，工业化和城市化对环境的破坏，现今才是最黑暗的年代。对软件开发来说也是如此，虽然遥远的 C 语言一统天下的日子，要比今天选 Java，选 .Net，还是选择脚本语言的多选题要简单得多，但是从工具和秩序上讲，过去的年代是黑暗的。

看看我经历的版本控制的“史前时代”吧。在大学里，代码分散地拷贝在各个软盘中，最终我会被搞糊涂，不知道哪个软盘中的代码是最优的，因为最新并非最优，失败的重构会毁掉原来尚能运作的代码。在我工作的第一年，代码的管理并未改观，还是以简单的目录拷贝进行数据的备份，三四个程序员利用文件服务器的共享目录进行协同，公共类库和头文件在操作过程中相互覆盖，痛苦不堪。很明显，那时我尚不知道版本控制系统为何物。我的版本控制史前时代一直延续到 2000 年，那时 CVS 已经诞生了 14 年，而我在那时对 CVS 还一无所知。

实际上，即便是在 CVS 出现之前的“史前时代”，也已经有了非常好用的源码比较和打补丁的工具：`diff` 和 `patch`，他们今天生命力依然顽强。大名鼎鼎的 Linus Torvalds 先生（Linux 之父）也对这两个工具偏爱有加，在 1991-2002 年之间，Linus 一直顽固地使用 `diff` 和 `patch` 管理着 Linux 的代码，即使不断有人提醒他 CVS 的存在[1]。

那么来看看 `diff` 和 `patch`，熟悉它们，将对理解版本控制系统（差异存储），使用版本控制系统（代码比较和冲突合并）都有莫大的好处。

1. 命令 diff 用于比较两个文本文件或目录的差异

先来构造两个文件：

文件 hello	文件 world
应该杜绝文章中的错别子[2]。	应该杜绝文章中的错别字。
但是无论使用	但是无论使用
* 全拼，双拼	* 全拼，双拼

[1] Linus Torvalds 于 2007-05-03 在 Google 的演讲：http://www.youtube.com/watch?v=4XpnKHJAok8

[2] 此处是故意将“字”写成“子”，以便两个文件进行差异比较。

* 还是五笔	* 还是五笔
是人就有可能犯错，软件更是如此。	是人就有可能犯错，软件更是如此。
犯了错，就要扣工资！	改正的成本可能会很高。
改正的成本可能会很高。	但是“只要眼球足够多，所有 Bug 都好捉”，这就是开源的哲学之一。

对这两个文件执行 diff 命令，并通过输出重定向，将差异保存在 *diff.txt* 文件中。

```
$ diff -u hello world > diff.txt
```

上面执行 *diff* 命令的 *-u* 参数很重要，使得差异输出中带有上下文。打开文件 *diff.txt* ，会看到其中的差异比较结果。为了说明方便，为每一行增添了行号。

```
 1 --- hello       2010-09-21 17:45:33.551610940 +0800
 2 +++ world       2010-09-21 17:44:46.343610465 +0800
 3 @@ -1,4 +1,4 @@
 4 -应该杜绝文章中的错别子。
 5 +应该杜绝文章中的错别字。
 6
 7  但是无论使用
 8  * 全拼，双拼
 9 @@ -6,6 +6,7 @@
10
11  是人就有可能犯错，软件更是如此。
12
13 -犯了错，就要扣工资！
14 -
15  改正的成本可能会很高。
16 +
17 +但是“只要眼球足够多，所有 Bug 都好捉”，
18 +这就是开源的哲学之一。
```

上面的差异文件，可以这么理解：

- 第 1、2 行，分别记录了比较的原始文件和目标文件的文件名及时间戳。以三个减号（---）开始的行标识的是原始文件，以三个加号（+++）开始的行标识的是目标文件。
- 在比较内容中，以减号（-）开始的行是只出现在原始文件中的行，例如：第 4、13、14 行。

- 在比较内容中，以加号（+）开始的行是只出现在目标文件中的行，例如：第 5、16-18 行。
- 在比较内容中，以空格开始的行，是在原始文件和目标文件中都出现的行，例如：第 6-8、10-12、15 行。这些行是用作差异比较的上下文。
- 第 3-8 行是第一个差异小节。每个差异小节以一行差异定位语句开始。第 3 行就是一条差异定位语句，其前后分别用两个@ 进行标识。
- 第 3 行定位语句中 *-1,4* 的含义是：本差异小节的内容相当于原始文件的从第 1 行开始的 4 行。而第 4、6、7、8 行是原始文件中的内容，加起来刚好是 4 行。
- 第 3 行定位语句中 *+1,4* 的含义是：本差异小节的内容相当于目标文件的从第 1 行开始的 4 行。而第 5、6、7、8 行是目标文件中的内容，加起来刚好是 4 行。
- 命令 *diff* 是基于行比较，所以即使改正了一个字，也显示为一整行的修改（参见差异文件第 4、5 行）。Git 对 *diff* 进行了扩展，还提供一种逐词比较的差异比较方法，参见本书第 2 篇“11.4.4 差异比较：git diff”小节。
- 第 9-18 行是第二个差异小节。第 9 行是一条差异定位语句。
- 第 9 行定位语句中 *-6,6* 的含义是：本差异小节的内容相当于原始文件的从第 6 行开始的 6 行。而第 10-15 行是原始文件中的内容，加起来刚好是 6 行。
- 第 9 行定位语句中 *+6,7* 的含义是：本差异小节的内容相当于目标文件的从第 6 行开始的 7 行。而第 10-12、15-18 行是目标文件中的内容，加起来刚好是 7 行。

2. 命令 patch 相当于 diff 的反向操作

有了 *hello* 和 *diff.txt* 文件，可以放心地将 *world* 文件删除或用 *hello* 文件将 *world* 文件覆盖。用下面的命令可以还原 *world* 文件：

```
$ cp hello world
$ patch world < diff.txt
```

也可以保留 *world* 和 *diff.txt* 文件，删除 *hello* 文件或用 *word* 文件将 *hello* 文件覆盖。用下面的命令可以恢复 *hello* 文件：

```
$ cp world hello
$ patch -R hello < diff.txt
```

命令 *diff* 和 *patch* 还可以对目录进行比较操作，这也就是 Linus 在 1991-2002 年用于维护 Linux 不同版本间差异的办法。可以用此命令，在没有版本控制系统的情况下，记录并保存改动前后的差异，还可以将差异文件注入版本控制系统（如果有的话）。

标准的 *diff* 和 *patch* 命令存在一个局限，就是不能对二进制文件进行处理。对二进制文件的修改或添加会在差异文件中缺失，进而丢失对二进制文件的改动或添加。Git 对差异文件格式提供了扩展支持，支持二进制文件的比较，解决了这个问题。这点可以参考本书第 7 篇“第 38 章 补丁中的二进制文件”的相关内容。

1.2 CVS —— 开启版本控制大爆发

CVS（Concurrent Versions System）[1]诞生于 1985 年，是由荷兰阿姆斯特丹 VU 大学的 Dick Grune 教授实现的。当时 Dick Grune 和两个学生共同开发一个项目，但是三个人的工作时间无法协调到一起，迫切需要一个记录和协同代码开发的工具软件。于是 Dick Grune 通过脚本语言对 RCS（一个针对单独文件的多版本管理工具）进行封装，设计出有史以来第一个被大规模使用的版本控制工具。在 Dick 教授的网站上介绍了 CVS 这段早期的历史。[2]

> “在 1985 年一个糟糕的秋日里，我站在校汽车站等车回家，脑海里一直纠结着一件事 —— 如何处理 RCS 文件、用户文件（工作区）和 Entries 文件的复杂关系，有的文件可能会缺失、冲突、删除，等等。我的头有些晕了，于是决定画一个大表，将复杂的关联画在其中看看出来的结果是什么样的……”

1986 年 Dick 通过新闻组发布了 CVS，1989 年由 Brian Berliner 将 CVS 用 C 语言重写。

从 CVS 的历史可以看出 CVS 不是设计出来的，而是被实际需要逼出来的，因此根据实用为上的原则，借用了已有的针对单一文件的多版本管理工具 RCS。CVS 采用客户端/服务器架构设计，版本库位于服务器端，实际上就是一个 RCS 文件容器。每一个 RCS 文件以 “,v” 作为文件名后缀，用于保存对应文件的历次更改历史。RCS 文件中只保留一个版本库的完全拷贝，其他历次更改仅将差异存储其中，使得存储变得非常有效率。我在 2008 年设计的一个 SVN 管理后台 pySvnManager[3]，实际上也采用了 RCS 作为 SVN 授权文件的变更记录的“数据库”。

图 1-1 展示了 CVS 版本控制系统的工作原理，可以看到作为 RCS 文件容器的 CVS 版本库和工作区目录结构的一一对应关系。

[1] http://www.nongnu.org/cvs/

[2] http://www.cs.vu.nl/~dick/CVS.html

[3] http://pysvnmanager.sourceforge.net/

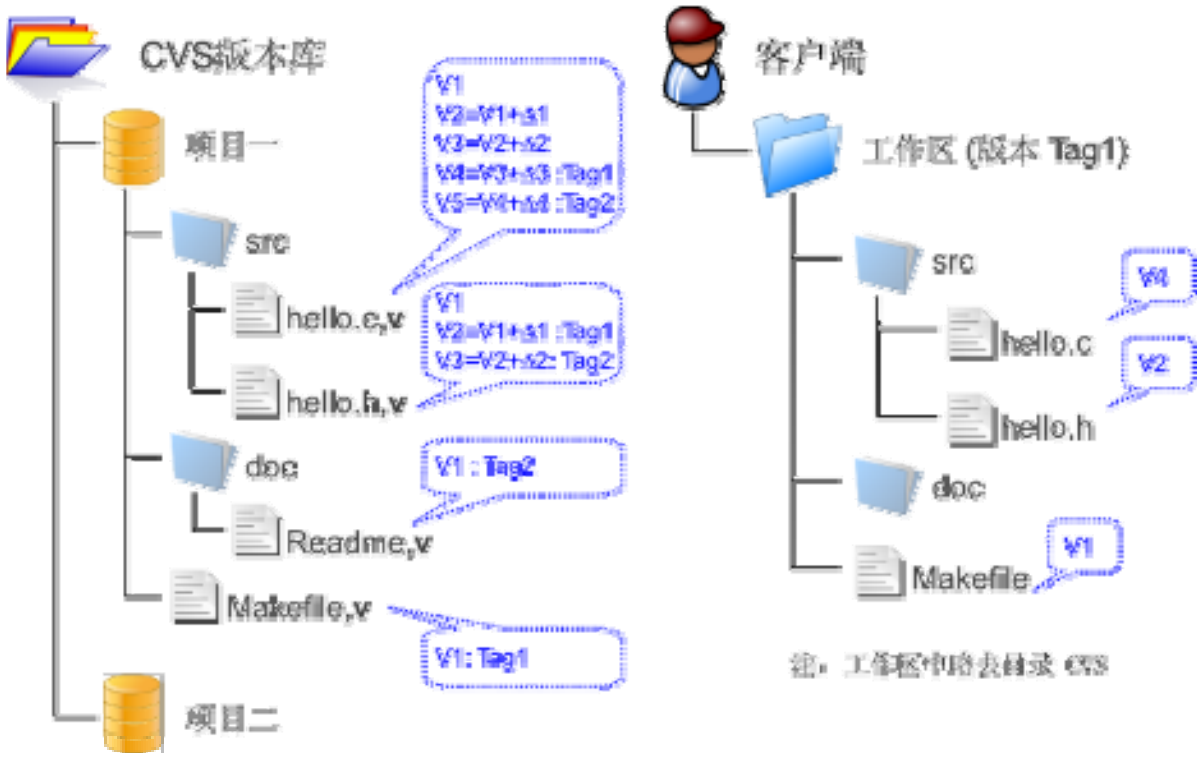

图 1-1：CVS 版本控制系统示意图

CVS 的这种实现方式的最大好处就是简单。把版本库中随便一个目录拿出来就可以成为另外一个版本库。如果将版本库中的一个 RCS 文件重命名，工作区检出的文件名也相应地会改变。这种低成本的服务器管理模式成为很多 CVS 粉丝至今不愿离开 CVS 的原因。

CVS 的出现让软件工程师认识到了原来还可以这样工作。CVS 成功地为后来的版本控制系统确立了标准，像提交（commit）、检入（checkin）、检出（checkout）、里程碑（tag）、分支（branch）等概念早在 CVS 中就已经确立。CVS 的命令行格式也被后来的版本控制系统竞相模仿。

在 2001 年，我正为使用 CVS 激动不已的时候，公司领导要求采用和美国研发部门同样的版本控制解决方案。于是，我的项目组率先进行了从 CVS 到该商业版本控制工具的迁移[1]。虽然商业版本控制工具有更漂亮的界面及更好的产品整合性，但是就版本控制本身而言，商业版本控制工具存在着如下缺陷。

- 采用黑盒子式的版本库设计。让人捉摸不透的版本库设计，最大的目的可能就是阻止用户再迁移到其他平台。
- 缺乏版本库整理工具。如果有一个文件（如记录核弹起爆密码的文件）检入到版本库中，就没有办法再彻底移除它。
- 商业版本控制工具很难为个人提供版本控制解决方案，除非个人愿意花费高昂的许可证费

[1] 于是就有了这篇文章：http://www.worldhello.net/doc/cvs_vs_starteam/

用。

- 商业版本控制工具注定是小众软件，对新员工的培训成本不可忽视。

而上述商业版本控制系统的缺点，恰恰是 CVS 及其他开源版本控制系统的强项。但在经历了最初的成功之后，CVS 也尽显疲态：

- 服务器端松散的 RCS 文件，导致在建立里程碑或分支时缺乏效率，服务器端文件越多，速度越慢。
- 分支和里程碑不可见，因为它们被分散地记录在服务器端的各个 RCS 文件中。
- 合并困难重重，因为缺乏对合并的追踪从而导致重复合并，引发严重冲突。
- 缺乏对原子提交的支持，会导致客户端向服务器端提交不完整的数据。
- 不能优化存储内容相同但文件名不同的文件，因为在服务器端每个文件都是单独进行差异存储的。
- 不能对文件和目录的重命名进行版本控制，虽然直接在服务器端修改 RCS 文件名可以让改名后的文件保持历史，但是这样做实际会破坏历史。

CVS 的成功开启了版本控制系统的大爆发，各式各样的版本控制系统如雨后春笋般地诞生了。新的版本控制系统或多或少地解决了 CVS 版本控制系统存在的问题。在这些版本控制系统中最典型的就是 Subversion（SVN）。

1.3 SVN —— 集中式版本控制集大成者

Subversion[1]，因其命令行工具名为 *svn* 因此通常被简称为 SVN。SVN 由 CollabNet 公司于 2000 年资助并发起开发，目的是创建一个更好用的版本控制系统以取代 CVS。前期 SVN 的开发使用 CVS 做版本控制，到了 2001 年，SVN 已经可以用于自己的版本控制了[2]。

我开始真正关注 SVN 是在 2005 年，那时 SVN 正经历着后端存储上的变革，即从 BDB（简单的关系型数据库）到 FSFS（文件数据库）的转变。FSFS 相对于 BDB 具有稳定性、免维护性，以及实现的可视性，我马上就被 SVN 吸引了。图 1-2 展示了 SVN 版本控制系统的工作原理。

[1] http://subversion.apache.org/

[2] http://svnbook.red-bean.com/en/1.5/svn.intro.whatis.html#svn.intro.history

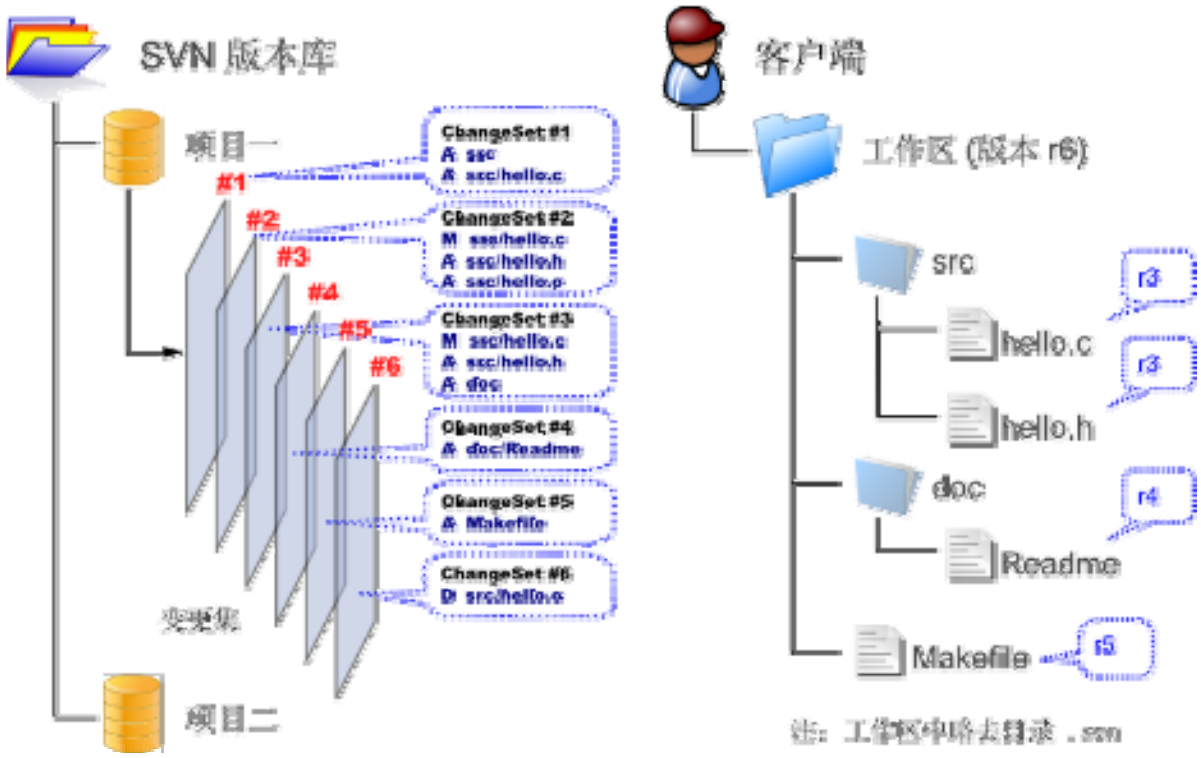

图 1-2：SVN 版本控制系统示意图

SVN 的每一次提交，都会在服务器端的 *db/revs* 和 *db/revprops* 目录下各创建一个以顺序数字编号命名的文件。其中 *db/revs* 目录下的文件（即变更集文件）记录与上一个提交之间的差异（字母 A 表示新增，M 表示修改，D 表示删除）。在 *db/revprops* 目录下的同名文件（没有在图 1-2 中体现）则保存着提交日志、作者、提交时间等信息。这样设计的好处有：

- 拥有全局版本号。每提交一次， SVN 的版本号就会自动加一。这为 SVN 的使用提供了极大的便利。回想 CVS 时代，每个文件都拥有各自独立的版本号（RCS 版本号），要想获得全局版本号，只能通过手工不断地建立里程碑来实现。
- 实现了原子提交。SVN 不会像 CVS 那样出现部分文件被提交而其他没有被提交的状态。
- 文件名不受限制。因为服务器端不再需要建立和客户端文件相似的文件名，这样，文件的命名就不再受服务器操作系统的字符集及大小写的限制。
- 文件和目录重命名也得到了支持。

SVN 最具有特色的功能是轻量级拷贝，例如将目录 *trunk* 拷贝为 *branches/v1.x* 只相当于在 *db/revs* 目录中的变更集文件中用特定的语法做了一下标注，无须真正的文件拷贝。SVN 使用轻量级拷贝的功能，轻松地解决了 CVS 存在的里程碑和分支的创建速度慢又不可见的问题，使用 SVN 创建里程碑和分支只在眨眼之间。

SVN 在版本库授权上也有改进，不再像 CVS 那样依赖操作系统本身对版本库目录和文件进行授权，而是采用授权文件的方式来实现。

SVN 还有一个创举，就是在工作区跟踪目录下（*.svn* 目录）为当前目录中的每一个文件都保存一份冗余的原始拷贝。这样做的好处是部分命令不再需要网络连接，例如文件修改的差异比较，以及错误更改的回退等。

正是由于 SVN 的这些闪亮的功能，使得 SVN 成为继 CVS 之后诞生的诸多版本控制系统中的集大成者，成为开源社区一时的新宠，也成为当时各个企业版本控制的最佳选择之一。

但是 SVN 相对 CVS 在本质上并没有突破，都属于集中式版本控制系统。就是一个项目只有唯一的一个版本库与之对应，所有的项目成员都通过网络向该服务器进行提交。这样的设计除了容易出现单点故障以外，单是查看日志、提交数据等操作的延迟，就足以让基于广域网协同工作的团队抓狂了。

除了集中式版本控制系统固有的问题外，SVN 的里程碑、分支的设计也被证明是一个错误，虽然这个错误使得 SVN 拥有了快速创建里程碑和分支的能力，但是这个错误导致了如下的更多问题。

- 项目文件在版本库中必须按照一定的目录结构进行部署，否则就可能无法建立里程碑和分支。

  我在项目咨询过程中就见过很多团队，直接在版本库的根目录下创建项目文件。这样的版本库布局，在需要创建里程碑和分支时就无从下手了，因为根目录是不能拷贝到子目录中的。所以 SVN 的用户在创建版本库时必须遵守一个古怪的约定：先创建三个顶级目录 */trunk*、*/tags* 和 */branches* 。

- 创建里程碑和分支会破坏精心设计的授权。

  SVN 的授权是基于目录的，分支和里程碑也被视为目录（和其他目录没有分别）。因此每次创建分支或里程碑时，就要将针对 */trunk* 目录及其子目录的授权在新建的分支或里程碑上重建。随着分支和里程碑数量的增多，授权愈加复杂，维护也愈加困难。

