Learn VersionControl

穷屌丝联盟

2013 年 12 月 9 日

Contents

List of Figures

List of Tables

Part I

Introduction

Revision control[1], also known as version control and source control (and an aspect of software configuration management), is the management of changes to documents, computer programs, large web sites, and other collections of information. Changes are usually identified by a number or letter code, termed the "revision number", "revision level", or simply "revision". For example, an initial set of files is "revision 1". When the first change is made, the resulting set is "revision 2", and so on. Each revision is associated with a timestamp and the person making the change. Revisions can be compared, restored, and with some types of files, merged.

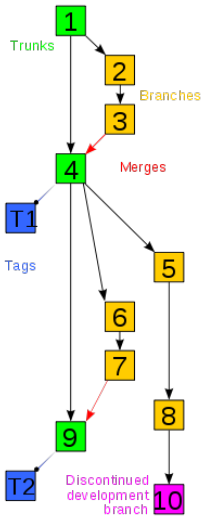


Figure 1: Example history graph of a revision-controlled project.

The need for a logical way to organize and control revisions has existed for almost as long as writing has existed, but revision control became much more important, and complicated, when the era of computing began. The numbering of book editions and of specification revisions are examples that date back to the print-only era. Today, the most capable (as well as complex) revision control systems are those used in software development, where a team of people may change the same files.

Version control systems (VCS) most commonly run as stand-alone applications, but revision control is also embedded in various types of software such as word processors and spreadsheets, e.g., Google Docs and Sheets(?) and in various content management systems, e.g., Wikipedia's Page history. Revision control allows for the ability to revert a document to a previous revision, which is critical for allowing editors to track each other's edits, correct mistakes, and defend against vandalism

[1]From Wikipedia, the free encyclopedia. This article needs additional citations for verification. Please help improve this article by adding citations to reliable sources. Unsourced material may be challenged and removed. (April 2011)

and spam.

Software tools for revision control are essential for the organization of multi-developer projects(?).

A version control system (or revision control system) is a system that tracks incremental versions (or revisions) of files and, in some cases, directories over time. Of course, merely tracking the various versions of a user's (or group of users') files and directories isn't very interesting in itself. What makes a version control system useful is the fact that it allows you to explore the changes which resulted in each of those versions and facilitates the arbitrary recall of the same.

In this section, we'll introduce some fairly high-level version control system components and concepts. We'll limit our discussion to modern version control systems—in today's interconnected world, there is very little point in acknowledging version control systems which cannot operate across wide-area networks.

In computer software engineering, revision control is any practice that tracks and provides control over changes to source code. Software developers sometimes use revision control software to maintain documentation and configuration files as well as source code.

As teams design, develop and deploy software, it is common for multiple versions of the same software to be deployed in different sites and for the software's developers to be working simultaneously on updates. Bugs or features of the software are often only present in certain versions (because of the fixing of some problems and the introduction of others as the program develops). Therefore, for the purposes of locating and fixing bugs, it is vitally important to be able to retrieve and run different versions of the software to determine in which version(s) the problem occurs. It may also be necessary to develop two versions of the software concurrently (for instance, where one version has bugs fixed, but no new features (branch), while the other version is where new features are worked on (trunk).

At the simplest level, developers could simply retain multiple copies of the different versions of the program, and label them appropriately. This simple approach has been used on many large software projects. While this method can work, it is inefficient as many near-identical copies of the program have to be maintained. This requires a lot of self-discipline on the part of developers, and often leads to mistakes. Consequently, systems to automate some or all of the revision control process have been developed.

Moreover, in software development, legal and business practice and other environments, it has become increasingly common for a single document or snippet of code to be edited by a team, the members of which may be geographically dispersed and may pursue different and even contrary inter-

ests. Sophisticated revision control that tracks and accounts for ownership of changes to documents and code may be extremely helpful or even indispensable in such situations.

Revision control may also track changes to configuration files, such as those typically stored in `/etc` or `/usr/local/etc` on Unix systems. This gives system administrators another way to easily track changes made and a way to roll back to earlier versions should the need arise.

Chapter 1

Structure

Revision control manages changes to a set of data over time. These changes can be structured in various ways.

Often the data is thought of as a collection of many individual items, such as files or documents, and changes to individual files are tracked. This accords with intuitions about separate files, but causes problems when identity changes, such as during renaming, splitting, or merging of files. Accordingly, some systems, such as git, instead consider changes to the data as a whole, which is less intuitive for simple changes, but simplifies more complex changes.

When data that is under revision control is modified, after being retrieved by checking out, this is not in general immediately reflected in the revision control system (in the repository), but must instead be checked in or committed. A copy outside revision control is known as a "working copy". As a simple example, when editing a computer file, the data stored in memory by the editing program is the working copy, which is committed by saving. Concretely, one may print out a document, edit it by hand, and only later manually input the changes into a computer and save it. For source code control, the working copy is instead a copy of all files in a particular revision, generally stored locally on the developer's computer;[1] in this case saving the file only changes the working copy, and checking into the repository is a separate step.

If multiple people are working on a single data set or document, they are implicitly creating branches of the data (in their working copies), and thus issues of merging arise, as discussed below. For simple collaborative document editing, this can be prevented by using file locking or simply avoiding working on the same document that someone else is working on.

Revision control systems are often centralized, with a single authoritative data store, the repos-

[1] In this case, edit buffers are a secondary form of working copy, and not referred to as such.

itory, and check-outs and check-ins done with reference to this central repository. Alternatively, in distributed revision control, no single repository is authoritative, and data can be checked out and checked into any repository. When checking into a different repository, this is interpreted as a merge or patch.

1.1 Graph structure

Example history graph of a revision-controlled project: trunk is in green, branches in yellow, and graph is not a tree due to presence of merges (the red arrows).

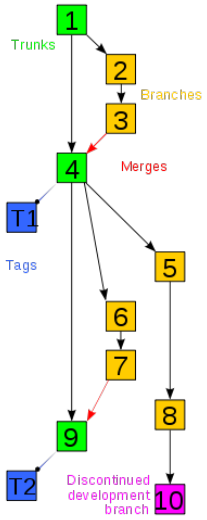


Figure 1.1: History graph of a revision-controlled project.

In terms of graph theory, revisions are generally thought of as a line of development (the trunk) with branches off of this, forming a directed tree, visualized as one or more parallel lines of development (the "mainlines" of the branches) branching off a trunk. In reality the structure is more complicated, forming a directed acyclic graph, but for many purposes "tree with merges" is an adequate approximation.

Revisions occur in sequence over time, and thus can be arranged in order, either by revision number or timestamp.[2] Revisions are based on past revisions, though it is possible to largely or

[2]In principle two revisions can have identical timestamp, and thus cannot be ordered on a line. This is generally the case for separate repositories, though is also possible for simultaneous changes to several branches in a single repository. In these cases, the revisions can be thought of as a set of separate lines, one per repository or branch (or branch within a repository).

completely replace an earlier revision, such as "delete all existing text, insert new text". In the simplest case, with no branching or undoing, each revision is based on its immediate predecessor alone, and they form a simple line, with a single latest version, the "HEAD" revision or tip.

In graph theory terms, drawing each revision as a point and each "derived revision" relationship as an arrow (conventionally pointing from older to newer, in the same direction as time), this is a linear graph. If there is branching, so multiple future revisions are based on a past revision, or undoing, so a revision can depend on a revision older than its immediate predecessor, then the resulting graph is instead a directed tree (each node can have more than one child), and has multiple tips, corresponding to the revisions without children ("latest revision on each branch").[3] In principle the resulting tree need not have a preferred tip ("main" latest revision) – just various different revisions – but in practice one tip is generally identified as HEAD. When a new revisions is based on HEAD, it is either identified as the new HEAD, or considered a new branch.[4] The list of revisions from the start to HEAD (in graph theory terms, the unique path in the tree, which forms a linear graph as before) is the trunk or mainline.[5] Conversely, when a revision can be based on more than one previous revision (when a node can have more than one parent), the resulting process is called a merge, and is one of the most complex aspects of revision control. This most often occurs when changes occur in multiple branches (most often two, but more are possible), which are then merged into a single branch incorporating both changes. If these changes overlap, it may be difficult or impossible to merge, and require manual intervention or rewriting.

In the presence of merges, the resulting graph is no longer a tree, as nodes can have multiple parents, but is instead a rooted directed acyclic graph (DAG). The graph is acyclic since parents are always backwards in time, and rooted because there is an oldest version. However, assuming that there is a trunk, merges from branches can be considered as "external" to the tree – the changes in the branch are packaged up as a patch, which is applied to HEAD (of the trunk), creating a new revision without any explicit reference to the branch, and preserving the tree structure. Thus, while the actual relations between versions form a DAG, this can be considered a tree plus merges, and the trunk itself is a line.

In distributed revision control, in the presence of multiple repositories these may be based on a single original version (a root of the tree), but there need not be an original root, and thus only a separate root (oldest revision) for each repository, for example if two people starting working on a project separately. Similarly in the presence of multiple data sets (multiple projects) that exchange

[3] The revision or repository "tree" should not be confused with the directory tree of files in a working copy.

[4] Note that if a new branch is based on HEAD, then topologically HEAD is no longer a tip, since it has a child.

[5] "Mainline" can also refer to the main path in a separate branch.

data or merge, there isn't a single root, though for simplicity one may think of one project as primary and the other as secondary, merged into the first with or without its own revision history.

Chapter 2

Specialized strategies

Engineering revision control developed from formalized processes based on tracking revisions of early blueprints or bluelines. This system of control implicitly allowed returning to any earlier state of the design, for cases in which an engineering dead-end was reached in the development of the design. A revision table was used to keep track of the changes made. Additionally, the modified areas of the drawing were highlighted using revision clouds.

Version control is also widespread in business and law. Indeed, "contract redline" and "legal blackline" are some of the earliest forms of revision control, and are still employed in business and law with varying degrees of sophistication. An entire industry has emerged to service the document revision control needs of business and other users, and some of the revision control technology employed in these circles is subtle, powerful, and innovative. The most sophisticated techniques are beginning to be used for the electronic tracking of changes to CAD files (PDM, Product Data Management), supplanting the "manual" electronic implementation of traditional revision control.

Chapter 3

Source-management models

Traditional revision control systems use a centralized model where all the revision control functions take place on a shared server. If two developers try to change the same file at the same time, without some method of managing access the developers may end up overwriting each other's work. Centralized revision control systems solve this problem in one of two different "source management models": file locking and version merging.

3.1 Atomic operations

Main article: Atomic commit

Computer scientists speak of atomic operations if the system is left in a consistent state even if the operation is interrupted. The commit operation is usually the most critical in this sense. Commits are operations that tell the revision control system you want to make a group of changes final and available to all users. Not all revision control systems have atomic commits; notably, the widely used CVS lacks this feature.

3.2 File locking

The simplest method of preventing "concurrent access" problems involves locking files so that only one developer at a time has write access to the central "repository" copies of those files. Once one developer "checks out" a file, others can read that file, but no one else may change that file until that developer "checks in" the updated version (or cancels the checkout).

File locking has both merits and drawbacks. It can provide some protection against difficult merge conflicts when a user is making radical changes to many sections of a large file (or group of

files). However, if the files are left exclusively locked for too long, other developers may be tempted to bypass the revision control software and change the files locally, leading to more serious problems.

3.3 Version merging

Main article: Merge (revision control)

Most version control systems allow multiple developers to edit the same file at the same time. The first developer to "check in" changes to the central repository always succeeds. The system may provide facilities to merge further changes into the central repository, and preserve the changes from the first developer when other developers check in.

Merging two files can be a very delicate operation, and usually possible only if the data structure is simple, as in text files. The result of a merge of two image files might not result in an image file at all. The second developer checking in code will need to take care with the merge, to make sure that the changes are compatible and that the merge operation does not introduce its own logic errors within the files. These problems limit the availability of automatic or semi-automatic merge operations mainly to simple text based documents, unless a specific merge plugin is available for the file types.

The concept of a reserved edit can provide an optional means to explicitly lock a file for exclusive write access, even when a merging capability exists.

3.4 Baselines, labels and tags

Most revision control tools will use only one of these similar terms (baseline, label, tag) to refer to the action of identifying a snapshot ("label the project") or the record of the snapshot ("try it with baseline X"). Typically only one of the terms baseline, label, or tag is used in documentation or discussion; they can be considered synonyms.

In most projects some snapshots are more significant than others, such as those used to indicate published releases, branches, or milestones.

When both the term baseline and either of label or tag are used together in the same context, label and tag usually refer to the mechanism within the tool of identifying or making the record of the snapshot, and baseline indicates the increased significance of any given label or tag.

Most formal discussion of configuration management uses the term baseline.

Chapter 4

Distributed revision control

Main article: Distributed revision control

Distributed revision control systems (DRCS) take a peer-to-peer approach, as opposed to the client-server approach of centralized systems. Rather than a single, central repository on which clients synchronize, each peer's working copy of the codebase is a bona-fide repository.[?] Distributed revision control conducts synchronization by exchanging patches (change-sets) from peer to peer. This results in some important differences from a centralized system:

- No canonical, reference copy of the codebase exists by default; only working copies.
- Common operations (such as commits, viewing history, and reverting changes) are fast, because there is no need to communicate with a central server[?].Rather, communication is only necessary when pushing or pulling changes to or from other peers.
- Each working copy effectively functions as a remote backup of the codebase and of its change-history, providing inherent protection against data loss.

Chapter 5

Integration

Some of the more advanced revision-control tools offer many other facilities, allowing deeper integration with other tools and software-engineering processes. Plugins are often available for IDEs such as Oracle JDeveloper, IntelliJ IDEA, Eclipse and Visual Studio. NetBeans IDE and Xcode come with integrated version control support.

Chapter 6

Common vocabulary

Terminology can vary from system to system, but some terms in common usage include:

- Baseline

  An approved revision of a document or source file from which subsequent changes can be made. See baselines, labels and tags.
- Branch

  A set of files under version control may be branched or forked at a point in time so that, from that time forward, two copies of those files may develop at different speeds or in different ways independently of each other.
- Change

  A change (or diff, or delta) represents a specific modification to a document under version control. The granularity of the modification considered a change varies between version control systems.
- Change list

  On many version control systems with atomic multi-change commits, a change list, change set, update, or patch identifies the set of changes made in a single commit. This can also represent a sequential view of the source code, allowing the examination of source "as of" any particular changelist ID.
- Checkout

  To check out (or co) is to create a local working copy from the repository. A user may specify a specific revision or obtain the latest. The term 'checkout' can also be used as a noun to describe the working copy.
- Commit

To commit (check in, ci or, more rarely, install, submit or record) is to write or merge the changes made in the working copy back to the repository. The terms 'commit' and 'checkin' can also be used as nouns to describe the new revision that is created as a result of committing.

- Conflict

  A conflict occurs when different parties make changes to the same document, and the system is unable to reconcile the changes. A user must resolve the conflict by combining the changes, or by selecting one change in favour of the other.

- Delta compression

  Most revision control software uses delta compression, which retains only the differences between successive versions of files. This allows for more efficient storage of many different versions of files.

- Dynamic stream

  A stream in which some or all file versions are mirrors of the parent stream's versions.

- Export

  exporting is the act of obtaining the files from the repository. It is similar to checking out except that it creates a clean directory tree without the version-control metadata used in a working copy. This is often used prior to publishing the contents, for example.

- Head

  Also sometimes called tip, this refers to the most recent commit, either to the trunk or to a branch. The trunk and each branch have their own head, though HEAD is sometimes loosely used to refer to the trunk.

- Import

  importing is the act of copying a local directory tree (that is not currently a working copy) into the repository for the first time.

- Label

  See tag.

- Mainline

  Similar to trunk, but there can be a mainline for each branch.

- Merge

  A merge or integration is an operation in which two sets of changes are applied to a file or set of files. Some sample scenarios are as follows:

  1. A user, working on a set of files, updates or syncs their working copy with changes made, and checked into the repository, by other users.

2. A user tries to check in files that have been updated by others since the files were checked out, and the revision control software automatically merges the files (typically, after prompting the user if it should proceed with the automatic merge, and in some cases only doing so if the merge can be clearly and reasonably resolved).
3. A set of files is branched, a problem that existed before the branching is fixed in one branch, and the fix is then merged into the other branch.
4. A branch is created, the code in the files is independently edited, and the updated branch is later incorporated into a single, unified trunk.

- Promote

  The act of copying file content from a less controlled location into a more controlled location. For example, from a user's workspace into a repository, or from a stream to its parent.
- Repository

  The repository is where files' current and historical data are stored, often on a server. Sometimes also called a depot (for example, by SVK, AccuRev and Perforce).
- Resolve

  The act of user intervention to address a conflict between different changes to the same document.
- Reverse integration

  The process of merging different team branches into the main trunk of the versioning system.
- Revision

  Also version: A version is any change in form. In SVK, a Revision is the state at a point in time of the entire tree in the repository.
- Share

  The act of making one file or folder available in multiple branches at the same time. When a shared file is changed in one branch, it is changed in other branches.
- Stream

  A container for branched files that has a known relationship to other such containers. Streams form a hierarchy; each stream can inherit various properties (like versions, namespace, workflow rules, subscribers, etc.) from its parent stream.
- Tag

  A tag or label refers to an important snapshot in time, consistent across many files. These files at that point may all be tagged with a user-friendly, meaningful name or revision number. See baselines, labels and tags.

- Trunk

  The unique line of development that is not a branch (sometimes also called Baseline, Mainline or Master)
- Update

  An update (or sync) merges changes made in the repository (by other people, for example) into the local working copy.[8] Update is also the term used by some CM tools (CM+, PLS, SMS) for the change package concept (see changelist).
- Working copy

  The working copy is the local copy of files from a repository, at a specific time or revision. All work done to the files in a repository is initially done on a working copy, hence the name. Conceptually, it is a sandbox.

Chapter 7

Atomic commit

An atomic commit is an operation in which a set of distinct changes is applied as a single operation. If the changes are applied then the atomic commit is said to have succeeded. If there is a failure before the atomic commit can be completed then all of the changes completed in the atomic commit are reversed. This ensures that the system is always left in a consistent state. The other key property of isolation comes from their nature as atomic operations. Isolation ensures that only one atomic commit is processed at a time. The most common uses of atomic commits are in database systems and revision control systems.

The problem with atomic commits is that they require coordination between multiple systems. As computer networks are unreliable services this means no algorithm can coordinate with all systems as proven in the Two Generals Problem. As databases become more and more distributed this coordination will increase the difficulty of making truly atomic commits.

7.1 Necessity for Atomic Commits

Atomic commits are essential for multi-step updates to data. This can be clearly shown in a simple example of a money transfer between two checking accounts.

This example is complicated by a transaction to check the balance of account Y during a transaction for transferring 100 dollars from account X to Y. To start, first 100 dollars is removed from account X. Second, 100 dollars is added to account Y. If the entire operation is not completed as one atomic commit, then several problems could occur. If the system fails in the middle of the operation, after removing the money from X and before adding into Y, then 100 dollars has just disappeared. Another issue is if the balance of Y is checked before the 100 dollars is added. The wrong balance

for Y will be reported.

With atomic commits neither of these cases can happen, in the first case of the system failure, the atomic commit would be rolled back and the money returned to X. In the second case, the request of the balance of Y cannot occur until the atomic commit is fully completed.

7.2 Database System

Atomic commits in database systems fulfil two of the key properties of ACID, atomicity and consistency. Consistency is only achieved if each change in the atomic commit is consistent.

As shown in the example atomic commits are critical to multistep operations in databases. Due to modern hardware design the physical disk on which the database resides true atomic commits cannot exist. The smallest area that can be written to on disk is known as a sector. A single database entry may span several different sectors. Only one sector can be written at a time. This writing limit is why true atomic commits are not possible. After the database entries in memory have been modified they are queued up to be written to disk. This means the same problems identified in the example have reoccurred. Any algorithmic solution to this problem will still encounter the Two Generals' Problem. The two-phase commit protocol and three-phase commit protocol attempt to solve this and some of the other problems associated with atomic commits.

The two-phase commit protocol requires a coordinator to maintain all the information needed to recover the original state of the database if something goes wrong. As the name indicates there are two phases, voting and commit.

During the voting phase each node writes the changes in the atomic commit to its own disk. The nodes then report their status to the coordinator. If any node does not report to the coordinator or their status message is lost the coordinator assumes the node' s write failed. Once all of the nodes have reported to the coordinator the second phase begins.

During the commit phase the coordinator sends a commit message to each of the nodes to record in their individual logs. Until this message is added to a node's log, any changes made will be recorded as incomplete. If any of the nodes reported a failure the coordinator will instead send a rollback message. This will remove any changes the nodes have written to disk.

The three-phase commit protocol seeks to remove the main problem with the two phase commit protocol which occurs if a coordinator and another node fail at the same time during the commit phase neither can tell what action should occur. To solve this problem a third phase is added to the protocol. The prepare to commit phase occurs after the voting phase and before the commit phase.

In the voting phase, similar to the two-phase commit, the coordinator requests that each node

is ready to commit. If any node fails the coordinator will timeout while waiting for the failed node. If this happens the coordinator sends an abort message to every node. The same action will be undertaken if any of the nodes return a failure message.

Upon receiving success messages from each node in the voting phase the prepare to commit phase begins. During this phase the coordinator sends a prepare message to each node. Each node must acknowledge the prepare message and reply. If any reply is missed or any node return that they are not prepared then the coordinator sends an abort message. Any node that does not receive a prepare message before the timeout expires aborts the commit.

After all nodes have replied to the prepare message then the commit phase begins. In this phase the coordinator sends a commit message to each node. When each node receives this message it performs the actual commit. If the commit message does not reach a node due to the message being lost or the coordinator fails they will perform the commit if the timeout expires. If the coordinator fails upon recovery it will send a commit message to each node.

7.3 Revision Control

The other area where atomic commits are employed is revision control systems. This allows multiple modified files to be uploaded and merged into the source. Most revision control systems support atomic commits (CVS and VSS are the major exceptions).

Like database systems commits may fail due to a problem in applying the changes on disk. Unlike a database system which overwrites any existing data with the data from the changeset, revision control systems merge the modification in the changeset into the existing data. If the system cannot complete the merge then the commit will be rejected. If a merge cannot be resolved by the revision control software it is up to the user to merge the changes. For revision control systems that support atomic commits, this failure in merging would result in a failed commit.

Atomic commits are crucial for maintaining a consistent state in the repository. Without atomic commits some changes a developer has made may be applied but other changes may not. If these changes have any kind of coupling this will result in errors. Atomic commits prevent this by not applying partial changes which would create these errors. Note that if the changes already contain errors, atomic commits offer no fix.

7.4 Atomic Commit Convention

When using a revision control systems a common convention is to use small commits. These are sometimes referred to as atomic commits as they (ideally) only affect a single aspect of the system. These atomic commits allow for greater understandability, less effort to roll back changes, easier bug identification.

The greater understandability comes from the small size and focused nature of the commit. It is much easier to understand what is changed and reasoning behind the changes if you are only looking for one kind of change. This becomes especially important when making format changes to the source code. If format and functional changes are combined it becomes very difficult to identify useful changes. Imagine if the spacing in a file is changed from using tabs to three spaces every tab in the file will show as having been changed. This becomes critical if some functional changes are also made as a reviewer may simply not see the functional changes.

If only atomic commits are made then commits that introduce errors become much simpler to identify. You are not required to look though every commit to see if it was the cause of the error, only the commits dealing with that functionality need to be examined. If the error is to be rolled back atomic commits again make the job much simpler. Instead of having to revert to the offending revision and remove the changes manually before integrating any later changes; the developer can simply revert any changes in the identified commit. This also means that a developer will not remove changes that did not cause the error by accident.

Atomic commits also allow bug fixes to be easily reviewed if only a single bug fixes committed at a time. Instead of having to check multiple potentially unrelated files the reviewer must only check files and changes that directly impact the bug being fixed. This also means that bug fixes can be easily packaged for testing as only the changes that fix the bug are in the commit.

Chapter 8

Merge

Merging (also called integration) in revision control, is a fundamental operation that reconciles multiple changes made to a revision-controlled collection of files. Most often, it is necessary when a file is modified by two people on two different computers at the same time. When two branches are merged, the result is a single collection of files that contains both sets of changes.

In some cases, the merge can be performed automatically, because there is sufficient history information to reconstruct the changes, and the changes do not conflict. In other cases, a person must decide exactly what the resulting files should contain. Many revision control software tools include merge capabilities.

8.1 Types of merges

There are two types of merges: automatic and manual.

Automatic merging is what revision control software does when it reconciles changes that have happened simultaneously (in a logical sense). Also, other pieces of software deploy automatic merging if they allow for editing the same content simultaneously. For instance, Wikipedia allows two people to edit the same article at the same time; when the latter contributor saves, their changes are merged into the article instead of overwriting the previous set of changes.

Manual merging is what people have to resort to (possibly assisted by merging tools) when they have to reconcile files that differ. For instance, if two systems have slightly differing versions of a configuration file and a user wants to have the good stuff in both, this can usually be achieved by merging the configuration files by hand, picking the wanted changes from both sources (this is also called two-way merging). Manual merging is also required when automatic merging runs into a

change conflict; for instance, very few automatic merge tools can merge two changes to the same line of code (say, one that changes a function name, and another that adds a comment). In these cases, revision control systems resort to the user to specify the intended merge result.

Merge algorithms are an area of active research, and consequently there are many different approaches to automatic merging, with subtle differences. The more notable merge algorithms include three-way merge, recursive three-way merge, fuzzy patch application, weave merge, and patch commutation.

8.1.1 Three-way merge

A three-way merge is performed after an automated difference analysis between a file 'A' and a file 'B' while also considering the origin, or common ancestor, of both files. It is a rough merging method, but really widely applicable since it only requires one common ancestor to reconstruct the changes that are to be merged.

The three-way merge uses the ancestor of the changed files to identify blocks of content that have changed in neither, one, or both of the derived versions. Blocks that have changed in neither are left as they are. Blocks that have changed in only one derivative use that changed version. If a block is changed in both derivatives, the changed version is used if it has the same content on both sides, but if the changes differ, it is marked as a conflict situation and left for the user to resolve.

Three-way merging is implemented by the ubiquitous diff3 program, and was the central innovation that allowed the switch from file-locking based revision control systems to merge-based revision control systems. It is extensively used by the Concurrent Versions System (CVS).

8.1.2 Recursive three-way merge

Widespread use of three-way merge based revision control tools has brought up cases where simple-minded three-way merging causes spurious conflicts and sometimes silently re-enacts reverted changes. Examples of such situations include criss-cross merges[1] and three-way rename conflicts.

The problems of three-way merge arise in situations where two derivative states do not have a unique latest common ancestor (LCA). To deal with these problems, the recursive three-way merge constructs a virtual ancestor by merging the non-unique ancestors first. This technique is used by the Git revision control tool.

Recursive three-way merge can only be used in situations where the tool has knowledge about the total ancestry DAG (directed acyclic graph) of the derivatives to be merged. Consequently, it cannot be used in situations where derivatives or merges do not fully specify their parent(s).

8.1.3 Fuzzy patch application

A patch is a file that contains a description of changes to a file. In the Unix world, there has been a tradition to disseminate changes to text files as patches in the format that is produced by "diff -u". This format can then be used by the patch program to re-apply (or remove) the changes into (or from) a text file, or a directory structure containing text files.

However, the patch program also has some facilities to apply the patch into a file that is not exactly similar as the origin file that was used to produce the patch. This process is called fuzzy patch application, and results in a kind of asymmetric three-way merge, where the changes in the patch are discarded if the patch program cannot find a place where to apply them.

Like CVS started as a set of scripts on diff3, GNU arch started as a set of scripts on patch. However, fuzzy patch application is a relatively untrustworthy method, sometimes misapplying patches that have too little context (especially ones that create a new file), sometimes refusing to apply deletions that both derivatives have done.

8.1.4 Weave merge

Weave merge is an algorithm that does not make use of a common ancestor for two files. Instead, it tracks how single lines are added and deleted in derivative versions of files, and produces the merged file on this information.

For each line in the derivative files, weave merge collects the following information: which lines precede it, which follow it, and whether it was deleted at some stage of either derivative's history. If either derivative has had the line deleted at some point, it must not be present in the merged version. For other lines, they must be present in the merged version.

The lines are sorted into an order where each line is after all lines that have preceded it at some point in history, and before all lines that have followed it at some point in history. If these constraints do not give a total ordering for all lines, then the lines that do not have an ordering with respect to each other are additions that conflict.

Weave merge was apparently used by the commercial revision control tool BitKeeper and can handle some of the problem cases where a three-way merge produces wrong or bad results. It is also one of the merge options of the GNU Bazaar revision control tool, and is used in Codeville.

8.1.5 Patch commutation

Patch commutation is used in Darcs to merge changes, and is also implemented in git (but called "rebasing"). Patch commutation merge means changing the order of patches (i.e. descriptions

of changes) so that they form a linear history. In effect, when two patches are made in the context of a common situation, upon merging, one of them is rewritten so that it appears to be done in the context of the other.