- 分支太随意从而导致混乱。SVN 的分支创建非常随意：可以基于 */trunk* 目录创建分支，也可以基于其他任何目录创建分支。因此 SVN 很难画出一个有意义的分支图。再加上一次提交可以同时包含针对不同分支的文件变更，使得事情变得更糟。

- 虽然在 SVN 1.5 之后拥有了合并追踪功能，但这个功能会因为混乱的分支管理而被抵消。

2009 年底，SVN 由 CollabNet 公司交由 Apache 社区管理，至此 SVN 成为了 Apache 组织的一个子项目[1]。这对 SVN 到底意味着什么？是开发的停滞，还是新的开始，结果如何我们将

[1] http://en.wikipedia.org/wiki/Apache_Subversion

拭目以待。

1.4 Git —— Linus 的第二个伟大作品

Linux 之父 Linus 是坚定的 CVS 反对者，他也同样地反对 SVN。这就是为什么在 1991-2002 这十余年间，Linus 宁可通过手工修补文件的方式维护代码，也迟迟不愿使用 CVS。我想在那个时期要想劝说 Linus 使用 CVS 只有一个办法：把 CVS 服务器请进 Linus 的卧室，并对外配以千兆带宽。

2002 年至 2005 年， Linus 顶着开源社区精英们的口诛笔伐，选择了一个商业版本控制系统 BitKeeper 作为 Linux 内核的代码管理工具[1]。BitKeeper 是一款不同于像 CVS/SVN 那样的集中式版本控制工具，而是一款分布式版本控制工具。

分布式版本控制系统最大的反传统之处在于，可以不需要集中式的版本库，每个人都工作在通过克隆操作建立的本地版本库中。也就是说每个人都拥有一个完整的版本库，所有操作包括查看提交日志、提交、创建里程碑和分支、合并分支、回退等都直接在本地完成而不需要网络连接。每个人都是本地版本库的主人，不再有谁能提交谁不能提交的限制，加之多样的协同工作模型（版本库间推送、拉回，及补丁文件传送等）让开源项目的参与度有爆发式增长。

2005 年发生的一件事最终导致了 Git 的诞生。在 2005 年 4 月 Andrew Tridgell，即大名鼎鼎的 Samba 的作者，试图尝试对 BitKeeper 反向工程，以开发一个能与 BitKeeper 交互的开源工具。这激怒了 BitKeeper 软件的所有者 BitMover 公司，要求收回对 Linux 社区免费使用 BitKeeper 的授权[2]。迫不得已， Linus 选择了自己开发一个分布式版本控制工具以替代 BitKeeper。以下是 Git 诞生大事记[3]：

- 2005 年 4 月 3 日，开始开发 Git。
- 2005 年 4 月 6 日，项目发布。
- 2005 年 4 月 7 日，Git 就可以作为自身的版本控制工具了。
- 2005 年 4 月 18 日，发生第一个多分支合并。
- 2005 年 4 月 29 日，Git 的性能就已经达到了 Linus 的预期。
- 2005 年 6 月 16 日，Linux 核心 2.6.12 发布，那时 Git 已经在维护 Linux 核心的源代码

[1] http://en.wikipedia.org/wiki/BitKeeper

[2] http://en.wikipedia.org/wiki/Andrew_Tridgell

[3] http://en.wikipedia.org/wiki/Git_%28software%29

了。

Linus 以一个文件系统专家和内核设计者的视角对 Git 进行了设计，其独特的设计，让 Git 拥有非凡的性能和最为优化的存储能力。完成原型设计后，在 2005 年 7 月 26 日，Linus 功成身退，将 Git 的维护交给另外一个 Git 的主要贡献者 Junio C Hamano[1]，直到现在。

最初的 Git 除了一些核心命令以外，其他的都用脚本语言开发，而且每个功能都作为一条独立的命令，例如克隆操作用 `git-clone` ，提交操作用命令 `git-commit` 。这导致 Git 拥有庞大的命令集，使用习惯也和其他版本控制系统格格不入。随着 Git 的开发者和使用者的增加，Git 也在逐渐演变，例如到 1.5.4 版本时，将一百多个独立的命令封装为一个 `git` 命令，使它看起来更像是一个独立的工具，而且 Git 的使用习惯也逐渐被普通用户所接受。

经过短短几年的发展，众多的开源项目都纷纷从 SVN 或其他版本控制系统迁移到 Git。虽然版本控制系统的迁移过程是痛苦的，但是因为迁移到 Git 会带来开发效率的极大提升，以及巨大的效益，所以很快就会忘记迁移的痛苦过程，并很快就会适应新的工作模式。在 Git 网站上列出了几个使用 Git 的重量级项目，个个都是人们耳熟能详的，除了 Git 和 Linux 内核外，还有：Perl、Eclipse、Gnome、KDE、Qt、Ruby on Rails、Android、PostgreSQL、Debian、X.org，当然还有 GitHub 的上百万个项目。

Git 虽然是在 Linux 下开发的，但现在已经可以跨平台运行在所有主流的操作系统上，包括 Linux、Mac OS X 和 Windows 等。可以说每一个使用计算机的用户都可以分享 Git 带来的便利和快乐。

[1] http://marc.info/?l=git&m=112243466603239

第2章 爱上 Git 的理由

本章通过一些典型应用展示 Git 作为版本控制系统的独特用法。对于不熟悉版本控制系统的读者，可以通过这些示例对版本控制拥有感性的认识。如果是有经验的读者，示例中的和 SVN 的对照可以让您体会到 Git 的神奇和强大。本章将列举 Git 的一些闪亮特性，期待能够让您爱上 Git。

2.1 每日工作备份

当我开始撰写本书时才明白写书真的是一个辛苦活。如何让辛苦的工作不会因为笔记本硬盘的意外损坏而丢失？如何防范灾害而不让一个篮子里的鸡蛋都毁于一旦？下面就介绍一下我在写本书时如何使用 Git 进行文稿备份的，请看图 2-1。

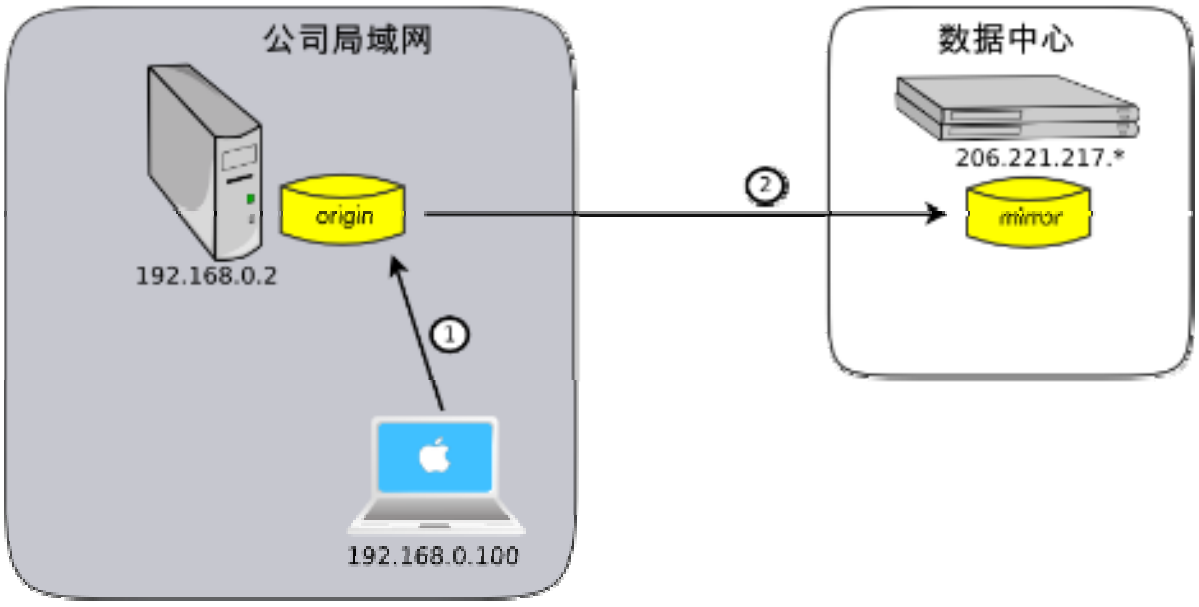

图 2-1：利用 Git 做数据的备份

如图 2-1，我的笔记本在公司局域网里的 IP 地址是 192.168.0.100，公司的 Git 服务器的 IP 地址是 192.168.0.2。公司使用动态 IP 上网因而没有固定的外网 IP，但是公司在数据中心有托管服务器，拥有固定的 IP 地址，其中一台服务器用作 Git 服务器镜像。

我的写书习惯大概是这样：一般在写完一个小节，或是画完一张图，我会执行下面的命令提交一次。每一天平均提交 3-5 次。提交是在笔记本本地完成的，因此在图中没有表示出来。

```
$ git add -u    # 如果创建了新文件，可以执行 git add -i 命令。
$ git commit
```

下班后，我会执行一次推送操作，将我在本地 Git 版本库中的提交同步到公司的 Git 服务器上。相当于图 2-1 中的步骤① 。

```
$ git push
```

因为公司的 Git 服务器和异地数据中心的 Git 服务器建立了镜像，所以每当我向公司内网服务器推送的时候，就会自动触发从内网服务器到外网 Git 服务器的镜像操作。相当于图 2-1 中的步骤②，步骤②是自动执行的，无须人工干预。图 2-1 中标记为 mirror 的版本库就是 Git 镜像版本库，该版本库只向用户提供只读访问服务，而不能对其进行写操作（推送）。

从图 2-1 中可以看出，我的每日工作保存有三个拷贝，一个在笔记本中，一个在公司内网的服务器上，还有一个在外网的镜像版本库中。鸡蛋分别装在了三个篮子里。

至于如何架设可以实时镜像的 Git 服务器，会在本书第 5 篇“第 30 章 Gitolite 服务架设”中予以介绍。

2.2 异地协同工作

为了能够加快写书的进度，熬夜是必须的，这就出现了在公司和在家两地工作同步的问题。图 2-2 用于说明我是如何解决两地工作同步的问题的。

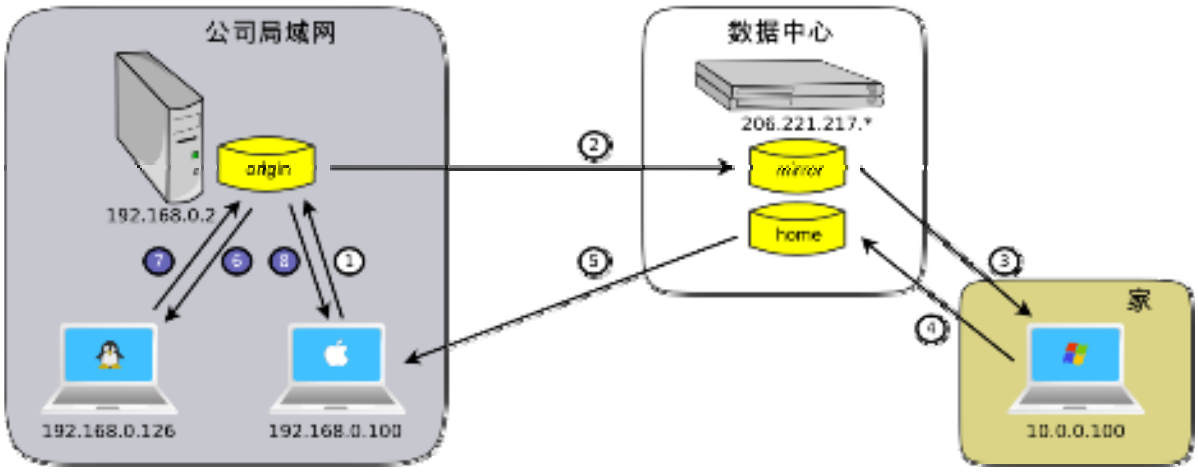

图 2-2：利用 Git 实现异地工作协同

我在家里的电脑 IP 地址是 10.0.0.100（家里也有一个小局域网）。如果在家里有时间工作的话，首先要做的就是图 2-2 中步骤③的操作：从 mirror 版本库同步数据到本地。只需要一条命令就好了：

```
$ git pull mirror master
```

然后在家里的电脑上编辑书稿并提交。当准备完成一天的工作时，就执行下面的命令，相当于

图 2-2 中步骤④的操作：将在家中的提交推送到标记为 home 的版本库中。

```
$ git push home
```

为什么还要再引入另外一个名为 home 的版本库呢？使用 mirror 版本库不好么？不要忘了 mirror 版本库只是一个镜像库，不能提供写操作。

当一早到公司，开始动笔写书之前，先要执行图 2-2 中步骤⑤的操作，从 home 版本库将家里做的提交同步到公司的电脑中。

```
$ git pull home master
```

公司的小崔是我这本书的忠实读者，我每有新章节出来，他都会执行图 2-2 中步骤⑥的工作，从公司内网服务器获取我最新的文稿。

```
$ git pull
```

一旦发现文字错误，小崔会直接在文稿中修改，然后推送到公司的服务器上（图 2-2 中步骤⑦）。当然他的这个推送也会自动同步到外网的 mirror 版本库。

```
$ git push
```

而我只要执行 *git pull* 操作就可以获得小崔对我文稿的修订（图 2-2 中的步骤⑧）。采用这种工作方式，文稿竟然分布在 5 台电脑上拥有 6 个拷贝，真可谓狡兔三窟。不，比狡兔还要多三窟。

在本节中，出现在 Git 命令中的 *mirror* 和 *home* 是和工作区关联的远程版本库。关于如何注册和使用远程版本库，请参见本书第 3 篇“第 19 章 远程版本库”中的内容。

2.3 现场版本控制

所谓现场版本控制，就是在客户现场或在产品部署的现场，进行源代码的修改，并在修改过程中进行版本控制，以便在完成修改后能够将修改结果甚至修改过程一并带走，并能够将修改结果合并至项目对应的代码库中。

1. SVN 的解决方案

如果使用 SVN 进行版本控制，首先要将服务器上部署的产品代码目录变成 SVN 工作区，这个过程并不简单而且会显得很繁琐，最后将改动结果导出也非常不方便，具体操作过程如下。

（1） 在其他位置建立一个 SVN 版本库。

```
$ svnadmin create /path/to/repos/project1
```

（2） 在需要版本控制的目录下检出刚刚建立的空版本库。

```
$ svn checkout file:///path/to/repos/project1 .
```

（3） 执行文件添加操作，然后执行提交操作。这个提交将是版本库中编号为 1 的提交。

```
$ svn add *
$ svn ci -m "initialized"
```

（4） 然后开始在工作区中修改文件，提交。

```
$ svn ci
```

（5） 如果对修改结果满意，可以通过创建补丁文件的方式将工作成果保存带走。但是 SVN 很难对每次提交逐一创建补丁，一般用下面的命令与最早的提交进行比较，以创建出一个大补丁文件。

```
$ svn diff -r1 > hacks.patch
```

上面用 SVN 将工作成果导出的过程存在一个致命的缺陷，就是 SVN 的补丁文件不支持二进制文件，因此采用补丁文件的方式有可能丢失数据，如新增或修改的图形文件会丢失。更为稳妥但也更为复杂的方式可能要用到 svnadmin 命令将版本库导出。命令如下：

```
$ svnadmin dump --incremental -r2:HEAD \
  /path/to/repos/project1/ > hacks.dump
```

将 svnadmin 命令创建的导出文件恢复到版本库中也非常具有挑战性，这里就不再详细说明了。还是来看看 Git 在这种情况下的表现吧。

2. Git 的解决方案

Git 对产品部署目录进行到工作区的转化相比 SVN 要更为简单，而且使用 Git 将提交历史导出也更为简练和实用，具体操作过程如下。

（1） 现场版本库创建。直接在需要版本控制的目录下执行 Git 版本库初始化命令。

```
$ git init
```

（2） 添加文件并提交。

```
$ git add -A
$ git commit -m "initialized"
```

（3） 为初始提交建立一个里程碑：“v1”。

```
$ git tag v1
```

（4） 然后开始在工作区中工作 —— 修改文件，提交。

```
$ git commit -a
```

（5） 当对修改结果满意，想将工作成果保存带走时，可以通过下面的命令，将从 v1 开始的历次提交逐一导出为补丁文件。转换的补丁文件都包含一个数字前缀，并提取提交日志信息作为文件名，而且补丁文件还提供对二进制文件的支持。下面命令的输出摘自本书第 3 篇“第 20 章 补丁文件交互”中的实例。

```
$ git format-patch v1..HEAD
0001-Fix-typo-help-to-help.patch
0002-Add-I18N-support.patch
0003-Translate-for-Chinese.patch
```

（6） 通过邮件将补丁文件发出。当然也可以通过其他方式将补丁文件带走。

```
$ git send-email *.patch
```

Git 创建的补丁文件使用了 Git 扩展格式，因此在导入时为了避免数据遗漏，要使用 Git 提供的命令而不能使用 GNU *patch* 命令。即使要导入的不是 Git 版本库，也可以使用 Git 命令，具体操作请参见本书第 7 篇“第 38 章 补丁中的二进制文件”中的相关内容。

2.4 避免引入辅助目录

很多版本控制系统，都要在工作区中引入辅助目录或文件，如 SVN 要在工作区的每一个子目录下都创建 *.svn* 目录，CVS 要在工作区的每一个子目录下都创建 *CVS* 目录。

这些辅助目录如果出现在服务器上，尤其是 Web 服务器上是非常危险的，因为这些辅助目录下的 *Entries* 文件会暴露出目录下的文件列表，让管理员精心配置的禁止目录浏览的努力全部白费。

还有，SVN 的 *.svn* 辅助目录下还存在文件的原始拷贝，在文件搜索时结果会加倍。如果您曾经在 SVN 的工作区用过 *grep* 命令进行内容查找，就会明白我指的是什么。

Git 没有这个问题，不会在子目录下引入讨厌的辅助目录或文件（*.gitignore* 和 *.gitattributes* 文件不算）。当然 Git 还是要在工作区的顶级目录下创建名为 *.git* 的目录（版本库目录），不过如果你认为唯一的一个 *.git* 目录也过于碍眼，可以将其放到工作区之外的任意目录。一旦这么做了，你在执行 Git 命令时，要通过命令行（ *--git-dir* ）或环境变量 *GIT_DIR* 为工作区指定版本库目录，甚至还要指定工作区目录。

Git 还专门提供了一个 *git grep* 命令，这样在工作区根目录下执行查找时，目录 *.git* 也不会对搜索造成影响。

关于辅助目录的详细讨论请参见本书第 4 章第 4.2 节中的内容。

2.5 重写提交说明

很多人可能如我一样，在敲下回车之后，才发现提交说明中出现了错别字，或忘记了写关联的 Bug ID。这就需要重写提交说明。

1. SVN 的解决方案

SVN 的提交说明默认是禁止更改的，因为 SVN 的提交说明属于不受版本控制的属性，一旦修改就不可恢复。我建议 SVN 的管理员只有在配置了版本库更改的外发邮件通知之后，再开放提交说明更改的功能。我发布于 SourceForge 上的 pySvnManager 项目，提供了 SVN 版本库图形化的钩子管理，会简化管理员的配置工作。

即使 SVN 管理员启用了允许更改提交说明的设置，修改提交说明也还是挺复杂的，看看下面的命令：

```
$ svn ps --revprop -r <REV> svn:log "new log message..."
```

2. Git 的解决方案

Git 修改提交说明很简单，而且提交说明的修改也是被追踪的。Git 修改最新提交的提交说明最为简单，使用一条名为修补提交的命令即可。

```
$ git commit --amend
```

这个命令如果不带“-m”参数，会进入提交说明编辑界面，修改原来的提交说明，直到满意为止。

如果要修改某个历史提交的提交说明，Git 也可以实现，但要用到另外一个命令：变基命令。例如要修改 <commit-id> 所标识提交的提交说明，执行下面的命令，并在弹出的变基索引文件中修改相应提交前面的动作的关键字。

```
$ git rebase -i <commit-id>^
```

关于如何使用交互式变基操作更改历史提交的提交说明，请参见本书第 2 篇“第 12 章 改变历史”中的内容。

2.6 想吃后悔药

假如提交的数据中不小心包含了一个不应该检入的虚拟机文件——大约有 1 个 GB！这时候，您会多么希望这个世界上有后悔药卖啊。

1. SVN 的解决方案

SVN 遇到这个问题该怎么办呢？删除错误加入的大文件，再提交，这样的操作是不能解决问题的。虽然表面上去掉了这个文件，但是它依然存在于历史中。

管理员可能是受影响最大的人，因为他要负责管理服务器的磁盘空间占用及版本库的备份。实际上这个问题也只有管理员才能解决，所以你必须向管理员坦白，让他帮你在服务器端彻底删除错误引入的大文件。我要告诉你的是，对于管理员，这并不是一个简单的活。

（1） SVN 管理员要是没有历史备份的话，只能从头用 *svnadmin dump* 导出整个版本库。
（2） 再用 *svndumpfilter* 命令过滤掉不应检入的大文件。
（3） 然后用 *svnadmin load* 重建版本库。

上面的操作描述中省略了一些窍门，因为要把窍门说清楚的话，这本书就不是讲 Git，而是讲 SVN 了。

2. Git 的解决方案

如果你用 Git，一切就会非常简单，而且你也不必去乞求管理员，因为使用 Git，每个人都是管理员。

如果是最新的提交引入了不该提交的大文件：*winxp.img*，操作起来会非常简单，还是用修补提交命令。

```
$ git rm --cached winxp.img
$ git commit --amend
```

如果是历史版本，例如是在 <commit-id> 所标识的提交中引入的文件，则需要使用变基操作。

```
$ git rebase -i <commit-id>^
```