Patch commutation requires that the exact changes that made derivative files are stored or can be reconstructed. From these exact changes it is possible to compute how one of them should be changed in order to rebase it on the other. For instance, if patch A adds line "X" after line 7 of file F and patch B adds line "Y" after line 310 of file F, B has to be rewritten if it is rebased on A: the line must be added on line 311 of file F, because the line added in A offsets the line numbers by one.

Patch commutation has been studied a lot formally, but the algorithms for dealing with merge conflicts in patch commutation still remain open research questions. However, patch commutation can be proven to produce "correct" merge results where other merge strategies are mostly heuristics that try to produce what users want to see.

The Unix program `flipdiff` from the "patchutils" package implements patch commutation for traditional patches produced by `diff -u`.

8.2 Trends

The technological advancements in the 3-way merge method have led to the increase in popularity among software development environments to institute concurrent modification through branching in their practices of software configuration management (SCM). In the early to mid-1990s branching was a discouraged practice in smaller software development groups due to the complexities and conflicts introduced through the merging process and the low availability of cost-effective 3-way merge tools. However, this practice was more in demand among larger groups merely due to the increased likelihood that two developers would need to modify the same file at the same time. Merging, at that time, was indeed a challenge and in some environments, additional proprietary conventions were introduced to simplify the necessary merge.

In the early 2000s, the increased availability of reliable 3-way merge tools reduced the time that software development groups had to spend concerning themselves with the technical limitations of their infrastructure. Even smaller software groups are more inclined to approach concurrent modification in their revision control systems. Software developers use a variety of tools and techniques to facilitate 3-way merges, including visual tools for viewing changes side by side.

Resolving conflicts in 3-way merges still remains one of the more taxing tasks of any software development team. This is especially because the person resolving the merge needs prior knowledge of the original code, the intermediate conflicting changes, and the result wanted.

Chapter 9

Fundamental Concepts

版本控制（Revision control）是维护工程蓝图的标准作法，能追踪工程蓝图从诞生一直到定案的过程。此外，版本控制也是一种软件工程技巧，借此能在软件开发的过程中，确保由不同人所编辑的同一程序文件都得到同步。

9.1 Overview

通过文档控制（documentation control），能记录任何工程项目内各个模块的改动历程，并为每次改动编上序号。

一种简单的版本控制形式如下：赋给图的初版一个版本等级“A”。当做了第一次改变后，版本等级改为“B”，以此类推。最简单的例子是，最初的版本指定为“1”，当做了改变之后，版本编号增加为“2”，以此类推。

借此，版本控制能提供项目的设计者，将设计回复到之前任一状态的选择权，这种选择权在设计过程进入死胡同时特别重要。

理论上所有的信息记录都可以加上版本控制，在过去的实践中，除了软件开发的流程，其它的领域中很少有使用较复杂的版本控制技巧与工具（虽然可能为其带来许多好处）。目前已有人开始用版本控制软件来管理 CAD 电子文件、电路板设计，来补足本来由人手工执行的传统版本控制。

9.2 Source-management models

软件设计师常会利用版本控制来追踪、维护源码、文件以及配置文件等等的改动，并且提供控制这些改动控制权的程序。

在最简单的情况下，软件设计师可以自己保留一个程序的许多不同版本，并且为它们做适当的编号。这种简单的方法已被用在很多大型的软件项目中。该方法虽然可行，但不够有效率。除了必须同时维护很多几乎一样的源码备份外；而且极度依赖软件设计师的自我修养与开发纪律，但这却常是导致错误发生的原因。

有时候，一个程序同时存有两个以上的版本也有其必要性，例如：在一个为了部署的版本中程序错误已经被修正、但没有加入新功能；在另一个开发版本则有新的功能正在开发、也有新的错误待解决，这使得同时间需要不同的版本并修改。

此外，为了找出只存在于某一特定版本中（为了修正了某些问题、或新加功能所导致）的程序错误、或找出程序错误出现的版本，软件除错者也必须借由比对不同版本的程序源码以找出问题的位置。

最简单的版本控制就是保留软件不同版本的数份 copy，并且适当编号。许多大型开发实例都是使用这种简单技巧。虽然这种方法能用，但是很没效率。一是因为保存的数份 copy 几乎完全一样，也因为这种方法要高度依靠开发者的自我纪律，而常导致错误。因此，有人开发出了将部分或全部版本控制工作自动化的版本控制系统。

1. 差分编码

   大部份的版本控制软件采用差分编码：只保留文件相继版本之间的差异，这个方法可以更有效的储存数个版本的文件。

2. 中央式系统与分布式系统

   大部分的软件开发实例，会有好几个开发人员同时工作。如果两个人员同时要改变同一个文件，而没有管理存取权限，很可能会覆盖彼此的工作。所以权限管理控制系统会在两种方法中择一解决：采用中央式系统，由中央权威管理存取权限；或是像分布式系统容许多个单位同时进行，包括同时更改同一文件。

   传统上版本控制系统都是采用中央式系统：所有版本控制的工作在一个服务器进行，由中央权威管理存取权限“锁上”文件库中的文件，一次只让一个开发者工作。2000 年后，TeamWare、BitKeeper 和 GNU 开始用分布式系统：开发者直接在各自的本地文件库工作，并容许多个开发者同时更改同一文件，而各个文件库有另一个合并各个改变的功能。这个方式让开发者能不靠网络也能继续工作，也让开发者有充分的版本控制能力，而不需经中央权威许可。分布式系统仍然可以有文件上锁功能。

   分布式系统 Linux 内核的发明人林纳斯 · 托瓦兹就是分布式版本控制系统的支持者，他开发了目前被开源社群广泛使用的分布式版本控制系统 Git。

3. 文件上锁

   文件上锁功能能对高难度的合并（例如大幅更改大文件或文件群的许多部份）提

供一些保护，但其他开发者仍然可以绕过版本控制系统改变文件（这本身就是很大的问题)。所以文件上锁功能带来的功效与副作用一直饱受争议。

4. 其他功能

   有些进步的版本控制工具提供更多功能，例如：

   - 管理谁能改变程序的哪个部位，
   - 提供某一个人控制权来审查哪些改变可以过关；
   - 与开发环境整合。

   维基百科用的 MediaWiki 也有版本控制的功能。

9.3 The Repository

At the core of the version control system is a repository, which is the central store of that system's data. The repository usually stores information in the form of a filesystem tree—a hierarchy of files and directories. Any number of clients connect to the repository, and then read or write to these files. By writing data, a client makes the information available to others; by reading data, the client receives information from others.

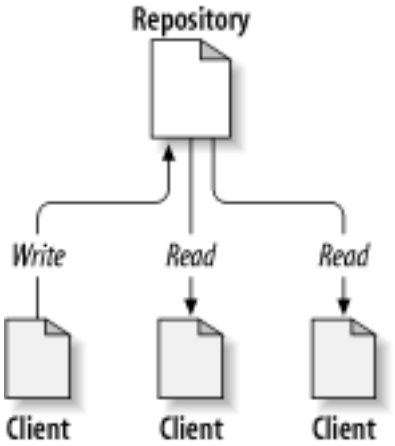


Figure 9.1: A typical client/server system

Why is this interesting? So far, this sounds like the definition of a typical file server. And indeed, the repository is a kind of file server, but it's not your usual breed. What makes the repository special is that as the files in the repository are changed, the repository remembers each version of those files.

这有什么意义吗? 说了这么多，其中，版本库听起来和一般的文件服务器没什么不同。事实上，版本库的确是一种文件服务器，但不是“一般”的文件服务器，它的特别

之处在于，它会记录每一次改变：每个文件的改变，甚至是目录树本身的改变，例如文件和目录的添加、删除和重新组织。

When a client reads data from the repository, it normally sees only the latest version of the filesystem tree. But what makes a version control client interesting is that it also has the ability to request previous states of the filesystem from the repository. A version control client can ask historical questions such as "What did this directory contain last Wednesday?" and "Who was the last person to change this file, and what changes did he make?" These are the sorts of questions that are at the heart of any version control system.

9.4 The Working Copy

A version control system's value comes from the fact that it tracks versions of files and directories, but the rest of the software universe doesn't operate on "versions of files and directories". Most software programs understand how to operate only on a single version of a specific type of file. So how does a version control user interact with an abstract—and, often, remote—repository full of multiple versions of various files in a concrete fashion? How does his or her word processing software, presentation software, source code editor, web design software, or some other program—all of which trade in the currency of simple data files—get access to such files? The answer is found in the version control construct known as a working copy.

A working copy is, quite literally, a local copy of a particular version of a user's VCS-managed data upon which that user is free to work. Working copies[5] appear to other software just as any other local directory full of files, so those programs don't have to be "version-control-aware" in order to read from and write to that data. The task of managing the working copy and communicating changes made to its contents to and from the repository falls squarely to the version control system's client software.

9.5 The Versioning Models

If the primary mission of a version control system is to track the various versions of digital information over time, a very close secondary mission in any modern version control system is to enable collaborative editing and sharing of that data. But different systems use different strategies to achieve this. It's important to understand these different strategies, for a couple of reasons. First, it will help you compare and contrast existing version control systems, in case you encounter other systems similar to Subversion. Beyond that, it will also help you make more effective use of Subversion,

since Subversion itself supports a couple of different ways of working.

9.5.1 The problem of file sharing

All version control systems have to solve the same fundamental problem: how will the system allow users to share information, but prevent them from accidentally stepping on each other's feet? It's all too easy for users to accidentally overwrite each other's changes in the repository.

所有的版本控制系统都需要解决这样一个基础问题：怎样让系统允许用户共享信息，而不会让他们因意外而互相干扰？版本库里意外覆盖别人的更改非常的容易。

Consider the scenario shown in Figure below, "The problem to avoid". Suppose we have two coworkers, Harry and Sally. They each decide to edit the same repository file at the same time. If Harry saves his changes to the repository first, it's possible that (a few moments later) Sally could accidentally overwrite them with her own new version of the file. While Harry's version of the file won't be lost forever (because the system remembers every change), any changes Harry made won't be present in Sally's newer version of the file, because she never saw Harry's changes to begin with. Harry's work is still effectively lost—or at least missing from the latest version of the file—and probably by accident. This is definitely a situation we want to avoid!

考虑下图中"需要避免的问题"的情景。假设我们有两个共同工作者，Harry 和 Sally。他们想同时编辑版本库里的同一个文件，如果 Harry 先保存他的修改，（过了一会）Sally 可能凑巧用自己的版本覆盖了这些文件，Harry 的更改不会永远消失（因为系统记录了每次修改），但 Harry 所有的修改不会出现在 Sally 新版本的文件中，因为她没有在开始的时候看到 Harry 的修改。所以 Harry 的工作还是丢失了——至少是从最新的版本中丢失了——而且可能是意外的。这就是我们要明确避免的情况！

9.5.2 The lock-modify-unlock solution

Many version control systems use a lock-modify-unlock model to address the problem of many authors clobbering each other's work. In this model, the repository allows only one person to change a file at a time. This exclusivity policy is managed using locks. Harry must "lock" a file before he can begin making changes to it. If Harry has locked a file, Sally cannot also lock it, and therefore cannot make any changes to that file. All she can do is wait for Harry to finish his changes, save the file and release his lock. After Harry unlocks the file, Sally can take her turn by locking the file. Then she may read the latest version of the file and edit it. Figure below, "The lock-modify-unlock solution" demonstrates this simple solution.

许多版本控制系统使用锁定 -修改 -解锁机制解决这种问题，在这样的模型里，在一

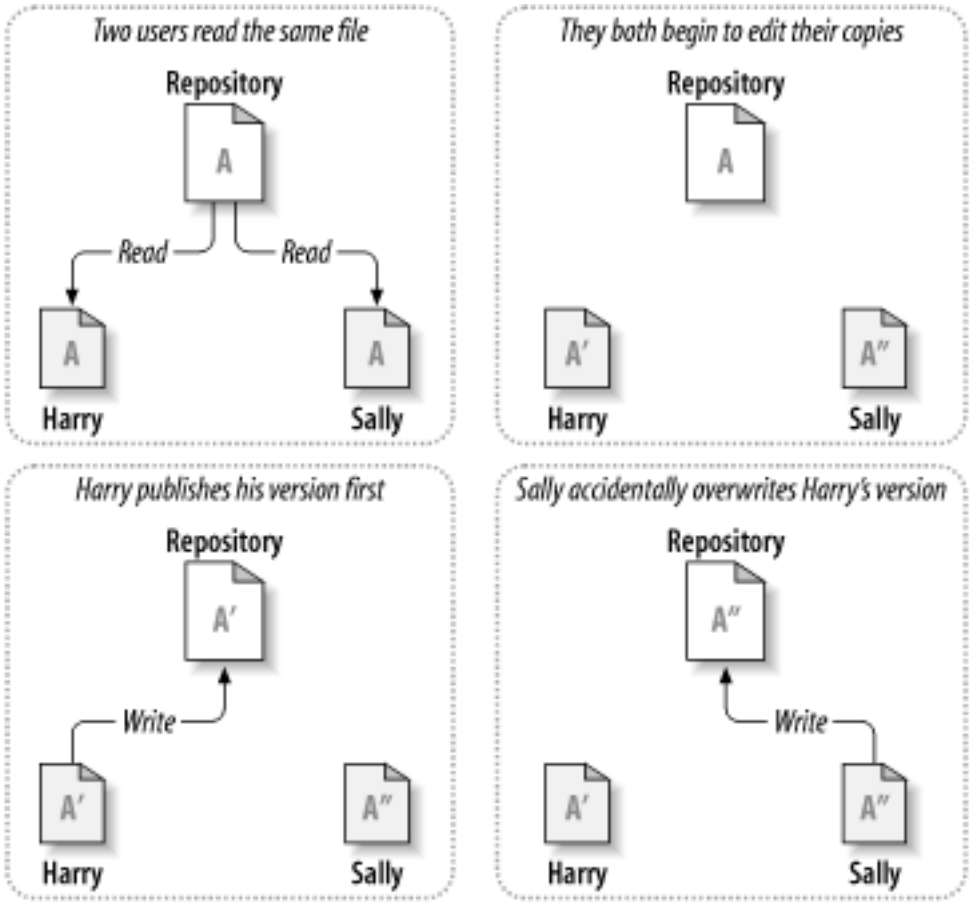


Figure 9.2: The problem to avoid

个时间段里版本库的一个文件只允许被一个人修改。首先在修改之前，Harry 要“锁定”住这个文件，锁定很像是从图书馆借一本书，如果 Harry 锁住这个文件，Sally 不能做任何修改，如果 Sally 想请求得到一个锁，版本库会拒绝这个请求。在 Harry 结束编辑并且放开这个锁之前，她只可以阅读文件。Harry 解锁后，就要换班了，Sally 得到自己的轮换位置，锁定并且开始编辑这个文件。

锁定 -修改 -解锁模型有一点问题就是限制太多，经常会成为用户的障碍：

- 锁定可能导致管理问题。
  有时候 Harry 会锁住文件然后忘了此事，这就是说 Sally 一直等待解锁来编辑这些文件，她在这里僵住了。然后 Harry 去旅行了，现在 Sally 只好去找管理员放开锁，这种情况会导致不必要的耽搁和时间浪费。
- 锁定可能导致不必要的串行开发。

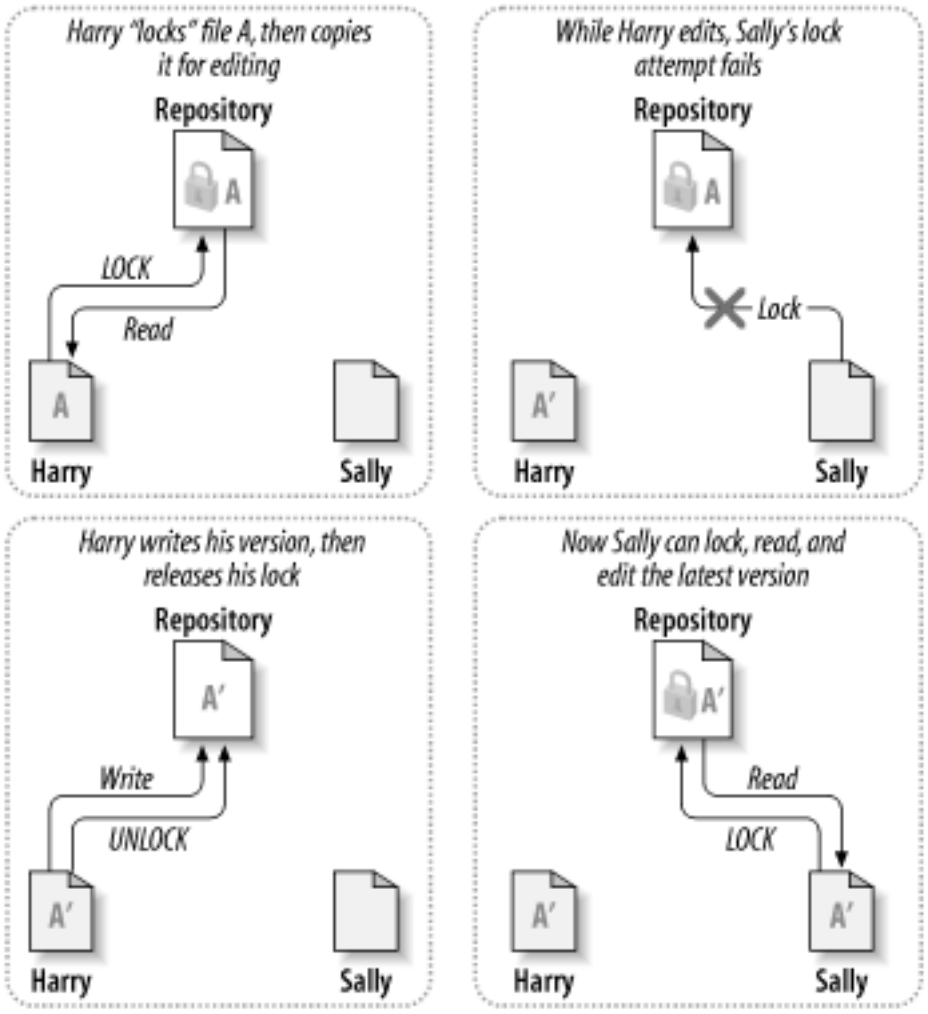


Figure 9.3: The lock-modify-unlock solution

如果 Harry 编辑一个文件的开始，Sally 想编辑同一个文件的结尾，这种修改不会冲突，设想修改可以正确的合并到一起，他们可以轻松的并行工作而没有太多的坏处，没有必要让他们轮流工作。

- 锁定可能导致错误的安全状态。
  假设 Harry 锁定和编辑一个文件 A，同时 Sally 锁定并编辑文件 B。但是如果 A 和 B 互相依赖，修改导致它们不兼容会怎么样呢？这样 A 和 B 不能正确的工作了，锁定机制对防止此类问题将无能为力—从而产生了一种处于安全状态的假相。很容易想象 Harry 和 Sally 都以为自己锁住了文件，而且从一个安全，孤立的情况开始工作，因而没有尽早发现他们不匹配的修改。锁定经常成为真正交流的替代品。

9.5.3 The copy-modify-merge solution

Subversion, CVS, and many other version control systems use a copy-modify-merge model as an alternative to locking. In this model, each user's client contacts the project repository and creates a personal working copy. Users then work simultaneously and independently, modifying their private copies. Finally, the private copies are merged together into a new, final version. The version control system often assists with the merging, but ultimately, a human being is responsible for making it happen correctly.

这是一个例子，Harry 和 Sally 为同一个项目各自建立了一个工作副本，工作是并行的，修改了同一个文件 A，Sally 首先保存修改到版本库，当 Harry 想去提交修改的时候，版本库提示文件 A 已经过期，换句话说，A 在他上次更新之后已经更改了，所以当他通过客户端请求合并版本库和他的工作副本之后，碰巧 Sally 的修改和他的不冲突，所以一旦他把所有的修改集成到一起，他可以将工作拷贝保存到版本库，下图“拷贝 -修改 -合并”方案”和图“拷贝 -修改 -合并”方案 (续)”展示了这一过程。

但是如果 Sally 和 Harry 的修改交迭了该怎么办？这种情况叫做冲突，这通常不是个大问题，当 Harry 告诉他的客户端去合并版本库的最新修改到自己的工作副本时，他的文件 A 就会处于冲突状态：他可以看到一对冲突的修改集，并手工的选择保留一组修改。需要注意的是软件不能自动的解决冲突，只有人可以理解并作出智能的选择，一旦 Harry 手工的解决了冲突——也许需要与 Sally 讨论—它可以安全的把合并的文件保存到版本库。

拷贝 -修改 -合并模型感觉有一点混乱，但在实践中，通常运行的很平稳，用户可以并行的工作，不必等待别人，当工作在同一个文件上时，也很少会有交迭发生，冲突并不频繁，处理冲突的时间远比等待解锁花费的时间少。

最后，一切都要归结到一条重要的因素：用户交流。当用户交流贫乏，语法和语义

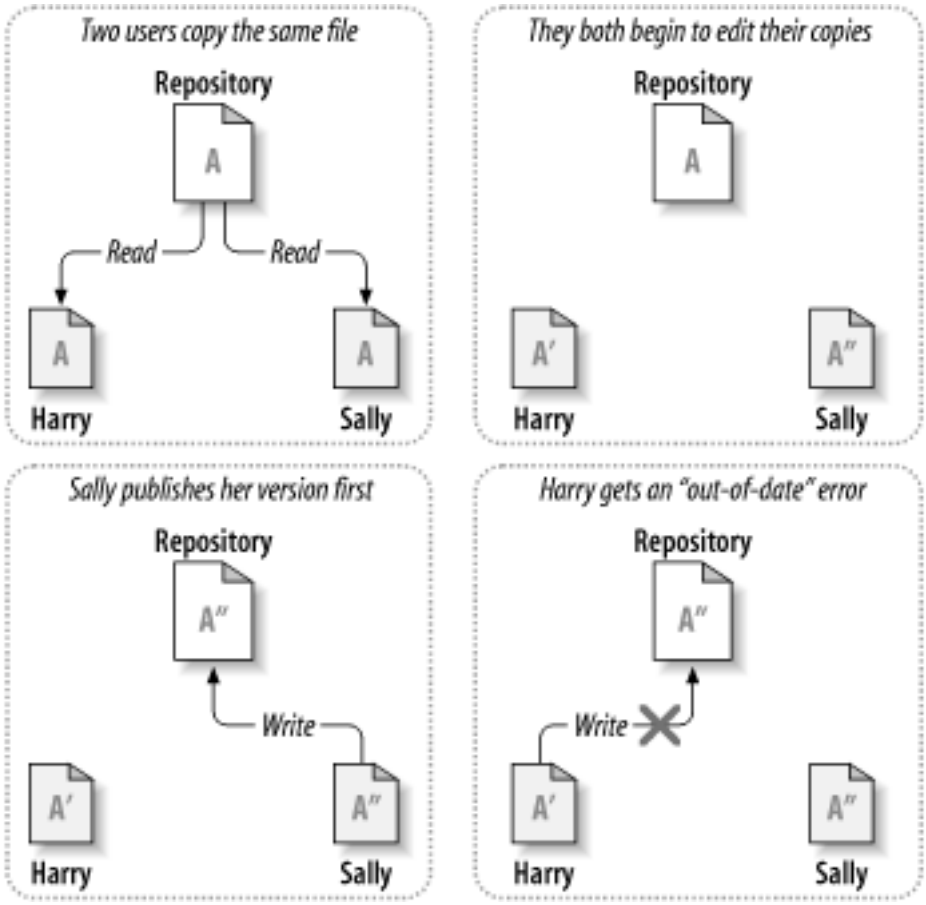


Figure 9.4: The copy-modify-merge solution

的冲突就会增加，没有系统可以强制用户完美的交流，没有系统可以检测语义上的冲突，所以没有任何证据能够承诺锁定系统可以防止冲突，实践中，锁定除了约束了生产力，并没有做什么事。

什么时候锁定是必需的

锁定 -修改 -解锁模型被认为不利于协作，但有时候锁定会更好。

拷贝 -修改 -合并模型假定文件是可以根据上下文合并的：就是版本库的文件主要是以行为基础的文本文件 (例如程序源代码)。但对于二进制格式，例如艺术品或声音，在这种情况下，十分有必要让用户轮流修改文件，如果没有线性的访问，有些人的许多工作就最终要被放弃。

尽管 Subversion 一直主要是一个拷贝 -修改 -合并系统，但是它也意识到了需要锁定一些文件，并且提供这种锁定机制。

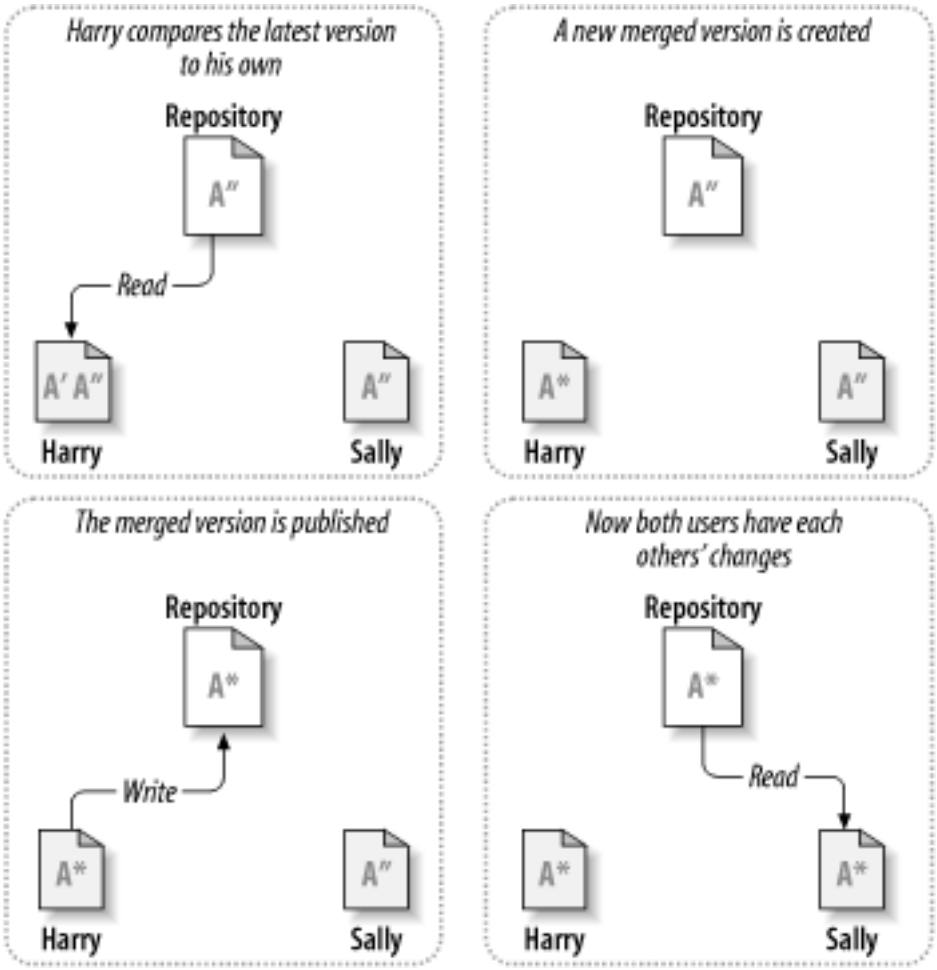


Figure 9.5: The copy-modify-merge solution (continued)

9.6 Concepts

- 基线（Baseline）
  基线是软件文档或源码（或其它产出物）的一个稳定版本，它是进一步开发的基础。
- 文件库（Repository）
  存储文件的新版本还有历史资料的地方，通常是在服务器上。有时候也叫 Depot（像是在 SVK、AccuRev 还有 Perforce 中）工作版本（Working copy）：从文件库中取出一个本地端（客户端）的复制，针对一个特定的时间或是版本。所有在文件库中的文件更改，都是从一个工作版本中修改而来的，这也是这名称的由来。观念上，

这是一个沙盒。

- 提交（Commit）
  将本地端的修改送回文件库。(由版本控制软件处理“跟上次更改相比，哪个文件又被更改”的事)
- 变更（Change）
  对一份文件作的特定更改。
- 变更记录（Change List）
- 取出（Check-Out）
  从文件库取出文件到本地端（客户端)。
- 更新（Update）
  将文件库的修改送到本地端（与提交相反)
- 合并（Merge / Integration）
  合并各个改变。
- 版次（Revision）
  一个 revision 或 version 指的是一系列版本变迁的其中之一。
- 导入（Import）
- 导出（Export）
- 冲突（Conflict）
  当两方更改同一份文件会发生冲突。

Chapter 10

Histroy

10.1 CVS

CVS（Concurrent Versions System）是一种版本控制系统，由 Dick Grune 于 1986 年开发，使用它，可以记录下源文件的历史。开始 CVS 只是一系列 shell 脚本，后来发展为协作版本系统或者并发版本系统，方便软件的开发者和使用者协同工作。

例如，修改软件时可能会不知不觉混进一些 bug，而且可能过了很久开发者才会察觉到它们的存在。有了 CVS，就可以很容易地恢复旧版本，并从中看出到底是哪个修改导致了这个 bug。有时这是很有用的。

当然也可以把曾经创建的每个文件的所有版本都保存下来，但这会浪费大量的磁盘空间，而 CVS 用一种聪明的办法把一个文件的所有版本保存在一个文件里，其中仅仅保存不同版本之间的差异。

如果你是项目开发组的一员，CVS 也会帮助你。除非极为小心，成员之间很容易互相覆盖文件。一些编辑器，如 GNU Emacs，会保证同一时间内同一文件绝不会被两个人修改。不幸的是，如果有人用了另外的编辑器，这种保护就没用了。CVS 用隔离开不同的开发者的办法解决了这个问题，在 CVS 中每个开发者在自己的目录里工作，等每一个开发者都完成了他们自己的工作后，CVS 会将它们合并到一起。

CVS 最初由 Dick Grune 在 1986 年 12 月以 shell 脚本的形式发布在comp.sources.unix的新闻组第 6 卷里。虽然当前的 CVS 中已经没什么代码来自于这些 shell 脚本了，但许多 CVS 的冲突解决算法是从它们来的。

1989 年 4 月，Brian Berliner 设计了 CVS 并编写了代码。之后 Jeff Polk 帮助 Brian 设计了 CVS 模块和销售商分支支持，可以从好多渠道得到 CVS，包括从因特网上自由下载。

很多开源或者自由软件项目都使用 CVS 作为其程序员之间的中心点，以便能够综

合各程序员的改进和更改。这些项目包括：Gnome、KDE、GIMP、Wine 等。现在 CVS 使用 GNU 通用公共许可证授权。

有一个关于 CVS 的邮件列表，名叫 info-CVS@GNU.org，写邮件到 info-CVS-request@GNU.org 来订阅或退订。

如果更喜欢新闻组（Usenet），在 news:GNU.CVS.help 有一个 info-CVS@GNU.org 的单向镜像（发送到邮件列表的邮件会自动转发到新闻组，反之则不行）。

news:comp.software.config-mgmt 比较适合于讨论 CVS（还有其它一些配置管理系统）。将来，可能会创立一个 comp.software.config-mgmt.CVS，但那要取决于在 news:comp.software.config-mgmt 上是否有足够的 CVS 讨论。

也可订阅 bug-CVS@GNU.org 邮件列表。要订阅它可以发邮件到 bug-CVS-request@GNU.org。对应 bug-CVS@GNU.org 有一个双向的新闻组镜像（发在新闻组的信息会自动转到邮件列表，反之亦然）名为 news:GNU.CVS.bug。

可以这样理解，CVS 是一个将一组文件放在层次目录树中以保持同步的系统，人们可以从 CVS 服务器上更新他们的本地层次树副本，并将修改的结果或新文件发回，或者删除旧文件。

如果大家曾经参与过多人协作开发的项目，大家肯定有这样的痛苦经历：由于多个人同时修改同一个文件，自己辛辛苦苦修改的程序被别人彻底删除了。另外，如果你的软件/程序已经发布了四个版本，而这时候用户需要你修改第三个版本的东西，也许你会因为只保留了最新版本而痛哭流涕。

如果你修改了别人的源程序，不过只是修改了很少的一部分，比如增加了一个方法，这时你想让其他人看到你对这个程序的修改部分，而不是让大家对全部程序都阅读一遍，那么采用与 CVS 搭配使用的文件比较工具将会大大提高工作效率。

采用 CVS 进行版本管理的另一个好处是，你不用自己备份自己的源程序，有了 CVS 这一切都变得异常简单，你所要做的仅仅是将自己的代码的每一个版本提交一份到服务器上，其他的一切都由 CVS 来完成。

CVS 基于客户端/服务器结构的行为使得其可容纳多用户，构成网络也很方便。这一特性使得 CVS 成为位于不同地点的人同时处理数据文件（特别是程序的源代码）时的首选（不过现已被 Git、SVN 等逐渐替代）。

10.1.1 CVS Limits

CVS 可以为你做很多，但不要指望它能为每一个人做每一件事情。

- CVS 不支持文件的复制和重新命名。
- 没有原子性提交（Atomic commit）

- CVS 只支持文本文件

具体来说：

1. CVS 不是一个 BUILD 系统。
   - 虽然用户的仓库（repository）和模块文件与用户的 BUILD 系统互相作用（例如：Makefiles），但它们本质上还是互相独立的[1]。
   - CVS 不能指导用户如何构造什么。它只是将用户所设计的一种树结构文件保存下来以备恢复之用。
   - CVS 不能决定如何在一个检出工作目录使用磁盘空间。如果用户在每一个目录中都写下 Makefile 或脚本，且必须知道其它一切的相对位置，有时不得不检出整个仓库。
   - 如果用户将工作模块化，并且建立了一个共享文件的 build 系统（通过 links、mounts、Makefiles 里的 VPATH 等），那么用户就可以随意安排磁盘的使用。
   - 不过要记住构建和维护这样一个系统是要做许多工作的。而 CVS 不善此道。
   - 当然了，应该在 CVS 下放一个工具来支持这样一个构造系统（脚本、Makefile 等）。
   - 当有些变化发生在 CVS 范围之外时，要想想什么文件需要重建。一个传统的方法是用 make 来构造，并用一些自动化的工具来产生 make 所用的相关文件。
2. CVS 不能替代管理。
   - 项目经理和项目负责人应经常与开发人员交流以确保他们时时记得进度表、合并点、分支名和发布日期。如果他们不这样做，CVS 也没用。
   - CVS 只是一个用来使用户的资源与步调一致的工具。但如果用户是风笛手和作曲家，记住没有哪种乐器会自己演奏或是作曲。
3. CVS 不能代替开发者之间的交流。
   - 在单个文件内遇到冲突时，大多数开发者不费多大力气就能解决它们。但更常见的 “冲突 (conflict)”，是那些难度较大、不在开发者之间进行交流就没法解决的问题。
   - 当在一个文件内或多个文件中同时发生变化时，CVS 并不知道何时它们会在逻辑上发生冲突。它的冲突（conflict）概念是纯粹文本意义上的，这种冲突会在同一个文件的两种变化十分接近以致于会破坏合并命令（如 diff3）。
   - CVS 决不会指出程序逻辑上非文本或分布式的冲突。
     例如：假如有人改变了在文件 A 中定义的函数 X 的参数。同时，别人在编辑文件 B，仍用旧参数调用 X 这个函数。此时产生的冲突 CVS 可就无能为力了。

[1] 关于结合 cvs 进行 build，请参考 Builds。

- 要养成经常阅读说明书和经常与你的同伴交谈的习惯。

4. CVS 没有变化控制。
   - 变化控制可以指许多事情。首先它的意思可以是 BUG 跟踪（bug-tracking），就是说它能维持一个数据库，其中包括已报告的 BUG 和每一个 BUG 状态 (是否已更正？在哪一个版本中？提交这个 BUG 的人是否认为已经更正？)。为了使 CVS 和一个外部的跟踪 BUG 系统协调一致，参考 rcsinfo 和 verifymsg 文件[2]。
   - 变化控制的另一个方面指跟踪这样的情况，即对好几个文件的改变实际上只是同一个逻辑变动。如果你在一次 `cvs commit` 操作中检入几个文件，CVS 会忘掉它们是一起检入的，它们共用一个 LOG 信息的事实只是把它们绑在一起而已。做一个 GNU 风格的 ChangeLog 可能会有点用。
   - 在一些系统中，变化控制的另一个方面是跟踪每个变化的状态的能力。一些变化由一个开发者写出，而另一些变化则由另一个开发者来作出评论，等等。一般来讲，用 CVS 来做，是产生一个 diff（用 `cvs diff` 或 `diff`），并且用电子邮件寄给某人，此人就可以用 `patch` 来应用它。这是非常灵活的，但依赖于 CVS 之外的机制可以保证事情不会崩溃。
5. CVS 不是自动测试程序。
   - 强制利用 `commitinfo` 文件测试套件应该是可能的。不过我没有听说过多少项目试图那样做或那里有微妙的陷阱。
6. CVS 没有内置的处理模型。
   - 有些系统提供一些方法确保变更或发布通过不同的步骤，以及各种所需的批准过程。一般地，可以用 CVS 来完成它，但是可能要多做点工作。有些情况下想用 commitinfo、loginfo、rcsinfo 或 verifymsg 文件，要求在 CVS 提交之前完成某些操作。也会考虑诸如 branches 和 tags 等特性是否能用在一个开发树中执行任务，然后仅当它们被证实就把某些修改合并到一棵稳定的树中。

10.2 Subversion

在 2000 年初，开发人员要写一个 CVS 的自由软件代替品，它保留 CVS 的基本思想，但没有它的错误和局限。

2000 年 2 月，他们联系了 Open Source Development with CVS（Coriolis, 1999）的作者 Karl Fogel，问他是否愿意为这个新项目工作。巧的是这时 Karl 已经在和他的朋友 Jim Blandy 讨论一个新的版本控制系统的设计。在 1995 年，两人开了一家提供 CVS 技术支持的公司，叫作 Cyclic Software。虽然公司已经卖掉了，他们仍然在日常工作中使用 CVS。

[2]参阅 Administrative files

在使用 CVS 时受到的束缚已经让 Jim 开始仔细思考管理版本化数据的更好的路子。他不仅已经起好了名字“Subversion”，而且有了 Subvesion 资料库的基本设计。当 CollabNet 打来电话时，Karl 立刻同意为这个项目工作。Jim 征得他的老板 RedHat Software 的同意，让他投入这个项目，而且没有时间限制。CollabNet 雇用了 Karl 和 Ben Collins-Sussman，从 5 月份开始详细设计。由于 Greg Stein 和 CollabNet 的 Brian Behlendorf 和 Jason Robbins 作了恰当的推动，Subversion 很快吸引了一个活跃的开发人员社区。这说明了许多人有相同的受制于 CVS 的经验，他们对终于有机会对它做点什么表示欢迎。

最初的设计团队设定了几个简单的目标。他们并不想在版本控制方法论上有新突破。他们只想修补 CVS。他们决定 Subversion 应该与 CVS 相似，保留相同的开发模型，但不复制 CVS 最明显的缺点。虽然它不一定是 CVS 的完全的替代品，但它应该和 CVS 相似，从而任何 CVS 用户可以不费什么力气的转换过来。

经过 14 个月的编码，在 2001 年 8 月 31 号，Subversion 可以“自我寄生”了。就是说，Subversion 开发人员停止使用 CVS 管理 Subversion 的源代码，开始使用 Subversion 代替。

虽然 CollabNet 发起了这个项目，而且仍然支助一大部分的工作（它为一些专职的 Subversion 开发人员发薪水）。但是 Subversion 像大部分开放源码的项目一样运作，由一个松散透明，鼓励能者多劳的规则管理。CollabNet 的版权许可证和 Debian FSG 完全兼容。换句话说，任何人可以免费下载，修改，按自己的意愿重新分发 Subversion，而不必得到来自 CollabNet 或其他任何人的许可。

虽然在 2006 年 Subversion 的使用族群仍然远少于传统的 CVS，但已经有许多开放原码团体决定将 CVS 转换为 Subversion。已经转换使用 Subversion 的包括了 FreeBSD、Apache Software Foundation、KDE、GNOME、GCC、Python、Samba、Mono 以及许多团体。许多开发团队换用 Subversion 是因为 Trac、SourceForge、CollabNet、CodeBeamer 等项目协同作业软件以及 Eclipse、NetBeans 等 IDE 提供 Subversion 的支持集成。除此之外，一些自由软件开发的协作网如 SourceForge.net 除了提供 CVS 外，现在也提供项目开发者使用 Subversion 作为源代码管理系统，JavaForge、Google Code 以及 BountySource 则以 Subversion 作为官方的源代码管理系统。

2009 年，绝大多数 CVS 服务已经改用 SVN。此时 CVS 早已经停止维护。不过 CVS 也有了合适的替代品。

2009 年 11 月，Subversion 被 Apache Incubator 项目所接收。

2010 年 1 月，正式成为 Apache 软件基金会的一个顶级项目。

10.2.1 Subversion Features

- 统一的版本号。CVS 是对每个文件顺序编排版本号，在某一时间各文件的版本号各不相同。而 Subversion 下，任何一次提交都会对所有文件增加到同一个新版本号，即使是提交并不涉及的文件。所以，各文件在某任意时间的版本号是相同的。版本号相同的文件构成软件的一个版本。
- 原子提交。一次提交不管是单个还是多个文件，都是作为一个整体提交的。在这当中发生的意外例如传输中断，不会引起数据库的不完整和数据损坏。
- 重命名、复制、删除文件等动作都保存在版本历史记录当中。
- 对于二进制文件，使用了节省空间的保存方法。(简单的理解，就是只保存和上一版本不同之处)
- 目录也有版本历史。整个目录树可以被移动或者复制，操作很简单，而且能够保留全部版本记录。
- 分支的开销非常小。
- 优化过的数据库访问，使得一些操作不必访问数据库就可以做到。这样减少了很多不必要的和数据库主机之间的网络流量。
- 支持元数据（Metadata）管理。每个目录或文件都可以定义属性（Property），它是一些隐藏的键值对，用户可以自定义属性内容，而且属性和文件内容一样在版本控制范围内。
- 支持 FSFS 和 Berkeley DB 两种资料库格式。

10.2.2 Subversion Limits

- 只能设置目录的访问权限，无法设置单个文件的访问权限。(目前可以通过辅助模块比如 wandisco access control 实现单文件访问)
- 数据库为二进制格式，无法方便的利用其它软件读取数据库的内容。

Subversion 可以管理任何类型的文件集——它并非是程序员专用的。

对于企业级应用，Subversion 还有其先天不足，比如对于多个地点的并行开发。Wandisco 公司为此开发了 subversion multisite，实现异地对等服务器自动同步，支持并行开发以及异地备份。

ALM（Application Lifecycle Management）是软件配置管理的未来趋势，各种软件版本工具包括 subversion 都要集成到其中。目前 UberSVN 是唯一的以 Subversion 为基础构建的 ALM 平台，并实现了协同开发以及社交化编码。

10.3 BitKeeper

BitKeeper 是一套分布式版本控制的软件。BitKeeper 的竞争主要是对其他系统 Git 和 Mercurial。BitKeeper 是由 BitMover 公司所开发，总部位于美国加州坎贝尔，总裁拉里麦沃伊，曾设计 TeamWare 。

BitKeeper 的许多概念是取自于 TeamWare。它的主要卖点是，它是一个分布式的版本控制工具，而不是 CVS 或 SVN。

10.4 Mercurial

Mercurial 是一个跨平台的分布式版本控制软件。Mercurial 主要由 Python 语言实现，不过也包含一个用 C 实现的二进制比较工具。Mercurial 一开始的主要运行平台是 Linux。现在 Mercurial 已经被移植到 Windows、Mac OS X 和大多数类 Unix 系统中。Mercurial 主要由一个命令行程序组成，但现在也有了图形用户界面。对 Mercurial 的所有操作都由用不同的关键字作为参数调用程序 hg 来实现，Hg 是参考水银的化学符号而取的名字。

Mackall 在 2005 年 4 月 19 日第一次发布了 Mercurial，其动机是当月早期 Bitmover 公司宣布撤销其免费版本的 BitKeeper。BitKeeper 已经被用于 Linux 内核的项目版本控制。Mackall 决定为 Linux 内核开发写一个分布式的版本控制软件来替代 BitKeeper。在该项目启动数天前，Linus Torvalds 基于类似的目的开始了另一个版本控制软件 Git。虽然 Linux 内核开发项目决定使用 Git 而不是 Mercurial，但 Mercurial 也使用于在很多其他的项目中[3]。

Mercurial 采用 SHA-1 散列算法来识别修订版本。Mercurial 使用一个基于 HTTP 的协议来接入网络中的版本库，旨在减少往返的提交、连接数和数据传输。Mercurial 也可以工作在 ssh 环境下，其协议和基于 HTTP 的协议非常相似。

Mercurial 的主要设计目标包括高性能、可扩展性、分散性、完全分布式合作开发、能同时高效地处理纯文本和二进制文件，以及分支和合并功能，以此同时保持系统的简洁性 [1] 。Mercurial 也包括一个集成的 Web 界面。

Mercurial 的创建者和主要开发人员是 Matt Mackal。其源代码采用 GNU 通用公共许可证第二版为授权，确保了 Mercurial 是一个自由软件。

Mercurial 图形用户界面有：Hgk（Tcl/Tk)。该程序作为 Mercurial 的插件而开发，现在被直接包含于正式版本中。界面可以通过命令命令'hg view' 来调用（如果安装了该扩展的话)。hgk 最初来源于名为 gitk 的类似工具。hgk 有一个名为 hgview 是纯 Python 编写

[3]Python 的开发人员宣布将从 Subversion 过渡到 Mercurial。不过并没有确定转换的时间，因为过渡小组在等待 hgsubversion 的开发。

的替代软件，同时提供 GTK 和 QT 界面。合并用的的工具包括 (h)gct (Qt) 和 Meld。转换插件可以将 CVS、GITDarcs、GIT、GNU Arch、Monotone、Perforce、Bazaar 和 Subversion 的版本库转换为 Mercurial 的版本库。

从 Mercurial 第 6 版开始支持 Netbeans，另外，TortoiseHg 提供了一个面向 Windows 的基于右键菜单的友好界面，也用于 GNOME 的 Nautilus 文件管理器。

10.5 Git

Git 最初的开发动力来自于 BitKeeper 和 Monotone。Git 最初只是作为一个可以被其他前端比如 Cogito 或 StGIT 包装的后端而开发的。不过，后来 Git 内核已经成熟到可以独立地用作版本控制。很多有名的软件都使用 Git 来进行版本控制，其中有 Linux 内核、X.Org 服务器和 OLPC 内核开发。

早期 Linux 的开发人员是使用 BitKeeper 来管理版本控制和维护源代码。2005 年的时候，开发 BitKeeper 的公司同 Linux 内核开源社区结束合作关系，并收回使用 BitKeeper 的权利。Torvalds 开始着手开发 Git 来替代 BitKeeper。

Git 是用于 Linux 内核开发的版本控制工具。与 CVS、Subversion 一类的集中式版本控制工具不同，它采用了分布式版本库的作法，不需要服务器端软件，就可以运作版本控制，使得源代码的发布和交流极其方便。Git 的速度很快，这对于诸如 Linux kernel 这样的大项目来说自然很重要。Git 最为出色的是它的合并追踪（merge tracing）能力。

作为开源自由原教旨主义项目，Git 没有对版本库的浏览和修改做任何的权限限制，通过其他工具也可以达到有限的权限控制，比如：gitosis, CodeBeamer MR。原本 Git 的使用范围只适用于 Linux / Unix 平台，但在 Windows 平台下的使用也逐渐成熟，这主要归功于 Cygwin、msysgit 环境与 TortoiseGit 这样易用的 GUI 工具。其实 Git 的源代码中已经加入了对 Cygwin 与 MinGW 编译环境的支援，且逐渐完善，为 Windows 使用者带来福音。

在 Windows 平台上有 msysgit 与 TortoiseGit 可资利用。TortoiseGit 还提供有 GUI，现在 git 也提供 windows 版本下载，。

10.5.1 Git Features

Git 和其他版本控制系统（如 CVS）有不少的差别，Git 本身关心档案的整体性是否有改变，但多数的 CVS，或 Subversion 系统则在乎档案内容的差异。因此 Git 更像一个档案系统，直接在本机上取得资料，不必连线到 host 端取资料回来。

Git 库结构如下：

- hooks：存储钩子的文件夹
- logs：存储日志的文件夹
- refs：存储指向各个分支的指针（SHA-1 标识）文件
- objects：存放 git 对象
- config：存放各种设置文档
- HEAD：指向当前所在分支的指针文件路径，一般指向 refs 下的某文件

10.5.2 Git Limits

实际上内核开发团队决定开始开发和使用 Git 来作为内核开发的版本控制系统的时候，世界开源社群的反对声音不少，最大的理由是 Git 太艰涩难懂，从 Git 的内部工作机制来说，的确是这样。但是随着开发的深入，Git 的正常使用都由一些友善的命令稿来执行，使 Git 变得非常好用。现在，越来越多的著名项目采用 Git 来管理项目开发，例如：wine、U-boot 等。

Part II

Version Control with CVS

CVS is a version control system, an important component of Source Configuration Management (SCM). Using it, user can record the history of sources files, and documents. It fills a similar role to the free software RCS, PRCS, and Aegis packages.

CVS is a production quality system in wide use around the world, including many free software projects.

While CVS stores individual file history in the same format as RCS, it offers the following significant advantages over RCS:

- It can run scripts which you can supply to log CVS operations or enforce site-specific polices.
- Client/server CVS enables developers scattered by geography or slow modems to function as a single team. The version history is stored on a single central server and the client machines have a copy of all the files that the developers are working on. Therefore, the network between the client and the server must be up to perform CVS operations (such as checkins or updates) but need not be up to edit or manipulate the current versions of the files. Clients can perform all the same operations which are available locally.
- In cases where several developers or teams want to each maintain their own version of the files, because of geography and/or policy, CVS's vendor branches can import a version from another team (even if they don't use CVS), and then CVS can merge the changes from the vendor branch with the latest files if that is what is desired.
- Unreserved checkouts, allowing more than one developer to work on the same files at the same time.
- CVS provides a flexible modules database that provides a symbolic mapping of names to components of a larger software distribution. It applies names to collections of directories and files. A single command can manipulate the entire collection.
- CVS servers run on most unix variants, and clients for Windows NT/95, OS/2 and VMS are also available. CVS will also operate in what is sometimes called server mode against local repositories on Windows 95/NT.

CVS 是典型的 C/S 结构的软件，因此它也分成服务器端和客户端两部分。与我们平时工作息息相关的部分是客户端，它也是我们天天与之打交道的部分，至于服务器端只要在最开始设定好，以后基本就不必再进行其他操作了。

CVS 的基本工作思路是这样的，在一台服务器上建立一个仓库，仓库里可以存放许多不同项目的源程序。由仓库管理员统一管理这些源程序，这样，就好像只有一个人在修改文件一样。避免了冲突，每个用户在使用仓库之前，首先要把仓库里的项目文件下载到本地。用户做的任何修改首先都是在本地进行，然后用 CVS 命令进行提交，由 CVS

仓库管理员统一修改，这样就可以做到跟踪文件变化、冲突控制等等。

使用 CVS 进行版本控制需要服务器端和客户端两个软件，在 Windows 操作系统下客户端软件就是 WinCvs，而服务器端软件是 CVSNT。

Chapter 11

CVS Mode

由于 CVS 是在 Unix 下发展起来的, 所以它的使用需要采用命令行的方式，这对于广大使用 Windows 操作系统的用户来说, 是极其不方便的, 因为命令行本来就难以记忆，而每个命令又带有很多参数，不过幸运的是 WinCvs 的出现，它是 Windows 操作系统下一个图形界面的 CVS 客户端工具，用户只需要使用菜单与工具按钮就可以完成同样的命令行操作，就象我们使用其他 Windows 下的软件一样, 大大降低了使用难度，同时它也提供了命令行的工作方式, 用户同样可以使用命令来完成同样的操作。

使用 CVS 的基本流程如下:

1. 首先要让 CVS 管理员给您分配一个用户名和密码, 先使用 WinCvs 登录 (Login) 到 CVS 服务器。
2. 把本地需要 CVS 管理的原始目录导入 (Import) 到 CVS 服务器上去, 使之成为 CVS 服务器上仓库 (Repository) 的一个 Module。
3. 在本地硬盘上创建一个工作目录。
4. 从 CVS 服务器的仓库 (Repository) 导出 (Checkout) 一个 Module 到本地硬盘的工作目录
5. 从 CVS 服务器同步 (Update) 你同事的修改到你本地工作目录。在工作目录上进行工作, 在这个过程中, 把文件的中间版本 (Revision) 提交 (Commit) 给 CVS 服务器。

当我们已经进行过上述流程，以后再使用时就简化成只需要步骤 1) 和步骤 5) 就可以了。也就是说，我们只需要在第一次使用 WinCvs 时需要进行这五步操作，以后再次使用时甚至连第一步都不需要，只是不断的重复第五步操作就可以了。

从 WinCvs 的工作模式可以知道，WinCvs 的工作涉及三个目录：一是原始目录，我们从这里把文件导入到 CVS 进行管理，从此以后这个目录下的文件就不再参与 WinCvs 活动了；二是 CVS 仓库目录，所有的 Module 都存放在这里，它可能是远程 Linux 下由

CVS 服务器管理员创建的，也可能是你自己在本地硬盘创建的，这决定于你工作在哪种模式下；三是您本地硬盘的工作目录，用户在这里对文件进行多次修改和提交。

对于在 Windows 下工作的用户来说，登陆使用的用户名和密码就是我们登陆系统时使用的用户名和密码，也就是说你需要知道服务器的登陆用户名和密码，如果一个开发小组有四个开发者，那么需要在服务器上创建四个账号（就是 Windows 的账号）分配给这四个开发者，每个开发者凭借自己的账号和密码来登陆进 CVS 服务器。一般来说这个操作只需要进行一次，以后再次打开 WinCvs 客户端时就可以直接进入了，因为成功登陆后将在你的目录建立一个.cvspass 文件, 所以以后就不用输入口令了。

Chapter 12

CVS Session

作为一种介绍 CVS 的方式，我们将使用 CVS 进行一次典型的工作会话。首先要明白的是 CVS 把所有的文件集中保存在一个仓库 repository[1] 中，在此我们假定已建立好一个仓库。

假定当前开发的是一个简单的编译器。源文件包括几个 C 文件和一个 Makefile。编译器叫 'tc'（Trivial Compiler），同时仓库已建立，因此有一个叫 'tc' 的模块。

12.1 Getting the source

首先要做的是得到一份 'tc' 源文件的工作副本。为此，使用checkout命令：

```
$ CVS checkout tc
```

这样会创建一个名为 tc 的新目录并把源文件复制到这个目录中。

```
$ cd tc
$ ls
CVS       Makefile  backend.c driver.c  frontend.c parser.c
```

CVS 目录是 CVS 内部使用的。通常，不要修改或删除里面的任何文件。

接下来，对 backend.c 进行修改，几小时后我们对编译器增加了优化遍数。这里，rcs 和 sccs 使用者应注意，不必锁住想编辑的文件，参阅 Multiple developers。

[1] 参阅 Repository。

12.2 Committing your changes

当检查了编译器还是可编译时，可以决定对 backend.c 做新的版本。CVS 会把新文件保存在仓库中并且使用同一仓库的任何人也可以得到它。

```
$ CVS commit backend.c
```

CVS 打开一个编辑器，让我们输入一个日志信息。于是，可以敲入“Added an optimization pass.”，保存这个临时文件，并退出编辑器。

环境变量 $CVSEDITOR 决定运行哪一个编辑器。如果没有设置 $CVSEDITOR 而设置了环境变量 $EDITOR，那么就会启用后者。如果两者都未设置，那么会启用一个默认编辑器，比如在 unix 系统上会是 vi，在 Windows NT/95 系统是 notepad。

做为补充，CVS 还会检测 $VISUAL 环境变量。这取决于是否需要和以后的 CVS 版本是否检测 $EDITOR 或把它忽略。不必多虑，你不设置 $VISUAL 或者将它与 $EDITOR 设成一样。

当 CVS 启动编辑器时，它包含了一个被修改的文件的名单。对于 CVS 客户，这个名单基于文件的修改时间和它最近一次被修改的时间的比较。因此，如果一个文件的修改时间改变了而内容没有，它就好象一个修改过的文件一样。处理这种情况最简单的办法是别管它—如果你继续提交的话，CVS 会检测到它的内容没有改变，从而把它作为未改变的文件来处理。下一次 update 会告知 CVS 此文件没有修改，并会重设时间戳以便此文件不会在以后的编辑会话中出现。

如果想避免打开一个编辑器，可以在命令行使用 ‘-m’ 标记来指定日志信息，如下所示：

```
$ CVS commit -m "Added an optimization pass" backend.c
```

12.3 Cleaning up

在转到其他程序任务之前，如果要删除 tc 的工作副本，一般采用如下方法：

```
$ cd ..
$ rm -r tc
```

但是更好的方法是使用release命令

```
$ cd ..
$ CVS release -d tc
M driver.c
```

```
? tc
You have [1] altered files in this repository.
Are you sure you want to release (and delete) directory `tc': n
** `release' aborted by user choice.
```

release 命令检查当前用户所做的所有更改。如果启用历史纪录功能，它会同时在历史档案文件中加入一条注释。参阅 history file。

当用 release 的 '-d' 参数时，它还将删除当前用户的工作副本。

在上面的例子中，release 命令向输出设备写了几行文字。'? tc' 意思是 CVS 不认识文件 tc。这一点不需要担心：tc 是一个可执行文件，它不应被存储在仓库中。关于如何消除这些警告，参阅 cvsignore。