执行交互式变基操作抛弃历史提交，版本库还不能立即瘦身，具体原因和解决方案请参见本书第 2 篇“第 14 章 Git 库管理”中的内容。除了使用变基操作，Git 还有更多的武器可以实现版本库的整理操作，具体请参见本书第 35 章第 35.4 节的内容。

2.7 更好用的提交列表

正确的版本控制系统的使用方法是，一次提交只干一件事：完成一个新功能、修改了一个 Bug、或是写完了一节的内容、或是添加了一幅图片，就执行一次提交。而不要在下班时才想起来要提交，那样的话版本控制系统就被降格为文件备份系统了。

但有时在同一个工作区中可能同时在做两件事情，一个是尚未完成的新功能，另外一个是解决刚刚发现的 Bug。很多版本控制系统没有提交列表的概念，或者要在命令行指定要提交的文件，或者默认把所有修改内容全部提交，破坏了一个提交干一件事的原则。

1. SVN 的解决方案

SVN 1.5 开始提供了变更列表（change list）的功能，通过引入一个新的命令 *svn changelist* 来实现。但是我从来就没有用过，因为：

- 定义一个变更列表太麻烦。例如不支持将当前所有改动的文件加入列表，也不支持将工作区中的新文件全部加入列表。
- 一个文件不能同时属于两个变更列表。两次变更不许有文件交叉，这样的限制太牵强。
- 变更列表是一次性的，提交之后自动消失。这样的设计没有问题，但是相比定义列表时的繁琐，以及提交时必须指定列表的繁琐，使用变更列表未免得不偿失。
- 再有，因为 Subversion 的提交不能撤销，如果在提交时忘了提供变更列表名称以针对特定的变更列表进行提交，错误的提交内容将无法补救。

总之，SVN 的变更列表尚不如鸡肋，食之无味，弃之不可惜。

2. Git 的解决方案

Git 通过提交暂存区实现对提交内容的定制，非常完美地实现了对工作区的修改内容进行筛选提交：

- 执行 `git add` 命令将修改内容加入提交暂存区。执行 `git add -u` 可以将所有修改过的文件加入暂存区。执行 `git add -A` 可以将本地删除文件和新增文件都登记到提交暂存区。
- 一个修改后的文件被登记到提交暂存区后，可以继续修改，继续修改的内容不会被提交，除非再对此文件再执行一次 `git add` 命令。即一个修改的文件可以拥有两个版本，在提交暂存区中有一个版本，在工作区中有另外一个版本。
- 执行 `git commit` 命令提交，无须设定什么变更列表，直接将登记在暂存区中的内容提交。
- Git 支持对提交的撤消，而且可以撤消任意多次。

只要使用 Git，就会时刻在和隐形的提交列表打交道。本书第 2 篇“第 5 章 Git 暂存区”会详细介绍 Git 的这一特性，相信你会爱上 Git 的这个特性。

2.8 更好的差异比较

Git 对差异比较进行了扩展，支持对二进制文件的差异比较，这是对 GNU 的 `diff` 和 `patch` 命令的重要补充。还有 Git 的差异比较除了支持基于行的差异比较外，还支持在一行内逐字比较的方式，当向 `git diff` 命令传递 `--word-diff` 参数时，就会进行逐字比较。

在上面介绍了工作区的文件修改可能会有两个不同的版本，一个是在提交暂存区，一个是在工作区。因此在执行 `git diff` 命令时会遇到令 Git 新手费解的现象。

- 修改后的文件在执行 `git diff` 命令时会看到修改造成的差异。
- 修改后的文件通过 `git add` 命令提交到暂存区后，再执行 `git diff` 命令会看不到该文件的差异。
- 继续对此文件进行修改，再执行 `git diff` 命令，会看到新的修改显示在差异中，而看不到旧的修改。
- 执行 `git diff --cached` 命令才可以看到添加到暂存区中的文件所做出的修改。

Git 差异比较的命令充满了魔法，本书第 5 章第 5.3 节会带您破解 Git 的 diff 魔法。一旦您

习惯了，就会非常喜欢 *git diff* 的这个行为。

2.9 工作进度保存

如果工作区的修改尚未完成时，忽然有一个紧急的任务，需要从一个干净的工作区开始新的工作，或要切换到别的分支进行工作，那么如何保存当前尚未完成的工作进度呢？

1. SVN 的解决方案

如果版本库规模不大，最好重新检出一个新的工作区，在新的工作区进行工作。否则，可以执行下面的操作。

```
$ svn diff > /path/to/saved/patch.file
$ svn revert -R
$ svn switch <new_branch>
```

在新的分支中工作完毕后，再切换回当前分支，将补丁文件重新应用到工作区。

```
$ svn switch <original_branch>
$ patch -p1 < /path/to/saved/patch.file
```

但是切记 SVN 的补丁文件不支持二进制文件，这种操作方法可能会丢失对二进制文件的更改！

2. Git 的解决方案

Git 提供了一个可以保存和恢复工作进度的命令 *git stash* 。这个命令非常方便地解决了这个难题。

在切换到新的工作分支之前，执行 *git stash* 保存工作进度，工作区就会变得非常干净，然后就可以切换到新的分支中了。

```
$ git stash
$ git checkout <new_branch>
```

新的工作分支修改完毕后，再切换回当前分支，调用 *git stash pop* 命令则可恢复之前保存的工作进度。

```
$ git checkout <orignal_branch>
$ git stash pop
```

本书第 2 篇“第 9 章 恢复进度”会为您揭开 *git stash* 命令的奥秘。

2.10 代理 SVN 提交实现移动式办公

使用像 SVN 一样的集中式版本控制系统，要求使用者和版本控制服务器之间要有网络连接，如果因为出差在外或在家办公访问不到版本控制服务器就无法提交。Git 属于分布式版本控制系统，不存在这样的问题。

当版本控制服务器无法实现从 SVN 到 Git 的迁移时，仍然可以使用 Git 进行工作。在这种情况下，Git 作为客户端来操作 SVN 服务器，实现在移动办公状态下的版本提交（当然是在本地 Git 库中提交）。当能够连通 SVN 服务器时，一次性将移动办公状态下的本地提交同步给 SVN 服务器。整个过程对于 SVN 来说是透明的，没有人知道你是使用 Git 在进行提交。

使用 Git 来操作 SVN 版本控制服务器的一般工作流程为：

（1） 访问 SVN 服务器，将 SVN 版本库克隆为一个本地的 Git 库，一个货真价实的 Git 库，不过其中包含针对 SVN 的扩展。

```
$ git svn clone <svn_repos_url>
```

（2） 使用 Git 命令操作本地克隆的版本库，例如提交就使用 *git commit* 命令。
（3） 当能够通过网络连接到 SVN 服务器，并想将本地提交同步给 SVN 服务器时，先获取 SVN 服务器上最新的提交，再执行变基操作，最后再将本地提交推送给 SVN 服务器。

```
$ git svn fetch
$ git svn rebase
$ git svn dcommit
```

本书第 4 篇“第 26 章 Git 和 SVN 协同模型”中会详细介绍这一话题。

2.11 无处不在的分页器

虽然拥有图形化的客户端，但 Git 的主要操作还是以命令行方式进行。使用命令行方式的好处一个是快，另外一个就是防止鼠标手的出现。Git 的命令行进行了大量的人性化设计，包括命令补全、彩色字符输出[1]等，不过最具特色的还是无处不在的分页器。

在操作其他版本控制系统的命令行时，如果命令的输出超过了一屏，为了能够逐屏显示，需要

[1] 须通过 Git 配置变量启用，如运行命令： *git config --global color.ui true*

在命令的后面加上一个管道符号将输出交给一个分页器。例如：

```
$ svn log | less
```

而 Git 则不用如此麻烦，因为每个 Git 命令自动带有一个分页器，默认使用 *less* 命令（*less -FRSX* ）进行分页。当一屏显示不下时启动分页器，这个分页器支持带颜色的字符输出，对于太长的行则采用截断方式处理。因为 *less* 分页器在翻屏时使用了 vi 风格的热键，如果您不熟悉 vi 的话，可能会遇到麻烦。下面是在分页器中常用的热键：

- 字母 q ：退出分页器。
- 字母 h ：显示分页器帮助。
- 按空格下翻一页，按字母 b 上翻一页。
- 字母 d 和 u ：分别代表向下翻动半页和向上翻动半页。
- 字母 j 和 k ：分别代表向上翻一行和向下翻一行。
- 如果行太长被截断，可以用左箭头和右箭头使得窗口内容左右滚动。
- 输入 */pattern* ：向下寻找和 pattern 匹配的内容。
- 输入 *?pattern* ：向上寻找和 pattern 匹配的内容。
- 字母 n 或 N：代表向前或向后继续寻找。
- 字母 g：跳到第一行；字母 G：跳到最后一行；输入数字再加字母 g：则跳转到对应的行。
- 输入 *!<command>* ：可以执行 Shell 命令。

如果不习惯分页器的长行截断模式而希望能够自动换行，可以通过设置 LESS 环境变量来实现。设置 LESS 环境变量如下：

```
$ export LESS=FRX
```

或者使用 Git 的方式，通过定义 Git 配置变量来改变分页器的默认行为。例如设置 *core.pager* 配置变量如下：

```
$ git config --global core.pager 'less -+$LESS -FRX'
```

2.12 快

您有项目托管在 sourceforge.net 的 CVS 或 SVN 服务器上么？或者因为公司的 SVN 服务器部署在另外一个城市需要经过互联网才能访问？

使用传统的集中式版本控制服务器，如果遇到上面的情况 —— 网络带宽没有保证，那么使用起来一定是慢得让人痛苦不堪。Git 作为分布式版本控制系统彻底解决了这个问题，几乎所有的操

作都在本地进行，而且还不是一般的快。

还有很多其他的分布式版本控制系统，如 Hg、Bazaar 等。和这些分布式版本控制系统相比，Git 在速度上也有优势，这源自于 Git 独特的版本库设计。第 2 篇的相关章节会向您展示 Git 独特的版本库设计。

其他很多版本控制系统，当输入检出、更新或克隆等命令后，只能双手合十然后望眼欲穿，因为整个操作过程就像是一个黑洞，不知道什么时候才能够完成。而 Git 在版本库克隆及与版本库同步的时候，能够实时地显示完成的进度，这不但是非常人性化的设计，更体现了 Git 的智能。Git 的智能协议源自于会话过程中在客户端和服务器端各自启用了一个会话的角色，按需传输以及获取进度。

第3章 安装 Git

Git 源自 Linux，现在已经可以部署在所有的主流平台之上，包括 Linux、Mac OS X 和 Windows。在开始我们的 Git 之旅之前，首先要做的就是安装 Git。

3.1 Linux 下安装和使用 Git

Git 诞生于 Linux 平台并作为版本控制系统率先服务于 Linux 核心，因此在 Linux 上安装 Git 是非常方便的。可以通过不同的方式在 Linux 上安装 Git。一种方法是通过 Linux 发行版的包管理器安装已经编译好的二进制格式的 Git 软件包。另外一种方式就是从 Git 源码开始安装。

3.1.1 包管理器方式安装

用 Linux 发行版的包管理器安装 Git 最为简单，而且会自动配置好命令补齐等功能，但安装的 Git 可能不是最新的版本。还有一点要注意，就是 Git 软件包在有的 Linux 发行版中可能不叫 git，而叫 git-core。这是因为有一款名为 GNU 交互工具[1]（GNU Interactive Tools）的 GNU 软件，在 Git 之前就在一些 Linux 发行版（Deian lenny）中占用了 git 的名称。为了以示区分，作为版本控制系统的 Git ，其软件包在这些平台就被命名为 git-core。不过因为作为版本控制系统的 Git 太有名了，最终导致在一些 Linux 发行版的最新版本中，将 GNU Interactive Tools 软件包由 git 改名为 gnuit，将 git-core 改名为 git。所以在下面介绍的在不同的 Linux 发行版中安装 Git 时，会看到有 git 和 git-core 两个不同的名称。

- Ubuntu 10.10（maverick）或更新的版本、Debian（squeeze）或更新的版本：

```
$ sudo aptitude install git
$ sudo aptitude install git-doc git-svn git-email git-gui gitk
```

其中 git 软件包包含了大部分 Git 命令，是必装的软件包。

软件包 git-svn、git-email、git-gui、gitk 本来也是 Git 软件包的一部分，但是因为有着不一样的软件包依赖（如更多的 perl 模组，tk 等），所以单独作为软件包发布。

软件包 git-doc 则包含了 Git 的 HTML 格式文档，可以选择安装。如果安装了 Git 的 HTML 格式的文档，则可以通过执行 *`git help -w <sub-command>`* 命令，自动用

[1] http://www.gnu.org/software/gnuit/

Web 浏览器打开相关子命令 *<sub-command>* 的 HTML 帮助。

- Ubuntu 10.04（lucid）或更老的版本、Debian（lenny）或更老的版本：

  在老版本的 Debian 中，软件包 git 实际上是 GNU Interactive Tools，而非作为版本控制系统的 Git。作为版本控制系统的 Git 在软件包 git-core 中。

```
$ sudo aptitude install git-core
$ sudo aptitude install git-doc git-svn git-email git-gui gitk
```

- RHEL、Fedora、CentOS：

```
$ yum install git
$ yum install git-svn git-email git-gui gitk
```

其他发行版安装 Git 的过程和上面介绍的方法类似。Git 软件包在这些发行版里或称为 git，或称为 git-core。

3.1.2 从源代码进行安装

访问 Git 的官方网站：http://git-scm.com/ 。下载 Git 源码包，例如：git-1.7.4.1.tar.bz2[1] 。安装过程如下：

（1） 展开源码包，并进入到相应的目录中。

```
$ tar -jxvf git-1.7.4.1.tar.bz2
$ cd git-1.7.4.1/
```

（2） 安装方法写在 INSTALL 文件当中，参照其中的指示完成安装。下面的命令将 Git 安装在 /usr/local/bin 中。

```
$ make prefix=/usr/local all
$ sudo make prefix=/usr/local install
```

（3） 安装 Git 文档（可选）。

编译的文档主要是 HTML 格式的文档，方便通过 *git help -w <sub-command>* 命令查看。实际上即使不安装 Git 文档，也可以使用 man 手册查看 Git 帮助，使用命令 *git help <sub-command>* 或 *git <sub-command> --help* 即可。

[1] 在您下载的时候可能已经有了更新的版本。

编译文档依赖 asciidoc，因此需要先安装 asciidoc（如果尚未安装的话），然后编译文档。在编译文档时要花费很多时间，要有耐心。

```
$ make prefix=/usr/local doc info
$ sudo make prefix=/usr/local install-doc install-html install-info
```

安装完毕之后，就可以在 */usr/local/bin* 下找到 *git* 命令。

3.1.3 从 Git 版本库进行安装

如果在本地克隆一个 Git 项目的版本库，就可以用版本库同步的方式获取最新版本的 Git，这样在下载不同版本的 Git 源代码时实际上采用了增量方式，非常的节省时间和空间。当然使用这种方法的前提是已经用其他方法安装好了 Git，具体操作过程如下。

（1） 克隆 Git 项目的版本库到本地。

```
$ git clone git://git.kernel.org/pub/scm/git/git.git
$ cd git
```

（2） 如果本地已经克隆过一个 Git 项目的版本库，直接在工作区中更新，以获得最新版本的 Git。

```
$ git pull
```

（3） 执行清理工作，避免前一次编译的遗留文件对编译造成影响。注意下面的操作将丢弃本地对 Git 代码的改动。

```
$ git clean -fdx
$ git reset --hard
```

（4） 查看 Git 的里程碑，选择最新的版本进行安装，例如 v1.7.4.1 。

```
$ git tag
...
v1.7.4.1
```

（5） 检出该版本的代码。

```
$ git checkout v1.7.4.1
```

（6） 执行安装。例如安装到 /usr/local 目录下。

```
$ make prefix=/usr/local all doc info
$ sudo make prefix=/usr/local install \
  install-doc install-html install-info
```

我在撰写本书的过程中，就通过 Git 版本库的方式安装，在 /opt/git 目录下安装了多个不同版本的 Git，以测试 Git 的兼容性。使用类似下面的脚本，可以批量安装不同版本的 Git。

```
#!/bin/sh

for ver in      \
    v1.5.0      \
    v1.7.3.5    \
    v1.7.4.1    \
; do
    echo "Begin install Git $ver.";
    git reset --hard
    git clean -fdx
    git checkout $ver || {
        echo "Checkout git $ver failed."; exit 1
    }
    make prefix=/opt/git/$ver all && \
    sudo make prefix=/opt/git/$ver install || {
        echo "Install git $ver failed."; exit 1
    }
    echo "Installed Git $ver."
done
```

3.1.4 命令补齐

Linux 的 shell 环境（bash）通过 bash-completion 软件包提供命令补齐功能，在录入命令参数时按一下或两下 TAB 键，实现参数的自动补齐或提示。例如输入 *`git com`* 后按下 TAB 键，会自动补齐为 *`git commit`* 。

通过包管理器方式安装 Git，一般都已经为 Git 配置好了自动补齐，但是如果是以源码编译的方式安装 Git，就需要为命令补齐多做些工作，具体操作过程如下。

（1） 将 Git 源码包中的命令补齐脚本复制到 bash-completion 对应的目录中。

```
$ cp contrib/completion/git-completion.bash \
      /etc/bash_completion.d/
```

（2） 重新加载自动补齐脚本，使之在当前的 shell 中生效。

```
$ . /etc/bash_completion
```

（3） 为了能够在终端开启时自动加载 bash_completion 脚本，需要在本地配置文件 *~/.bash_profile* 或全局文件 */etc/bashrc* 中添加下面的内容。

```
if [ -f /etc/bash_completion ]; then
  . /etc/bash_completion
fi
```

3.1.5 中文支持

Git 的本地化做的并不完善，命令的输出及命令的帮助还只能输出英文，也许在未来的版本中会使用 gettext 实现本地化，就像目前对 git-gui 命令所做的那样。

使用中文的用户最关心的问题还有：是否可以在提交说明中使用中文？是否可以使用中文文件名或目录名？是否可以使用中文来命名分支或里程碑？简单地说，可以在提交说明中使用中文，但是需要为 Git 做些设置。至于用中文来命名文件、目录和引用，只有在使用 UTF-8 字符集的环境下（Linux、Mac OS X、Windows 下的 Cygwin）才可以，否则尽量避免使用。

1. UTF-8 字符集

Linux 平台的中文用户一般会使用 UTF-8 字符集，Git 在 UTF-8 字符集下可以工作得非常好：

- 在提交时，可以在提交说明中输入中文。
- 显示提交历史，能够正常显示提交说明中的中文字符。
- 可以添加中文文件名的文件，并可以在同样使用 UTF-8 字符集的 Linux 环境中克隆及检出。
- 可以创建带有中文字符的里程碑名称。

但是默认设置下，带有中文文件名的文件，在工作区状态输出、查看历史更改概要，以及在补丁文件中，文件名不能正确显示为中文，而是用若干 8 进制字符编码来显示中文，如下：

```
$ git status -s
?? "\350\257\264\346\230\216.txt"
$ printf "\350\257\264\346\230\216.txt\n"
```

```
说明.txt
```

通过将变量 *core.quotepath* 设置为 *false* ，就可以解决中文文件名在这些 Git 命令输出中的显示问题。

```
$ git config --global core.quotepath false
$ git status -s
?? 说明.txt
```

2. GBK 字符集

但如果 Linux 平台采用非 UTF-8 的字符集，例如用 zh_CN.GBK 字符集编码（有人这么做么？），就要另外再做些工作了。

- 设置提交说明显示所使用的字符集为 *gbk*，这样使用 *git log* 查看提交说明才能够正确显示其中的中文。

  ```
  $ git config --global i18n.logOutputEncoding gbk
  ```

- 设置录入提交说明时所使用的字符集，以便在 commit 对象中正确标注字符集。

  Git 在提交时并不会对提交说明进行从 GBK 字符集到 UTF-8 的转换，但是可以在提交说明中标注所使用的字符集，因此在非 UTF-8 字符集的平台录入中文，需要用下面的指令设置录入提交说明的字符集，以便在 commit 对象中嵌入正确的编码说明。

  ```
  $ git config --global i18n.commitEncoding gbk
  ```

3.2 Mac OS X 下安装和使用 Git

Mac OS X 被称为最人性化的操作系统，工作在 Mac 上是件非常惬意的事情，工作中又怎能没有 Git 呢？

3.2.1 以二进制发布包的方式安装

Git 在 Mac OS X 中也有好几种安装方法。最简单的方式是安装 *.dmg* 格式的安装包。

访问 git-osx-installer 的官方网站：http://code.google.com/p/git-osx-installer/，下载 Git 安装包。安装包带有 *.dmg* 扩展名，是苹果磁盘镜像（Apple Disk Image）格式的软件发布包。从官方网站上下载文件名类似 git-<version>-<arch>-leopard.dmg 的安装包文件，例如：

git-1.7.4-x86_64-leopard.dmg 是 64 位的安装包，git-1.7.4-i386-leopard.dmg 是 32 位的安装包。建议选择 64 位的软件包，因为 Mac OS X 10.6 雪豹完美地兼容 32 位和 64 位（开机按住键盘数字 3 和 2 进入 32 位系统，按住 6 和 4 进入 64 位系统），即使在核心处于 32 位的架构下，也可以放心地运行 64 位的软件包。

苹果的 *.dmg* 格式的软件包实际上是一个磁盘映像，安装起来非常方便，点击该文件就可以直接挂载到 Finder 中，并打开，如图 3-1 所示。

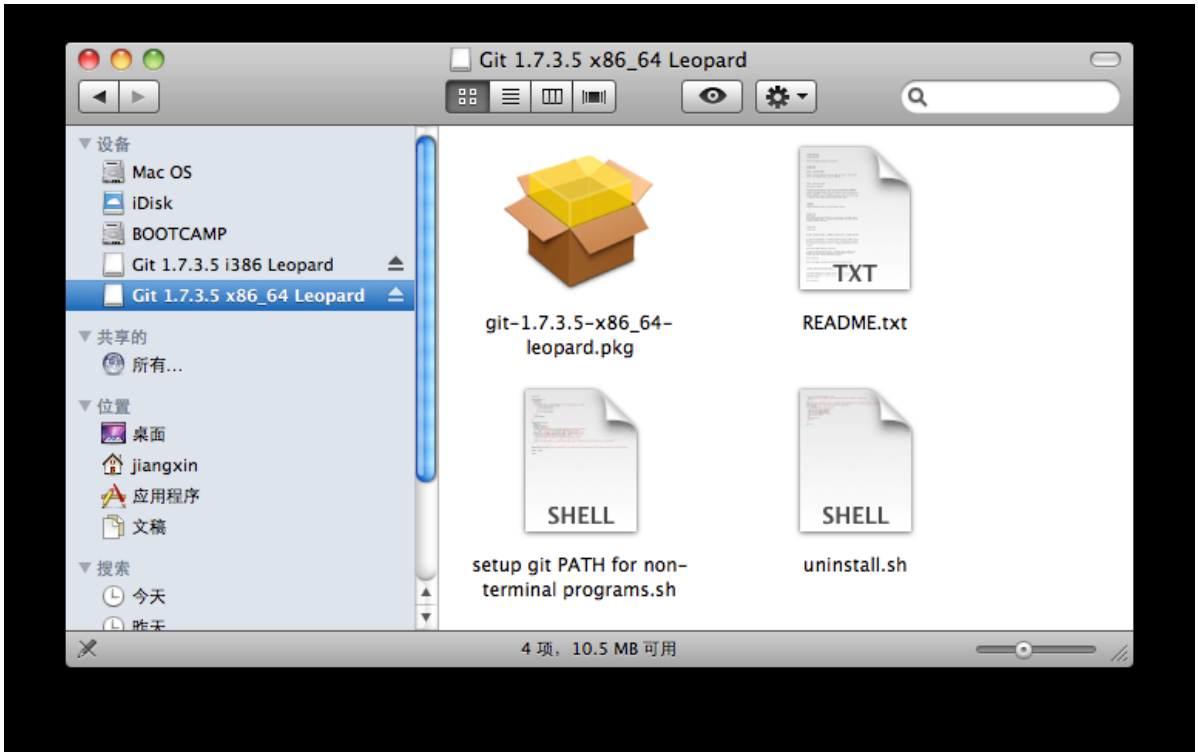

图 3-1：在 Mac OS X 下打开 .dmg 格式磁盘镜像

图 3-1 中显示为拆包图标的文件（扩展名为 *.pkg*）就是 Git 的安装程序，另外的两个脚本程序，一个用于应用的卸载（*uninstall.sh*），另外一个带有长长文件名的脚本在 Git 安装后再执行，为非终端应用注册 Git 的安装路径，这是因为 Git 部署在标准的系统路径之外 */usr/local/git/bin* 。

点击扩展名为 *.pkg* 的安装程序，开始 Git 的安装（以安装 Git1.7.3.5 版本为例），根据提示按步骤完成安装，如图 3-2 所示。

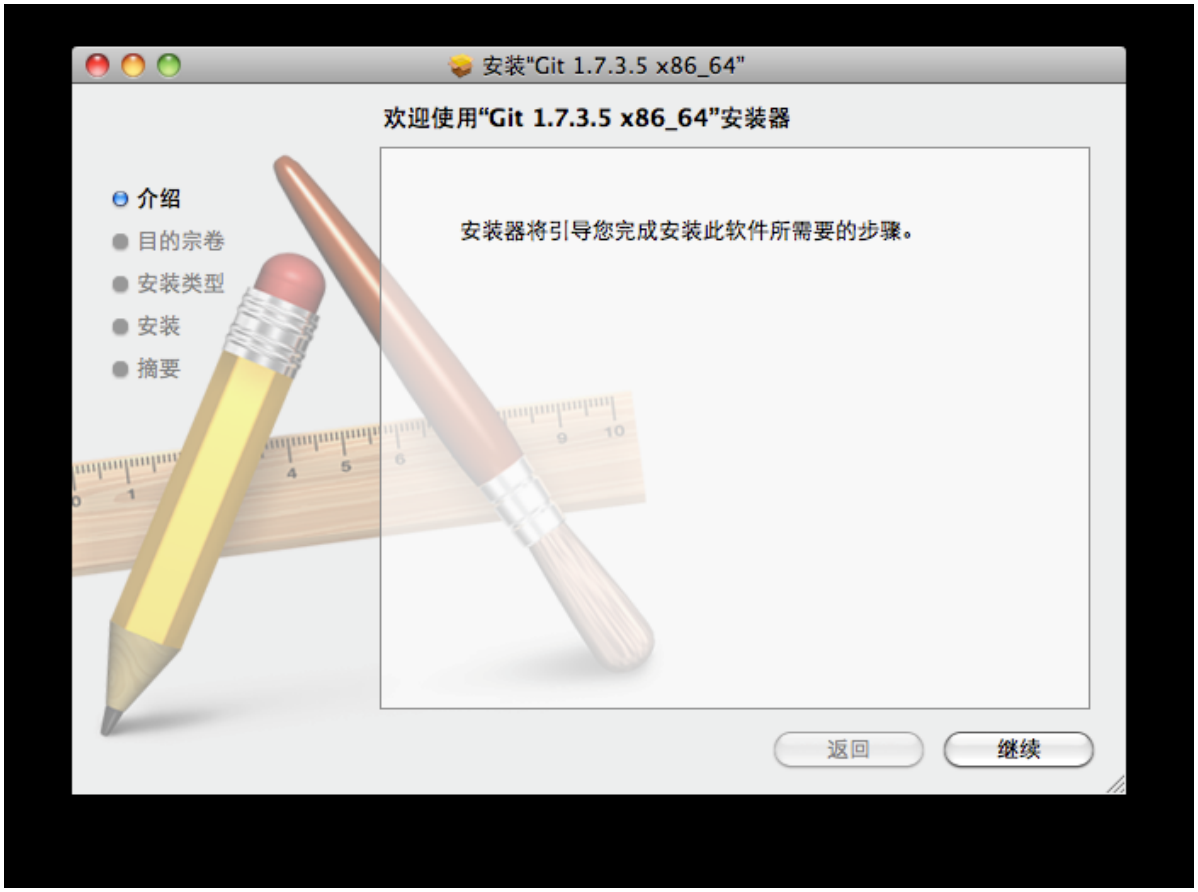

图 3-2：在 Mac OS X 下安装 Git

安装完毕，Git 会被安装到 `/usr/local/git/bin/` 目录下。重启终端程序，这样 `/etc/paths.d/git` 文件为 PATH 环境变量中添加的新路径注册才能生效，然后就可以在终端直接运行 `git` 命令了。

3.2.2 安装 Xcode

Mac OS X 基于 Unix 内核开发，因此也可以很方便地通过源码编译的方式进行安装，但是默认安装的 Mac OS X 缺乏相应的开发工具，需要安装苹果提供的 Xcode 软件包。在 Mac 随机附送的光盘（Mac OS X Install DVD）的可选安装文件夹下就有 Xcode 的安装包（如图 3-3 所示），通过随机光盘安装 Xcode 可以省去了网络下载的麻烦，要知道 Xcode 有 3GB 以上。

图 3-3：在 Mac OS X 下安装 Xcode

3.2.3 使用 Homebrew 安装 Git

Mac OS X 有好几个包管理器，用于对一些开源软件在 Mac OS X 上的安装和升级进行管理。有传统的 MacPorts[1]、Fink[2]，还有更为简单易用的 Homebrew[3]。下面就介绍一下如何通过 Homebrew 包管理器，以源码包编译的方式安装 Git。

Homebrew 用 ruby 语言开发，支持千余种开源软件在 Mac OS X 中的部署和管理。Homebrew

[1] http://www.macports.org/

[2] http://www.finkproject.org/

[3] http://mxcl.github.com/homebrew/

项目托管在 Github 上，网址为：https://github.com/mxcl/homebrew 。

首先是安装 Homebrew，执行下面的命令：

```
$ ruby -e \
"$(curl -fsSL https://gist.github.com/raw/323731/install_homebrew.rb)"
```

安装完成后，Homebrew 的主程序安装在 */usr/local/bin/brew* 中，在目录 */usr/local/Library/Formula/* 下保存了 Homebrew 支持的所有软件的安装指引文件。

执行下面的命令，通过 Homebrew 安装 Git。

```
$ brew install git
```

使用 Homebrew 方式安装，Git 被安装在 */usr/local/Cellar/git/<version>/* 中，可执行程序自动在 */usr/local/bin* 目录下创建符号连接，可以直接在终端程序中访问。

通过 *brew list* 命令可以查看安装的开源软件包。

```
$ brew list
git
```

也可以查看某个软件包安装的详细路径和安装内容。

```
$ brew list git
/usr/local/Cellar/git/1.7.4.1/bin/gitk
...
```

3.2.4 从 Git 源码进行安装

如果需要安装历史版本的 Git 或是安装尚在开发中的未发布版本的 Git，就需要从源码安装或通过克隆 Git 源码库进行安装。既然 Homebrew 就是通过源码编译方式安装 Git 的，那么也应该可以直接从源码进行安装，但是使用 Homebrew 安装 Git 和直接通过 Git 源码安装并不完全等同，例如 Homebrew 安装 Git 的过程中，是通过下载已经编译好的 Git 文档包进行安装，而非从头对文档进行编译。

直接通过源码安装 Git 软件及文档，遇到的主要问题就是对文档的编译，因为 Xcode 没有提供 Git 文档编译所需要的相关工具。但是这些工具可以通过 Homebrew 进行安装，下面安装工具软件的过程可能会遇到一些小麻烦，不过大多可以通过参考命令输出予以解决。

```
$ brew install asciidoc
$ brew install docbook2x
$ brew install xmlto
```

当编译源码及文档的工具部署完全后，就可以通过源码编译 Git。

```
$ make prefix=/usr/local all doc info
$ sudo make prefix=/usr/local install \
  install-doc install-html install-info
```

3.2.5 命令补齐

Git 通过 bash-completion 软件包实现命令自动补齐，在 Mac OS X 下可以通过 Homebrew 进行安装。

```
$ brew search completion
bash-completion
$ brew install bash-completion
...
Add the following lines to your ~/.bash_profile file:
if [ -f `brew --prefix`/etc/bash_completion ]; then
  . `brew --prefix`/etc/bash_completion
fi
...
```

根据 bash-completion 安装过程中的提示，修改文件 *~/.bash_profile* ，并在其中加入如下内容，以便在终端加载时自动启用命令补齐。

```
if [ -f `brew --prefix`/etc/bash_completion ]; then
  . `brew --prefix`/etc/bash_completion
fi
```

将 Git 的命令补齐脚本拷贝到 bash-completion 对应的目录中。

```
$ cp contrib/completion/git-completion.bash \
      `brew --prefix`/etc/bash_completion.d/
```

不用重启终端程序，只需要运行下面的命令，即可立即在当前的 shell 中加载命令补齐。

```
. `brew --prefix`/etc/bash_completion
```

3.2.6 其他辅助工具的安装

本书中还会用到一些常用的 GNU 或其他开源软件，在 Mac OS X 下也可以通过 Homebrew 进行安装。这些软件包有：

- gnupg：数字签名和加密工具。在为 Git 版本库建立签名里程碑时会用到。
- md5sha1sum：生成 MD5 或 SHA1 摘要。在研究 Git 版本库的对象时会用到。
- cvs2svn：CVS 版本库迁移到 SVN 或 Git 的工具。在版本库迁移时会用到。

- stgit：Git 的补丁和提交管理工具。
- quilt：一种补丁管理工具。在介绍 StGit 时会用到。

在 Mac OS X 下能够使用到的 Git 图形工具除了 Git 软件包自带的 *gitk* 和 *git gui* 之外，还可以安装 GitX。下载地址为：

- GitX 的原始版本：http://gitx.frim.nl/
- 或 GitX 的一个分支版本，提供增强的功能：https://github.com/brotherbard/gitx/downloads

Git 的图形工具一般需要在本地克隆版本库的工作区中执行，为了能和 Mac OS X 有更好的整合，可以安装插件实现和 Finder 的整合。在 git-osx-installer 的官方网站：http://code.google.com/p/git-osx-installer/ 上有两个以 *OpenInGitGui-* 和 *OpenInGitX-* 为前缀的软件包，可以分别实现和 *git gui* 及 *gitx* 的整合：在 Finder 中进入工作区目录，点击对应插件的图标，启动 *git gui* 或 *gitx* 。

3.2.7 中文支持

由于 Mac OS X 采用 Unix 内核，在中文支持上和 Linux 相近，具体内容请参照前面 3.1.5 节介绍的在 Linux 下安装 Git 的相关内容。

3.3 Windows 下安装和使用 Git（Cygwin 篇）

在 Windows 下安装和使用 Git 有两个不同的方案，通过安装 msysGit[1] 或 Cygwin[2] 来使用 Git。在这两种不同的方案下，Git 的使用和在 Linux 下的使用完全一致。再有一个就是基于 msysGit 的图形界面软件 —— TortoiseGit[3]，也就是在 CVS 和 SVN 时代就已经广为人知的 Tortoise 系列软件的 Git 版本。TortoiseGit 提供和资源管理器的整合，提供 Git 操作的图形化界面。

先介绍通过 Cygwin 来使用 Git 的原因，不是因为这是最便捷的方法，如果需要在 Windows 中快速安装和使用 Git，下节介绍的 msysGit 也许是最佳方法。之所以将 Cygwin 放在前面介绍是因为本书在介绍 Git 原理的部分，以及介绍其他 Git 相关软件的部分用到了大量的开源工具，

[1] http://code.google.com/p/msysgit/

[2] http://www.cygwin.com/

[3] http://code.google.com/p/tortoisegit/

在 Cygwin 下很容易获得这些开源工具，而 msysGit 的 MSYS[1]（Minimal SYStem，最简系统）则不能满足我们的需要。因此我建议使用 Windows 平台的读者，在跟随本书学习 Git 的过程中首选 Cygwin，当对 Git 有了一定经验后，无论是 msysGit 还是 TortoiseGit 您都会应对自如。

Cygwin 是一款伟大的软件，通过一个小小的 DLL（cygwin1.dll）建立了 Linux 和 Windows 系统调用及 API 之间的转换，实现了 Linux 下绝大多数软件到 Windows 的迁移。Cygwin 通过 cygwin1.dll 所建立的中间层和诸如 VMWare[2]、VirtualBox[3] 等的虚拟机软件完全不同，不会独占系统资源。像 VMWare 等虚拟机，只要启动一个虚拟机（操作系统），即使不在其中执行任何命令，同样也会占用大量的系统资源：内存、CPU 时间，等等。

Cygwin 还提供了一个强大易用的包管理工具（setup.exe），实现了几千个开源软件包在 Cygwin 下便捷的安装和升级，Git 就是 Cygwin 下支持的几千个开源软件中的一员。

我对 Cygwin 有着深厚的感情，Cygwin 让我在 Windows 平台能用 Linux 的方式更有效率地做事，使用 Linux 风格的控制台替换 Windows 黑乎乎的、冷冰冰的、由 cmd.exe 提供的命令行。Cygwin 帮助我逐渐摆脱对 Windows 的依赖，当我完全转换到 Linux 平台时，没有感到一丝的障碍。

3.3.1 安装 Cygwin

安装 Cygwin 非常简单，访问其官方网站 http://www.cygwin.com/ ，下载安装程序 —— 一个只有几百 KB 的 *`setup.exe`* ，即可开始安装。

安装过程会让用户选择安装模式，可以选择网络安装、仅下载，或者通过本地软件包缓存（在安装时自动在本地目录下建立软件包缓存）进行安装。如果是第一次安装 Cygwin，因为本地尚没有软件包缓存，当然只能选择从网络安装，如图 3-4 所示。

[1] http://www.mingw.org/wiki/msys

[2] http://www.vmware.com/

[3] http://www.virtualbox.org/

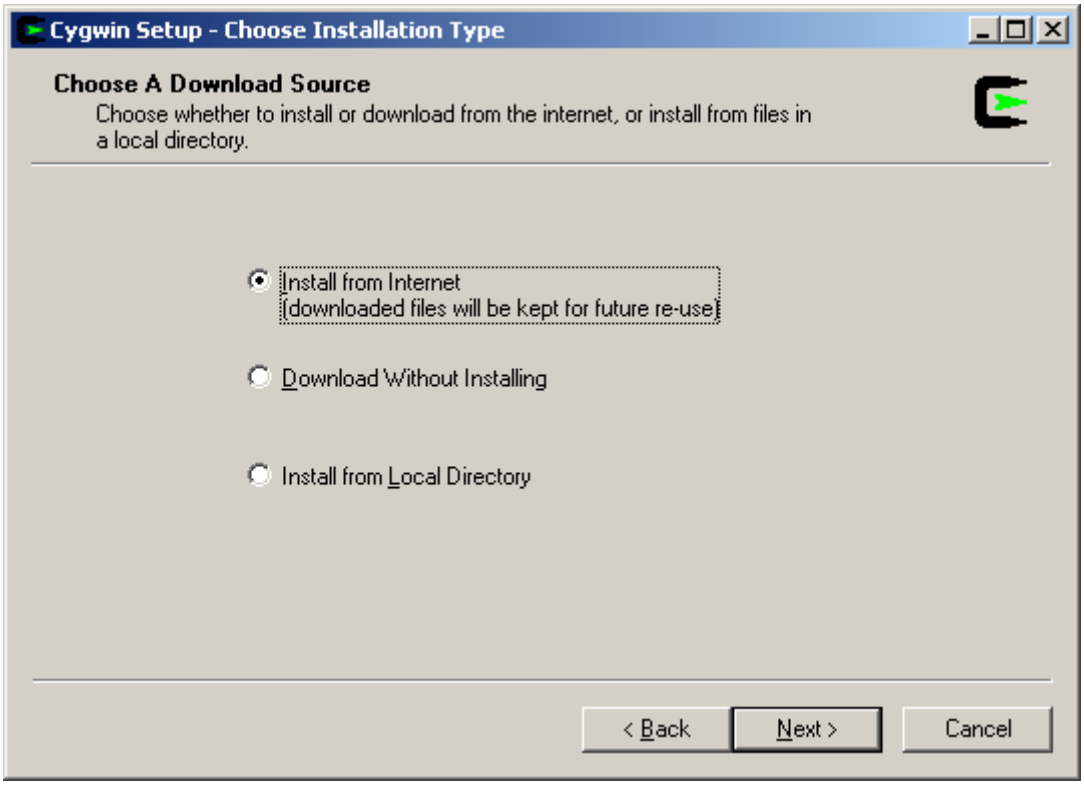

图 3-4：选择安装模式

接下来，Cygwin 询问安装目录，默认为 `C:\cygwin` ，如图 3-5 所示。这个目录将作为 Cygwin shell 环境的根目录（根卷），Windows 的各个盘符将挂载在根卷的一个特殊目录之下。