另外，参阅 release output，可以得到 release 所有可能输出的完整说明。

'M driver.c' 更严重一些。这表明 driver.c 这个文件在检出后已经被更改过了。

release 命令通常在结尾处告诉用户在工作目录有多少源文件的副本已经被更改了，然后在删除或在历史文件中进行注解之前让用户确认。

当 release 要求用户确认时，如果只是安全的演练的情况下，按 n <RET> 就可以了。

12.4 Viewing differences

如果忘记了 driver.c 是否被更改，想查询一下这个文件的更改情况。

```
$ cd tc
$ CVS diff driver.c
```

这个命令diff可以检查 driver.c 的检出版本和工作目录中版本的差异。看到输出后要是记起增加了一个命令行选项使优化可以执行。提交之后，记得要 release 这个模块。

```
$ CVS commit -m "Added an optimization pass" driver.c
Checking in driver.c;
/usr/local/cvsroot/tc/driver.c,v <-- driver.c
new revision: 1.2; previous revision: 1.1
done
$ cd ..
$ CVS release -d tc
? tc
You have [0] altered files in this repository.
Are you sure you want to release (and delete) directory `tc': y
```

Chapter 13

CVS Concept

CVS 是很早的时候在 Unix 下发展起来的，它使用的术语比较特殊，需要先熟悉和理解，这是使用 CVS 的第一步。

- Repository：仓库
  它是 CVS 服务器（可能在远程，也可能在本地）的根目录，我们所有的工作都保存在这个仓库中，包括源代码和这些代码的全部历史。可以把 Repository 想象成一个仓库，仓库中有许多“木桶”，每个“木桶”就是我们的一个让 CVS 管理起来的工程。对于 CVS 来说，这些“木桶”之间是没有什么关联的，删除一个“木桶”不会影响别的“木桶”。我们所想象的木桶，在 CVS 术语中，又叫模块（Module）。
- Module：模块
  就是上面我们所想象的仓库中的“木桶”，里面放的是一个项目的所有文件（包括源代码，文档文件，资源文件等等）。在物理上，Module 是 CVS 服务器根目录下的第一级子目录。
- Import：导入
  我们本地有一个软件项目，里面有许多各种类型的文件，都需要用 CVS 进行版本管理，那么第一步就是把这个软件项目的整个目录结构都 Import 到 CVS 的仓库中去。经过这种导入，CVS 将为用户的项目创建一个新的“木桶”——Module，即模块。
- Checkout：导出
  指将仓库中的一个“木桶”（Module，模块）中的东西导出到本地的工作目录下，然后我们可以在 WinCvs 的管理下，进行工作，修改其中的内容。
- Commit：提交修改
  我们在本地的工作目录下，对工程中的文件进行修改，这些修改需要提交给 CVS

的仓库。这个过程就叫 Commit。可以 Commit 一个文件，也可以 Commit 整个目录。

- Update：同步
  它与 Commit 相对应，是从仓库中的“木桶”（模块）中下载同事修改过的文件（别忘记项目是有许多人共同参与的），如果这个文件在本地也有，就会更新本地的拷贝，如果本地没有，就会把新文件下载到本地。
- Revision：文件版本
  这是 CVS 中一个需要特别注意的概念，它指的是单个文件的版本，而不是整个项目的版本。基本上，单个文件每次的修改，经过 Commit 之后，它的 Revision 都要改变一次，比如从 1.1 到 1.2 到 1.3 等等。特别要注意，单个文件的版本（Revision）与整个工程产品的版本（Version，或者 Release）可以没有任何关系。例如，整个产品现在发行 1.0 版本（Version 1.0）了，但是产品的源代码文件中，有的文件版本（Revision）可能是 1.9，有的是 2.1，等等。这很容易理解，因为为了发行产品 V1.0，我们需要对源代码进行多次修改编译。
- Release：发行版本
  整个产品的版本，例如 VC5.0，VC6.0 等。
- Tag：标签
  在一个开发的特定期，对一个文件或者多个文件给的符号名，一般是有意义的字符串，如“stable”，“release_1_0”等。比如，我们对某个文件的 1.5 版本加上标签：“memory_bug_fixed”，借助这个有意义的标签，我们可以理解 1.5 版本解决了内存 Bug，所以说 Tag 赋予了版本一些文字含义。

以上这些概念是我们在使用 CVS 时经常遇到的，大家将会在实际使用中进一步加深对这些概念的理解。

Chapter 14

CVS Repository

CVS 仓库（repository）存储了用于版本控制的所有文件和目录的副本。

通常，用户决不直接访问仓库里的任何文件。代之以，使用 CVS 命令从仓库取得文件副本放到工作目录中，并对该副本进行工作。当完成了一系列修改后，再把它们提交（commit）到仓库。仓库将保存用户对文件的所有修改情况，包括用户做了什么样的修改和什么时候进行的修改，以及诸如此类的信息。注意，仓库不是工作目录的子目录，反之亦然；它们应该在各自独立的位置。

CVS 有很多方法访问仓库，可以是本地计算机，也可以是隔壁房间里的或是世界另一端的计算机。为了区别访问仓库的方法，仓库的名称可以用（access method）开始。例如，访问方法：`:local:`是访一个仓库目录，这样仓库`:local:/usr/local/cvsroot`的意思就是仓库是在运行 CVS 的计算机上的`/usr/local/cvsroot`中。要获得更多关于访问方法，见 Remote repositories

如果省略了访问方法，并且仓库以‘/’，那么`:local:`就是默认的方法。如果没有以‘/’，那么`:ext:`或者`:server:`就是默认的方法。例如，有一个仓库在`/usr/local/cvsroot`下，可以用`/usr/local/cvsroot`替代`:local:/usr/local/cvsroot`。但如果本地仓库（例如在 Windows NT 下）在c:\src\cvsroot ，那就要指定访问方法`:local:c:/src/cvsroot` 。

仓库分成两个部分，其中 $CVSROOT/CVSROOT 包含关于 CVS 的管理文件，其它目录包含实际用户定义的模块。

14.1 Specifying a repository

有几种方法告诉 CVS 仓库的位置。一种方法是通过命令行显式指明仓库，具体方法是用 `-d`（表示“目录”）选项：

```
CVS -d /usr/local/cvsroot checkout yoyodyne/tc
```

也可以给环境变量 $CVSROOT 设置绝对路径来代表仓库的位置，在本例中 $CVSROOT 被设置成 /usr/local/cvsroot。对于 csh 和 tcsh 用户，在.cshrc 或.tcshrc 文件中加入下面语句即可设置 $CVSROOT 的值：

```
setenv CVSROOT /usr/local/cvsroot
```

sh 与 bash 用户则应在他们的.profile 或者.bashrc 加入下面语句：

```
CVSROOT=/usr/local/cvsroot
export CVSROOT
```

用 -d 指定的仓库将跨越环境变量 $CVSROOT。一旦从仓库里检出了一份工作副本，那么它将记住仓库的位置（信息被记录在工作副本的 CVS/Root 文件里）。

用 -d 选项和 CVS/Root 文件都会跨越环境变量 $CVSROOT。如果 -d 选项和 CVS/Root 文件指定的仓库位置不同的话，则使用前者。当然，为了正确操作它们应该是引用同一仓库的两个方法。

14.2 Repository storage

对大部分使用者来讲，不必太在意 CVS 是如何在仓库中存储数据的。实际上，在过去的时间中存储的格式一直在变化，而且以后也会继续变化。因为在大多数情况下用户是通过 CVS 命令来访问仓库，所以这些变化带来的影响不大。

但是，在某些情况下，了解 CVS 是如何在仓库中存储数据就显得十分必要，例如，你需要追踪 CVS 锁[1]，或者需要处理仓库中的文件权限。

14.2.1 Repository files

仓库中全部目录结构完全对应于工作目录中的目录结构。比如，假设仓库在

```
/usr/local/cvsroot
```

下面是一个可能的目录结构（只显示目录）：

```
/usr
 |
 +--local
 |    |
```

[1]参阅 Concurrency

```
|     +--cvsroot
|     |     |
|     |     +--CVSROOT
              |     (administrative files)
              |
              +--gnu
              |     |
              |     +--diff
              |     |     (source code to GNU diff)
              |     |
              |     +--rcs
              |     |     (source code to rcs)
              |     |
              |     +--cvs
              |           (source code to CVS)
              |
              +--yoyodyne
                     |
                     +--tc
                     |     |
                     |     +--man
                     |     |
                     |     +--testing
                     |
                     +--(other Yoyodyne software)
```

与目录一起的是在版本控制下的每个文件的历史文件（history files）。这些文件的名称是在对应在工作目录中的文件名称后面加上‘,v’。下面是仓库中 yoyodyne/tc 目录的可能情况：

```
$CVSROOT
     |
     +--yoyodyne
     |   |
     |   +--tc
```

```
|   |   |
            +--Makefile,v
            +--backend.c,v
            +--driver.c,v
            +--frontend.c,v
            +--parser.c,v
            +--man
            |     |
            |     +--tc.1,v
            |
            +--testing
                   |
                   +--testpgm.t,v
                   +--test2.t,v
```

历史文件中包括有足够的信息来再创建文件的任何一个修订版，另外还记录有所有提交信息的日志，其中包括提交者的用户名。这些历史文件就是以前的 RCS files，因为第一个以这种存储文件格式方式来进行版本控制的系统是 rcs。关于文件格式的详细描述请看 rcs 发布的 man 中关于 rcsfile(5) 的信息，或者是 CVS 源代码中的 doc/RCSFILES 文件。这种文件格式已经应用的十分广泛——除了 CVS 和 rcs 之外还有很多其它的版本控制系统至少可以导入这种格式的历史文件。

CVS 中用的文件格式与标准的 rcs 文件格式有一些差别。最大的差异在于魔术分支[2]。另外 CVS 中有效的标签名称是 rcs 所能接受的一个子集[3]。

14.2.2 File permissions

所有的 ‘,v’ 文件全部被设置成只读，并且，用户不能改变那些文件的权限。在仓库里的目录对于在每个目录里都有修改文件权限的人来说才是可写的。这通常意味着用户必须建立由一个能在项目里编辑文件的人组成的 unix 组[4]，并且设置仓库以便确保确实是这个组拥有这一目录[5]。

这也意味着用户仅能控制每个目录下文件的访问。

[2]关于更多信息见 Magic branch numbers

[3]参见 Tags

[4]参见 group(5)。

[5]在有些系统上，用户还需设置仓库的 set-group-ID-on-execution 标识位[6]，这样才能使新建的文件和目录正确得到父目录的 group-ID 而不是当前进程的。

注意，那些用户必须也由写访问来检出文件，因为 CVS 需要建立锁定文件（lock file）[7]。对于一些只读访问的目录，用户也可以利用 CVSROOT/config 的 LockDir 来将锁定文件放到仓库以外的地方[8]。

还要注意，用户必须有 CVSROOT/val-tags 文件的写访问权限。CVS 用它来纪录标签的标记名是有效的（有时当标签被使用，以及被创建时就被更新）。

每个 rcs 文件将被最后一个提交它的人拥有。其实这并不重要，它的实质在于谁拥有这目录。

CVS 试图为每一个添加在这个目录树里的新目录设置合理的文件权限，但是当用户需要一个新目录与它的父目录有不同的权限的时候，必须手动设置它。假如用户设置那些在仓库里由 CVS 建立目录或者（和）文件所使用的控制文件权限的 CVSUMASK 环境变量。CVSUMASK 不影响在工作目录中的文件权限；这样的文件有此种权限，典型地对于新近创建的文件，除有时候 CVS 以只读创建它们之外[9]。

注意，使用客户机/服务器的 CVS[10]，没有好的方法来设置 CVSUMASK；在客户机上的设置是无效的。如果正与 rsh 连接着，那正如操作系统文档中说明的那样，可以把 CVSUMASK 设置在.bashrc 或.cshrc 里。这个操作也许会在 CVS 将来的版本里发生变化，不要依靠在客户机上的没有效果的 CVSUMASK 设置。

使用 pserver，在 cvsroot 目录，以及在这个目录树里那些目录上，通常需要更加严格的设置权限[11]。

有些操作系统具有这种特性：允许让某一特定程序运行时完成程序访问者不能完成的任务。比如，unix 中的设置用户 id（setuid）或设置组 id（setgid）特性，或者 VMS 的安装映像特性。CVS 就不是为使用这类特性而编写的，因此，试图以这种方式安装 CVS，CVS 将只能提供针对偶然性失误的保护；任何想要获得这一方法的人都可以轻易做到，而且，根据设置的不同，还可以获得比 CVS 更多的东西。或者，也许希望考虑 pserver，就可能提供虚假的安全感或者打开一个比我们试图要弥补的安全漏洞更大的漏洞方面而言，它也具有相同的特性，因此，如果考虑这一选项的话，一定要仔细阅读有关 pserver 安全方面的文档[12]。

[7]参阅 Concurrency
[8]参阅 config。
[9]参见 Setting a watch，Global options 或 CVSREAD，Environment variables
[10]参阅 Remote repositories。
[11]参见 Password authentication security。
[12]参见 Password authentication security。

14.2.3 Windows permissions

针对 Windwos 操作系统（如 Windows95、WindowsNT，还有这个系列以后的操作系统，有可能还针对 OS/2 操作系统，但不确定）的一些文件权限问题。

如果正在使用一个在区域网中的 CVS 系统，仓库在一个由 Samba 提供服务的网络文件系统上，就可能会碰到一些文件权限问题。在 Samba 配置中允许`WRITE=YES`据说可以解决这个问题[13]。

14.2.4 Attic

可能会注意到 CVS 时常会把 rcs 文件存放在 Attic 目录中。例如，如果 cvsroot 是 /usr/local/cvsroot，且我们讨论的文件 backend.c 在 yoyodyne/tc 目录中，于是这个文件通常应该在下面这个位置上

```
/usr/local/cvsroot/yoyodyne/tc/backend.c,v
```

但是，如果它在 Attic 目录中，就应该是下面这个位置。

```
/usr/local/cvsroot/yoyodyne/tc/Attic/backend.c,v
```

从用户的观点来看，一个文件是否在 Attic 中是没有关系的；CVS 会自动跟踪文件的，如果需要它会到 Attic 目录中寻找。不过用户需要知道的是，当且仅当在树干中的主修订版（head revision）处于 dead 状态[14]，CVS 才会把 RCS 文件存放在 Attic 目录中。例如，用户在一个分支中加入一个文件，那么就会存在一个处于 dead 状态的树干修订版，而分支中的修订本处于非 dead 状态。

14.2.5 CVS in repository

在仓库中的每一个目录中都存在一个 CVS 目录，其中存放一些象文件属性这样的信息（在 CVS 目录中有一个 CVS/fileattr 文件）。在将来会有附加的文件放入到这个目录中，实现应默默忽略附加的文件。这个特性只会出现在 CVS 1.7 版本之后[15]。

fileattr 文件有一系列如下格式的条目（entries）组成（其中 '{' 和 '}' 意思是括号内文字可以没有或重复多次）：

ent-type filename <tab> *attrname* = *attrval* ; *attrname* = *attrval* <linefeed>

- *ent-type* 为 'F' 表示文件，该条目设置文件的属性。

[13]需要申明的是：我们没有研究过允许这个选项会带来什么样的影响，同时也不知道有没有其他的方法可以解决这个问题。如果你发现有解决的方法，请及时告诉我们。

[14]dead 状态指在这个修订版中文件被删除或没有被加入进来。

[15]详细请看 Watches Compatibility。

- *ent-type* 为 'D'，并且 filename 为空，给新添加的文件指定默认属性。

其余 *ent-type-type* 留给以后扩展使用。CVS 1.9 和更老的版本在任何写文件属性的时候会把它们删除。CVS 1.10 和后续版本将其保留。

注意，每行的先后次序不重要；任何程序可以按它的习惯重新排列。

当前无法作到文件名中带 TAB 或换行，*attrname* 里的 '='，*attrval* 里的 ';'，等等[16]。

习惯上在 *attrname* 名前加 '_' 表示在 CVS 中有特殊含义；其余为用户自定义类型的属性（或将是，一旦实现开始支持用户定义的属性）。

内置属性：

- _watched
  出现意思是该文件已被监视 (watched)，并且应以只读检出。
- _watchers
  监视此文件的用户。值为 *watcher* > *type* [, *watcher* > *type*]，其中 *watcher* 是用户名，*type* 是空或由 '+' 隔开的 edit，unedit，commit（什么都没有就是空, 无需"none" 或"all" 关键字）。
- _editors
  编辑此文件的用户。值为 *editor* > *val* [, *editor* > *val*]，其中 *editor* 是用户名，*val* 为 *time*+*hostname*+*pathname*，其中 *time* 是 CVS edit 执行的时间（或类似的），*hostname* 和 *pathname* 是工作目录。

例子：

```
Ffile1 _watched=;_watchers=joe>edit,mary>commit
Ffile2 _watched=;_editors=sue>8 Jan 1975+workstn1+/home/sue/CVS
D _watched=
```

意味着文件 file1 应按只读检出。另外 joe 监视此文件的 edit 而 mary 监视此文件 commit。文件 file2 应按只读检出；sue 在 8 Jan 1975 开始编辑它，目录是 /home/sue/CVS，机器为 workstn1。其余文件都是按只读检出。作为演示，'D'、'Ffile1'、'Ffile2' 后面有一个空格，其实该用 TAB 字符而无空格。

14.2.6 Locks

关于 CVS 锁用于用户可见特性方面的介绍，请参看 Concurrency。接下来的内容是针对那些想制作工具软件的朋友，这些软件可以访问 CVS 仓库，但不会与其它访问同一库的工具冲突。如果发现被这里的一些名词，如"读锁（read lock）"、"写锁（write lock）"

[16] 注意：有些实现还无法处理任何字段中的 NULL 字符，但实现已注意此事。

和“死锁（deadlock)”，弄得混淆不清，建议可以去阅读一些关于操作系统和数据库的资料。

在仓库中任何一个以 #CVS.rfl. 开头的文件是有个读锁。以 #CVS.pfl 开头的文件就是个可升级读锁。以 #CVS.wfl 开头的文件是个写锁。老版本的 CVS（在 cvs1.5 之前的版本）也会创建一些以 #CVS.tfl 开头的文件，在这里我们不讨论它们。目录 #CVS.lock 的作用是一个主锁 (master lock)。它的意思是在创建其它种类锁之前用户必须获得这个主锁。

为了得到一个读锁，首先创建 #CVS.lock 目录。这样的一个操作必须是一个原子操作 (在大多数操作系统下面创建目录都是可以的)。如果失败，是因为这个目录已经存在，这个操作会等待一会再重新执行。在得到 #CVS.lock 之后，会创建一个文件，以 #CVS.rfl. 开头，后面为用户选择的信息（例如主机名和进程识别号)。接着删除 #CVS.lock 目录以释放主锁。接下来开始读仓库。读操作完成后删除 #CVS.rfl. 文件以释放读锁。

可升级读锁是在其他并行论述中没有提及的概念。它们允许两个 (或多个) 在第一次 (读) 时对文件锁设为读的文件认证，然后在最后认证必要的时候将读锁升级为写锁，并仍然假设自此首次读时没有改变。例如，CVS 使用可升级读锁来防止提交和打标签认证时读处理的相互干扰。它只可以在写认证时锁住一个单独的目录。

为了获得一个可升级读锁，首先创建 #CVS.lock 目录，就像普通读锁一样。然后检查那里是否有以 #CVS.pfl 开头的文件。如果存在，删除 #CVS.lock 主目录，等一会儿（CVS 在锁之间会等 30 秒）再试。如果不存在其他的可升级锁，就创建一个 #CVS.pfl 名字以开头的文件，后面跟着用户自己定义的信息（比如，CVS 是使用创建锁的 CVS 服务器进程所在的主机名和进程 id 号)。如果版本低于 1.12.4 的 CVS 直接访问用户的仓库（没有通过 1.12.4 或更新版本的 CVS 服务器)，用户应该创建一个读锁，因为旧版本的 CVS 会忽略可升级读锁创建自己的写锁。接着删除 #CVS.lock 主目录以使其它进程可以获得读锁。

为了得到一个写锁，同读锁一样首先创建 #CVS.lock 目录。接下来检查是否不存在以 #CVS.rfl. 和 #CVS.pfl 开头的文件，这些是不属于试图获取写锁的进程。如果存在就删除 #CVS.lock 目录，等待一会再重试。如果没有其他进程的读或可升级读锁，就创建一个文件，以 #CVS.wfl 开头，后面为你选择的信息（同样，CVS 使用主机名和服务器进程识别号)。删除你自己的 #CVS.pfl 文件。继续保持住 #CVS.lock 锁。开始进行写仓库操作。当操作结束，首先删除 #CVS.wfl 文件，接着删除 #CVS.lock 目录。需要说明的是不像 #CVS.rfl 文件，#CVS.wfl 文件仅仅只有提示作用；不能锁住操作，而这个功能是由 #CVS.lock 来完成的。

注意，每一个锁（读锁或者写锁）仅仅只锁住仓库中的单一目录，包括 Attic 目录和 CVS 目录，但是不包括在版本控制中代表其它目录的子目录。如果要锁住整个目录树，用户必须锁住每一个目录（如果用户在锁任何一个目录时出错，为了避免死锁就必须在

重试之前释放整个目录树)。

还需注意的是 CVS 希望用写锁来控制任何 foo,v 文件的访问。rcs 有一个计划，让,foo, 文件具有锁的作用，但 CVS 并没有这样实现，而且建议使用 CVS 的写锁[17]。

14.2.7 CVSROOT storage

$CVSROOT/CVSROOT 目录包含有各种管理文件。在某些方面这个目录就象仓库中任何其它目录一样；它包含了 rcs 文件，后缀名为 ‘,v’，许多 CVS 命令以同样的方式对它进行操作。当然，还是有点小小的差异。

对每一个管理文件，还有 rcs 文件，还有一个检出的副本。例如，有一个 rcs 文件 loginfo,v 和一个包含有对 loginfo,v 的最新修改的文件 loginfo。当你检入一个管理文件时，CVS 就会显示

```
CVS commit: Rebuilding administrative file database
```

并在 $CVSROOT/CVSROOT 目录中更新已检出的副本。如果它没这样做，那肯定出了问题[18]。为了使用户自己的文件加进用这种方式更新文件中，可以把它们加入到管理文件的 checkoutlist 中[19]。

在默认的情况下，modules 文件按上面所说的方式运作。如果 modules 文件很大，把它作为一个纯文本文件存储可能会使得查找模块变慢[20]。因此，对 CVS 源代码进行合适的编辑，人们可以把模块文件保存在一个使用 ndbm 界面如 Berkeley db 或者 GDBM 的数据库中。如果使用这个选项，那么模块数据库将被存在文件 modules.db，modules.pag 和/或 modules.dir 中[21]。

14.3 Working directory storage

当我们讨论 cvs 的内部结构时，它就变得越来越可见了，我们可能讨论 cvs 把什么东西放在工作目录的 CVS 目录中。对仓库而言，cvs 处理该信息而通常用户可以通过 cvs 命令来访问这些信息。但是在某些情况下，更常用的是让外部程序可以浏览这些信息，例如 jCVS 图形界面和 emacs 的 VC 软件包。这些程序如果要和其它使用这些文件的程序协同工作的话，应该遵循本节讨论的推荐标准，相关的这些程序包括提到的程序的未来版本和 cvs 的命令行客户端。

[17]关于此更多的讨论和基本原理的信息请参看 CVS 源码中的 rcs_internal_lockfile 注释。
[18]参阅 BUGS。
[19]参阅 checkoutlist。
[20]不知道现在人们是否还象当初 CVS 有此特点时那样关心这个问题；也没有看过有关的评测。
[21]关于各种管理文件的意义，参考 Administrative files。

CVS 目录包含以下若干个文件。为了保证未来版本的可扩展性，读取此目录的程序应忽略在此目录中但没有在此归档的文件。

文件以系统习惯的文本方式保存。这意味着文件在不同存储文本文件习惯的系统之间工作目录是不能移植的。这是故意的，理论上 cvs 管理的文件在这种系统之间也无法移植。

- Root
  该文件包含当前 cvs 根目录[22]。
- Repository
  该文件包含当前目录对应的仓库里的目录。该路径可以是绝对路径也可以是相对路径；从 1.3 版本开始，cvs 可以读取这两种格式的路径。相对路径名相对于根目录并且容易解析，但是绝对路径更通用，实现应该支持这两种格式中的任意一种。例如，执行以下命令以后

  ```
  cvs -d :local:/usr/local/cvsroot checkout yoyodyne/tc
  ```

  Root 应该包含

  ```
  :local:/usr/local/cvsroot
  ```

  Repository 同时包含

  ```
  /usr/local/cvsroot/yoyodyne/tc
  ```

  或者

  ```
  yoyodyne/tc
  ```

  如果特定的工作目录不与仓库的目录相一致，Repository 应当包含 CVSROOT/Emptydir。
- Entries
  该文件列出了工作目录中的文件和子目录。每一行的第一个字符代表该行的类别。为了保证未来版本的可扩展性，如果一行的第一个字符不可识别，读取文件的程序应默认忽略该行。如果第一个字符是 '/'，则格式如下：

  ```
  /name/revision/timestamp[+conflict]/options/tagdate
  ```

  其中：
  1. '[' 和 ']' 不是格式的一部分，仅仅是代表 '+' 和 conflict 项是可选的。
  2. name 是目录中文件的名字。

[22]参考 Specifying a repository。

3. revision 是正在编辑的文件派生的修订版本号，'0' 代表新添加的文件，'-'revision 代表删除的文件。
4. timestamp 为时间戳，表示 cvs 创建文件的时间；如果时间戳和文件修改的时间不一致，意味着文件已经被修改过。时间戳以 ISO 标准的 C 函数 asctime() 的格式存储 (例如，'Sun Apr 7 01:29:26 1996')。用户也可以不以这种式写入字符串来表示文件已经被修改过，比如，合并结果（'Result of merge'）。但这不是特殊情况；判断一个文件是否被修改，一个程序只要读取改文件的时间戳 timestamp 并进行字符串比较就可以了。如有冲突，文件做了冲突标记之后会在文件的修改时间上设一个 conflict[23]。
5. conflict 表示是否存在版本冲突。如果 conflict 和文件实际的修改时间相同表示用户还没有解决版本冲突问题。
6. options 包含可选项 (例如对二进制文件可以使用 '-kb')。
7. tagdate 含有 'T' 后面跟标签名，或 'D' 表示日期，后面跟是 sticky 标签或日期。

注意：如果 timestamp 包含一对以空格分隔的时间戳而不是一个，应该按照早于 cvs1.5 的版本处理（这里没有相关文档）。

CVS/Entries（本地或全球）里时间戳（timestamp）的时区（timezone）应该与操作系统使用的时区相同。例如，Unix 上文件的时间戳为 UT，CVS/Entries 里的时间戳也应如此。vms 上时间戳为本地时间，这样在 vms 上的 cvs 应该使用本地时间。这样文件就不会因为时区（例如，夏时制）改变而显示被修改。

如果 Entries 的某一行的第一个字符是 'D'，则该行代表一个子目录。如果一行只有字符 'D' 则表示生成 Entries 的程序没有记录子目录 (因此，如果有这样的一行并且没有别的以字符 'D' 开头的行，可以知道没有子目录)。否则，行是这样的：

```
D/name/filler1/filler2/filler3/filler4
```

其中：name 是子目录的名称，为了保证未来版本的可扩展性，所有的 filler 域都应该被忽略。修改 Entries 文件的程序应保留这些域。

Entries 文件中行的次序无关紧要。

- Entries.Log

  这个文件不纪录任何与 Entries 无关的信息，但是它能使你更新一些信息而不必重写整个 Entries 文件，并且对信息的安全提供保护，甚至写 Entries 和 Entries.Log 期间程序突然发生中断仍能保证数据的安全。读 Entries 的程序应该也检查 Entries.Log。如果后者存在，它将按照 Entries.Log 所纪录的信息应用到 Entries。在使用这些内容后，推荐的操作是重写 Entries 并且删除 Entries.Log。Entries.Log 的格式为一个单字

[23]参阅 Conflicts example。

符的命令加一个空格，接着与 Entries 行的格式一样。单字符命令有下面几种：'A' 表示 entry 已经被加入，'R' 表示 entry 已经被删除，其它的字符表示该行应被忽略不计 (为以后扩展)。如果 entries.log 中行的第二个字符不是空格，那么，它是由一个老版本的 cvs 系统生成的 (本文档不做介绍)。
如果程序选择只是读而不写则可以忽略 Entries.Log。