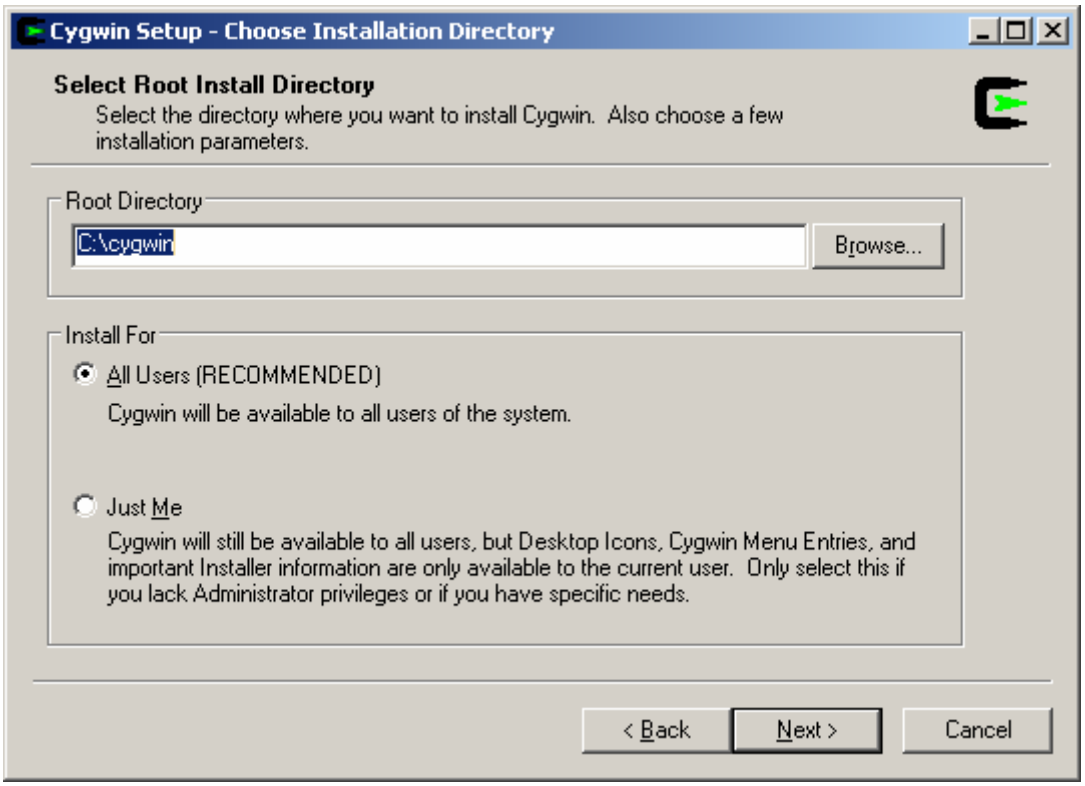

图 3-5：选择安装目录

询问本地软件包缓存目录，默认是 `setup.exe` 所处的目录，如图 3-6 所示。

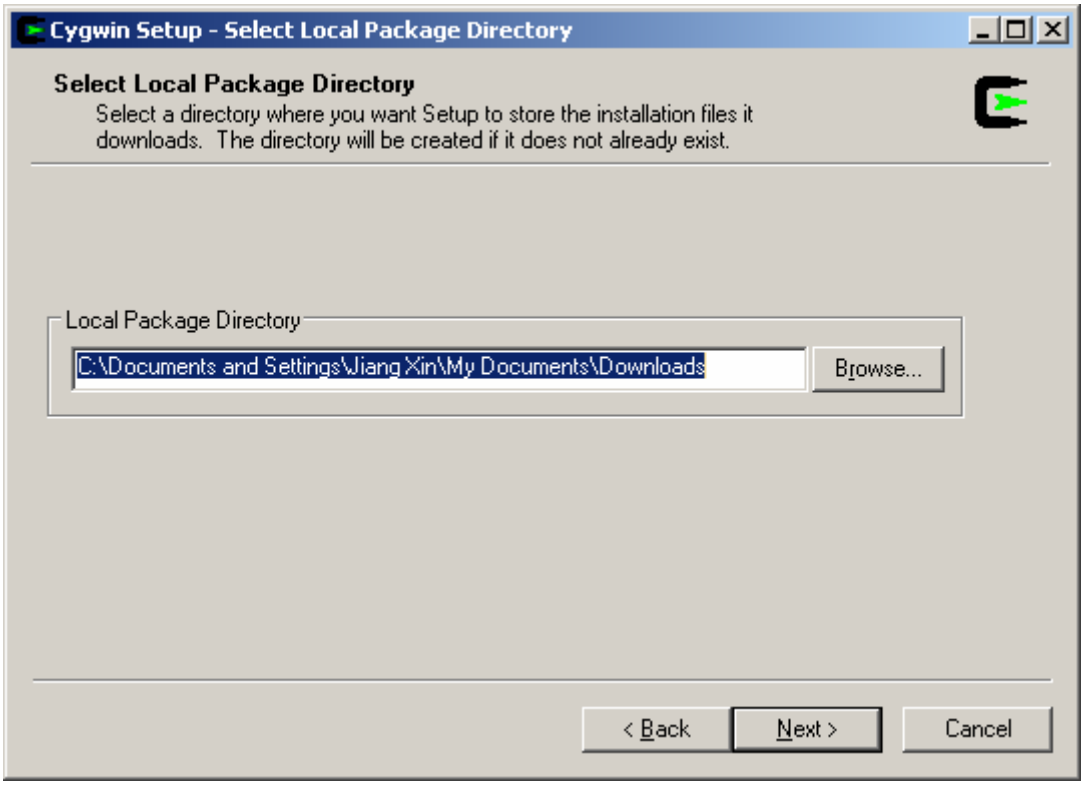

图 3-6：选择本地软件包缓存目录

询问网络连接方式是否使用代理等，如图 3-7 所示。默认会选择第一项：“直接网络连接”。如果一个团队有很多人要使用 Cygwin，架设一个能够提供软件包缓存的 HTTP 代理服务器会节省大量的网络带宽和大把的时间。在 Debian/Ubuntu 下用 apt-cacher-ng[1] 就可以非常简单地搭建一个软件包代理服务器。图 3-7 显示的就是我在公司内网中安装 Cygwin 时使用内网的服务器 `bj.ossxp.com` 作为 HTTP 代理的截图，端口设置为 9999，因为这是 apt-cacher-ng 的默认端口。

[1] http://www.unix-ag.uni-kl.de/~bloch/acng/

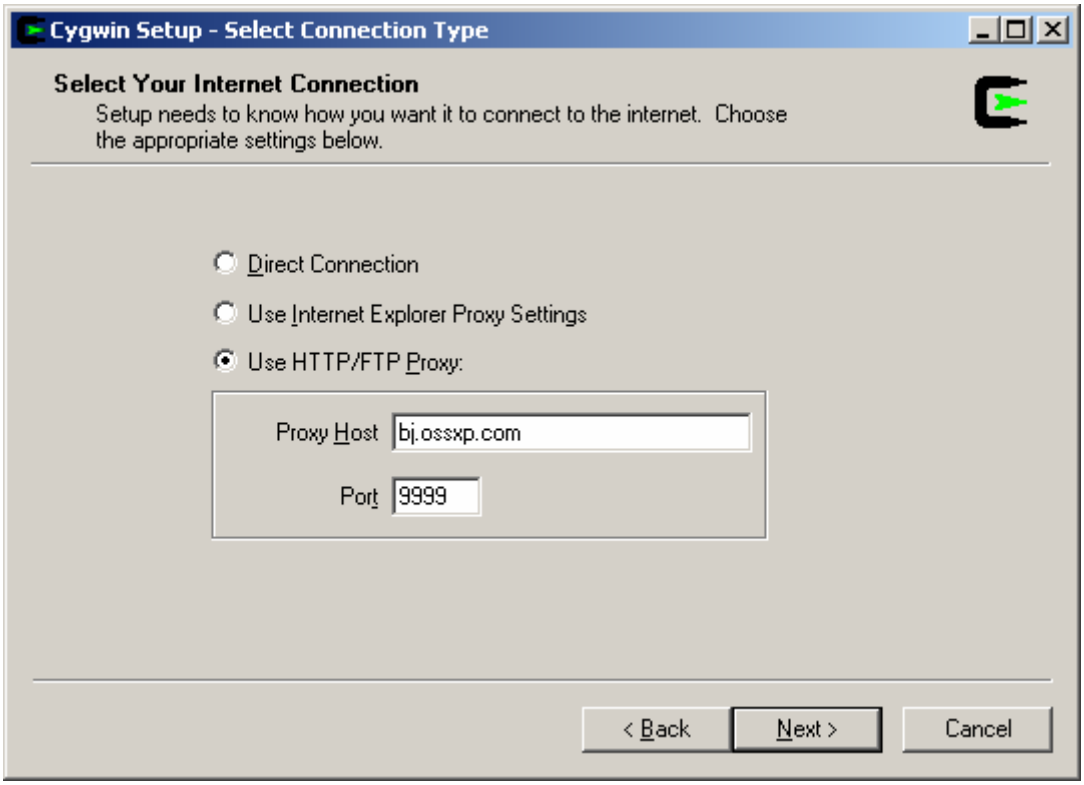

图 3-7：是否使用代理下载 Cygwin 软件包

选择一个 Cygwin 源，如图 3-8 所示。如果在上一个步骤中选择使用 HTTP 代理服务器，就必须选择 HTTP 协议的 Cygwin 源。

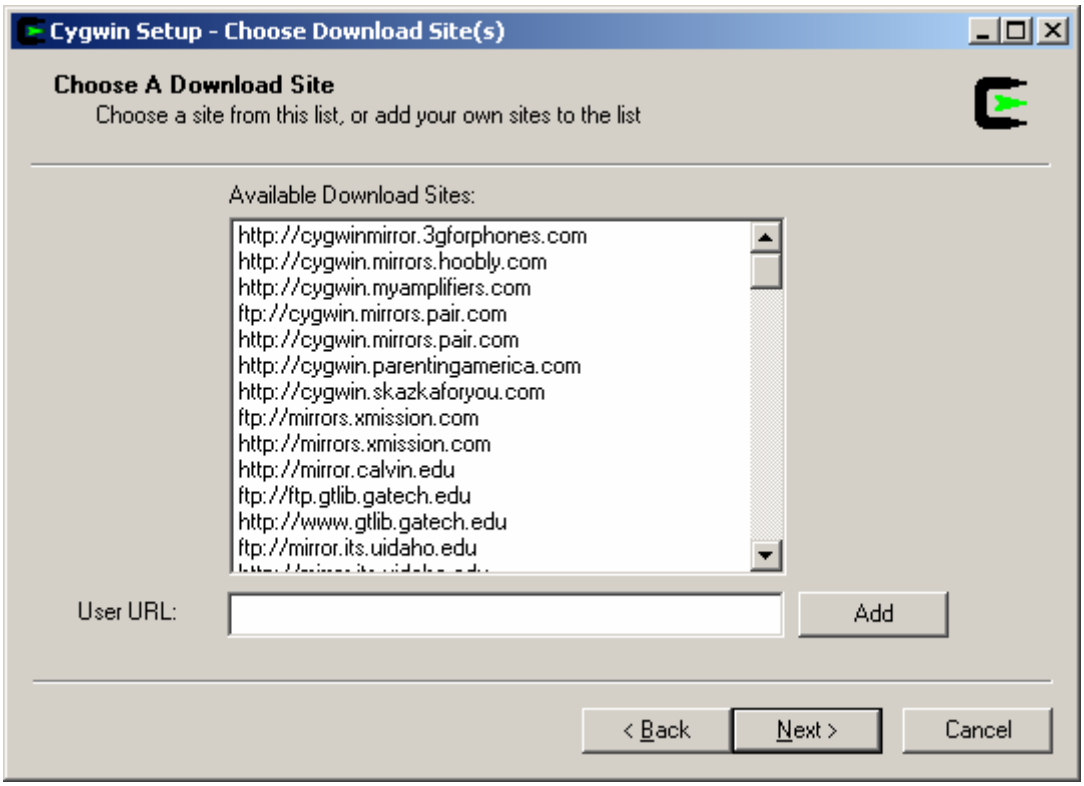

图 3-8：选择 Cygwin 源

接下来就会从所选的 Cygwin 源下载软件包索引文件，然后显示软件包管理器界面，如图 3-9 所示。

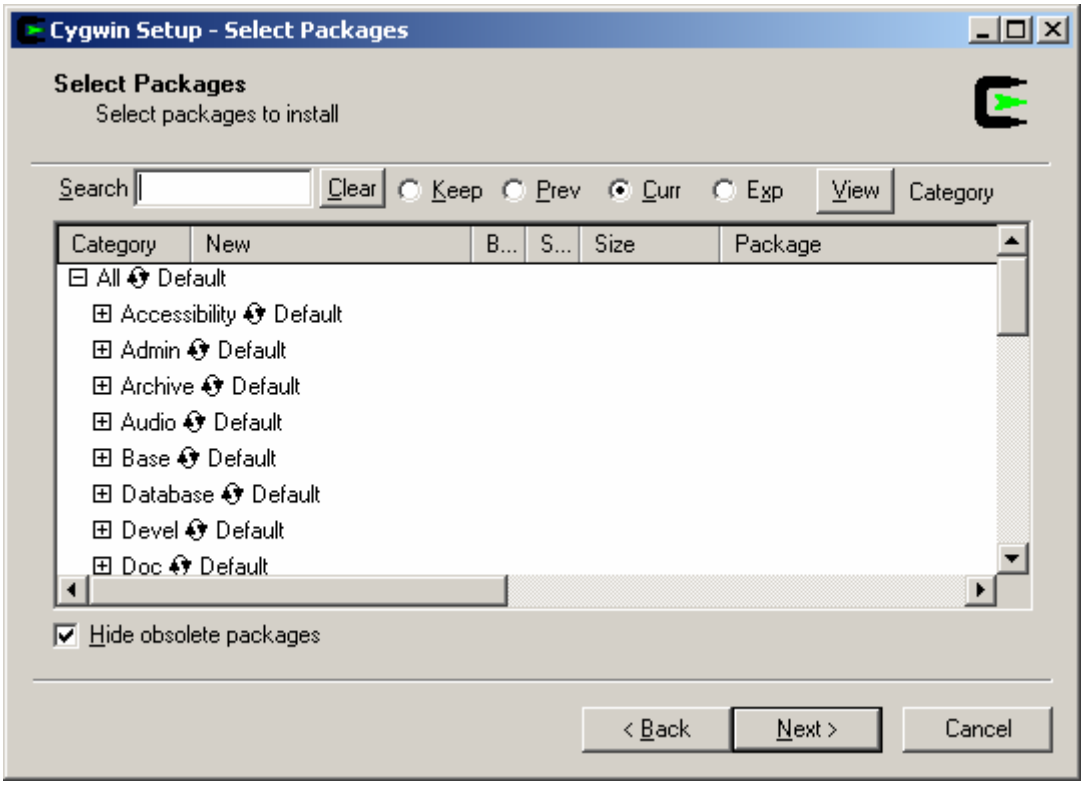

图 3-9：Cygwin 软件包管理器

Cygwin 的软件包管理器非常强大和易用（如果习惯了其界面）。软件包归类于各个分组中，点击分组前的加号就可以展开分组。在展开的 Admin 分组中，如图 3-10 所示（这个截图不是首次安装 Cygwin 的截图），有的软件包如 *libattr1* 已经安装过了，因为没有新版本而标记为“Keep”（保持）。至于没有安装过并且不准备安装的软件包则标记为 “Skip”（跳过）。

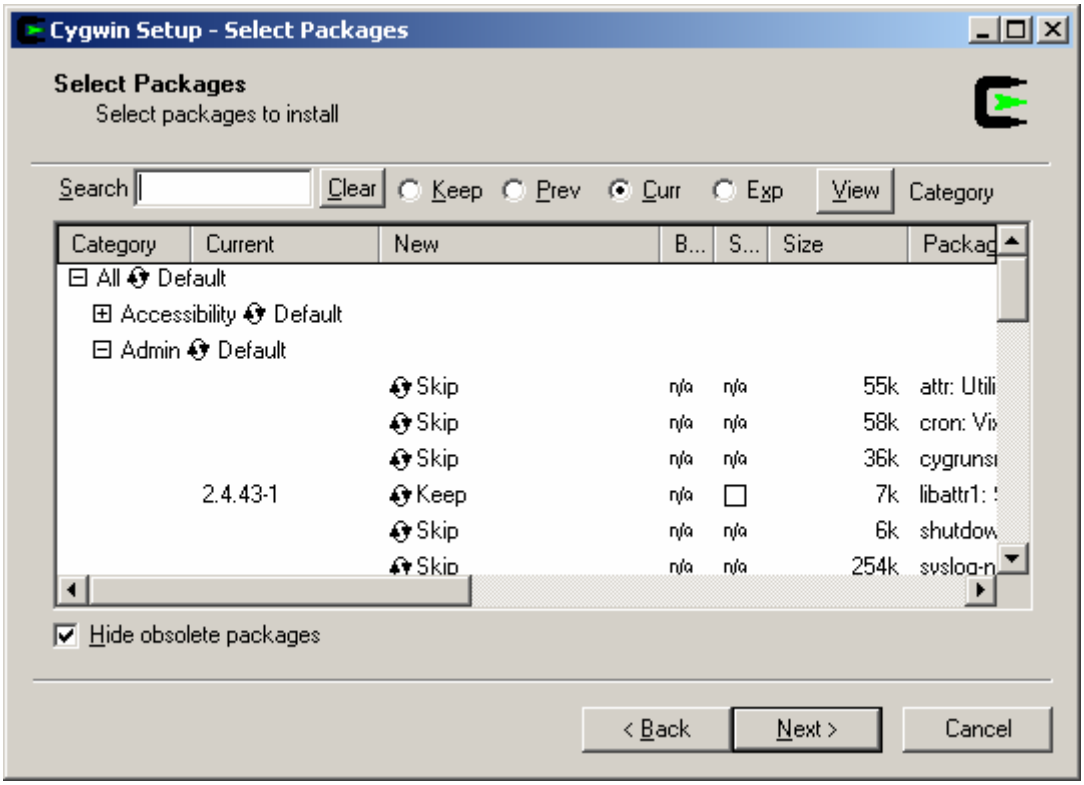

图 3-10：Cygwin 软件包管理器展开分组

点击分组名称后面的动作名称（文字“Default”），会进行软件包安装动作的切换。例如图 3-11，将 Admin 分组的安装动作由“Default”（默认）切换为“Install”（安装），会看到 Admin 分组下的所有软件包都标记为安装（显示具体要安装的软件包版本号）。也可以通过鼠标点击，单独为软件包进行安装动作的设定：可以强制重新安装、安装旧版本，或者不安装。

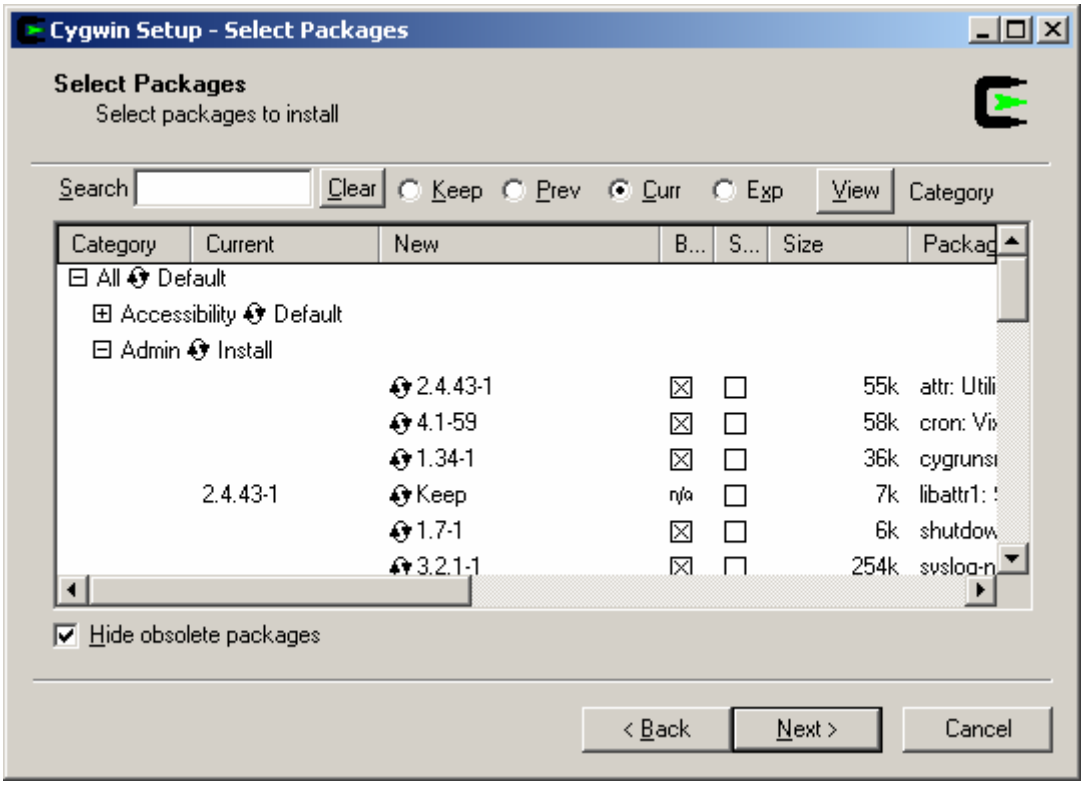

图 3-11：安装某一分组下所有软件

当通过软件包管理器对要安装的软件包定制完毕后，点击下一步，开始下载软件包、安装软件包和软件包后处理（postinstall），直至完成安装。根据选择的软件包的多少、网络情况，以及是否架设有代理服务器等，首次安装 Cygwin 的时间可能从几分钟到几个小时不等。

3.3.2 安装 Git

默认安装的 Cygwin 没有安装 Git 软件包。如果在首次安装过程中忘记通过包管理器选择安装 Git 或其他相关软件包，可以在安装后再次运行 Cygwin 的安装程序 `setup.exe` 。当再次进入 Cygwin 包管理器界面时，在搜索框中输入 git，如图 3-12 所示。

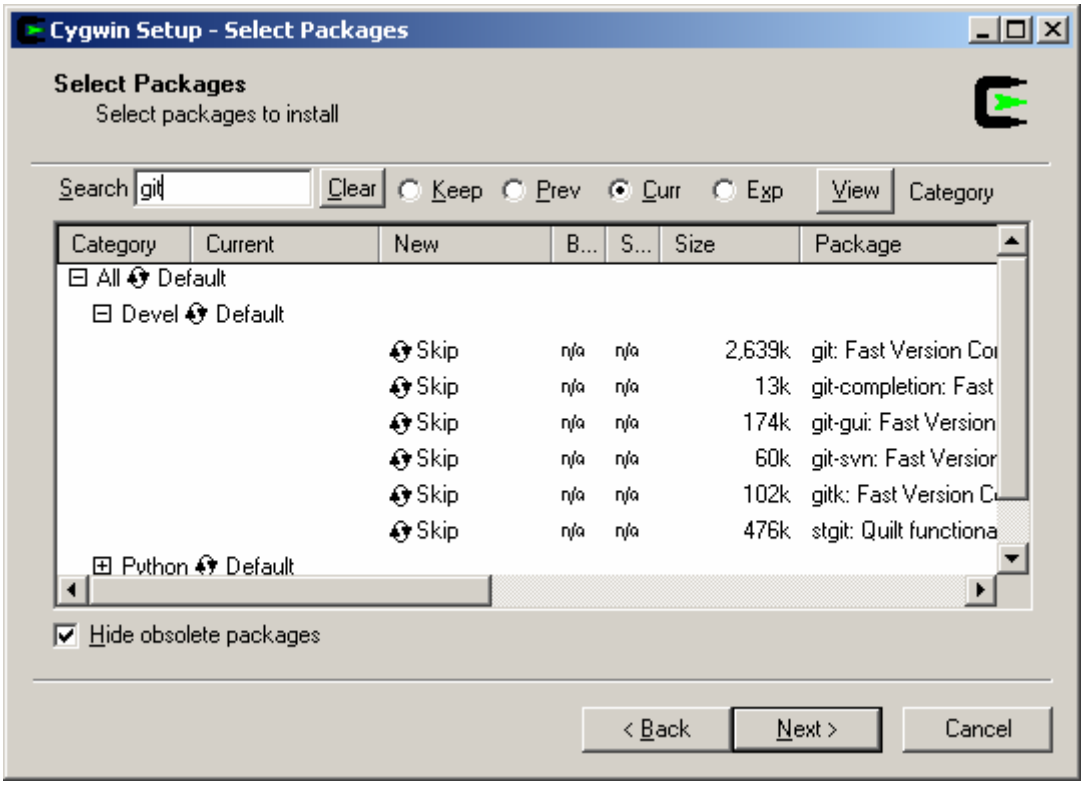

图 3-12：Cygwin 软件包管理器中搜索 git

从图 3-12 中可以看出在 Cygwin 中包含了很多和 Git 相关的软件包，把这些 Git 相关的软件包全部都安装上吧，如图 3-13 所示。

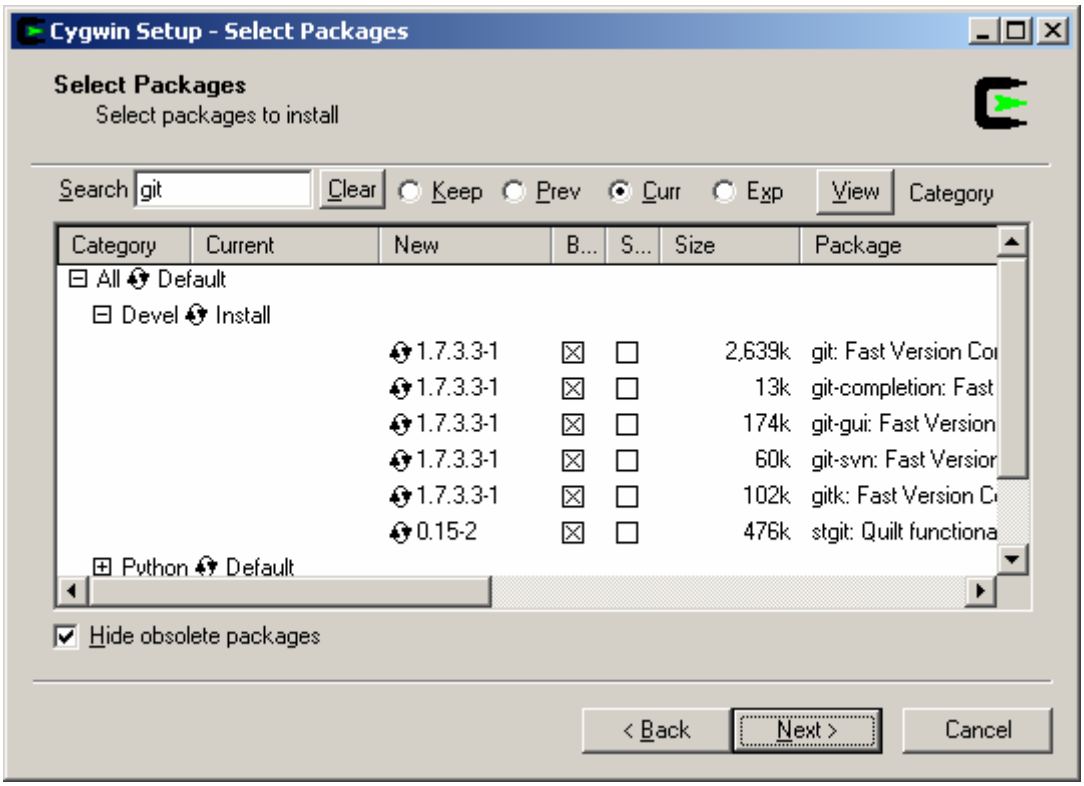

图 3-13：Cygwin 软件包管理器中安装 git

需要安装的其他软件包还有：

- git-completion：提供 Git 命令的自动补齐功能。安装该软件包会自动安装其所依赖的 bash-completion 软件包。
- openssh：SSH 客户端，为访问 SSH 协议的版本库提供支持。
- vim：Git 默认的编辑器。

3.3.3 Cygwin 的配置和使用

运行 Cygwin，就会进入 shell 环境中，见到熟悉的 Linux 提示符，如图 3-14 所示。

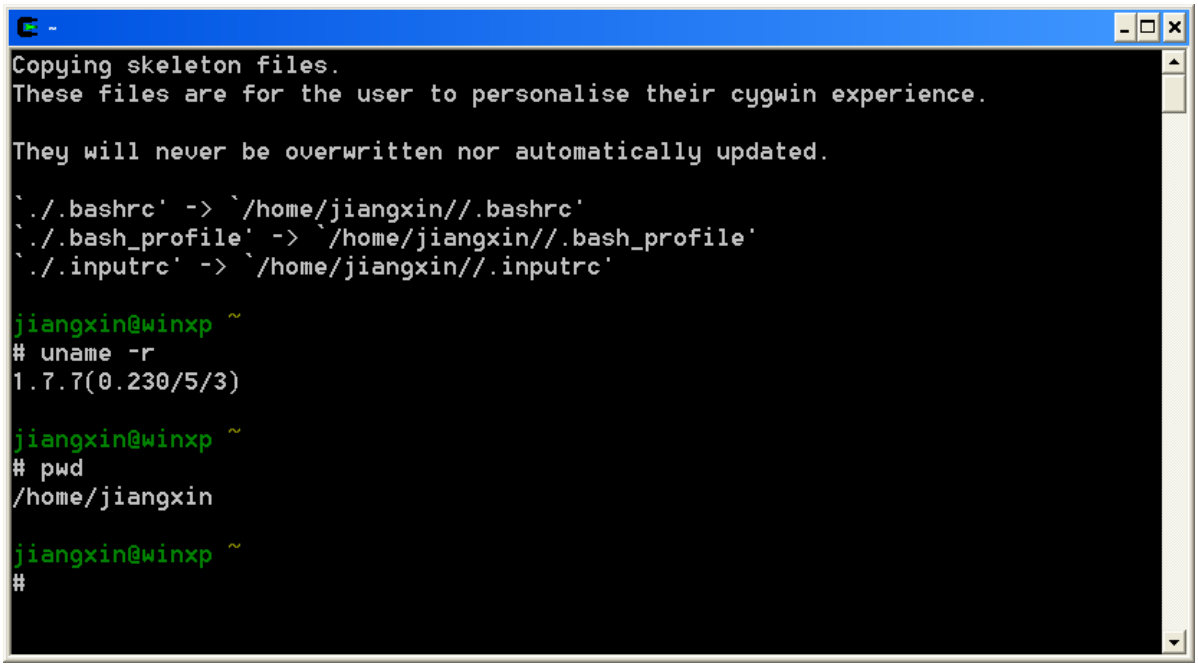

图 3-14：运行 Cygwin

可以通过执行 *cygcheck* 命令来查看 Cygwin 中安装的软件包的版本，例如查看 cygwin 软件包本身的版本：

```
$ cygcheck -c cygwin
Cygwin Package Information
Package              Version        Status
cygwin               1.7.7-1        OK
```

1. 如何访问 Windows 的盘符

刚刚接触 Cygwin 的用户遇到的头一个问题就是 Cygwin 如何访问 Windows 的各个磁盘目录，以及在 Windows 平台如何访问 Cygwin 中的目录？

执行 *mount* 命令，可以看到 Windows 下的盘符被映射到 */cygdrive* 特殊目录下。

```
$ mount
C:/cygwin/bin on /usr/bin type ntfs (binary,auto)
C:/cygwin/lib on /usr/lib type ntfs (binary,auto)
C:/cygwin on / type ntfs (binary,auto)
C: on /cygdrive/c type ntfs (binary,posix=0,user,noumount,auto)
D: on /cygdrive/d type ntfs (binary,posix=0,user,noumount,auto)
```

也就是说在 Cygwin 中以路径 */cygdrive/c/Windows* 来访问 Windows 下的 *C:\Windows* 目录。实际上 Cygwin 提供一个命令 *cygpath* 实现 Windows 平台和 Cygwin 之间目录名称的变换。如下：

```
$ cygpath -u C:\\Windows
/cygdrive/c/Windows

$ cygpath -w ~/
C:\cygwin\home\jiangxin\
```

从上面的示例也可以看出，Cygwin 下的用户主目录（即 */home/jiangxin/* ）相当于 Windows 下的 *C:\cygwin\home\jiangxin* 目录。

2. 用户主目录不一致的问题

如果其他某些软件(如 msysGit)为 Windows 设置了 HOME 环境变量，会影响到 Cygwin 中用户主目录的设置，甚至造成在 Cygwin 中不同的命令有不同的用户主目录的设置。例如：Cygwin 下 Git 的用户主目录被设置为 */cygdrive/c/Documents and Settings/jiangxin*，而 SSH 客户端软件的主目录被设置为 */home/jiangxin*，这会造成用户的困惑。

出现这种情况，是因为 Cygwin 确定用户主目录有几个不同的依据，要按照顺序确定主目录：首先查看系统的 HOME 环境变量，其次查看 */etc/passwd* 中为用户设置的主目录。有的软件遵照这个原则，而有些 Cygwin 应用如 SSH，却没有使用 HOME 环境变量而是直接使用 */etc/passwd* 中的设置。要想避免在同一个 Cygwin 环境下有两个不同的用户主目录设置，可以采用下面两种方法。

- 方法 1：修改 Cygwin 启动的批处理文件（如：*C:\cygwin\Cygwin.bat* ），在批处理的开头添加如下的一行，就可以防止其他软件在 Windows 引入的 HOME 环境变量被带入到 Cygwin 中。

  ```
  set HOME=
  ```

- 方法 2：如果希望使用 HOME 环境变量指向的主目录，则通过手工编辑 /etc/passwd 文件，将其中的用户主目录修改成 HOME 环境变量所指向的目录[1]。

3. 命令行补齐忽略文件名大小写

Windows 的文件系统忽略文件名大小写，在 Cygwin 下最好对命令行补齐进行相关设置以忽略大小写，这样使用起来更方便。

编辑文件 *~/.inputrc* ，在其中添加设置“set completion-ignore-case on”，或者取消已有的相关设置前面的井号注释符。修改完毕后，再重新进入 Cygwin，就可以实现命令行补齐对文件名大小写的忽略。

4. 忽略文件权限的可执行位

Linux、Unix、Mac OS X 下的可执行文件对文件权限都有特殊的设置（设置文件的可执行位），Git 可以跟踪文件的可执行位，即在添加文件时会把文件的权限也记录在其中。在 Windows 上，缺乏对文件可执行位的支持和需要，虽然 Cygwin 可以模拟 Linux 下的文件授权并对文件的可执行位进行支持，但一来为支持文件权限而调用 Cygwin 的 stat() 和 lstat() 函数会比 Windows 自身的 Win32 API 要慢两倍[2]，二来对于非跨平台的项目也没有必要对文件权限位进行跟踪，还有其他 Windows 下的工具及操作可能会破坏文件的可执行位，导致 Cygwin 下的 Git 认为文件的权限更改需要重新提交。通过下面的配置，可以禁止 Git 对文件权限的跟踪：

```
$ git config --system core.fileMode false
```

在此模式下，当已添加到版本库中的文件其权限的可执行位改变时，该文件不会显示有改动。新增到版本库的文件，无论文件本身是否设置为可执行，都以 100644 的权限（忽略可执行位）进行添加。

关于 Cygwin 的更多定制和帮助，请参见网址：http://www.cygwin.com/cygwin-ug-net/ 。

[1] http://www.cygwin.com/cygwin-ug-net/ntsec.html

[2] git-config(1) 用户手册中关于 core.ignoreCygwinFSTricks 的介绍

3.3.4 Cygwin 下 Git 的中文支持

Cygwin 的当前版本 1.7.x，对中文的支持非常好。无需任何配置就可以在 Cygwin 的窗口内输入中文，以及执行 `ls` 命令显示中文文件名。这与我记忆中的 6、7 年前的 Cygwin 1.5.x 完全不一样了。老版本的 Cygwin 还需要做一些工作才能在控制台输入中文和显示中文，但是最新版本的 Cygwin 已经完全不需要了。反倒是后面要介绍的 msysGit 的 shell 环境仍然需要做出类似（老版本 Cygwin）的改动才能够正常显示和输入中文。

Cygwin 默认使用 UTF-8 字符集，并巧妙地和 Windows 系统的字符集进行转换。在 Cygwin 下执行 `locale` 命令查看 Cygwin 下正在使用的字符集。

```
$ locale
LANG=C.UTF-8
LC_CTYPE="C.UTF-8"
LC_NUMERIC="C.UTF-8"
LC_TIME="C.UTF-8"
LC_COLLATE="C.UTF-8"
LC_MONETARY="C.UTF-8"
LC_MESSAGES="C.UTF-8"
LC_ALL=
```

正因为如此，Cygwin 下的 Git 对中文的支持非常出色，虽然中文 Windows 本身使用 GBK 字符集，但是在 Cygwin 下，Git 的行为就如同工作在 UTF-8 字符集的 Linux 下，对中文的支持非常的好：

- 在提交时，可以在提交说明中输入中文。
- 显示提交历史，能够正常显示提交说明中的中文字符。
- 可以添加中文文件名的文件，并可以在使用 UTF-8 字符集的 Linux 环境中克隆及检出。
- 可以创建带有中文字符的里程碑名称。

但是和 Linux 平台一样，在默认设置下，带有中文文件名的文件，在工作区状态输出、查看历史更改概要，以及在补丁文件中，文件名不能正确地显示为中文，而是用若干 8 进制字符编码来显示，如下：

```
$ git status -s
?? "\350\257\264\346\230\216.txt"
$ printf "\350\257\264\346\230\216.txt"
说明.txt
```

通过设置配置变量 `core.quotepath` 为 `false` ，就可以解决中文文件名在这些 Git 命令

输出中的显示问题。

```
$ git config --global core.quotepath false
$ git status -s
?? 说明.txt
```

3.3.5 Cygwin 下 Git 访问 SSH 服务

在本书第 5 篇“第 29 章 使用 SSH 协议”中介绍的以公钥认证的方式访问 Git 服务，是 Git 写操作最重要的服务。以公钥认证方式访问 SSH 协议的 Git 服务器时无须输入口令，而且更安全。使用公钥认证就涉及到如何创建公钥/私钥对，以及在 SSH 连接时应该选择哪一个私钥的问题（如果建立有多个私钥）。

Cygwin 下的 openssh 软件包提供的 ssh 命令和 Linux 下的没有什么区别，也提供了 ssh-keygen 命令来管理 SSH 公钥/私钥对。但是 Cygwin 当前的 openssh（版本号：5.7p1-1）有一个 Bug，在用 Git 克隆使用 SSH 协议的版本库时偶尔会中断，无法完成版本库克隆。如下：

```
$ git clone git@bj.ossxp.com:ossxp/gitbook.git
Cloning into gitbook...
remote: Counting objects: 3486, done.
remote: Compressing objects: 100% (1759/1759), done.
fatal: The remote end hung up unexpectedly MiB | 3.03 MiB/s
fatal: early EOFs:  75% (2615/3486), 13.97 MiB | 3.03 MiB/s
fatal: index-pack failed
```

如果您也遇到同样的问题，建议使用 PuTTY 提供的 plink.exe 作为 SSH 客户端，替代存在问题的 Cygwin 自带的 ssh 命令。

1. 安装 PuTTY

PuTTY 是 Windows 下的一个开源软件，提供 SSH 客户端服务，还包括公钥管理的相关工具。访问 PuTTY 的主页（http://www.chiark.greenend.org.uk/~sgtatham/putty/），下载并安装 PuTTY。安装完毕会发现 PuTTY 软件包包含了好几个可执行程序，下面几个命令会用于和 Git 的整合。

- Plink：即 plink.exe，是命令行的 SSH 客户端，用于替代 ssh 命令。默认安装于 *C:\Program Files\PuTTY\plink.exe* 中。
- PuTTYgen：用于管理 PuTTY 格式的私钥，也可以用于将 openssh 格式的私钥转换为 PuTTY 格式的私钥。

- Pageant：是 SSH 认证代理，运行于后台，负责为 SSH 连接提供私钥访问服务。

2. PuTTY 格式的私钥

PuTTY 使用专有格式的私钥文件（扩展名为 *.ppk* ），而不能直接使用 openssh 格式的私钥。即用 openssh 的 ssh-keygen 命令创建的私钥不能直接被 PuTTY 拿过来使用，必需经过转换。程序 PuTTYgen 可以实现私钥格式的转换。

运行 PuTTYgen 程序，如图 3-15 所示。

图 3-15：运行 PuTTYgen 程序

PuTTYgen 既可以重新创建私钥文件，也可以通过点击加载按钮（load）读取 openssh 格式的私钥文件，从而可以将其转换为 PuTTY 格式的私钥。点击加载按钮，会弹出文件选择对话框，选择 openssh 格式的私钥文件（如文件 id_rsa），如果转换成功，会显示如图 3-16 的界面。

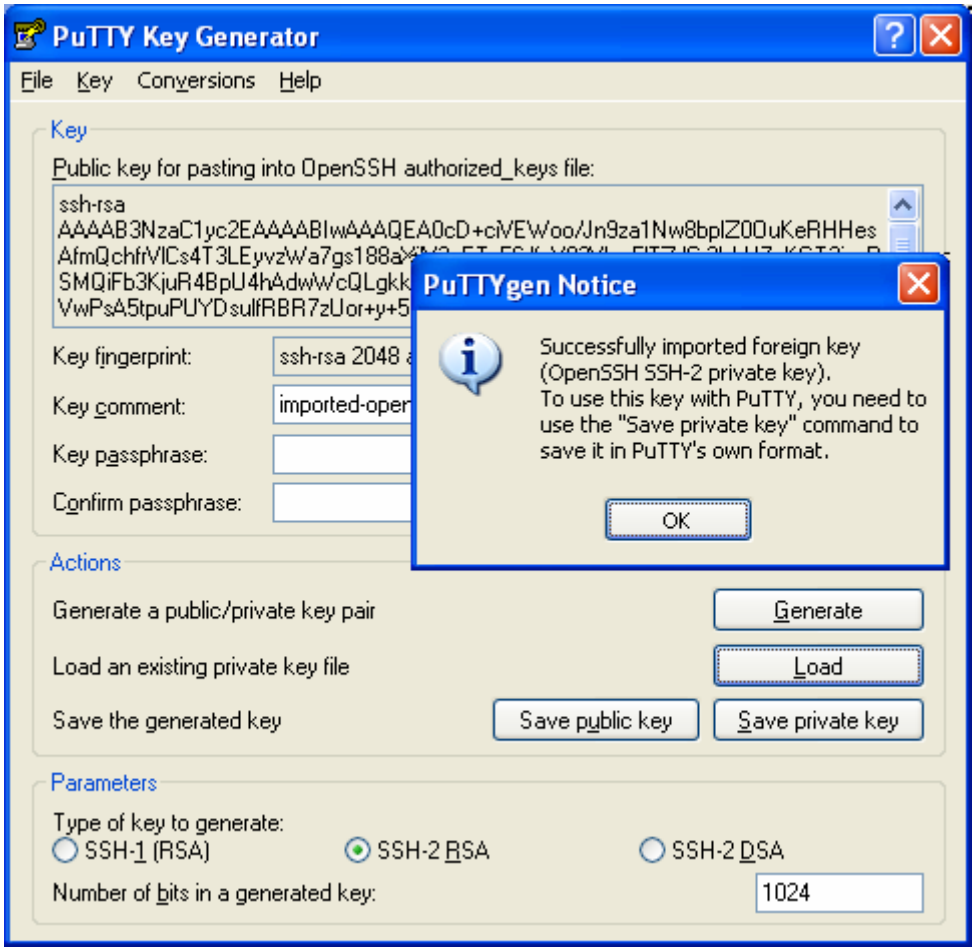

图 3-16：PuTTYgen 完成私钥加载

然后点击 “Save private key”（保存私钥），就可以将私钥保存为 PuTTY 的 `.ppk` 格式的私钥。例如将私钥保存到文件 `~/.ssh/jiangxin-cygwin.ppk` 中。

3. Git 使用 Pageant 进行公钥认证

Git 在使用命令行工具 Plink（`plink.exe`）作为 SSH 客户端访问 SSH 协议的版本库服务器时，如何选择公钥呢？使用 Pageant 是一个非常好的选择。Pageant 是 PuTTY 软件包中的代理软件，为各个 PuTTY 应用提供私钥请求，当 Plink 连接 SSH 服务器需要请求公钥认证时，Pageant 就会给 Plink 提供相应的私钥。

运行 Pageant ，启动后显示为托盘区中的一个图标，在后台运行。使用鼠标右键单击 Pageant 的图标，就会显示弹出菜单，如图 3-17 所示。

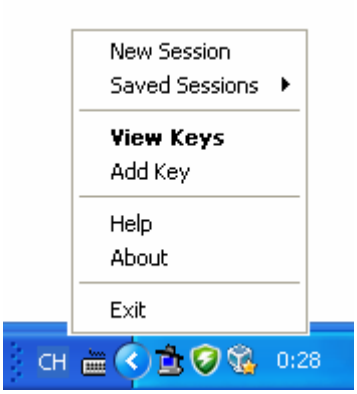

图 3-17：Pageant 的弹出菜单

点击弹出菜单中的 “Add Key”（添加私钥）按钮，会弹出文件选择框，选择扩展名为 *.ppk* 的 PuTTY 格式的公钥，即完成了 Pageant 的私钥准备工作。

接下来，还需要对 Git 进行设置，设置 Git 使用 *plink.exe* 作为 SSH 客户端，而不是默认的 *ssh* 命令。通过设置 GIT_SSH 环境变量即可实现。

```
$ export GIT_SSH=/cygdrive/c/Program\ Files/PuTTY/plink.exe
```