- Entries.Backup
  这是个临时文件。推荐的作法是将一个新的 Entries 写到 Entries.Backup，然后将它更名为 Entries（如可能，自动实现)。
- Entries.Static
  此文件唯一实质性的事情就是它的存在与否。如果它存在，那么，意味着只有部分的目录被得到，并且 cvs 将不在该目录中创建附加文件。可以使用 update 带 '-d' 选项的命令来删除它，该命令将得到附加的文件并且删除 Entries.Static 文件。
- Tag
  这个文件包含每一目录的粘性标签或时间[24]。一个分支标记的第一个字符是 'T'，对于一个非分支标记则是 'N', 或对于一个日期则是 'D'，或别的字符意味着为了将来扩充而文件应该被默默忽略。这个字符后接的是标签或日期。注意，每一目录新增加的粘性标签或日期被用来向新增加的文件施加作用这种事情；它们可能不与在单个的文件上的粘标签或日期一样。
- Notify
  这个文件存储还没有发送给服务器的通知（如 edit 或 unedit 等)。这里还没有归档它的格式。
- Notify.tmp
  该文件是针对 Notify 的，正如 Entries.Backup 针对于 Entries 一样。即在更新 Notify 文件时，先将新内容写入 Notify.tmp 中，然后将（自动）该文件更名为 Notify。
- Base
  如果监视器处在使用状态，那么一个 edit 命令在 Base 目录下保存原来的副本。这样即使系统不能与服务器进行通信，也允许 unedit 命令操作。
- Baserev
  这个文件列出了对 Base 目录下文件的每次修订。格式为：

  ```
  Bname/rev/expansion
  ```

  其中 expansion 将被忽略，允许将来扩展。
- Baserev.tmp

[24]关于粘性标签或日期的一般信息，见 Sticky tags。

该文件是针对 Baserev 的，正如 Entries.Backup 针对于 Entries 一样。即，为了写 Baserev 文件，先将新内容写入 Baserev.tmp 中，然后将（如可能则自动）其更名为 Baserev。

- Template
  这个文件包含由 rcsinfo 文件指定的模版[25]。它只用于客户端；非客户/服务器模式的 cvs 直接参考 rcsinfo。

14.4 Intro administrative files

目录 $CVSROOT/CVSROOT 包含了一些管理文件 administrative files。完整的说明参阅 Administrative files. 你可以使用 cvs 而不需要这些文件中的任何一个，但是当 modules 文件得到适当的设置后，一些命令将会工作得更好。

这些文件中最重要的文件是 modules。它定义了在仓库中的所有模块。下面是一个 modules 文件的例子。

```
CVSROOT CVSROOT
modules       CVSROOT modules
cvs                   gnu/cvs
rcs                   gnu/rcs
diff                  gnu/diff
tc                    yoyodyne/tc
```

modules 是定向排列的行。在它简单的格式中每一行包含了模块的名字、空格和模块所在的目录。目录是相对于 $CVSROOT 的路径。上例中的最后四行就是这样的例子。

定义称为 'modules' 模块的行使用不在这里解释的特性[26]。

可以用编辑其它任何模块的方式来编辑管理文件。使用 '`cvs checkout CVSROOT`' 以取得一个工作副本，编辑它，并且用平常的方式提交修改。

提交有错误的管理文件是可能的。用户可以经常修正这些错误并且保存到一个新的修订版中，但是有时管理文件中一些特殊的严重错误会使得它不能保存到一个新的修订版中。

14.5 Multiple repositories

在某些情况下，有多个仓库是个好办法，比如，你有两个开发小组在不同项目中工作而且没有共享代码。为获得多个仓库你所要做的就是指定适当的仓库，可用 CVSROOT

[25]参阅 rcsinfo。
[26]想了解所有可用的特性的解释，参阅 modules

的环境变量，cvs 加上 '-d' 选项，或者（一旦你获得一个检出的工作目录）简单地让 cvs 使用工作路径中的仓库信息[27]。

拥有多个库的最大优点是它们可以存在于不同的服务器上。在 cvs1.10 版本上，单一命令不能从同的库重置目录。用 cvs 的开发版本，你可以从不同的服务器上把代码检出到你的工作目录中。cvs 会重复并处理连接上有关的主机以执行请求的命令的所有细节。下面是一个创建此类工作目录的例子：

```
cvs -d server1:/cvs co dir1
cd dir1
cvs -d server2:/root co sdir
cvs update
```

`cvs co` 命令创建工作目录，然后用`cvs update` 命令连上 server2，更新 dir1/sdir 子目录，对 server1 也做相应的更新。

14.6 Creating a repository

要建立一个 cvs 仓库，首先挑选一台想存储源文件修订历史的机器和磁盘，不需要很好 cpu 和内存，大部分机器都会满足这个要求[28]。

如何估计所需空间的大小，如果要从另一个系统导入 rcs 文件，文件的大小接近库初始时的大小，或如果没有任何版本的历史，单凭经验来说，可以考虑三倍于 cvs 仓库的代码大小的空间（最终可能发现不合适，但这不是暂时的）。在一台开发者工作的机器上，用户希望分配给每个开发者一个工作目录所需的磁盘空间（任一完整的子目录或部分目录，根据每一个开发者的使用需要）。

所有需要使用 cvs 的机器都该在服务器上或本地模式下访问（直接或通过网络文件系统）仓库；客户机通过 cvs 协议就不需要任何别的认证。不能经过 cvs 读改只有读访问的库；cvs 可以根据需要建立加锁文件[29]。

为了创建一个库，运行 cvs init 命令。在以通常办法指定的 cvs 根目录下建立一个空白的库[30]。例如，

```
cvs -d /usr/local/cvsroot init
```

[27]参阅 Specifying a repository。
[28]细节参阅 Server requirements。
[29]参阅 Concurrency。
[30]参阅 Repository。

`cvs init` 并不更改库中已有的文件，所以在一个已初始化过的库中运行`cvs init`没有任何的损害。

`cvs init` 将启用历史记录；如果不希望这样，在运行 cvs init 后删除历史文件[31]。

14.7 Backing up

和一般文件相比，仓库里的文件也没有什么特别的有魔力；在很大程度上它们可以和一般文件一样备份。然而，还是需要考虑以下问题。

首先要注意，用户不能在备份的时候使用 cvs，或在备份进行的时候使用备份程序锁定 cvs。要停止使用 cvs，可阻止其它机器登录进仓库，即关掉 cvs 服务器，或采用类似的机制来停止 cvs 服务。详细作法取决于操作系统和 cvs 的设置。如果要锁定 cvs，就需要创建 #cvs.rfl 来锁定每一个仓库的目录[32]。总之，如果没有进行以上的操作就进行备份的话，操作的结果是不可预知的。当从备份恢复的时候，仓库可以处于不一致的状态，但是这也可以进行手动修复而并不特别困难。

当从备份中恢复仓库的时候，假设仓库里的某些修改是在备份后进行的，没有受到故障影响的工作目录可能指向一个仓库里并不存在的版本。在这种目录里运行 cvs 将典型地引发一个错误消息。要把使那些修改取回放进仓库可以按以下操作进行：

- 创建一个新的工作目录。
- 将出错前工作目录的所有文件复制到新的工作目录（当然不要复制 CVS 目录的内容）。
- 在新的工作目录中，使用 cvs update 和 cvs diff 等命令弄清楚修改了什么，然后弄好后把这些修改的内容提交进仓库。

14.8 Moving a repository

正如备份仓库中的文件和备份任何其他文件非常相似一样，从一个地方将仓库移动到另一个地方也和移动其它类型的文件的方式非常相似。

所需要考虑的主要的事就是工作目录指着仓库。处理一个移动过的仓库的最简单的办法是取得一个移动后的新的工作目录。当然，会想确定旧的工作目录在移动前已经被提交过，或者有一些其它的途径确定没有遗漏任何修改。如果确实想重新使用已存在的工作目录，那有可能需要手工修改 CVS/Repository 中的文件。可以参考 Working directory

[31]参阅 history file。

[32]参考 Concurrency 找更多关于 cvs 锁定的细节。

storage，取得关于 CVS/Repository 和 CVS/Root 中文件的信息，但是除非是想自找麻烦，这样做可能并不值得。

14.9 Remote repositories

源代码的工作副本也可以存放在与仓库不同的机器上。这种方式 cvs 以 client/server 的模式运作。先在一台可以安装工作目录的机器上运行 cvs，它被称为客户机（client），然后告诉它将连到一台安装有仓库的机器，它被称为服务器（server）。通常，使用远程仓库除了仓库名称格式不同，其它方面与本地机没什么区别：

[:*method*:][[*user*][:*password*]@]*hostname*[:[*port*]]/path/to/repository 在检出期间不推荐在仓库名中带有口令这种方式，因为它会使 cvs 把纯文本口令副本保存到每个工作目录中。先 cvs login 代替[33]。

具体安装细节要看连到服务器的方式。

14.9.1 Server requirements

对什么样的机器适合作为一个服务器的快速回答是需求适度——具有 32M 或甚至更少内存的服务器可以处理相当大的具有合理活动数量的源码树。

当然，要精确定位的话就比较复杂。通过评估已知耗用内存比较大的方面就足以确定服务器的内存需求。主要有两个方面在此归档；相比之下其它方面占用内存较少[34]。

第一个方面大内存消耗是当使用 cvs 服务器时进行大量的检出。对于每一个要求服务的客户机，服务器都提供两个进程。一般在子进程中应当少占用内存。而在父进程中的内存消耗，尤其是在网络连接比较慢的时候，内存需求就要比单一目录的尺寸大，大 2M 或更多。

每个 cvs 服务器的尺寸乘以用户期待同时有活动的服务器数应该能给出一个对于服务器内存需求的概念。大多数情况下，父进程消耗的内存多半可以是交换空间而不是物理内存。

大内存消耗的第二个方面是 diff，当检入大批量文件时。还有对二进制文件操作时。经验规则要求：在校验文件时提供最大文件尺寸的 10 倍大小的内存，尽管 5 倍大小的内存就够了。例如，要检入一个 10M 的文件，这时就需要 100M 的内存空间以在机器上做检入（C/S 机制或其它机制的 cvs 服务器）。这可以是交换空间而不是物理内存。因为，对服务器内存的占用是暂时的，没有必要专门同时为这样多个检入提供内存。

[33]参阅 Password authentication client。

[34]当然，如果你认为不像 BUGS 中描述的那样，请告诉我们，以便于我们可以更新这个文档。

对于客户机的资源消耗甚至更适度—有足够能力运行操作系统的任何机器应该不会有大问题。

要想了解 CVS 服务器对硬盘的要求，参看 Creating a repository。

14.9.2 The connection method

在最简单的形式下，仓库字符串[35]中 method 部分可以是 'ext', 'fork', 'gserver', 'kserver', 'local', 'pserver' 之一，在某些平台上是 'server'。

如果没有指定 method，并且仓库名以 '/' 开头，那么默认为 local。如果没有指定 method，但仓库名不以 '/' 开头，则根据平台默认为 ext 或 server[36]。

ext, fork, gserver 和 pserver 连接方式都能使用连接选项，可在 method 字符串中指定，比如：

:*method*[;*option*=*arg*...]:*other_connection_data*

虽然 cvs 对于某些 arg 的大小写敏感，但对 method 或 option 的大小写不敏感。可以使用的方式选项如下：

- proxy=hostname
- proxyport=port

  这两种方式选项可用于通过 HTTP 隧道的 web 代理。hostname 应为代理服务器名，port 为端口。port 默认使用 8080。

  注意：HTTP 代理服务器与 cvs 写代理服务器不同[37]。

  例如，连接通过 8000 端口的 web 代理，应该使用：

  :pserver;proxy=www.myproxy.net;proxyport=8000:*pserver_connection_string*

  注意：在上面的例子中，pserver_connection_string 是需要连接和认证的 CVS 服务器，如在下面章节注明的密码认证 gserver 和 kserver。上面例子中只是用来演示仓库名中的 method 部分。

  这些选项首次出现在 cvs 版本 1.12.7 中，对 gserver 和 pserver 有效。
- CVS_RSH=path

  该选项可用于 ext 方式中指明 cvs 客户端寻找远端 shell 的路径，用作连接 cvs 服务器和以更高的优先权覆盖 $CVS_RSH 环境变量中指定的路径[38]。例如，通过本地 /path/to/ssh/command 命令连接到 cvs 服务器，可以通过 CVS_RSH 选项在以下 path 中指定：

[35]参阅 Remote repositories。
[36]'ext' 与 'server' 方式说明见 Connecting via rsh。
[37]参阅 Write proxies 了解 cvs 上的写代理。
[38]参阅 Connecting via rsh。

```
:ext;CVS_RSH=/path/to/ssh/command:ext_connection_string
```

该选项首次出现在 cvs 版本 1.12.11 中，只对 ext 连接方式有效。

- CVS_SERVER=path
  该选项可用于 ext 和 fork 方式，指明 cvs 服务器上的 cvs 可执行文件的路径，并以更高的优先权覆盖 $CVS_SERVER 环境变量中指定的路径[39]。例如，选择 /path/to/cvs/command 可执行文件作为 cvs 服务器上的的 cvs 应用程序，可以通过 CVS_SERVER 选项在以下 path 中指定：

  ```
  :ext;CVS_SERVER=/path/to/cvs/command:ext_connection_string
  ```

  或者，选择 'cvs-1.12.11' 作为可执行文件名，假设它在 cvs 服务器 $PATH 上存在：

  ```
  :ext;CVS_SERVER=cvs-1.12.11:ext_connection_string
  ```

  该选项首次出现在 cvs 版本 1.12.11 中，对 ext 和 fork 连接方式有效。
- Redirect=boolean-state
  Redirect 选项用于 cvs 客户端允许 cvs 服务器可以将其重定向到其他的 cvs 服务器，通常用在写代理设置的写请求。
  在 CVSROOT/config 文件中可以使用的布尔值[40]都可以在 boolean-state 中指定。例如，'on', 'off', 'true' 和 'false' 对 boolean-state 都是有效值。Redirect 选项的默认值为 'on'。
  该选项对没有第二服务器的 cvs 服务器没有任何作用[41]。
  该选项首次出现在 cvs 版本 1.12.11 中，只对 ext 连接方式有效。
  作为更进一步的例子，组合 CVS_RSH 和 CVS_SERVER 选项，连接方式可以如下：

  ```
  :ext;CVS_RSH=/path/to/ssh/command;CVS_SERVER=/path/to/cvs/command:
  ```

  它的意思是，毋须 CVS_SERVER 或 CVS_RSH 环境变量设置正确[42]。

14.9.3 Connecting via rsh

cvs 使用 'rsh' 协议执行这些操作，因此远程用户主机需要建立.rhosts 来控制本地用户的访问。注意，cvs 为此目的使用的程序可以用 –with-rsh 标志进行配置来指定。

例如，假设当前用户是本地机上 'toe.example.com' 上的用户 'mozart'，服务器是 'faun.example.org'。首先，在服务器上 'bach' 主目录下的.rhosts 的文件中加入下面的内容：

[39]参阅 Connecting via rsh。
[40]参阅 config。
[41]了解更多的写代理与第二服务器，请参阅 Write proxies。
[42]参阅 Connecting via rsh 了解更多环境变量的信息。

```
toe.example.com mozart
```

再用以下命令从本地机测试 'rsh'

```
rsh -l bach faun.example.org 'echo $PATH'
```

接着应该确保 rsh 可以找到服务器。作到确保，上面例子中 rsh 打印的路径应包括服务器上 cvs 程序所在的目录。你需要在.bashrc，.cshrc 中设置路径而不是在.login 或者.profile 中。同时，你需要在客户机上设置环境变量 CVS_SERVER 指向你希望访问的服务器，例如：/usr/local/bin/cvs-1.6。对于 ext 和 fork 方式，你可以在 CVSROOT 中指定 CVS_SERVER 作为选项，这样你可以为不同的根使用不同的服务器[43]。

不需要编辑 inetd.conf 或者启动一个 cvs 守护进程。

有两种方法可以在 rsh 中使用 CVSROOT。:server: 指定一个内部 rsh 客户，这种方法仅仅被某些 cvs 端口支持。:ext: 指定一个外部的 rsh 程序。按照默认，这是 rsh(除非使用 –with-rsh 标志去配置)，但是可以通过设置 CVS_RSH 环境变量用别的程序来访问远程服务器（例如，在 HP-UX 9 上的 remsh，因为在 HP-UX 9 上 rsh 有一些不同)。这个程序必须是一个可以在客户机和服务器之间来回传送数据而并不修改数据的程序，比如，Windows NT 的 rsh 就不适合作为这样的程序，因为它默认地是在 CRLF 和 LF 之间传送数据的。OS/2 的 cvs 通过 '-b' 给 rsh 来实现这种传递，但是由于这会对标准 rsh 程序以外的程序引起潜在的问题，这种方法在未来可望被改变。如果设置 CVS_RSH 为 SSH 或者使用其它替代程序，本节中其余部分关于.rhosts 的例子可能会不适用；建议参考替代程序的文档。

可以选择在 CVSROOT 中指定 CVS_RSH 选项，这样可以为不同的根使用不同的值。例如，在 ext 方式下允许一些根使用 CVS_RSH=remsh，其他的使用 CVS_RSH=ssh[44]。

继续我们的例子，假如希望访问服务器 faun.example.org 上的仓库 /usr/local/cvsroot/ 中的模块 foo，可以使用以下命令：

```
cvs -d :ext:bach@faun.example.org:/usr/local/cvsroot checkout foo
```

注：如果用户在本地机和远程主机上的用户名相同，bach@ 可以被忽略。

14.9.4 Password authenticated

cvs 客户端也可以通过一种口令协议连接服务器。这在 rsh 不可用的时候（例如，服务器在防火墙后面）和无法使用网络安全协议 Kerberos 的时候尤其有用。

要使用这种方法，必须在服务器和客户端进行一些调整。

[43]参阅 Remote repositories 了解更多信息。
[44]参阅 Remote repositories 了解更多信息。

Password authentication server

首先，用户可能需要 $CVSROOT 和 $CVSROOT/CVSROOT 上设定权限[45]。

在服务器端，需要编辑文件 /etc/inetd.conf 让 inetd 它在正确的端口上收到一个连接的时候运行 cvs pserver 命令。按照默认，端口号是 2401；当然，如果客户端用 CVS_AUTH_PORT 重新编译了就会有一些不同。也可以在 CVSROOT 变量 (参阅 Remote repositories) 指定或用 CVS_CLIENT_PORT 环境变量[46]跨越。

如果用户的 inetd 在 /etc/inetd.conf 中允许原始的端口号定义，那么下面的命令就足够了（所有命令在 inetd.conf 中都是单行）[47]：

```
2401 stream tcp nowait root /usr/local/bin/cvs
cvs -f --allow-root=/usr/cvsroot pserver
```

'–allow-root' 选项指定一个允许的 cvsroot 目录。试图使用一个另外的 cvsroot 目录的客户将被拒绝连接。如果用户希望设置多个允许的 cvsroot 目录，重复设置这一选项就行了[48]。

如果用户希望 inetd 使用符号服务名称来代替原始端口号，那么把它放进 /etc/services 里：

```
cvspserver    2401/tcp
```

并且在 inetd.conf 中使用 cvspserver 代替 2401。

如果用户的系统用 xinetd 替代 inetd，步骤有些不同。

创建一个叫 /etc/xinetd.d/cvspserver 的文件，包含下列内容：

```
service cvspserver
{
   port        = 2401
   socket_type = stream
   protocol    = tcp
   wait        = no
   user        = root
   passenv     = PATH
   server      = /usr/local/bin/cvs
```

[45]参见 Password authentication security。

[46]参阅 Environment variables。

[47]注：用户也可以使用 '-T' 选项来指定一个临时目录。

[48]不幸的是，许多版本的 inetd 对参数的数量和/或命令长度小有限制。通常解决方法是在 inetd 中运行一个 shell 脚本，然后在脚本中调用 cvs 所需的参数。

```
    server_args = -f --allow-root=/usr/cvsroot pserver
}
```

注：如果已在 /etc/services 中定义 cvspserver，可以省略 port 这一行。

做了上面的修改之后，需要重新启动 inetd，或者采取其它可以使它重新读取初始化文件手段[49]。

因为客户端以明文存储和传输口令（几乎是——参见 Password authentication security），通常使用了一个独立的 cvsCVS 口令文件，这样人们访问仓库的时候不会危及定期口令的安全。这个文件是 $CVSROOT/CVSROOT/passwd[50]。除了只用较少的域（cvs 用户名称，可有可无的口令和服务器的用户名称）以外，它的格式和 Unix 上的 /etc/passwd 相似，采用分号分隔。这里是个有五个表项的例子 passwd 文件[51]：

```
anonymous:
bach:ULtgRLXo7NRxs
spwang:1sOp854gDF3DY
melissa:tGX1fS8sun6rY:pubcvs
qproj:XR4EZcEs0szik:pubcvs
```

例子中第一行允许任何 cvs 客户机使用 anonymous 来访问，口令可以用任意字符还可以用空口令[52]。

如果它们提供各自的明文口令第二、三行允许对 bach 和 spwang 进行访问。

如果它提供正确的口令第四行允许对 melissa 访问，但它的 cvs 操作将实际运行在服务器侧的 pubcvs 用户名下。虽然系统上不须有 melissa 用户名，但必须使用 pubcvs。

第五行显示系统用户标识可以共享：作为 qproj 成功认证的任何客户实际将按 pubcvs 来运行，和 melissa 所做的一样。用这种方法用户可以在他们的仓库中为每个项目创建一单个共享的系统用户，并且在 $CVSROOT/CVSROOT/passwd 文件中为每个开发者设置一行。每行的 cvs 用户名应该不同，而系统用户名应该相同。用不同的 cvs 用户名目的是在 cvs 将在其名字下记录其动作：当 melissa 提交了项目中的一个修改，检入被记录在项目历史中的 melissa 名下，而不是 pubcvs 下。而共享一个系统用户的原因是可以安排仓库中相关区域的权限，只有那个帐户有写权限。

如果系统用户域存在的时候，全部口令鉴定 cvs 命令按该用户运行；如果没有指定系统用户，cvs 简单地取 cvs 用户名作为系统用户名并且按此户名运行命令。无论哪种情

[49]如果设置这有麻烦，参阅 Connection。

[50]参阅 Intro administrative files。

[51]口令使用 Unix 标准的 crypt() 函数加密，因此它可以直接从常规 Unix /etc/passwd 文件粘贴过来。

[52]这是一个站点允许匿名进行只读访问的典型用法；“只读（read-only）”部分信息的做法，参看 Read-only access。

况，如果系统上没有这样的用户，于是 cvs 操作会失败（不管客户机是否提供了正确的口令）。

口令和系统用户字段都可以省略（如果系统用户字段省略，那么与口令分隔的冒号也省略）。例如，这将是有效的 $CVSROOT/CVSROOT/passwd 文件：

```
anonymous::pubcvs
fish:rKa5jzULzmhOo:kfogel
sussman:1sOp854gDF3DY
```

当口令字段为空，客户的认证可以用包括空字符串在内的任何口令。但 cvs 用户名后的分隔冒号必需存在，即使口令为空。

cvs 还可借助于系统认证。认证口令的时候，服务器先检查 $CVSROOT/CVSROOT/-passwd 文件中的用户。如果找到用户，就按上面说的方法使用该项进行认证。如果没有找到，或 cvs passwd 文件不存在，服务器将采用操作系统的用户认证机制[53]。

除非系统中有 PAM(Pluggable Authentication Modules)，或者 cvs 服务器端编译的时候配置[54]，否则默认的回调（callback）是查找 /etc/passwd 中的口令。在此情况下，将磋商以 PAM 代替。这意味着可以配置 cvs 使用 PAM 提供的各种口令认证方式（可能是 UNIX password，NIS，LDAP，或其他），这取决于 PAM 的全局配置文件（通常在 /etc/pam.conf 或可能是 /etc/pam.d/cvs）[55]。

注意，PAM 是 cvs 中使用的实验特性，鼓励反馈信息。如果用户使用了 PAM 支持，请发邮件到 cvs 的邮件列表（info-cvs@gnu.org 或 bug-cvs@gnu.org）。

警告：采用 PAM 可以给系统管理员更多的 cvs 认证灵活性，但并不比别的方法有更高的安全性。详见下。

CVS 在 PAM 配置文件里需要“auth”，“account”和“session”模块。因此典型的 PAM 配置是在文件 /etc/pam.conf 里模拟标准的 cvs 系统 /etc/passwd 认证：

```
cvs       auth       required   pam_unix.so
cvs       account    required   pam_unix.so
cvs       session    required   pam_unix.so
```

对等的 /etc/pam.d/cvs 文件里包括

```
auth          required   pam_unix.so
account       required   pam_unix.so
session       required   pam_unix.so
```

[53]这种机制可以通过设置 cvs config 文件中的 SystemAuth=no 来禁止，参阅 config。
[54]使用./configure –enable-pam - 参考 INSTALL 中的说明。
[55]详细配置 PAM 的方法请参考 PAM 的文档。

一些系统要求给出模块的完整路径，因此 pam_unix.so（Linux）改为类似 /usr/lib/security/$ISA/pam_unix.so.1（Sun Solaris）[56]。

PAM 中指出的服务名“cvs”是默认配置中的服务名，也可以在编译前使用./configure –with-hardcoded-pam-service-name=<pam-service-name> 重新设置。cvs 也可以被设置成调用其他的名字./configure –without-hardcoded-pam-service-name，但这个特性只有当前用户有权控制 cvs 用什么名字时才有效。

要知道，在使用系统认证时会有安全漏洞：cvs 操作时是使用系统中用户的正规注册口令，而该口令网络传输中是采用明码方式[57]，这点对使用 PAM 认证会有更多的问题，因为系统口令的源头是某些像 LDAP 的中心服务，而 LDAP 它还被用于认证别的服务。

另一方面，PAM 使得定期地修改口令非常容易。如果对全部活动给用户一个口令系统的选项，用户经常定期改变他们的口令。

在非 PAM 的配置中，口令是保存在 CVSROOT/passwd 文件里面，因为只有管理员用户（或有等同的权限）才有此文件的修改访问权限，所以想用定期改变口令会很困难。可以采用设计的网页程序或 set-uid 程序来更新文件，或者是提交请求让管理员进行手动修改来更新。第一种情况下，必须记住去定期更新各自的口令可能是很困难的。第二种情况下，手工的本性将是只有绝对必要时才会去修改口令。

注意，PAM 管理员应避免在 cvs 证明/授权（authentication/authorization）中采用配置一次性口令 OTP。如果用 OTP，管理员最好考虑其他的 Client/Server 访问方式[58]。

现在，cvs passwd 文件里加入口令的唯一方法是从别处粘贴而来。也许有一天，会有一条命令`cvs passwd`。

不像其他在 $CVSROOT/CVSROOT 目录下的文件，可以在不是通过 cvs 来编辑 passwd 的地方进行编辑。这是因为使 passwd 文件检出为人们的工作副本可能的安全风险[59]。

Password authentication client

为了通过口令验证服务器远程在仓库上运行 cvs 命令，需要指定 pserver 协议，可选的用户名，仓库主机，可选的端口号，仓库路径。例如：

```
cvs -d :pserver:faun.example.org:/usr/local/cvsroot checkout someproj
```

或

[56]参考 cvs 源码发布子目录 contrib/pam 中更多的配置例子。
[57]参见 Password authentication security。
[58]参阅 Remote repositories 获得其他方法的列表。
[59]如果的确需要将 $CVSROOT/CVSROOT 中的 passwd 检出，参看 checkoutlist。

```
CVSROOT=:pserver:bach@faun.example.org:2401/usr/local/cvsroot
cvs checkout someproj
```

然而，除非用户连接到公共访问仓库（比如，不需要口令的用户名），首先要提供用口令或登录 log in。登录用仓库校验你的口令并将它保存起来。这是通过 login 命令完成，如果在 $CVSROOT 没有提供口令，它将交互提示输入：

```
cvs -d :pserver:bach@faun.example.org:/usr/local/cvsroot login
CVS password:
```

或

```
cvs -d :pserver:bach:p4ss30rd@faun.example.org:/usr/local/cvsroot login
```

输入口令后，cvs 用服务器进行验证。如果认证成功，username、host、repository 和 password 的组合被永久记录下来，这样以后与仓库的事务就不再要求执行 cvs login 命令。(如果认证失败，cvs 申诉口令错误，并不记录任何信息。)

按照默认，记录存储在 $HOME/.cvspass 文件中。它的格式是人可读的，一定程度上是人可编辑的，但注意，口令不是以明文存储的—通常经过了编码来保护它们避免被无心发现的危险（例如，被碰巧看到这个文件的系统管理员或其他人无心地看到）。

用户可以用设置 CVS_PASSFILE 环境变量改变来改变该文件的默认位置。如果使用这个变量，需要确保在 cvs login 之前将其设置。如在运行 cvs login 后将其设置，以后的 cvs 命令将无法找到口令传输给服务器。