上面在设置 GIT_SSH 环境变量的过程中，使用了 Cygwin 格式的路径，而非 Windows 格式，这是因为 Git 是在 Cygwin 的环境中调用 *plink.exe* 命令的，当然要使用 Cygwin 能够理解的路径。

然后就可以用 Git 访问 SSH 协议的 Git 服务器了。运行在后台的 Pageant 会在需要的时候为 plink.exe 提供私钥访问服务。但在首次连接一个使用 SSH 协议的 Git 服务器的时候，很可能会因为远程 SSH 服务器的公钥没有经过确认而导致 git 命令执行失败，如下所示。

```
$ git clone git@bj.ossxp.com:ossxp/gitbook.git
Cloning into gitbook...
The server's host key is not cached in the registry. You
have no guarantee that the server is the computer you
think it is.
The server's rsa2 key fingerprint is:
ssh-rsa 2048 49:eb:04:30:70:ab:b3:28:42:03:19:fe:82:f8:1a:00
Connection abandoned.
fatal: The remote end hung up unexpectedly
```

这是因为首次连接一个 SSH 服务器时，要对其公钥进行确认（以防止被钓鱼），而运行于 Git 下的 *plink.exe* 没有机会从用户那里获取输入，以建立对该 SSH 服务器公钥的信任，因此 Git 访问失败。解决办法非常简单，就是直接运行 *plink.exe* 连接一次远程 SSH 服务器，对公钥确认进行应答。操作如下：

```
$ /cygdrive/c/Program\ Files/PuTTY/plink.exe git@bj.ossxp.com
The server's host key is not cached in the registry. You
have no guarantee that the server is the computer you
think it is.
The server's rsa2 key fingerprint is:
ssh-rsa 2048 49:eb:04:30:70:ab:b3:28:42:03:19:fe:82:f8:1a:00
If you trust this host, enter "y" to add the key to
PuTTY's cache and carry on connecting.
If you want to carry on connecting just once, without
adding the key to the cache, enter "n".
If you do not trust this host, press Return to abandon the
connection.
Store key in cache? (y/n)
```

输入“y”，将公钥保存在信任链中，以后再和该主机连接就不会弹出该确认应答了。当然 Git 命令也就可以成功执行了。

4. 使用自定义 SSH 脚本取代 Pageant

使用 Pageant 还要在每次启动 Pageant 时手动选择私钥文件，比较麻烦。实际上可以创建一个脚本对 *plink.exe* 进行封装，在封装的脚本中使用 *-i* 参数指定私钥文件。

例如，创建脚本 *~/bin/ssh-jiangxin*，文件内容如下：

```
#!/bin/sh

/cygdrive/c/Program\ Files/PuTTY/plink.exe -T -i \
    c:/cygwin/home/jiangxin/.ssh/jiangxin-cygwin.ppk $*
```

设置该脚本为可执行脚本。

```
$ chmod a+x ~/bin/ssh-jiangxin
```

使用下面的命令通过该脚本和远程 SSH 服务器连接：

```
$ ~/bin/ssh-jiangxin git@bj.ossxp.com
Using username "git".
hello jiangxin, the gitolite version here is v1.5.5-9-g4c11bd8
the gitolite config gives you the following access:
     R          gistore-bj.ossxp.com/.*$
     R          gistore-ossxp.com/.*$
  C  R  W       ossxp/.*$
     R  W       test/repo1
     R  W       test/repo2
     R  W       test/repo3
```

```
    @R @W       test/repo4
@C @R  W        users/jiangxin/.+$
```

设置 GIT_SSH 变量，使之指向新建立的脚本，然后就可以脱离 Pageant 来连接 SSH 协议的 Git 库了。

```
$ export GIT_SSH=~/bin/ssh-jiangxin
```

3.4 Windows 下安装和使用 Git（msysGit 篇）

运行在 Cygwin 下的 Git 不直接使用 Windows 的系统调用，而是通过二传手 *cygwin1.dll* 来进行的。虽然 Cygwin 的 Git 能够在 Windows 下的 *cmd.exe* 命令窗口中运行得非常好，但 Cygwin 下的 Git 并不能被看作是 Windows 下的原生程序。相比 Cygwin 下的 Git，msysGit 才是原生的 Windows 程序，msysGit 下运行的 Git 是直接通过 Windows 的系统调用来运行的。

msysGit 名字前面的四个字母来源于 MSYS[1] 项目。MSYS 项目源自于 MinGW[2]（Minimalist GNU for Windows，最简 GNU 工具集），通过增加了一个 bash 提供的 shell 环境及其他相关的工具软件，组成了一个最简系统（Minimal SYStem），简称 MSYS。利用 MinGW 提供的工具，以及 Git 针对 MinGW 的一个分支版本，在 Windows 平台为 Git 编译出一个原生应用，结合 MSYS 就组成了 msysGit。

3.4.1 安装 msysGit

安装 msysGit 非常简单，访问 msysGit 的项目主页（http://code.google.com/p/msysgit/），下载 msysGit。最简单的方式是下载名为 *Git-<VERSION>-preview<DATE>.exe* 的软件包，如：*Git-1.7.3.1-preview20101002.exe* 。如果您有时间和耐心，想要观察 Git 是如何在 Windows 上被编译为原生应用的，也可以下载带 *msysGit-fullinstall-* 前缀的软件包。

点击下载的安装程序（如 *Git-1.7.3.1-preview20101002.exe* ）开始安装，如图 3-18 所示。

[1] http://www.mingw.org/wiki/msys

[2] http://www.mingw.org/

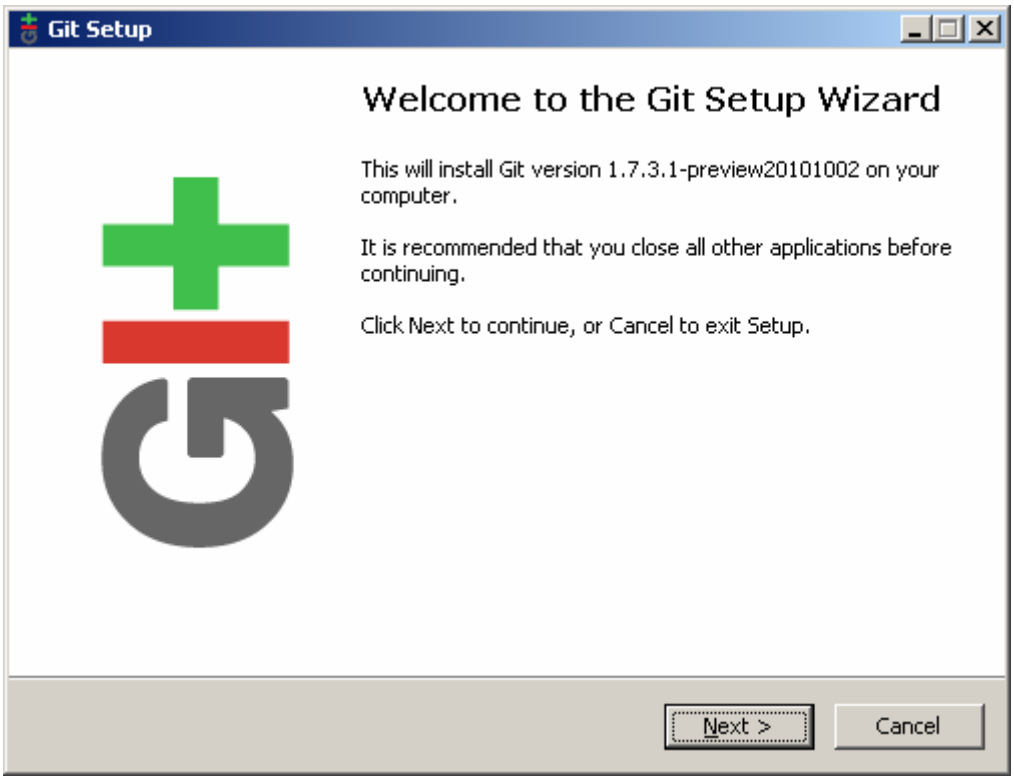

图 3-18：启动 msysGit 安装

默认安装到 `C:\Program Files\Git` 目录中，如图 3-19 所示。

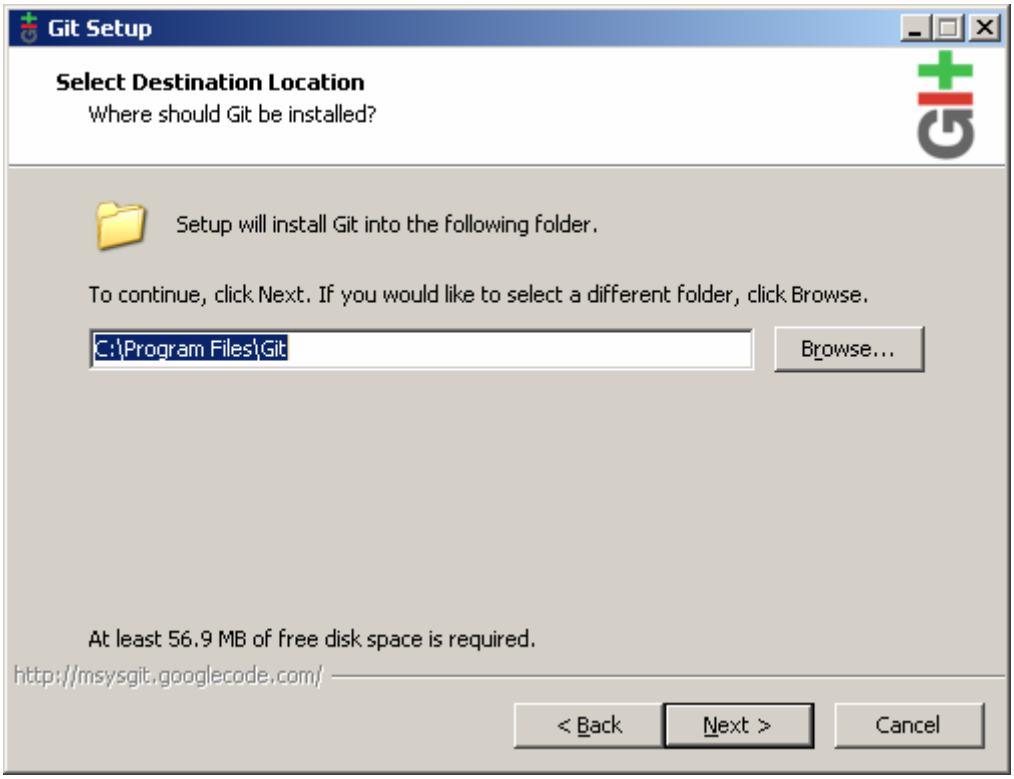

图 3-19：选择 msysGit 的安装目录

在安装过程中会询问是否修改环境变量，如图 3-20 所示。默认选择“Use Git Bash Only”，即

只在 msysGit 提供的 shell 环境（类似 Cygwin）中使用 Git，不修改环境变量。注意如果选择最后一项，会将 msysGit 所有的可执行程序全部加入 Windows 的 PATH 路径中，有的命令会覆盖 Windows 相同文件名的程序（如 find.exe 和 sort.exe）。而且如果选择最后一项，还会为 Windows 添加 HOME 环境变量，如果安装有 Cygwin，Cygwin 就会受到 msysGit 引入的 HOME 环境变量的影响（参见前面 3.3.3 节的相关讨论）。

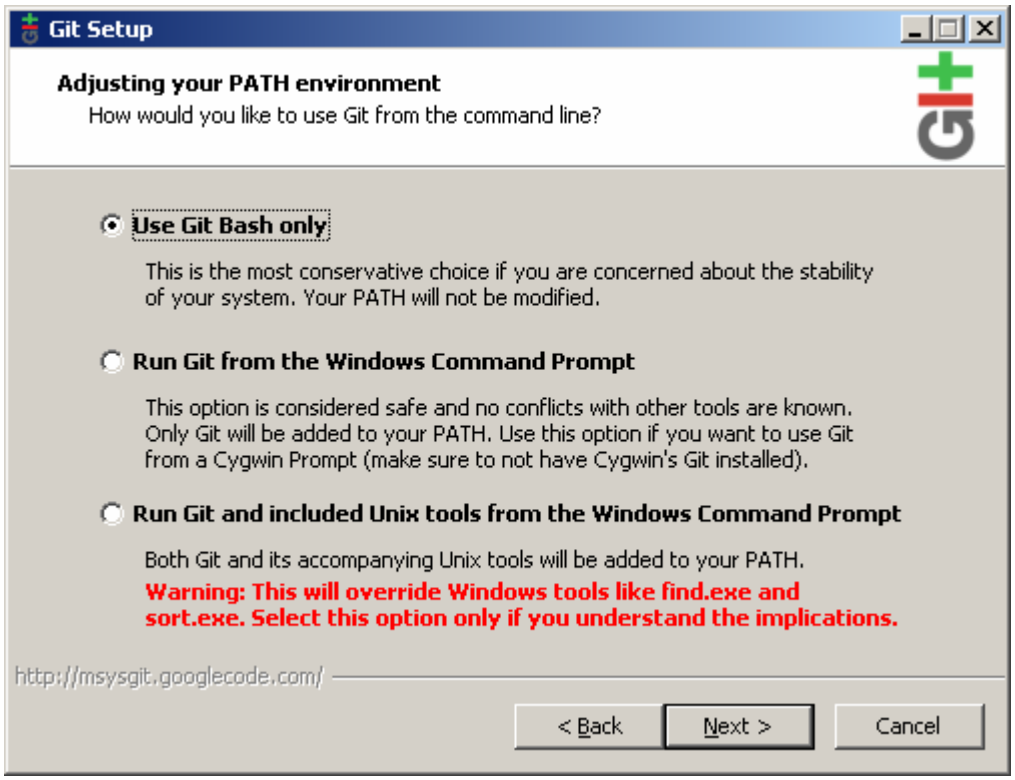

图 3-20：是否修改系统的环境变量

还会询问换行符的转换方式，使用默认设置就可以了，如图 3-21 所示。关于换行符转换的内容，请参见本书第 8 篇的相关章节。

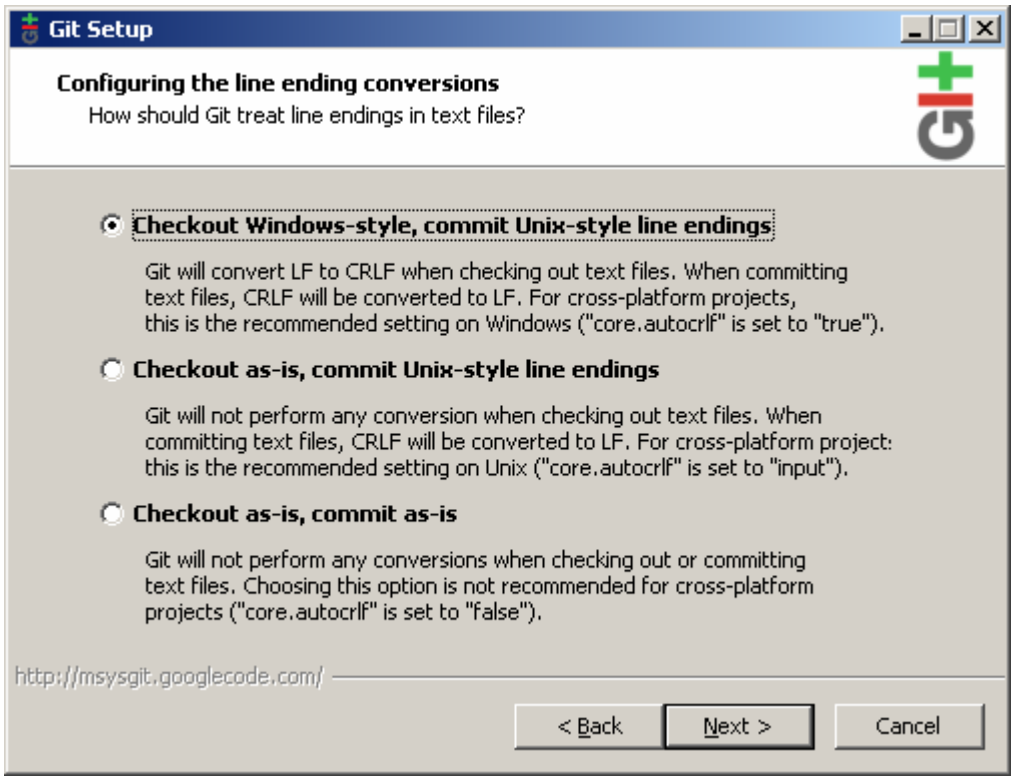

图 3-21：换行符转换方式

根据提示，完成 msysGit 的安装。

3.4.2 msysGit 的配置和使用

完成 msysGit 的安装后，点击 Git Bash 图标，启动 msysGit，如图 3-22。会发现 Git Bash 的界面和 Cygwin 的非常相像。

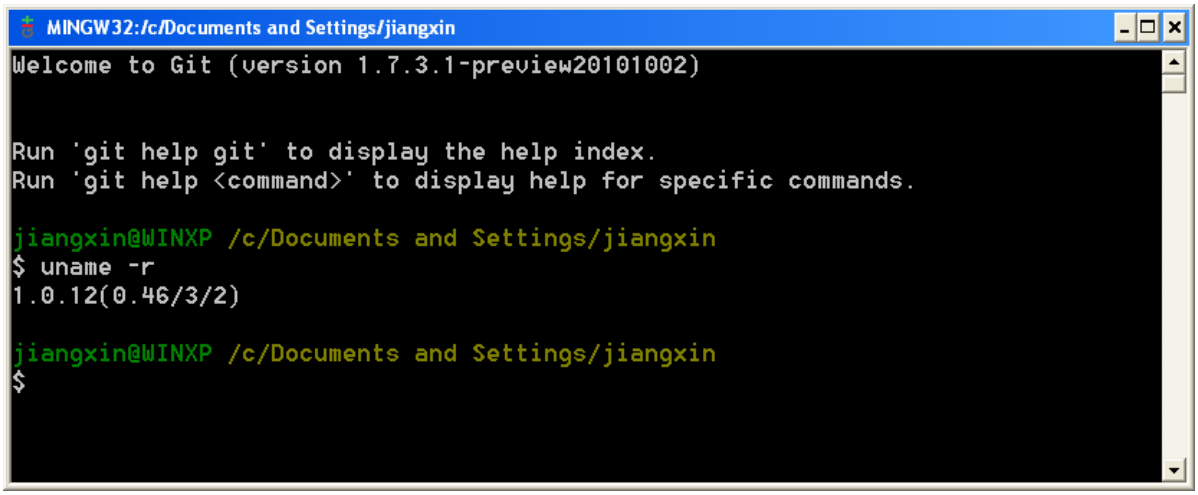

图 3-22：启动 Git Bash

1. 如何访问 Windows 的盘符

在 msysGit 下访问 Windows 的各个盘符，要比 Cygwin 简单，直接通过“*/c*”即可访问 Windows 的 C: 盘，用“*/d*”即可访问 Windows 的 D: 盘。

```
$ ls -ld /c/Windows
drwxr-xr-x  233 jiangxin Administ        0 Jan 31 00:44 /c/Windows
```

至于 msysGit 的根目录，实际上就是 msysGit 的安装目录，如：“C:\Program Files\Git”。

2. 命令行补齐和忽略文件大小写

msysGit 默认已经安装并启用了 Git 的命令行补齐功能，是通过在文件 */etc/profile* 中加载相应的脚本实现的。

```
. /etc/git-completion.bash
```

msysGit 还支持在命令行补齐时忽略文件名大小写。这是因为 msysGit 已经在配置文件 */etc/inputrc* 中包含了下列的设置：

```
set completion-ignore-case on
```

3.4.3 msysGit 中 shell 环境的中文支持

在介绍 Cygwin 的章节中曾经提到过，msysGit 的 shell 环境的中文支持相当于老版本的 Cygwin[1]，需要配置才能够录入中文和显示中文。

1. 中文录入问题

默认安装的 msysGit 的 shell 环境无法输入中文。为了能在 shell 界面中输入中文，需要修改配置文件 */etc/inputrc* ，增加或修改相关的配置如下：

```
# disable/enable 8bit input
set meta-flag on
set input-meta on
set output-meta on
```

[1] MSYS 是源自于 Cygwin1.3 的轻量级分支（参考 http://www.mingw.org/）。

```
set convert-meta off
```

关闭 Git Bash 再重启，就可以在 msysGit 的 shell 环境中输入中文了。

```
$ echo 您好
您好
```

2. 分页器中文输出问题

对 */etc/inputrc* 进行正确的配置之后，能够在 shell 下输入中文，但是执行下面的命令会显示乱码。这显然是 *less* 分页器命令导致的问题。

```
$ echo 您好 | less
<C4><FA><BA><C3>
```

通过管道符调用分页器命令 *less* 后，原本的中文输出变成了乱码显示。因为 Git 大量地使用 *less* 命令作为分页器，这将会导致 Git 很多命令的输出都会出现中文乱码的问题。之所以 *less* 命令会出现乱码，是因为该命令没有把中文当作正常的字符，可以通过设置 LESSCHARSET 环境变量，将 UTF-8 编码字符视为正规字符显示，则中文就能正常显示了。下面的操作，可以使得 *less* 分页器中的中文正常显示。

```
$ export LESSCHARSET=utf-8
$ echo 您好 | less
您好
```

编辑配置文件 */etc/profile* ，将对环境变量 LESSCHARSET 的设置加入其中，以便 msysGit 的 shell 环境一启动即加载。

```
declare -x LESSCHARSET=utf-8
```