一旦登录之后，所有的 cvs 使用该远程仓库和用户名的命令都使用保存的口令认证。这样，如

```
cvs -d :pserver:bach@faun.example.org:/usr/local/cvsroot checkout foo
```

将正常工作（除非服务器端更改了口令，那样需要重新运行 cvs login）。

注意，如果在仓库规格说明中没注明 ':pserver:'，cvs 会采用 rsh 来连接服务器[60]。

当然，如果用户已经有检出一个工作副本并在里面运行过 cvs 命令，以后将不再显式指明仓库，因为 cvs 可以从工作副本的 CVS 子目录推断仓库。

使用`cvs logout` 命令可以将远程仓库的口令从 CVS_PASSFILE 中删除。

Password authentication security

口令存储在客户端是以明文的一般编码方式存储的，也是以同样的编码方式传输。这种编码仅仅可以避免口令被无心发现的危险（例如，被碰巧看到这个文件的系统管理

[60]参阅 Connecting via rsh。

员看到)，它甚至不能阻止一个想获得口令的初级攻击者。

分开的 cvs 口令文件[61]允许用户使用与注册不同的口令获取仓库的访问。在另一方面，一旦用户拥有对仓库的非只读权限，那么该用户就可以在服务器上通过一系列手段运行程序。因而，仓库的访问比系统相同权限要宽得多。可以通过修改 cvs 来防止这一情况，但是在本文写作以前，还没有人这样做过。

注意，因为 $CVSROOT/CVSROOT 目录包含 passwd 和其它与安全审核相关的文件，因此必须象严格控制 /etc 目录的许可权限一样严格地控制这个目录的许可权限。同样适用于 $CVSROOT 目录本身和它在文件目录树结构中的任何上级目录。任何对这些目录具有写访问的用户将有能力成为系统上的任何用户。注意，如果没有使用 pserver 的话，这些权限典型地比应该使用的更加牢固。

概括地讲，任何获得口令的用户就获得仓库的访问权限 (同时也获得了一般系统的访问权限)。对任何可以嗅探网络数据包或者能够读取被保护的文件（例如，只读文件）的人该口令都是有用的[62]。

14.9.5 GSSAPI authenticated

GSSAPI 是一套类似 Kerberos 5 的通用网络安全系统接口。如果用户拥有一套 GSSAPI 库，就可以通过 tcp 连接直接建立 cvs 连接，由 GSSAPI 进行安全鉴别。

要做到这一点，需要在 GSSAPI 的支持下重新编译 cvs；当配置 cvs 的时候，它会检测使用 Kerberos 5 的 GSSAPI 库是否存在。也可以使用 `--with-gssapi` 标志来配置。

这样的连接就使用 GSSAPI 来进行安全鉴别了，但是按照默认并不鉴别消息流。用户必须使用`-a` 全局选项来要求消息流进行安全鉴别。

按照默认不加密传输数据。加密支持必须同时编译进客户端和服务器端；使用`--enable-encrypt` 配置选项来把它开启。最后必须使用`-x` 全局选项来要求加密。

在服务器端处理 GSSAPI 连接的服务器和口令鉴别服务器应该是同一台服务器[63]，如果已经使用了诸如基于 Kerberos 的 GSSAPI 机制来提供功能强大的安全鉴别功能了，可能希望停用通过明文口令鉴别的功能。要做到这一点，创建一个空的 CVSROOT/passwd 口令文件，并且在配置文件里设置 SystemAuth=no[64]。

GSSAPI 服务器使用 cvs/hostname 的主名称，其中主机名称 hostname 是服务器主机的规范名称。必须按要求通过 GSSAPI 机制把它设置起来。

要使用 GSSAPI 进行连接，要在命令中使用 ':gserver:' 方法。例如，

[61]参阅 Password authentication server。
[62]如果希望获得更高的安全性能，使用 Kerberos 网络安全系统。
[63]参阅 Password authentication server。
[64]参阅 config。

```
cvs -d :gserver:faun.example.org:/usr/local/cvsroot checkout foo
```

14.9.6 Kerberos authenticated

就像 Connecting via rsh 中描述的，使用 Kerberos 最容易的方式是使用基于 Kerberos 的 rsh。使用 rsh 最大的缺点是所有的数据都需要通过另外的程序传递，因此这种方式会慢一些。如果安装了 Kerberos 就可以通过直接的 tcp 连接建立连接，使用 Kerberos 进行安全鉴别。

本节的讨论都是基于 Kerberos 网络安全系统第四版的。在前一节中已经讨论过基于 GSSAPI 通用网络安全接口的 Kerberos 第五版支持。

要作到这一点，cvs 需要在 Kerberos 的支持下编译；在配置 cvs 的时候，它会自动检测 Kerberos 是否存在，也可以使用`--with-krb4` 标记来进行配置。

按照默认不加密传输数据。加密支持必须同时加入到客户端和服务器端，然后使用`--enable-encryption` 配置选项来把它开启。最后必须使用 -x 全局选项来要求加密。

按照默认，CVS 客户端会尝试对 1999 端口进行连接。

当用户希望使用 cvs 的时候，以通常方式获取一张入场券（一般 kinit），它必须是一张允许你登录进服务器机器这的入场券，然后就准备启程：

```
cvs -d :kserver:faun.example.org:/usr/local/cvsroot checkout foo
```

cvs 的早期版本在通过 rsh 连接的时候会不履行；但现在的版本将不会这样了。

14.9.7 Connecting via fork

这种访问方法允许你用远程登录协议访问本机磁盘上的仓库。也就是说除了远程 cvs 方式的一些 bug 外其他的跟:local: 没什么两样。

取决于爱好你可以在日常操作中选择:local: 或:fork:。当然:fork: 最初是为了测试或调试 cvs 和远程访问协议的。我们特别回避了网络相关的连接、超时、和在别的远程访问方法中固有的认证问题，但还是创建了使用远程协议的连接。

要用 fork 方法连接，使用 ':fork:' 和本机仓库路径名。例如：

```
cvs -d :fork:/usr/local/cvsroot checkout foo
```

与:ext: 一样，服务器默认调用 'cvs'，也可在 CVS_SERVER 环境变量中指定。

14.9.8 Write proxies

cvs 可以通过多台 cvs 服务器配置成分布式。对于单独的 primary server，它的实现借助于一个或多个 write proxies，或者是 secondary servers。

当 cvs 客户端访问第二服务器时，只请求读操作，那么第二服务器会处理所有的请求。如果客户端发送写请求，那么，如果客户端支持重定向，第二服务器要求客户端将写请求重定向到第一服务器。否则第二服务器作为可以处理写请求的第一服务器的一个传输代理。

这样，第一服务器可以配置多个写代理的第二服务器，有效地将读负载分配到第二服务器，限制第一服务器的写负载并将变更传递到第二服务器。

第一服务器不会自动将变更推向第二服务器。它必须通过 loginfo, postadmin, posttag, & postwatch 脚本配置[65]，就像下面这样：

```
ALL       rsync -gopr -essh ./ secondary:/cvsroot/%p &
```

用户也许希望在运行上面例子中 'rsync' 写第二服务器和读第一服务器的时候给目录加锁，但如何配置超出了本文档范畴。

写代理的另外一个优点是，在连接第一服务器比较慢或者它定期断开时，用户仍然可以从第二服务器执行快速的读操作。只在写操作时才需要连接到第一服务器。

为了配置写代理，第一服务器必须在 CVSROOT/config 里指定 'PrimaryServer' 选项[66]。为了使传输代理模式有效，所有的第二服务器必须运行相同版本的 cvs 服务器，或者至少给客户端提供与第一服务器相同的支持需求列表。它对于转发不是必须的。

一旦设置了第一服务器，第二服务器可以通过下面进行配置：

1. 复制主仓库到新位置。
2. 在第一服务器上建立 loginfo, postadmin, posttag 和 postwatch 文件用于传播修改到新的第二服务器。
3. 像配置其他 CVS 服务器 (参阅 Remote repositories) 那样配置远程访问到第二服务器。
4. 如果两台服务器上 cvs 仓库的路径不同，并且不希望客户端处理 '`Redirect`' (CVS 1.12.9 及更早的客户端不能处理 '`Redirect`')，务必确保传递 `--allow-root=secondary-cvsroot` 给所有的调用第二服务器。

•

再次请注意，通过代理写，要求指定 `--allow-root=secondary-cvsroot` 到所有的调用第二服务器，不只是 'pserver' 调用。这也许需要为用户的服务器包装一个执行脚本。

[65]参阅 Trigger Scripts。
[66]参阅 config。

14.10 Read-only access

使用口令验证服务器授予用户以只读存储仓库的访问是可能的 (参阅 Password authenticated)。（其他访问方法对只读用户没有显式支持，因为这些方法都假定（用户）必须注册访问仓库机器，并且因而用户只能做在局部文件许可权限范围内的任何事情。）

一个有只读访问的用户只能做不改变仓库的 cvs 操作，除了某些“管理”文件（例如加锁文件和历史文件）。结合用户别名使用这个特征是相当令人满意的 (参阅 Password authentication server)。

不象 cvs 的早先版本，只读用户只能读取存储仓库，而不能在服务器上执行程序或另外获得难以预料的访问级别。准确一点说，已知的漏洞已经被堵上了。由于这些是新特征，没有做过一个全面的安全检查，你应该使用被许可使用的安全级别。

为一位用户指定只读访问有两种方式：“包含”和“排除”。

“包含”意思是在文件 $CVSROOT/CVSROOT/readers 内列出特定的使用者的名字，这个文件是一个以简单换行符分隔的用户列表。这里有一个简单的 readers 文件[67]：

```
melissa
splotnik
jrandom
```

“排除”意指显示列出有写访问权限的每一位用户——如果文件

```
$CVSROOT/CVSROOT/writers
```

存在，那么只有列入该文件内的用户有写访问权限，而没有列入其内的其他人只有读取访问权限（当然，即使只读权限的用户仍需被列在 cvs passwd 文件之内）。writers 文件与 readers 文件有相同的格式。

注意：如果 cvs passwd 文件将 CVS 用户映射到系统用户[68]，要确信拒绝或允许使用 cvs 用户名而不是系统用户名来进行只读访问。也就是说，readers 和 writers 文件包含有 cvs 用户名，这些用户名可以与系统用户名一致或不一致。

这里有一个关于服务器在决定是否允许只读或只写访问行为的完整描述：

如果 readers 文件存在，并且这个用户名被列在内，那么它会获得只读访问权限。如果 writers 存在，并且用户没有被列入其中，那么它也只能获得只读访问权限（这是正确的，即使 readers 存在但用户没有被列入其中）。否则，它将获取完全读写访问权限。

[67]不要忘记在最后一个使用者之后的换行符。

[68]参阅 Password authentication server。

当然，如果两个文件同时记录了同一个用户，这样会发生冲突。这种情况可以通过较为保守的方式解决，多给仓库保护总比少给保护好：此用户会获得只读访问权限。

14.11 Server temporary directory

运行期间，cvs 服务器会创建临时目录。这些临时目录被称为

```
cvs-servpid
```

其中 pid 指的是服务器的进程标识号。它们位于 '`-t`' 全局选项[69]指定的目录中，tmpdir 环境变量[70]，或如不行就使用 /tmp。

大多数情况下，当服务器完成时，不论它正常或是非正常结束，都要删除这个临时目录。然而，有一些情况下，服务器不或不能删除临时目录，比如：

- 如果服务器由于一个内部错误而导致退出的话，服务器将保留这个临时目录以供将来查错
- 如果服务器的进程被“杀掉”，那么它没有办法清除该目录（如，UNIX 系统的 'kill -KILL' 命令）。
- 如果系统没有依照正常退出次序被关闭的时候，系统在正常退出时会通知服务器清除临时目录。

万一出现以上的情况，要必须人工删除这些 cvs-servpid 目录。只要从 pid 进程号查出没有服务器运行时，人工删除操作就是安全的。

[69]参阅 Global options。
[70]参阅 Environment variables。

Chapter 15

Starting a new project

因为更改文件名并且把它们移动到另一个目录中不是经常发生的，因此用户在开始一个新项目时要做的第一件事是考虑项目的文件组织。更改文件名或移动文件并非不可能，但增加了理解上潜在的费解，并且 cvs 在更改名字的目录上特别的敏感[1]。

下一步做的事取决于手中的情况。

15.1 Setting up the files

第一步是在仓库中创建文件。这可以使用多种不同的方法来完成。

15.1.1 From files

本方法对那里已经存在文件的老项目是有用的。

当开始使用 cvs 时，用户可能已经有几个项目可以置于 cvs 控制下了。在这种情况下，最容易的方法就是使用：导入 `import` 命令。通过一个例子是最容易解释如何使用它的。

假定现在有一些想放到 cvs 中的文件在 wdir 中，并且想让它们放在仓库中的如下目录：$CVSROOT/yoyodyne/rdir，可以这样做：

```
$ cd wdir
$ cvs import -m "Imported sources" yoyodyne/rdir yoyo start
```

如果没有使用 '`-m`' 标志记录一个日志信息，cvs 将启动一个编辑器并且提示输入信息。'yoyo' 字符串是销售商标签（vendor tag），而 '`start`' 是发行标签（release tag）。它们

[1]参阅 Moving files。

没有什么特别的意义，仅仅是因为 cvs 的需要才放在这里[2]。

现在可以检查一下它是否有效了，然后可以删除你原来的源目录。

```
$ cd ..
$ cvs checkout yoyodyne/rdir  # Explanation below
$ diff -r wdir yoyodyne/rdir
$ rm -r wdir
```

删除原来的源目录是个好想法，这样避免偶然编辑这个 wdir 目录，而不在 cvs 控制之中。当然，删除之前作个备份也是明智之举。

`checkout` 命令能使用模块名作为参数（正如前面所有例子）或是一个相对于 $CVSROOT 的路径，如同上面的例子。

检查 cvs 设置的 $CVSROOT 目录权限情况是否合适是一个好主意，并使它们属于某一个特定的组[3]。

如果想导入的一些文件是二进制代码，可以使用一些特殊的方法表明这些文件是否是二进制文件[4]。

15.1.2 From other version control systems

从其它版本控制系统创建文件。

如果用户有一个其它版本控制系统维护的项目，例如 rcs，也许希望把这些文件从那个项目放到 cvs 中，并且要保留这些文件的历史。

- 来自 RCS

  如果已使用 rcs，找到 rcs 文件—通常是一个名叫 foo.c.c 的文件会有 RCS/foo.c,v 的 rcsRCS 文件[5]。如果文件目录不存在，那在 cvs 中创建相应的目录。然后把这些文件复制到 cvs 的仓库目录中（在仓库中的名字必须是带 ‘,v’ 的源文件；这些文件直接放在 cvs 中的这个目录下，而非 RCS 子目录中）。这是一个为数不多的直接访问 cvs 仓库的情况，而没使用 cvs 命令。然后就可以把它们在新的目录下检出（checkout）了。

  当用户把 rcs 文件移进 cvs 中时，rcs 文件应在未被锁定的状态，否则移动操作时 cvs 将会出现一些问题。
- 从其它版本控制工具

[2]参阅 Tracking sources，以得到更多的这方面信息。
[3]参阅 File permissions。
[4]参阅 Wrappers。
[5]但它有可能在其它地方；细节请查阅 rcs 的文档。

许多版本控制工具都可以导出标准格式的 rcs 文件。如果当前的版本控制工具可以做到这一点，导出 rcs 文件，然后按照上面的例子做就可以了。
如果不能做，那么必须要写一个脚本文件来，用命令行界面 i 对别的系统每次检出一个版本然后把它放到 cvs 中去。下面提到的 sccs2rcs 脚本就是一个很好的例子。

- 来自 SCCS
  有一个 sccs2rcs 的脚本文件可以做把 sccs 的文件转换成 rcs 文件，这个文件放在 cvs 发行目录的 contrib 目录中[6]。
- 来自 PVCS
  在 cvs 的源码发行目录 contrib 中有一个叫 pvcs_to_rcs 的脚本可以把 pvcs 转换到 rcs 文件。必须在一台同时有 pvcs 和 rcs 的机器上运行它，并且，正如其它在 contrib 目录中的其它脚本一样不被支持[7]

15.1.3 From scratch

从无到有建立一个目录树。
建立一个新的项目，最容易的方法是建立一个空的目录树，如下所示：

```
$ mkdir tc
$ mkdir tc/man
$ mkdir tc/testing
```

在这之后，可以用 import 在仓库创建对应的（空）目录结构：

```
$ cd tc
$ cvs import -m "Created directory structure" yoyodyne/dir yoyo start
```

这会使 yoyodyne/dir 作为一个目录添加到 $CVSROOT。
然后，如果出现则用 add 命令把文件（或目录）添加到仓库中[8]。

15.2 Defining the module

下一步是在 modules 文件中定义模块。这不是严格需要的，但模块能容易地把相关的目录和文件关联起来。

在简单情况下，这些步骤足以定义一个模块。

[6]注意：必须在一台同时安装了 rcs 和 sccs 的机器上运行它，并且，正如其它在 contrib 目录中的其它脚本一样不被支持（不同的用户的情形也许不同）。
[7]不同的用户的情形也许不同，参看脚本中的注释以得到更多细节。
[8]需要检查 cvs 设置的 $CVSROOT 中目录的权限是否合理。

1. 得到 modules 文件的工作副本。
   $cvscheckoutCVSROOT/modules$ cd CVSROOT
2. 编辑这个文件并插入定义模块的行[9]。可以使用下面的行定义模块 'tc'：

```
tc  yoyodyne/tc
```

3. 提交对模块文件的更改。

```
$ cvs commit -m "Added the tc module." modules
```

4. 发布（release）模块文件。

```
$ cd ..
$ cvs release -d CVSROOT
```

[9]参阅 Intro administrative files 和 modules，有它的详细描述。

Chapter 16

Revisions

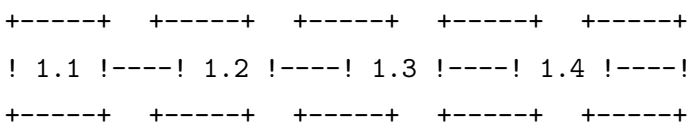
对于大多数 cvs 用户来说，不需要考虑修订号；他们只要知道 cvs 已经自动地加上了类似 1.1，1.2 之类的修订号就可以了。但是，还是有些用户想进一步了解 cvs 是如何控制版本号的。

如果用户想跟踪许多文件的一系列版本号，例如一个特别的发布版本，他使用了一个特殊的标签（tag）作为修订号，这个符号标记的功能和每个文件的数值修订号的功能是相同的。

16.1 Revision numbers

修订号的意义

每个文件的版本都有一唯一的 revision number。修订号的形式一般是这样的：‘1.1’，‘1.2’，‘1.3.2.2’ 甚至是 ‘1.3.2.2.4.5’。一个修订号总有偶数个用句号分隔的十进制数。按照默认，文件第一个修订号是 1.1。每个新的修订号的最右边的数会比它的上一个修订号的最右边的数大 1。下图显示了一些修订号，较新的版本在右边。



还有一种可能就是修订号包含了不止一个句点，如 ‘1.3.2.2’。这种修订号表示修订是在分支[1]上进行的[2]。

[1]参阅 Branching and merging。

[2]对于这样的修订号的详细解释见 Branches and revisions。

16.2 Versions revisions releases

一个文件可以有几个版本 (version)，就像上面描述的。同样，一个软件产品也可以有几个版本。一个软件产品常给以比如 '4.1.1' 这样的一个版本号。

在本文档中版本第一个含义就是“修订（revisions)”，版本的第二个含义就是“发行(releases)”。为避免混淆，本文档中几乎不用“version（版本)”这个词。

16.3 Assigning revisions

分配版本号

按照默认，cvs 分配数值形式的版本号，在分配的时候，使第一个数字保持相同，顺序增加第二个数字。例如，1.1、1.2、1.3 等等。

增加一个新文件的时候，第二个数字将总是为“1”，而第一个数字将等于该目录中任何文件版本号第一个值的最大值。例如，当前目录包含版本号为 1.7、3.1 和 4.12 的文件，则增加的文件的版本号应为 4.1。（当使用客户/服务器 cvs 时, 仅考虑实际上发送到服务器的文件。)

通常没有理由必要关心版本号—最好把它们看作 cvs 维护的内部数值，而标签提供了一种更好地分辨产品版本的方式[3]。然而，如果希望设置数值型的版本号，可以在命令 cvs commit 中使用 '`-r`' 选项。'`-r`' 选项同时包含 '`-f`' 选项的功能，即使文件没有被修改也会被提交。

例如，要把用户所有的文件（包括那些没有被修改的文件）带进 3.0 版本，可以调用以下命令：

```
$ cvs commit -r 3.0
```

注意，使用 '`-r`' 选项时指定的版本号必须大于已有的版本号。也就是说，如果 3.0 版存在，就不能使用 '`cvs commit -r 1.3`' 提交。如果需要并行地维护几个不同的版本，则需要使用分支来进行管理[4]。

16.4 Tags

符号修订号

版本号生存自己的一生。它们与软件产品的发行版本号所做的全然没有任何关系。根据你使用 cvs 的方式，在软件的两个发行版之间，版本号可能被 cvs 修改多次。作为一

[3]参阅 Tags

[4]参阅 Branching and merging。

个例子，由 rcs5.6 管理的源文件可能有如下版本号：

```
ci.c          5.21
co.c          5.9
ident.c       5.3
rcs.c         5.12
rcsbase.h     5.11
rcsdiff.c     5.10
rcsedit.c     5.11
rcsfcmp.c     5.9
rcsgen.c      5.10
rcslex.c      5.11
rcsmap.c      5.2
rcsutil.c     5.10
```

可以使用 tag 命令来给一个文件的某个版本分配一个符号名。在状态命令 status 中可以使用 '-v' 标志来查看一个文件的代表版本号的所有标签。标签名称必须以大写或者小写字母开始，可以包括大小写字母、数字、'-' 和 '_'。两个标签名 BASE 和 HEAD 是保留为 cvs 使用。预料在将来可能采用特殊的方式命名 cvs 的保留字来避免冲突，例如，以 '.' 开始而不是采用象 BASE 和 HEAD 一类的名称。

基于程序和发行版本号的数值版本号的信息，可能在命名标签的时候选择一些命名惯例。例如，有人可能采用程序的名称，在后面直接加上 '-' 而不是 '.'，这样 cvs1.9 的标签可以被命名为 cvs1-9。如果选择一致命名约定，于是不必经常猜测标签名称是 cvs-1-9 还是 cvs1_9 或者是别的什么。甚至用户也许还希望把自己的命名约定实施到 taginfo 文件中去[5]。

下面的例子说明了怎样给一个文件添加标签。命令必须在模块的工作目录中发出。也就是说，你应该在 backend.c 文件所在的目录中发出该命令。

```
$ cvs tag rel-0-4 backend.c
T backend.c
$ cvs status -v backend.c
===================================================================
File: backend.c      Status: Up-to-date
```

[5]参阅 taginfo。

```
Version:          1.4    Tue Dec 1 14:39:01 1992
RCS Version:      1.4    /u/cvsroot/yoyodyne/tc/backend.c,v
Sticky Tag:       (none)
Sticky Date:      (none)
Sticky Options:   (none)

Existing Tags:
    rel-0-4                  (revision: 1.4)
```

要了解完整的 cvs tag 语法以及各种选项的用法，参看 Invoking CVS。

很少对单个孤立文件添加标签。一种更常见的用法是在产品开发周期中的各个里程碑任务完成后对一个模块的所有文件添加标签，比如在发行版完成的时候。

```
$ cvs tag rel-1-0 .
cvs tag: Tagging .
T Makefile
T backend.c
T driver.c
T frontend.c
T parser.c
```

当把一个目录作为 cvs 的一个参数的时候，该命令不仅对该目录下的所有文件执行操作，而且也会递归地对该目录下的所有子目录中的文件执行操作[6]。

在 checkout 命令中使用 '`-r`' 标志可以检出一个模块某个版本。下面的命令可以很容易地取出模块 'tc' 1.0 发行版的所有源文件：

```
$ cvs checkout -r rel-1-0 tc
```

比如，如果有人宣称在那个版本里有个 bug，但用户在当前工作的副本中是找不到那个 bug ，这往往是很有用的。

用户也可以按时间检出一个模块[7]。当对那些命令指定 '`-r`' 或 '`-D`' 的选项，则需要留意粘性标签（sticky tags）[8]。

当用户用同一个标签标记多个文件的时候，可以想象标签以文件名为横轴，以版本号为纵轴绘制了一个曲线图（或者也可以想象成在一个由文件名和版本号组成的矩阵里面绘制的曲线)。以下面的 5 个文件为例：

[6]参阅 Recursive behavior。
[7]参阅 checkout options。
[8]参看 Sticky tags。

```
file1  file2  file3  file4  file5

1.1    1.1    1.1    1.1 /--1.1*    <-*- TAG
1.2*-  1.2    1.2   -1.2*-
1.3 \- 1.3*-  1.3  / 1.3
1.4        \  1.4 /  1.4
            \-1.5*-  1.5
              1.6
```

在过去的某个时候带 * 的版本号已被标记上标签。可以把标签想象成一条经过所有被标记的文件的曲线。当抓住线就得到所有标签标记的版本了。也可以通过另一种方式来看待这一点：把被同一个标签标记的所有版本号经过的曲线拉直，然后直直地看过去，就象下图所示的一样：

```
file1  file2  file3  file4  file5

              1.1
              1.2
       1.1    1.3                  _
1.1    1.2    1.4    1.1           /
1.2*----1.3*----1.5*----1.2*----1.1* (--- <--- Look here
1.3           1.6    1.3           \_
1.4                  1.4
                     1.5
```

16.5 Tagging the working directory

cvs tag 命令

前面节的例子展示了打标签的常规用法。也就是在没有选项的情况下运行 cvs tag 来选择当前工作目录下检出文件的版本。比如，工作目录中 backend.c 副本的是从版本 1.4 检出的，那么 cvs 打标签的版本也是 1.4。请注意，tag 会立即作用于仓库中的 1.4 版本；打标记不像修改一个文件，或其它首先修改工作目录然后运行 cvs commit 把修改传送到仓库的操作。

这里有一个隐患，cvs tag 命令是对已经提交到仓库的版本进行操作，这很有可能与工作目录中本地修改了的文件不同。为了防止这种错误，给 `cvs tag` 加上 '`-c`' 选项。如

果有任何本地修改过了的文件，cvs 在其打标签之前将给出错误信息而退出：

```
$ cvs tag -c rel-0-4
cvs tag: backend.c is locally modified
cvs [tag aborted]: correct the above errors first!
```

16.6 Tagging by date/tag

：cvs rtag 命令

cvs rtag 命令用来对仓库一定的日期或时间打标签（或者给最后版本）。rtag 的特点是直接对仓库内容操作（它不需要事先检出也不会去寻找工作目录）。

下面选项指定打标签的日期或版本[9]。

- `-D date`
  为不晚于 date 的最新版本打标签。
- `-f`
  仅配合 '`-D`' 或 '`-r`' 标志使用。如找不到匹配的版本，使用最新的版本（替代忽略文件）。
- `-r tag[:date]`
  打标签给给已有的标签 tag，或者是指定 date 并且 tag 为分支标签，来自分支 tag 的版本已存在于 date[10]。

cvs tag 命令也可以用同样的 '`-r`'、'`-D`' 和 '`-f`' 选项按版本或日期指定文件。然而，此特点可能不是用户想要的。原因是 cvs tag 选择基于工作目录中存在的文件来打标签，而不是按给定的标记或时间的已存在文件来打标签。因此，最好使用 cvs rtag 命令。可能的例外情况像：

```
cvs tag -r 1.4 stable backend.c
```

16.7 Modifying tags

：添加、重命名和删除标签

通常不会去修改标签。它们的存在是为了纪录仓库的历史，如果删除或修改本身就是违反了初衷。

[9]参见 Common options 得到完整的解释。
[10]参阅 Common options。

然而，可能有人使用临时标签，或者在别处误打了标签。因此，用户就可以进行删除、移动更名一个标签。

警告：本节的命令具有危险性，他们会永久性的抹去历史纪录信息并且一旦出错无法恢复。如果是 cvs 管理员，应当利用 taginfo 限制这些命令[11]。

要删除标签，在 `cvs tag` 或 `cvs rtag` 后面加上 '`-d`' 选项。例如：

```
cvs rtag -d rel-0-4 tc
```

将非分支（non-branch）标签 rel-0-4 从模块 tc 上删除。如果给定的名称是仓库的一个分支标签（branch tags），该标签不会被删除并返回警告信息。当用户明确地知道自己在干什么，加上 `-B` 选项就能删除分支标签。反过来，这时如果遇到非分支标签将不会被删除并返回警告信息。

警告：移动分支标签是非常危险的！当需要使用 `-B` 选项时，想清楚并咨询 cvs 管理员（如果不是管理员）。当然有别的办法来完成想完成做的事。

move 一个标签，是将这个名字赋给另外的版本。例如，stable 标签现在是指向文件 backend.c 的版本 1.4 上，我们想将它指向版本 1.6 上面。要移动一个非分支标签，在 cvs tag 或 cvs rtag 命令后面加上 '`-F`' 选项。例如，可以这样做刚提到的任务：

```
cvs tag -r 1.6 -F stable backend.c
```

如果在仓库中遇到给定名字的任何分支标签，发出警告并不妨碍分支标签。如果明确要移动该分支标签，可以指定 `-B` 选项。那样，给定名字遇到的非分支标签将被忽略并返回警告消息。

警告：移动分支标签是非常危险的！当需要使用 `-B` 选项时，想清楚并咨询 cvs 管理员（如果不是管理员）。当然有别的办法来完成想完成做的事。

当我们说 rename 一个标签，是说给一个版本上的已有标签名改名。例如，有人将标签名写错了并要更正它（希望其他人还没有使用这个错误的标签）。要改标签名，先用带有 '`-r`' 选项的 cvs rtag 命令创建一个新标签，然后删除名字。(告诫：此方法将与分支标记一起不工作。) 改名后标签位置与老标签相同。例如：

```
cvs rtag -r old-name-0-4 rel-0-4 tc
cvs rtag -d old-name-0-4 tc
```

16.8 Tagging add/remove

标签与添加和删除文件

[11]参阅 taginfo。

准确地讲清楚打标签与添加、删除文件之间的相互关系不那么容易；其他部分的 cvs 对于跟踪文件存在与否都处理的不错。默认作法是，打标签只对有版本的文件进行。文件如果尚不存在或者被删除了将被忽略，cvs 对没有标签的文件认为它在打标签时不存在。

但这种作法会丢失一些信息。例如，假设有一个文件被添加，然后又给删除了。那么对于没有标签的文件，cvs 没法知道打标签是在添加前还是在删除后执行的。如果是用 cvs rtag 加 '`-r`' 选项，cvs 对已删除的文件也能打标签，这样就避免了上述问题。例如，你可以用 `-r HEAD` 来给版本树的头打标签。

为了处理添加、删除的文件，cvs rtag 命令有一个 '`-a`' 选项来避免给删除的文件打标签。例如，可以与移动标签时结合 '`-F`' 同时使用该选项。要是没有 '`-a`' 选项而移动标签，标签仍然会指向被删除文件的旧版本，而不能正确反映文件已经被删除。对采用 '`-r`' 选项的操作，上面解释过，一般不认为需要这样。

16.9 Sticky tags

某些标签是持久的

有时文件的一份工作副本的版本有另外的数据关联，例如它恰好位于一个分支上[12]，或者由于采用 '`checkout -D`' 或者 '`update -D`' 这样的命令，所以版本号受到日期的限制。因为这样的数据是持久的 – 也就是说，这些数据适用于工作副本中随后的一系列命令——我们称这些数据是粘性的 sticky。

在大多数情况下，粘着性是 cvs 较费解的一面，用户无需考虑过多。然而，即使你用不着使用这一特性，也需要知道一些粘性标签（例如，怎样避免它们!）。

用户可以使用状态 status 命令来查看是否设置任何粘性标签和日期：

```
$ cvs status driver.c
===================================================================
File: driver.c        Status: Up-to-date

   Version:           1.7.2.1 Sat Dec 5 19:35:03 1992
   RCS Version:       1.7.2.1 /u/cvsroot/yoyodyne/tc/driver.c,v
   Sticky Tag:        rel-1-0-patches (branch: 1.7.2)
   Sticky Date:       (none)
   Sticky Options:    (none)
```

[12]参阅 Branching and merging。

用户可以使用 '`cvs update -A`' 命令来去掉粘性标签。'`-A`' 选项将把本地的改变合并进树干顶部的文件，并且忽略中间的任何粘性标签、设置日期或选项[13]。

就像 Accessing branches 中讨论的一样，粘性标签常常用于识别当前是在哪一个分支上工作。然而，粘性标签也用于没有分支的情况。例如，假设想通过不更新你的工作目录内容的方法来避免受到其他成员所做的不稳定的改变。虽然可以通过不使用更新命令 `cvs update` 来做到这一点。但是如果只是希望不更新所有文件中的一小部分，粘性标签就有用了。如果检出某些版本（例如 1.4）它就成为粘性的。之后的 `cvs update` 命令不会取得最新版本，直到使用 `cvs update -A` 选项重置标签。同样地，如果在 update 或 checkout 命令中使用 '`-D`' 选项来设置 sticky date，这个日期也会被用于未来获得版本的操作中。

在很多情况下，用户希望取回一个不设置粘性标签的文件的老版本。方法是在用 checkout 或者 update 的时候使用 '`-p`' 选项，使用该选项可以把文件的内容发送到标准输出。例如：

```
$ cvs update -p -r 1.1 file1 >file1
===================================================================
Checking out file1
RCS: /tmp/cvs-sanity/cvsroot/first-dir/Attic/file1,v
VERS: 1.1
***************
$
```

然而，如果想要取消一个先前的检入（在本例中，将 file1 返回到版本 1.1），这不是一个最简单的方法。这种情况下应该是使用 update 命令的 '`-j`' 选项，进一步讨论参看 Merging two revisions。

[13]参考 update 获得更多的 cvs update 操作信息。

Chapter 17

Branching and merging

cvs 允许用户把修改隔离在各自的开发线上，这就是分支（branch）。当用户改变一个分支中的文件时，这些更改不会出现在开发主干（main trunk）和其它分支中。

在这之后可以使用 merging 把这些变更从一个分支移动到另一个分支（或主干）。合并首先使用 `cvs update -j` 命令，将这些变更合并到工作目录，然后用户可以提交这个版本，这样也可以将这些变更作用于其它的分支。

17.1 Branches motivation

假定 tc 的发行版 1.0 已完成，当前正在继续开发 tc，计划在 2 个月后创建发行 1.1 的版本。不久客户开始抱怨说代码有些问题，于是开发者检出了 1.0 的发行版[1]，找到了这个错误（这将会有一个小小的更正）。但是，当前代码的版本是处在一个不稳的状态，并且在下一个月才能有希望稳定下来。这样就没法基于最新源代码去发行一个修复错误的版本。

这种情况下就可以去为所有构成 tc 的 1.0 发行版文件创建版本树的一个分支（branch）。然后就可以修改这分支而不影响到主干。当修订完成时，开发者可以选定是否要把它同主干合并或继续保留在这个分支里。

17.2 Creating a branch

使用 `tag -b` 去建立一个分支；例如，假定现在有一个工作副本：

```
$ cvs tag -b rel-1-0-patches
```

[1]参阅 Tags。

这将基于工作副本的当前版本分离出一个分支，并分配 'rel-1-0-patches' 名字给该分支。

有一点对理解分支很重要，分支是在 CVS 仓库中创建，而非在工作副本中创建。正如上面的例子，基于当前版本创建一个分支不会自动把当前的工作副本切换到新的分支上[2]。

使用 rtag 命令可以不参考任何工作副本而创建一个分支：

```
$ cvs rtag -b -r rel-1-0 rel-1-0-patches tc
```

'`-r rel-1-0`' 说明这个分支是基于有 'rel-1-0' 这个标签的文件。这不是从最新的版本建立分支——对需要从老的版本分出一个分支很有用（例如，给以前一个认为稳定的发行版改 bug）。

跟使用 'tag' 命令一样，这个 '`-b`' 标志告诉 rtag 去创建一个分支（而不只是这个版本的符号连接）。注意，标记 'rel-1-0' 可能对不同的文件有不同的版本数字。

因此，这个命令的结果是为 'tc' 模块建立了一个命名为 'rel-1-0-patches' 的新版本分支，它是基于标记为 'rel-1-0' 的版本树。

17.3 Accessing branches

可以通过两种方式恢复分支：重新从仓库检出一份或是从现有的工作副本切换过去。

为了从仓库检出 i 一个分支，使用 'checkout' 命令并带上 '`-r`' 标志，后面是这个分支的标签（tag）名[3]：

```
$ cvs checkout -r rel-1-0-patches tc
```

或者如果已经有了一个工作副本，可以使用 '`update -r`' 命令切转到这个分支：

```
$ cvs update -r rel-1-0-patches tc
```

或者使用另一个等效的命令：

```
$ cd tc
$ cvs update -r rel-1-0-patches
```

这对工作副本为主干代码或是其它分支都是有效的——上面的命令将把它切换到指名的分支。同 'update' 命令相类似，'`update -r`' 合并用户所做的任何改变，通知用户出现的冲突。

[2]关于任何做的情况请看 Accessing branches。

[3]参阅 Creating a branch。

一旦用户的工作副本已经转向一个特定的分支，它将一直保持在这个分支内，除非用户又做了其它的操作。这意味着从这个工作副本提交的变更将加到这个分支的新版本中，而不影响到主干版本和其它分支。

想看一个工作副本是基于哪一个分支，可以使用 'status' 命令。在它们输出中查找一个 'Sticky tag' 的域[4]——那就是 cvs 告诉用户当前工作文件分支号的方式：

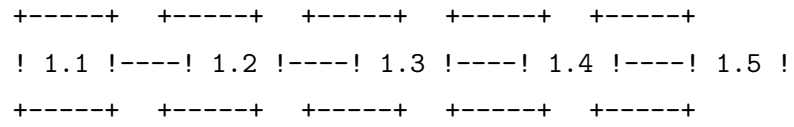
```
$ cvs status -v driver.c backend.c
===================================================================
File: driver.c       Status: Up-to-date

   Version:          1.7    Sat Dec 5 18:25:54 1992
   RCS Version:      1.7    /u/cvsroot/yoyodyne/tc/driver.c,v
   Sticky Tag:       rel-1-0-patches (branch: 1.7.2)
   Sticky Date:      (none)
   Sticky Options:   (none)

   Existing Tags:
      rel-1-0-patches         (branch: 1.7.2)
      rel-1-0                 (revision: 1.7)

===================================================================
File: backend.c      Status: Up-to-date

   Version:          1.4    Tue Dec 1 14:39:01 1992
   RCS Version:      1.4    /u/cvsroot/yoyodyne/tc/backend.c,v
   Sticky Tag:       rel-1-0-patches (branch: 1.4.2)
   Sticky Date:      (none)
   Sticky Options:   (none)

   Existing Tags:
      rel-1-0-patches         (branch: 1.4.2)
      rel-1-0                 (revision: 1.4)
      rel-0-4                 (revision: 1.4)
```

[4]参阅 Sticky tags。

请不要因为每个文件的分支号不同（‘1.7.2’ 和 ‘1.4.2’）而迷惑。分支的标签是相同的，‘rel-1-0-patches’，所以这些文件是在相同的分支上。数字简单地反映在每个文件的版本历史中在制造分支的点。在以上的例子中，分支建立之前，‘driver.c’ 比 ‘backend.c’ 有更多的变更，因此它们的版本编号是不同的[5]。

17.4 Branches and revisions

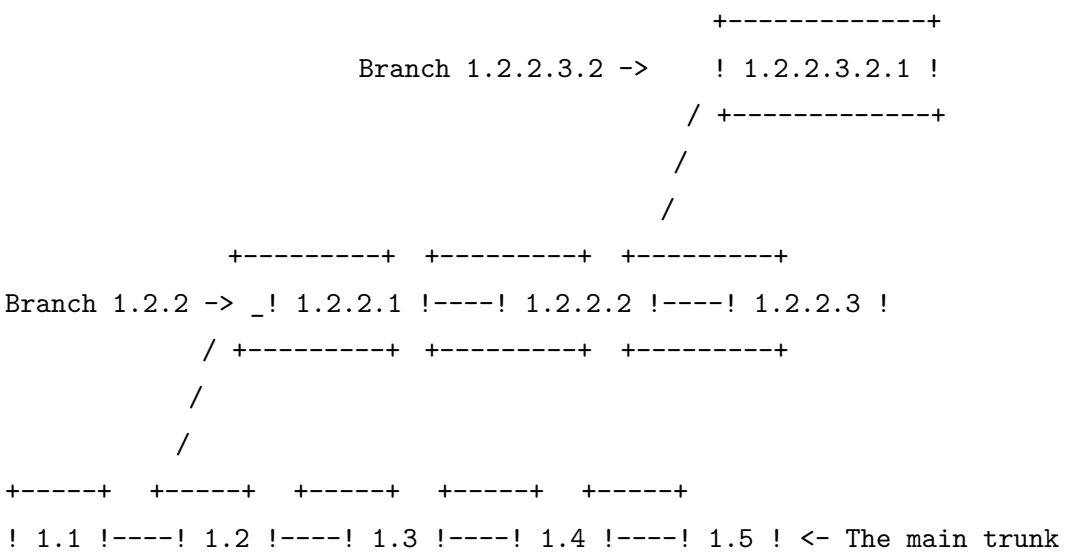
通常，一个文件的修订版本历史是一个增长线[6]：



然而，cvs 并不局限于线性的开发。版本树（revision tree）可以分为不同的分支（branches），每一个分支可以是一个独立的自我维护的开发线。而在一个分支中的变更可以很容易地移回到主干中。

每一个分支均有一个分支号（branch number），由奇数个“.”分开的十进制数组成。把一个整数追加到对应分支赖以分离出的版本号上来创建分支号。使用分支号允许从一个特定版本分离出多个分支。

所有的分支上的版本都把序号追加到分支号上来构成版本号。下面的例子将展示这一点。



[5]参阅 Branches and revisions 了解如何构建分支号的细节。

[6]参阅 Revision numbers。

```
+-----+   +-----+   +-----+   +-----+   +-----+
              !
              !
              !   +---------+  +---------+  +---------+
Branch 1.2.4 -> +---! 1.2.4.1 !----! 1.2.4.2 !----! 1.2.4.3 !
                  +---------+  +---------+  +---------+
```

虽然如何创建具体分支号的细节通常不是用户需要考虑的，但这里是它如何工作的：

当 cvs 创建一个分支号的时候它取一个未用的偶整数，用 2 开始。这样当用户想从 6.4 的版本创建分支时分支号将为 6.4.2。以零结尾的所有分支号（如 6.4.0）被 cvs 内部使用[7]。分支 1.1.1 有特别的含义[8]。

17.5 Magic branch numbers

这一节描述 cvs 的魔术分支（magic branches）特性。在大多数情况下，用户不用考虑魔术分支号，cvs 将为他们进行处理。然而，在一些特定条件下，它将显现出来，因此理解它如何工作将是有用的。

外表上，分支号码将由奇数个“.”分隔的十进制整数组成[9]。然而那并非完全是这样的。由于效率的原因，cvs 有时插入一个额外的“0”在右末的第二个位置（1.2.4 变为 1.2.0.4，8.9.10.11.12 变为 8.9.10.11.0.12 等）。

cvs 将会很好的将这些称为魔术分支隐蔽在背后进行，但在一些地方这种隐蔽并不完全：

- 魔术分支号会出现在 cvs log 的输出中。
- 用户不能够对 cvs admin 指定符号分支名。

用户可以使用 admin 命令去为一个分支重新分配一个 rcs 希望的那样的符号名。如果 R4patches 是一个分配给分支 1.4.2（魔术分支号为 1.4.0.2）的一个 numbers.c 文件的命名，用户可以使用如下命令：

```
$ cvs admin -NR4patches:1.4.2 numbers.c
```

至少有一个版本已经提交到这个分支时它才会有效。务必非常小心不要把一个标签（tag）分配给了一个错误标识号，现在没法看到昨天的一个标签是如何分配的。

[7]参阅 Magic branch numbers。
[8]参阅 Tracking sources。
[9]参阅 Revision numbers。

17.6 Merging a branch

用户可以把另一个分支上的修改合并到其工作副本，只要在 update 子命令中加 '`-j branchname`' 的标志。使用一个 '`-j branchname`' 选项，它把在分支的最大公共祖先（GCA）和目的修订版之间所做的改变（在下面简单的情况下 GCA 是分支在那里分岔的点）和在那个分支上的最新修订版合并进你的工作副本。

'`-j`' 的意思是“join”。

Consider this revision tree:

```
+-----+   +-----+   +-----+   +-----+
! 1.1 !---! 1.2 !---! 1.3 !---! 1.4 ! <- The main trunk
+-----+   +-----+   +-----+   +-----+
              !
              !
              !   +---------+  +---------+
Branch R1fix -> +---! 1.2.2.1 !----! 1.2.2.2 !
                    +---------+  +---------+
```

分支 1.2.2 分配了一个 'R1fix' 的标签（符号名）。下面的例子假定模块 'mod' 只包含一个文件 m.c。

```
$ cvs checkout mod          # Retrieve the latest revision, 1.4

$ cvs update -j R1fix m.c   # Merge all changes made on the branch,
                            # i.e. the changes between revision 1.2
                            # and 1.2.2.2, into your working copy
                            # of the file.

$ cvs commit -m "Included R1fix" # Create revision 1.5.
```

在合并时可能会发生冲突。如果这种情况发生，用户应该在提交新版本之前解决它[10]。

如果用户的源文件中包含关键字[11]，可能会得到比严格意义上的冲突更多的冲突信息[12]。

[10]参阅 Conflicts example。
[11]参阅 Keyword substitution。
[12]参见 Merging and keywords，去了解如何避免这个问题。

checkout 命令也支持使用 '`-j branchname`' 标志。下面的例子同上面所用的例子有相的效果：

```
$ cvs checkout -j R1fix mod
$ cvs commit -m "Included R1fix"
```

注意使用 `update -j tagname` 也许行但结果可能不是我们想要的[13]。

17.7 Merging more than once

继续我们的例子，现在版本树看起来是这样的：

```
+-----+   +-----+   +-----+   +-----+   +-----+
! 1.1 !---! 1.2 !---! 1.3 !---! 1.4 !---! 1.5 ! <- The main trunk
+-----+   +-----+   +-----+   +-----+   +-----+
              !                       *
              !                      *
              !   +---------+  +---------+
Branch R1fix -> +---! 1.2.2.1 !----! 1.2.2.2 !
                  +---------+  +---------+
```

正如上面所讨论的，星号线表示从 'R1fix' 分支到主干的合并。
现在我们继续开发 'R1fix' 分支：

```
+-----+   +-----+   +-----+   +-----+   +-----+
! 1.1 !---! 1.2 !---! 1.3 !---! 1.4 !---! 1.5 ! <- The main trunk
+-----+   +-----+   +-----+   +-----+   +-----+
              !                       *
              !                      *
              !   +---------+  +---------+  +---------+
Branch R1fix -> +---! 1.2.2.1 !----! 1.2.2.2 !----! 1.2.2.3 !
                  +---------+  +---------+  +---------+
```

然后你可能会希望合并新的变更到主干中去。如果用户仍使用 `cvs update -j R1fix m.c`，cvs 将试图合并已经合并过的东西，这可能导致一些不希望发生的副作用。

因此，必须指定我们只希望把未被合并分支的改变合并进树干。这样需要使用两个 '`-j`' 参数，cvs 合并从第一个版本到第二个版本的变化。例如，在这种情况下最简单的方

[13]详见 Merging adds and removals。

法应该是

```
cvs update -j 1.2.2.2 -j R1fix m.c # Merge changes from 1.2.2.2 to the
                                   # head of the R1fix branch
```

用此法的问题是用户需要手工指定 1.2.2.2 的版本号。一个更好的方法是使用最后完成合并的日期:

```
cvs update -j R1fix:yesterday -j R1fix m.c
```

在每一次合并进树干后加一个标签给 R1fix 分支，然后就可以使用该标签为以后的合并的方式也挺好:

```
cvs update -j merged_from_R1fix_to_trunk -j R1fix m.c
```

17.8 Merging two revisions

使用两个 '`-j revision`' 标志，update（和 checkout）命令能把两个任意不同的版本的差异合并进当前的工作文件。

```
$ cvs update -j 1.5 -j 1.3 backend.c
```

将把 1.5 版本恢复到 1.3 版本。注意修订版本的次序!

如果在操作多个文件时使用这个选择项，用户必须了解在不同的文件之间，版本的数字可能是完全不同的。我们几乎总是使用符号标签而不是使用版本号来完成多个文件的操作。

使用两个 '`-j`' 操作也能恢复增加或删除的文件。例如，假定有一个叫 file1 的文件存在于 1.1 版本中，然后又删除了它（因此增加了一个 dead 版本 1.2)。如果现在又打算增加它，并用它原先的内容。下面是如何操作的:

```
$ cvs update -j 1.2 -j 1.1 file1
U file1
$ cvs commit -m test
Checking in file1;
/tmp/cvs-sanity/cvsroot/first-dir/file1,v <-- file1
new revision: 1.3; previous revision: 1.2
done
$
```

17.9 Merging adds and removals

如果在合并时做的改变涉及到添加或删除一些文件，`update -j` 将反映这些变化。例如：

```
cvs update -A
touch a b c
cvs add a b c ; cvs ci -m "added" a b c
cvs tag -b branchtag
cvs update -r branchtag
touch d ; cvs add d
rm a ; cvs rm a
cvs ci -m "added d, removed a"
cvs update -A
cvs update -jbranchtag
```

在执行这些命令且‘`cvs commit`’——完成之后，文件 a 将被删除，而文件 d 将被加入到主分支。

注意，当用静态标签（‘`-j tagname`’）而不是动态标签（‘`-j branchname`’）从一个分支合并改变时，cvs 一般不会删除文件，因为 cvs 不会自动给 dead 版本添加静态标签。除非静态标签是手工添加到 dead 版本上的。使用分支标签从分支合并所有改变或使用两个静态标签作为合并端点合并都会在合并中将企图的修改传播开。

17.10 Merging and keywords

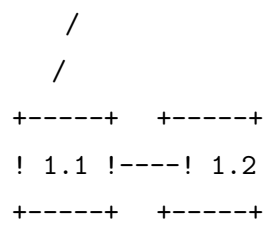
如果要合并的文件中包含关键字[14]，就会得到一大堆冲突，这是因为关键字与合并的版本关联。

因此，需要在合并的命令行里面指定‘`-kk`’开关[15]。用只替换关键字名而不是其展开的值的办法，这个选项确保正合并的版本互相是是相同的，从而避免产生假的冲突。

例如，假定你有一个这样的文件：

```
      +---------+
     _! 1.1.2.1 ! <- br1
    / +---------+
```

[14]参阅 Keyword substitution。
[15]参阅 Substitution modes。



用户的工作目录当前在树干上（版本 1.2)。合并时会得到下面的信息：

```
$ cat file1
key $Revision: 1.2 $
. . .
$ cvs update -j br1
U file1
RCS file: /cvsroot/first-dir/file1,v
retrieving revision 1.1
retrieving revision 1.1.2.1
Merging differences between 1.1 and 1.1.2.1 into file1
rcsmerge: warning: conflicts during merge
$ cat file1
<<<<<<< file1
key $Revision: 1.2 $
=======
key $Revision: 1.1.2.1 $
>>>>>>> 1.1.2.1
. . .
```

产生这些信息是由于合并尝试将 1.1 与 1.1.2.1 之间的差异合并到你的工作目录。因为版本关键字从 Revision: 1.1 变为 Revision: 1.1.2.1，cvs 试图把该改变合并进你的工作目录，而与工作目录里已包含 Revision: 1.2 的事实冲突。

下面是使用 '-kk' 后的结果：

```
$ cat file1
key $Revision: 1.2 $
. . .
$ cvs update -kk -j br1
U file1
```

```
RCS file: /cvsroot/first-dir/file1,v
retrieving revision 1.1
retrieving revision 1.1.2.1
Merging differences between 1.1 and 1.1.2.1 into file1
$ cat file1
key $Revision$
. . .
```

这时在文件中 1.1 和 1.1.2.1 版本的关键字双双扩展为明码 Revision，因此把它们之间的改变合并进工作目录不需要改变什么。也就不会有冲突产生。

警告：在 cvs 1.12.2 之前的版本中，合并时使用 '`-kk`' 会有严重的问题。那就是 '`-kk`' 会跨越任何仓库中归档文件中设置的默认关键字扩展模式。对一些用户很不幸，这会造成二进制文件（默认关键字扩展模式设置为 '`-kb`'）的损坏。因此，当仓库中包含二进制文件时，解决冲突必须使用手工修改的方法来替代合并命令中的 '`-kk`'。

在 cvs 1.12.2 之后的版本中，命令行对任何 cvs 命令提供的关键字扩展模式不再跨越为二进制文件设置的 '`-kb`' 关键字扩展模式，然而它还是将跨越其它默认关键字扩展模式。现在即使仓库中包含二进制文件，你也可以在合并中安全地使用 '`-kk`' 来防止含有 RCS 关键字行的假性冲突。

Chapter 18

Recursive behavior

如果用户指定一个目录作为参数，几乎所有的 cvs 子命令都会在该目录中递归地执行。例如，考虑这个目录结构：

```
$HOME
  |
  +--tc
  |   |
      +--CVS
      |     (internal cvs files)
      +--Makefile
      +--backend.c
      +--driver.c
      +--frontend.c
      +--parser.c
      +--man
      |    |
      |    +--CVS
      |    |  (internal cvs files)
      |    +--tc.1
      |
      +--testing
           |
           +--CVS
```

```
|  (internal cvs files)
+--testpgm.t
+--test2.t
```

如果 tc 是当前工作目录，则以下操作为真：

- ‘`cvs update testing`’ 等价于

  `cvs update testing/testpgm.t testing/test2.t`
- ‘`cvs update testing man`’ 会更新子目录下的所有文件
- 使用 ‘cvs update .’ 或者只使用 ‘cvs update’ 会更新 tc 目录中的所有文件

如果不给 update 命令赋参数，它会更新当前工作目录中的所有文件和所有子目录。换句话说，. 是 update 命令的默认参数。这点对其它 cvs 子命令也适用，不仅仅是 update 命令。

使用选项 ‘`-l`’ 可以关闭 cvs 子命令的递归行为。相反地，如果在 `/.cvsrc`（参阅 `/.cvsrc`）中指定 ‘`-l`’ 选项，可以使用 ‘`-R`’ 选项来强制递归。

```
$ cvs update -l      # Don't update files in subdirectories
```

Chapter 19

Adding and removing

在项目的开发过程中，常常需要添加新文件。同时还要删除或重命名文件，对目录也一样。用户需要紧记，在这些情况下是使用 cvs 记录下发生了什么变化而不是用它做不可恢复的修改，这和修改已有的文件一样。实现这一点的机制在很大程度上依赖于 cvs 运行的环境。

19.1 Adding files

按照以下步骤，可以添加一个新文件到的一个目录。

- 用户必须有目录的一个工作副本[1]。
- 在工作副本的目录中创建一个新文件。
- 使用 'cvs add filename' 命令告诉 cvs 我们希望对该文件进行版本控制。如果该文件包含二进制数据，需要指定 '`-kb`' 选项[2]。
- 使用 'cvs commit filename' 命令真正把该文件提交进仓库。在这一步前，别的开发人员都看不到这个新加入的文件。

也可以使用 add 命令新建一个目录。

不象其它命令，add 命令的执行方式不是递归的。你必须指明你准备添加到仓库中的文件名和路径。而且，每个目录在添加新文件到其中之前，还必须单独地添加进仓库。

```
$ mkdir -p foo/bar
$ cp ~/myfile foo/bar/myfile
$ cvs add foo foo/bar
```

[1]参阅 Getting the source。
[2]参阅 Binary files。

```
$ cvs add foo/bar/myfile
```

Command: cvs add [-k kflag] [-m message] files ...

将文件 files 添加进仓库。命令中指定的 add 文件或目录必须在当前目录中存在。要将整个新的目录结构（例如，来自第三方的文件）添加进源码仓库，使用 import 命令[3]。

被添加的文件直到用户使用 commit 命令确认修改，才被放进仓库。要 add 一个被 remove 命令删掉的文件将取消删除 remove，除非 commit 干预[4]。

指定选项 '-k' 选项修改以后默认检出的方式[5]。

使用 '-m' 选项可以同时添加文件的描述性信息。这种描述出现在历史记录（如果启用它的话，请参阅 history file）中。在文件被提交的时候，这些描述性信息也会被存储在仓库的版本历史中。用 log 命令可以显示这些描述。使用 'admin -t' 命令可以修改描述性信息。参阅 admin. 如果用户忽略 '-m description' 标志，会自动使用一个空字符串。不会向你提示描述信息。

例如，下面的例子把文件 backend.c 添加到仓库：

```
$ cvs add backend.c
$ cvs commit -m "Early version. Not yet compilable." backend.c
```

当添加一个文件的时候它仅仅被添加到你当前工作的分支上[6]。但是稍后如果想也可以把添加的内容合并到另外的分支去[7]。

19.2 Removing files

各个目录产生变化。有新的文件不断地添加进来，也有一些不再需要的文件被删除。在某些情况下，用户可能仍然希望能获得某一个老发行版的正确的副本。

通过下面的方式，用户可以在当前版本中删除一个文件而仍然在老版本中保留它：

- 确保文件中不存在没有被提交的修改[8]。用户也可以使用 status 命令或者 update 命令。如果在没有提交的情况下删除了一个文件，当然无法重新从仓库中获得修改后的内容。
- 从目录的工作副本删除文件。例如可以使用 rm。
- 使用 'cvs remove filename' 命令告诉 cvs 删除该文件。

[3]参阅 import。
[4]参阅 Removing files。
[5]参阅 Substitution modes 来获得更多信息。
[6]参阅 Branching and merging。
[7]参阅 Merging adds and removals。
[8]参阅 Viewing differences, 寻找其做法。

- 使用 'cvs commit filename' 命令实际执行从仓库删除该文件。

当用户提交文件的删除操作后，cvs 将记录该文件不再存在。一个文件可能存在于仓库的一些分支里，而不在另外的分支里，或者在后来加入同名的另一个文件。根据在 checkout 或 update 命令中指定 '-r' 和 '-D' 选项，cvs 可以正确地创建和不创建该文件。

Command: cvs remove [options] files ...