3. ls 命令对中文文件名的显示

最常用的显示目录和文件名列表的命令 *ls* 对中文文件名的显示也有问题。下面的命令创建了一个中文文件名的文件，显示文件内容中的中文没有问题，但是在显示文件名时，会显示一串问号。

```
$ echo 您好 > 您好.txt

$ cat \*.txt
您好
```

```
$ ls \*.txt
????.txt
```

实际上只要在 *ls* 命令后添加参数 *--show-control-chars* 即可正确显示中文。

```
$ ls --show-control-chars *.txt
您好.txt
```

为方便起见，可以为 *ls* 命令设置一个别名，这样就不必在输入 *ls* 命令时输入长长的参数了。

```
$ alias ls="ls --show-control-chars"

$ ls \*.txt
您好.txt
```

将上面的 alias 命令添加到配置文件 */etc/profile* 中，实现在每次运行 Git Bash 时自动加载。

3.4.4 msysGit 中 Git 的中文支持

非常遗憾的是 msysGit 中的 Git 对中文支持没有 Cygwin 中的 Git 做的那么好，msysGit 中的 Git 对中文支持的程度，就相当于前面讨论过的 Linux 使用了 GBK 字符集时 Git 的情况。

- 使用未经配置的 msysGit 提交，如果在提交说明中包含中文，从 Linux 平台或其他 UTF-8 字符集平台上查看提交说明会显示为乱码。
- 同样，从 Linux 平台或其他使用 UTF-8 字符集平台进行的提交，若提交说明包含中文，在未经配置的 msysGit 中也会显示为乱码。
- 如果使用 msysGit 向版本库中添加带有中文文件名的文件，在 Linux（或其他 UTF-8）平台检出文件名会显示为乱码，反之亦然。
- 不能创建带有中文字符的引用（里程碑、分支等）。

如果希望版本库中出现使用中文文件名的文件，最好不要使用 msysGit，而是使用 Cygwin 下的 Git。而如果只是想在提交说明中使用中文，经过一定的设置后 msysGit 还是可以实现的。

为了解决提交说明显示为乱码的问题，msysGit 要为 Git 设置参数 i18n.logOutputEncoding，将提交说明的输出编码设置为 gbk。

```
$ git config --system i18n.logOutputEncoding gbk
```

Git 在提交时并不会对提交说明进行从 GBK 字符集到 UTF-8 字符集的转换，但是可以在提

交说明中标注所使用的字符集，因此在非 UTF-8 字符集的平台录入中文，需要用下面的指令设置录入提交说明的字符集，以便在 commit 对象中嵌入正确的编码说明。

```
$ git config --system i18n.commitEncoding gbk
```

同样，为了让带有中文文件名的文件，在工作区状态输出、查看历史更改概要，以及在补丁文件中能够正常显示，要为 Git 设置 *core.quotepath* 配置变量，将其设置为 *false*。但是要注意在 msysGit 中添加中文文件名的文件，只能在 msysGit 环境中正确显示，而在其他环境（如 Linux、Mac OS X、Cygwin）中文件名会出现乱码。

```
$ git config --system core.quotepath false
$ git status -s
?? 说明.txt
```

注意：如果同时安装了 Cygwin 和 msysGit（可能配置了相同的用户主目录），或者因为中文支持问题而需要单独为 TortoiseGit 准备一套 msysGit 时，为了保证不同的 msysGit 之间，以及和 Cygwin 之间的配置互不影响，需要在配置 Git 环境时使用 *--system* 参数。这是因为不同的 msysGit 安装及 Cygwin 的系统级配置文件位置不同，但是用户级配置文件位置却可能重合。

3.4.5 使用 SSH 协议

msysGit 软件包包含的 ssh 命令和 Linux 下的没有什么区别，也提供 ssh-keygen 命令管理 SSH 公钥/私钥对。在使用 msysGit 的 ssh 命令时，没有遇到 Cygwin 中的 ssh 命令（版本号：5.7p1-1）不稳定的问题，即 msysGit 下的 ssh 命令可以非常稳定的工作。

如果需要和 Windows 有更好的整合，希望使用图形化工具管理公钥，也可以使用 PuTTY 提供的 plink.exe 作为 SSH 客户端。关于如何使用 PuTTY 请参见 3.3.5 节中 Cygwin 和 PuTTY 整合的相关内容。

3.4.6 TortoiseGit 的安装和使用

TortoiseGit 提供了 Git 和 Windows 资源管理器的整合，提供了 Git 的图形化操作界面。像其他 Tortoise 系列产品（TortoiseCVS、TortoiseSVN）一样，在资源管理器中显示的 Git 工作区目录和文件的图标附加了标识版本控制状态的图像，可以非常直观地看到哪些文件被更改了需要提交。通过扩展后的右键菜单，可以非常方便地在资源管理器中操作 Git 版本库。

TortoiseGit 是对 msysGit 命令行的封装，因此需要先安装 msysGit。安装 TortoiseGit 非常简

单，访问网站 http://code.google.com/p/tortoisegit/ ，下载安装包，然后根据提示完成安装。

安装过程中会询问要使用的 SSH 客户端，如图 3-23。默认使用内置的 TortoisePLink（来自 PuTTY 项目）作为 SSH 客户端。

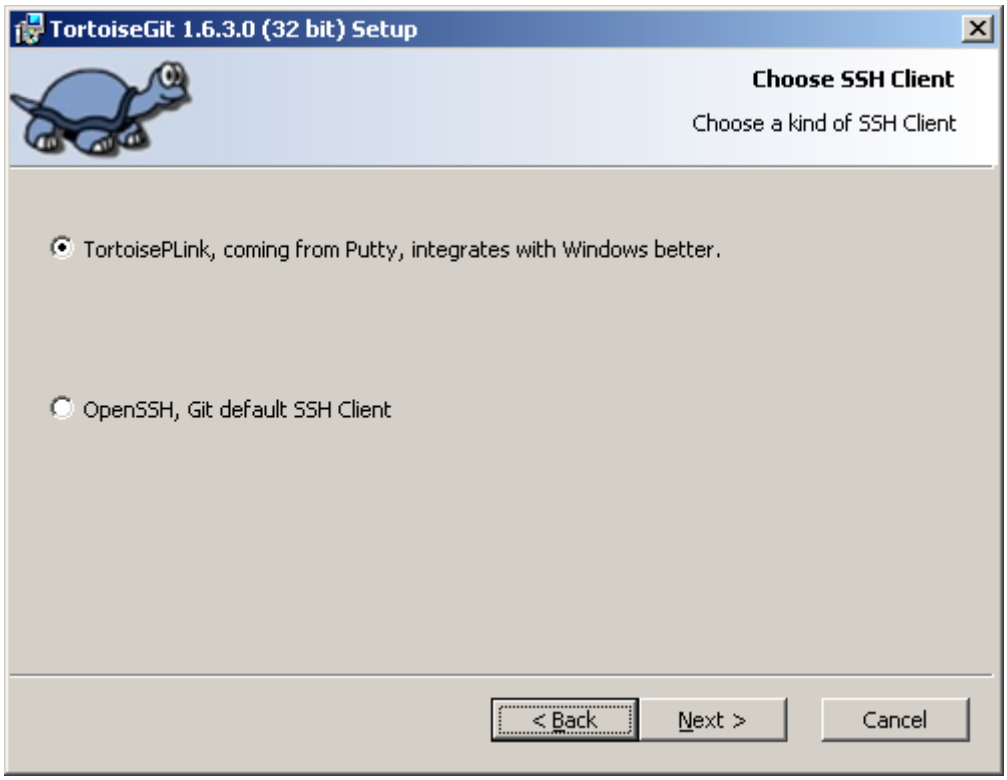

图 3-23：选择 SSH 客户端

TortoisePLink 和 TortoiseGit 的整合性更好，可以直接通过对话框设置 SSH 私钥（PuTTY 格式），而无须再到字符界面去配置 SSH 私钥和其他配置文件。如果安装过程中选择了 OpenSSH，可以在安装完之后，通过 TortoiseGit 的设置对话框重新选择 TortoisePLink 作为默认 SSH 客户端程序，如图 3-24 所示。

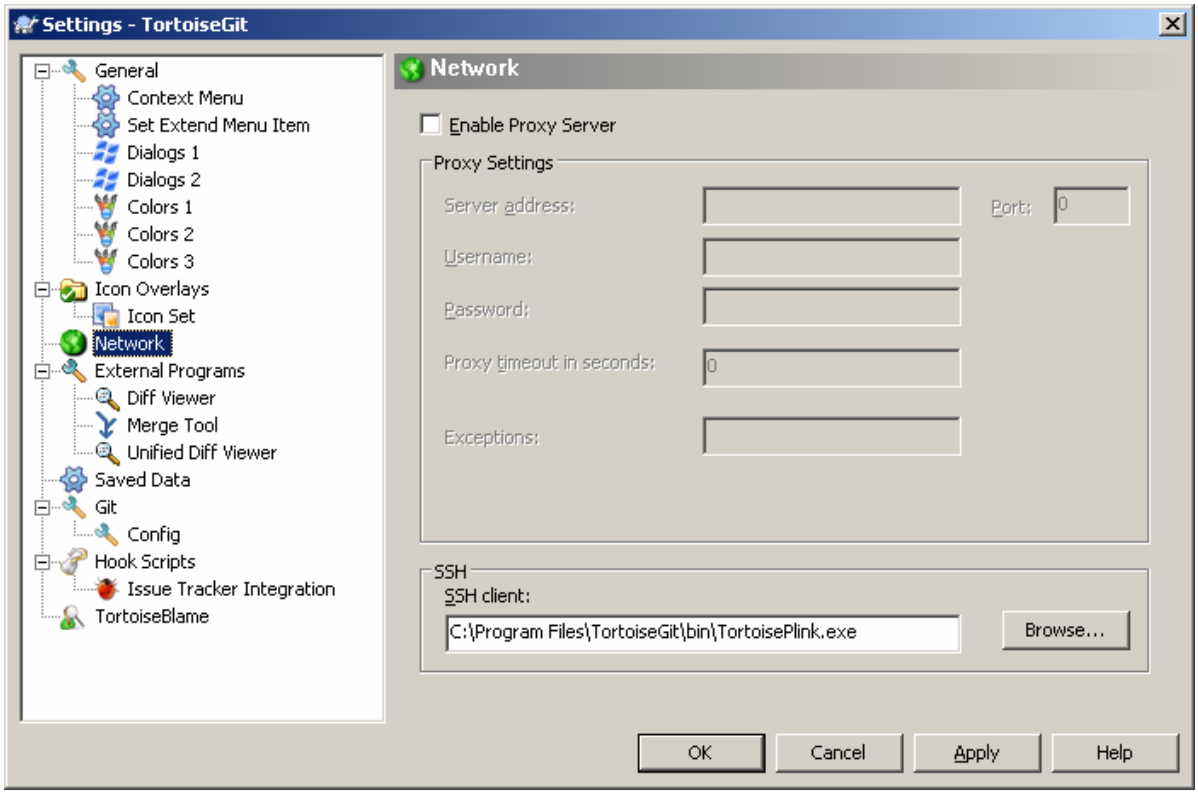

图 3-24：更改默认 SSH 客户端

当配置使用 TortoisePLink 作为默认 SSH 客户端后，在执行克隆操作时，可以在 TortoiseGit 操作界面中选择一个 PuTTY 格式的私钥文件进行认证，如图 3-25。

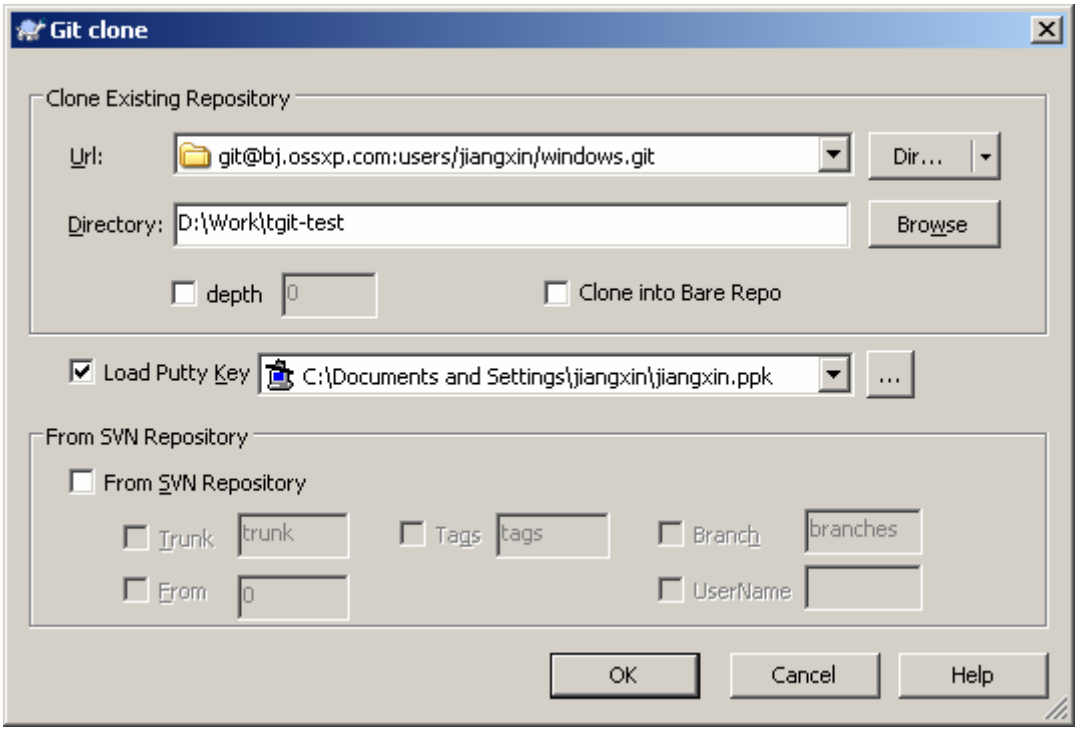

图 3-25：克隆操作选择 PuTTY 格式的私钥文件

如果需要更换连接一个服务器的 SSH 私钥，可以通过 Git 远程服务器配置界面对私钥文件

进行重新设置，如图 3-26。

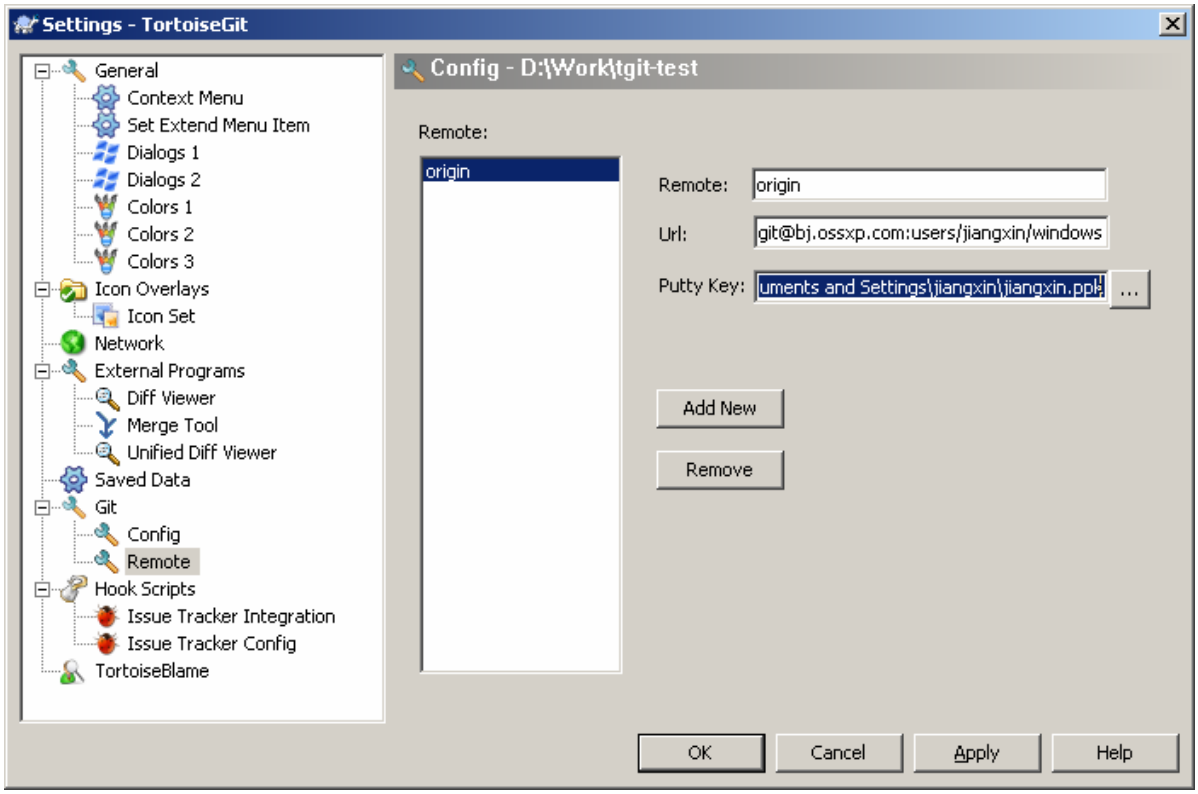

图 3-26：更换连接远程 SSH 服务器的私钥

如果系统中安装有多个 msysGit 拷贝，可以通过 TortoiseGit 的配置界面进行选择，如图 3-27。

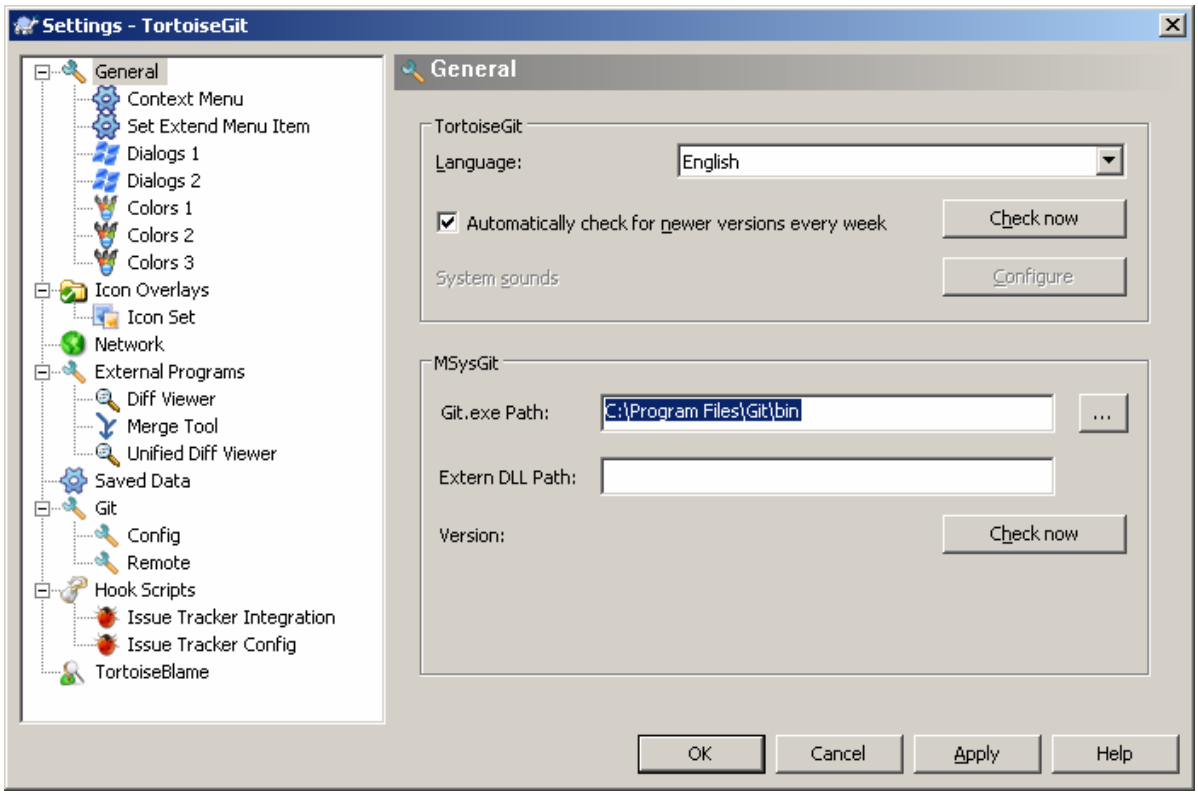

图 3-27：配置 msysGit 的可执行程序位置

3.4.7 TortoiseGit 的中文支持

TortoiseGit 虽然在底层调用了 msysGit，但是 TortoiseGit 的中文支持和 msysGit 是有区别的，甚至前面介绍 msysGit 中文支持时所进行的配制会破坏 TortoiseGit。

TortoiseGit 在提交时，会将提交说明转换为 UTF-8 字符集，因此无须对 i18n.commitEncoding 变量进行设置。相反，如果设置了 i18n.commitEncoding 为 gbk 或其他，则在提交对象中会包含错误的编码设置，有可能为提交说明的显示带来麻烦。

TortoiseGit 在显示提交说明时，认为所有的提交说明都是 UTF-8 编码，会转换为合适的 Windows 本地字符集显示，而无须设置 i18n.logOutputEncoding 变量。因为当前版本的 TortoiseGit 没有对提交对象中的 encoding 设置进行检查，因此使用 GBK 字符集的提交说明中的中文不能正常显示。

因此，如果需要同时使用 msysGit 的文字界面 Git Bash 及 TortoiseGit，并需要在提交说明中使用中文，可以安装两套 msysGit，并确保 TortoiseGit 关联的 msysGit 没有对 i18n.commitEncoding 进行设置。

TortoiseGit 对使用中文命名的文件和目录的支持和 msysGit 一样，都存在缺陷，因此应当避免在 msysGit 和 TortoiseGit 中添加用中文命名的文件和目录，如果确实需要，可以使用 Cygwin。