策划从仓库中删除的文件（还没有从工作目录中删除的文件不包括）。本命令不会真正从仓库里删除文件直到用户提交了删除操作[9]。

下面是删除多个文件的例子：

```
$ cd test
$ rm *.c
$ cvs remove
cvs remove: Removing .
cvs remove: scheduling a.c for removal
cvs remove: scheduling b.c for removal
cvs remove: use 'cvs commit' to remove these files permanently
$ cvs ci -m "Removed unneeded files"
cvs commit: Examining .
cvs commit: Committing .
```

为了简化操作，可以通过 cvs remove 命令 '-f' 选项一步就从仓库中删除文件。例如，上面的例子也可以 xiang 像这样：

```
$ cd test
$ cvs remove -f *.c
cvs remove: scheduling a.c for removal
cvs remove: scheduling b.c for removal
cvs remove: use 'cvs commit' to remove these files permanently
$ cvs ci -m "Removed unneeded files"
cvs commit: Examining .
cvs commit: Committing .
```

如果用户执行 remove 文件之后，在提交以前又改变了主意，可以使用 add 命令来取消 remove 操作。

```
$ ls
```

[9]查看 Invoking CVS 所有可用的选项。

```
CVS  ja.h oj.c
$ rm oj.c
$ cvs remove oj.c
cvs remove: scheduling oj.c for removal
cvs remove: use 'cvs commit' to remove this file permanently
$ cvs add oj.c
U oj.c
cvs add: oj.c, version 1.1.1.1, resurrected
```

如果在运行 remove 前意识到了错误，可以使用 update 命令来恢复文件：

```
$ rm oj.c
$ cvs update oj.c
cvs update: warning: oj.c was lost
U oj.c
```

如果用户只是在当前工作的分支上删除了文件[10]。如果想要也可以将删除后的内容合并到别的分支[11]。

19.3 Removing directories

删除目录和删除文件概念上有些类似—用户既希望一个目录在当前工作目录中不存在，同时又希望在存在过的目录中取出老版本。

删除目录的方法就是删除目录下的所有文件。用户不能直接删除目录本身；目前 cvs 中也没有方法可以办到这一点。可以在 cvs update 或 cvs checkout 命令中使用 '-P' 选项来让 cvs 删除工作目录中的空目录 (注意 cvs export 命令总是删除空目录的)。（注意 cvs export 命令总是删除空目录的。)

可能最好的办法就是每次在上面的命令中指定 '-P' 选项；如果用户希望在工作目录中保留一个空目录，在该目录中随便放一个文件（例如放一个.keepme）文件来防止它被带 '-P' 的命令删掉。

注意，checkout 命令的 '-r' 和 '-D' 选项隐含了 '-P' 选项。这使 cvs 可以正确地创建目录，或者不考虑在该目录中是否有用户检出的某个版本的文件。

[10]参阅 Branching and merging。
[11]参阅 Merging adds and removals。

19.4 Moving files

移动文件到另一个目录或者重命名并不困难，但是其中一些方法的工作方式并不是太明了[12]。

下面的例子假设文件 old 被重命名为 new。

19.4.1 Outside

重命名的常规方法

移动一个文件的常规方法是把文件从 old 复制到 new，然后使用 cvs 命令从仓库里删除 old 文件，并且把 new 文件添加进去。

```
$ mv old new
$ cvs remove old
$ cvs add new
$ cvs commit -m "Renamed old to new" old new
```

这是移动文件最简单的方法，而且不会出错，它还可以清楚地留下操作的记录。注意，用户必须指定文件新的或者老的名称来访问历史，这取决于用户想要访问哪一部分。例如，cvs log old 命令将给出文件被重命名前的所有记录。

当文件以新 new 名称被提交以后，它的版本号将从 1.1 开始重新分配，用户如果觉得这一点不便的话，可以在提交命令中使用 '-r tag' 选项[13]。

19.4.2 Inside

一种巧妙的备用方法

这种做法是比较危险的，因为它涉及移动在仓库里的文件。在动手之前，请完整地阅读这一节内容!

```
$ cd $CVSROOT/dir
$ mv old,v new,v
```

优点:

- 记载这些改变的日志保持完整。
- 这些修订号不受影响。

缺点:

[12]移动和重命名目录更困难一些。参阅 Moving directories。

[13]参阅 Assigning revisions 获得更多细节。

- 将不能轻易地从仓库提取旧发行版[14]。
- 当这个文件被改名时，将不会产生对应的日志信息。
- 如果在移动这些历史文件的时候有人恰巧正在访问它，将会发生十分糟糕的事情，所以在移动这些文件的时候，请确保没有任何 cvs 命令运行访问到这个文件。

19.4.3 Rename by copying

还有一种巧妙的备用方法

这里同样包含了对仓库的直接修改。这样做是安全的，但并非没有坏处。

```
# Copy the rcs file inside the repository
$ cd $CVSROOT/dir
$ cp old,v new,v
# Remove the old file
$ cd ~/dir
$ rm old
$ cvs remove old
$ cvs commit old
# Remove all tags from new
$ cvs update new
$ cvs log new          # Remember the non-branch tag names
$ cvs tag -d tag1 new
$ cvs tag -d tag2 new
...
```

去掉标签，你就能检出老的版本。

优点：

- 检出旧版本工作正常，只要你用 '-r tag' 而不是 '-D date' 来取回这些版本。
- 记载这些改变的日志保持完整。
- 这些修订号不受影响。

缺点：

- 很难看清文件改名的历史情况。

[14]仓库里留下的文件是个新 new 文件，甚至在修订版中改名之前的版本也这样。

19.5 Moving directories

重命名或者移动一个目录的常规方法就是象 Outside 中描述的一样。然后象 Removing directories 一节中描述的一样使用 '-P' 检出。

如果用户希望在仓库里重命名或者删除一个目录，可以采用以下方法：

1. 首先，通知使用包含该目录的模块的所有用户目录要改名。并在整理前要求他们提交已做的工作，删除该它们的工作副本。
2. 在仓库里重命名该目录。

```
$ cd $CVSROOT/parent-dir
$ mv old-dir new-dir
```

3. 必要的话，修改 cvs 的管理文件（例如，重命名了整个模块)。
4. 通知所有用户可以检出该模块继续工作了。

如果有人没有移除自己本地的被重命名模块的工作副本，cvs 将不会工作直到他删除了仓库里没有的目录为止。

最好只是移动目录下的文件而不移动目录。如果移动了目录就可能无法从仓库里正确地获得老版本的代码，因为文件的以前版本的内容可能依赖于某个被移动了的目录的名称。

Chapter 20

History browsing

一旦当我们用 cvs 存储了一个文件的版本控制的历史——如它是怎样被改变的，以及什么时候、被什么人改变等后，那就可以用很多种方法来查看这些历史。

20.1 log messages

日志消息

无论何时都应该在提交文件时加上日志信息。

要查看提交时记录的 log 信息，可以使用 cvs log 命令[1]。

20.2 history database

历史数据库

可以使用这些历史文件[2]记录 cvs 的动作。要从历史文件得到信息，使用 cvs history 命令[3]。

注意：要控制哪些文件纪录日志，可以设置 CVSROOT/config 文件中的 'LogHistory' 关键字[4]。

20.3 user-defined logging

用户定义的日志

[1]参阅 log。
[2]参阅 history file。
[3]参阅 history。
[4]参阅 config。

用户可以定制 cvs 来记录所选择的任何一种动作。这种机制是通过在不同的时候执行相应脚本来实现的。脚本可能在一个列出信息和创建人的文件后面附加一条信息，也可以发送电子邮件给开发组的其他程序员，或许在一个给定的新闻组上发布一条信息。要记录提交操作，可以使用 loginfo 文件[5]，要记录 tag，使用 taginfo 文件[6]。

要分别记录 commits、checkouts、exports 和 tags 操作，可以在模块文件中分别使用 '-i'、'-o'、'-e' 和 '-t' 选项。要通过更灵活的方式把信息通知给不同的用户，要求尽可能少地采用更新集中脚本的方式，这种情况下应该使用 cvs watch add 命令[7]，这个命令即使在没有使用 cvs watch on 的时候也有用。

[5]参阅 loginfo。
[6]参阅 taginfo。
[7]参阅 Getting Notified。

Chapter 21

CVS Practice

21.1 Binary files

cvs 通常都是用来存储文本文件的。对于文本文件，cvs 可以合并修订版，并且可以按照人的可读性显示不同修订版本之间的差异以及类似的操作。但如果用户放弃这些功能的话，cvs 也可以存储二进制文件。例如，某个人或许在 cvs 里保存一个既有文本又有图像的 WEB 站点。

21.1.1 Binary why

什么时候需要对二进制文件进行管理是很明显的：用户通常使用的是二进制文件，把它们放到版本控制时，要注意的一些额外的问题。

版本控制的一个基本功能是显示两个修订版之间的差异。例如，如果某人检入了一个文件的新版本，也许我们想看看这个版本有什么变化并想确定这些改变的好坏。对于文本文件 cvs 是通过 cvs diff 命令来提供该功能。

对于二进制文件，一种可能的做法是：首先抽取这两的修订版本，然后使用 cvs 外部工具（例如，字处理软件通常都具有这样的功能）来比较它们。如果没有这样的工具，那么必须通过其他的机制来跟踪改变，例如督促人们认真记录日志信息，并希望他们实际所作的改动就是他们想要改的。

版本控制系统的另一个功能是合并两个修订版。对于 cvs，两种环境下发生这种情况。一是用户所作的改变是在不同的工作目录下[1]。第二是是某人用 'update -j' 命令显式合并时[2]。

[1]参阅 Multiple developers。
[2]参阅 Branching and merging。

对于文本文件，cvs 可以独立地合并，如果有冲突还可以给出冲突信号。而对于二进制文件，cvs 的最好做法是提供两个文件的不同副本，让用户自己解决冲突。用户可以任选其中一个副本，或者通过能处理该格式文件的合并工具来解决这个问题。注意，让用户解决合并，主要靠用户不会偶然忽略一些改变，因此会有潜在的错误发生。

如果不喜欢上述处理方式的话，最好的解决办法是避免合并。如何避免由于不同的工作目录引起的合并，参见 Multiple developers 里关于保留的检出方法（文件加锁）的讨论。为了避免由于分支引起的合并，还要限制使用分支。

21.1.2 Binary howto

使用 cvs 来存贮二进制文件有两个问题。第一是默认情况下，cvs 将会在仓库的规范保存形式（只有换行符）和客户端使用的操作系统合适形式（例如 Windows NT 中回车符跟在换行符后面）之间进行行尾转换。

第二是一个二进制文件有可能会包含有看起来像关键字 (参阅 Keyword substitution) 的数据，所以关键字扩展必须关闭。

在一些 CVS 命令中使用的选项 '-kb'，可以确保 cvs 不进行行尾转换和关键字扩展。

下面一个例子说明如何使用 '-kb' 标志创建一个新的文件：

```
$ echo '$Id$' > kotest
$ cvs add -kb -m"A test file" kotest
$ cvs ci -m"First checkin; contains a keyword" kotest
```

如果一个文件意外地没有用 '-kb' 添加，你可以使用 cvs admin 命令去恢复。例如：

```
$ echo '$Id$' > kotest
$ cvs add -m"A test file" kotest
$ cvs ci -m"First checkin; contains a keyword" kotest
$ cvs admin -kb kotest
$ cvs update -A kotest
# For non-unix systems:
# Copy in a good copy of the file from outside CVS
$ cvs commit -m "make it binary" kotest
```

当用户检入 kotest 文件时，这个文件不使用二进制形式保存，因为用户并没有当成二进制文件检入。cvs admin -kb 命令默认地为这文件设置关键字替换方法，但它并不改变用户拥有的这个文件的工作副本。如果用户需要处理行尾（也就是说，当前在一个非 unix 系统中使用 cvs），那么需要检入文件的新副本，如上面 cvs commit 命令所示。在

Unix 中可使用 cvs update -A 命令满足需要。（注意，用户可以使用 cvs log 检测文件的默认关键字替换模式，使用 cvs status 检测工作副本的关键字替换模式。）

虽然如此，在使用 cvs admin -k 来改变关键字扩展时，记住关键字扩展模式是不会受版本控制的。这就是说，例如，如果用户有一个旧版本的文本文件以及一个相同名字的新版本的二进制文件，cvs 没办法根据用户要取出的版本取出文本模式或是二进制模式的文件。现在还没有很好的办法解决这个问题。

也可以设置默认值，让 cvs add 和 cvs import 根据名字来决定文件是否以二进制模式处理；例如，可以让文件名以 '.exe' 结尾是二进制的[3]。现在没有办法让 cvs 根据内容检测一个文件是否为二进制。设计这样一个特性的主要困难是，二进制和非二进制文件之间的区别并非明确的，且使用的准则随操作系统而有很大的不同。

21.2 Multiple developers

当软件项目不再由一个人来开发的时候，事情总会变得复杂起来。因为经常会碰到两个人同时编辑一份文件的情况发生。有一种解决方案，文件加锁（file locking）或者节制检出（reserved checkouts），就是每份文件每次只允许一个人来编辑。这种方法在很多版本控制软件中使用，比如 rcs 和 sccs。当前在 cvs 中通常可以用 cvs admin -l 命令来做到节制检出[4]。这点在 cvs 里不如监视（watch）特性做的好，后文有述，但对于寻找“节制检出”手段的人来说也够用。

cvs 1.12.10 以后的版本，获得大部分节制检出功能的另外一种技术是打开“advisory locks”。为了打开“advisory locks”，所有的开发人员要将"edit -c", "commit -c" 放入它们的.cvsrc 文件中。然后在仓库里面打开监视。如果有人已经编辑此文件，这将阻止他们执行 cvs edit。这也可以配合普通的监视，使用适当的过程（非软件强制），避免两个人同时编辑。

cvs 默认的模型为无保留检出（unreserved checkouts）。这种模型下，多个开发人员可以同时编辑自己的工作副本（working copy）。第一个提交修改的人无法自动得知其他人已经开始编辑它。其他人在提交此文件的时候将得到错误消息。然后他们必须要用 cvs 命令使自己的工作副本用仓库修订版本来更新。这一过程几乎是自动的。

cvs 还支援各种便利的沟通方式，而不是像节制检出那样用强制的规则。

本章余下部分将解释这些工作模式，和一些涉及在它们间选择的问题。

[3]参阅 Wrappers。
[4]参阅 admin options。

21.2.1 File status

文件可以处于多种状态

依据用户对检出文件的操作和其他人对仓库文件的操作，可以将文件分成各种状态。这些状态可以用 status 命令看到：

- Up-to-date 该文件与该分支在仓库里面的最新版本文件内容一致。
- Locally Modified 编辑了文件，并且还没有提交改变。
- Locally Added 用 add 命令添加了该文件，但没有提交改变。
- Locally Removed 用 remove 命令删除了该文件，但没有提交改变。
- Needs Checkout 另有人向仓库提交了较新的版本。这个名字有些误导；一般应该用 update 而不是 checkout 来得到新近的版本。
- Needs Patch 与需要检出类似，但 cvs 服务器只发送补丁 (patch) 而不是整个文件。发补丁和发整个文件做的是同样的事情。
- Needs Merge 另有人向仓库提交了较新的版本，而你已经对该文件做了修改。
- Unresolved Conflict 与此新文件名字相同的文件从第二工作区添加到仓库中。该文件要移走才能执行 update 命令。
- File had conflicts on merge 类似于 Locally Modified，除先前的 update 命令造成了冲突外。如果你还没有那么做，需要按 Conflicts example 描述的解决这个冲突。
- Unknown cvs 对该文件一无所知。例如，创建了一个新文件而还没有执行 add 命令。

为了说清文件的状态，status 命令还报告工作目录中文件的 Working revision，它是该文件编辑前的版本，和一个 Repository revision，它是该文件在仓库里面使用分支的最新版本。'Commit Identifier' 影响 commit 的单一提交。

status 命令的选项列于 Invoking CVS。关于它的 Sticky tag 和 Sticky date 输出信息，见 Sticky tags。关于它的 Sticky options 输出信息，见 update options 中 '-k' 选项。

你可以认为 status 和 update 命令有互补性。用 update 将文件更新，而用 status 可以得到 update 能做什么的提示（当然，仓库的状态可能会在实际运行 update 命令时有所改变)。其实，如果你想要使命令显示比 status 命令显示的更简洁的格式文件状态的信息，可以调用

```
$ cvs -n -q update
```

选项 '-n' 是要求不做实质更新，仅仅显示状态信息；而 '-q' 选项用来避免打印每个目录名[5]。

[5]关于 update 命令及其选项的更多信息，见 Invoking CVS。

21.2.2 Updating a file

使文件更新

当需要更新或合并文件时，使用 update 命令。对于没有更新的文件这近似于使用 checkout 命令：将文件的最新版本从仓库中取出，并放到你的工作目录中。

使用 update 命令时决不会让你对文件所做的修改丢失。如果没有较新的版本存在，运行 update 没有效果。如果你编辑了一个文件并且有较新的版本可用，cvs 将把全部修改合并进工作目录中。

例如，设想检出版本 1.4 并开始编辑。与此同时另外一个人提交了版本 1.5，后来又提交了 1.6。如果你这时对该文件运行 update 命令，cvs 将版本 1.4 到 1.6 之间的所有修改并入到你的文件中。

如果版本 1.4 到 1.6 做的修改与你的改动太靠近，就有了重叠 overlap 的部分。这种情况下打印一个警告，并结果的文件含有加上特殊的标记的重叠各行的改写本[6]。

21.2.3 Conflicts example

一个有益的例子

假设 driver.c 文件的 1.4 包含：

```
#include <stdio.h>

void main()
{
    parse();
    if (nerr == 0)
        gencode();
    else
        fprintf(stderr, "No code generated.\n");
    exit(nerr == 0 ? 0 : 1);
}
```

而 driver.c 的 1.6 版中包含：

```
#include <stdio.h>

int main(int argc,
```

[6]参阅 update，以了解 update 命令的完整描述。

```
        char **argv)
{
    parse();
    if (argc != 1)
    {
        fprintf(stderr, "tc: No args expected.\n");
        exit(1);
    }
    if (nerr == 0)
        gencode();
    else
        fprintf(stderr, "No code generated.\n");
    exit(!!nerr);
}
```

工作目录中 driver.c，它是基于版本 1.4。执行 'cvs update' 命令前包含：

```
#include <stdlib.h>
#include <stdio.h>

void main()
{
    init_scanner();
    parse();
    if (nerr == 0)
        gencode();
    else
        fprintf(stderr, "No code generated.\n");
    exit(nerr == 0 ? EXIT_SUCCESS : EXIT_FAILURE);
}
```

运行 'cvs update' 命令：

```
$ cvs update driver.c
RCS file: /usr/local/cvsroot/yoyodyne/tc/driver.c,v
retrieving revision 1.4
```

```
retrieving revision 1.6
Merging differences between 1.4 and 1.6 into driver.c
rcsmerge warning: overlaps during merge
cvs update: conflicts found in driver.c
C driver.c
```

cvs 告诉用户有某些冲突存在。先前的工作文件被不修改地保存在.#driver.c.1.4 中。driver.c 的新版本文件内容为：

```
#include <stdlib.h>
#include <stdio.h>

int main(int argc,
         char **argv)
{
    init_scanner();
    parse();
    if (argc != 1)
    {
        fprintf(stderr, "tc: No args expected.\n");
        exit(1);
    }
    if (nerr == 0)
        gencode();
    else
        fprintf(stderr, "No code generated.\n");
<<<<<<< driver.c
    exit(nerr == 0 ? EXIT_SUCCESS : EXIT_FAILURE);
=======
    exit(!!nerr);
>>>>>>> 1.6
}
```

上面显示了没有重复的地方合并的情况和重复的地方用 '«««<', '=======' 和 '»»»>' 做了清晰的标记。

编辑文件，去掉标记和错误的行来解决冲突。假定最终文件内容为：

```
#include <stdlib.h>
#include <stdio.h>

int main(int argc,
         char **argv)
{
    init_scanner();
    parse();
    if (argc != 1)
    {
        fprintf(stderr, "tc: No args expected.\n");
        exit(1);
    }
    if (nerr == 0)
        gencode();
    else
        fprintf(stderr, "No code generated.\n");
    exit(nerr == 0 ? EXIT_SUCCESS : EXIT_FAILURE);
}
```

接着按版本 1.7 提交修改。

```
$ cvs commit -m "Initialize scanner. Use symbolic exit values." driver.c
Checking in driver.c;
/usr/local/cvsroot/yoyodyne/tc/driver.c,v <-- driver.c
new revision: 1.7; previous revision: 1.6
done
```

出于保护的目的，如果你没有处理冲突 cvs 将拒绝提交。解决冲突后你自然会改变文件的时间戳。在以前版本的 cvs 中，还要求文件不能包含冲突标记。但因为文件中可能会出现合理的冲突标记（比如出现 '»»»>' 字符但又不是冲突标记），现在版本的 cvs 只发出警告但还会将文件提交。

假如用户有 1.04 版或以后的 pcl-cvs（gnu Emacs 的 cvs 前端），就可以用 Emacs 软件包的合并功能解决冲突[7]。

[7]查看文档中的 pcl-cvs。

21.2.4 Informing others

合作需要相互通知

提交一个文件的新版本后通知其他的开发人员通常会有用的。modules 的 '-i' 选项，或 loginfo 文件，可以被用于自动执行这种处理。参阅 modules. 参阅 loginfo. 例如，利用 cvs 这一功能分咐 cvs 发邮件给所有的开发者一个消息，或者把消息发到本地新闻组里面。

21.2.5 Concurrency

同时仓库的存取

如果多个开发者试图同时运行 cvs，其中一个人会得到下面消息：

```
[11:43:23] waiting for bach's lock in /usr/local/cvsroot/foo
```

cvs 会每 30 秒尝试一次，然后或者完成操作，或者继续显示上面的信息。如果锁定时间看起来不正常，询问一下上面信息里显示的加锁的人是否他正在执行命令。如果他们不在运行 CVS 命令，查看仓库里面在消息中提到的目录并删除它们所属以 #cvs.rfl、#cvs.wfl 或 #cvs.lock 开始的文件。

注意，这里的 lock 只是用来保护 cvs 的内部数据结构，跟 rcs 里面的 lock 没有任何关系—rcs 里面的是用来保留检出的[8]。

除非有人加了锁来防止别人读或写，仓库可以在同一时间被多人读取。

或许有人希望能有这样的仓库：

一个人用一个命令提交的所有修改，在另外一个人执行更新命令时要么得到全部的变更，要么一个都没有。

但是 cvs 没有这样的性能。例如，给出下列文件

```
a/one.c
a/two.c
b/three.c
b/four.c
```

如有人运行

```
cvs ci a/two.c b/three.c
```

并且另一个人同时运行 `cvs update` 命令，运行 update 的人可能只得到 b/three.c 里面的变更而没有得到 a/two.c 的变更。

[8]参阅 Multiple developers。

21.2.6 Watches

跟踪谁在编辑文件的机制

对于大多数开发团队，使用 cvs 默认的模式就可以很好地满足要求。有时用户可以检入修改以寻找是否遇到其它修改的干预，他们会自行解决然后再提交。而另外一些团队倾向于需要知道哪些文件正由谁在编辑，这样如果两个人要编辑同一个文件，他们可以选择讨论谁在干什么而不会在提交时感到意外。本节将讨论的特点允许这种协作方式，从而保持同一时间由两个开发者编辑同一个文件。

开发人员最好有这样的习惯，编辑文件前先使用 `cvs edit`（而不是 chmod）命令，不再使用的工作目录用 `cvs release`（而不是 rm）命令来释放。但 cvs 本身没有强制要求这样去做。

如果一个开发组希望强制监视，在开发组员和服务器都是使用 cvs 1.12.10 以后的版本时，他们可以打开“advisory lock”。

要打开“advisory lock”，所有的开发人员要将”`edit -c`” 和”`commit -c`” 放入.cvsrc 文件中，除非使用监视或.cvsrc 文件里面有”cvs -r”，所有的文件都是只读。这阻止了多开发人员同时编辑一个文件 (除非使用 ‘`-f`’ 覆盖)。

Setting a watch

为了启用监视特性，首先指定哪些文件需要被监视。

Command: cvs watch on [-lR] [files]...

要求开发者在编辑指定的 files 前先执行 `cvs edit` 命令。cvs 在创建工作副本时将这些 files 设成只读，并保持到开发人员运行 `cvs edit` 命令。

当 files 包括目录名，cvs 会监视仓库中对应目录里所有文件，并对后来添加的文件也默认监视；这样允许用户设置基于每目录的通知机制。除非指定 `-l` 选项，目录里面的内容是递归处理的。如果在 /.cvsrc[9]里设置了 `-l` 选项，使用 `-R` 选项可以 i 强制递归。

如果省略 files，默认是对当前目录。

Command: cvs watch off [-lR] [files]...

检出时不创建 files 为只读；开发人员也就不再要使用 `cvs edit` 和 `cvs unedit` 命令。

处理 files 和选项的方法和 `cvs watch on` 相同。

[9]参阅 /.cvsrc

Getting Notified

用户可以告诉 cvs 你想得到关于在一个文件上所进行动作的通知。虽然可以不用 cvs watch on 命令，但如果要提醒开发人员使用 cvs edit 命令，还是应该用 cvs watch on 命令。

Command: cvs watch add [-lR] [-a action]... [files]...

将当前用户加入到 files 完成的通知列表。

- `-a` 选项指定通知给用户的 cvs 事件类型。action 类型可以是：
  - `edit` 另一个用户对被监视的文件使用 cvs edit 命令（见下文）。
  - `commit` 另一个用户提交了 files 的修改。
  - `unedit` 另一个用户放弃编辑文件（而不是提交修改）。有以下几种方式可以做此事：
    * 对该文件使用 `cvs unedit` 命令（见下文）
    * 对该文件的父目录使用 `cvs release` 命令[10]（或者递归到该文件的某层目录）
    * 删除该文件，然后用 `cvs update` 命令重建
  - all 上面的所有事件。
  - none 在上面没有。（对 `cvs edit` 命令很有用，见下文。）

`-a` 选项可以出现多次，也可以没有。如果省略，默认使用 `all`。

处理 `files` 和选项的方法和 `cvs watch on` 相同。

Command: cvs watch remove [-lR] [-a action]... [files]...

删除由 `cvs watch add` 命令确立的通知；参数相同。如果使用 `-a` 选项，只删除指定的动作的监视。

一旦通知的条件成立，cvs 会调用 notify 管理文件。编辑 notify 文件的方法与其他管理文件相同[11]。该文件也符合管理文件的语法习惯[12]，文件中每一行为一个正则表达式加一个要执行的命令。命令中包含 '%s' 用来替换所要通知的用户；其余的是通知需要使用的附加信息，它们作为命令的标准输入。标准用法是将放在 notify 文件的一行中：

```
ALL mail %s -s "CVS notification"
```

这条命令使用电子邮件通知用户。

注意，如果直接使用这种方法，用户在服务器机器上接收通知。当然，可以写一个 notify 脚本将通知指向其他的地方，但为了使用方便，cvs 允许给每个用户指定一个相关的地址。做法是在 CVSROOT 目录下创建一个 users 文件，每行的格式为 `user:value`。

[10]参阅 release。
[11]参阅 Intro administrative files。
[12]参阅 syntax。

这样 cvs 不是把被通知用户名传递给 notify，而是传递 value 中的值（一般是其它机器上的 email 地址）。

cvs 不会对用户自己做的修改进行通知。现在的检测是基于触发事件的用户名与被通知的用户名。通常，实际上监视特性只跟踪每个用户的一个编辑。或许分别监视每个目录会更有用，所以这方面将来也许改变。

Editing files

因为被监视的文件在检出时文件属性为只读，你不能直接去编辑它。为使它成可读写并通知其他人你打算进行编辑，使用 `cvs edit` 命令。一些系统把这一过程叫 checkout，但 cvs 使用该术语于获得源码的一份副本[13]，而这些系统将此称为 get 或 fetch 操作。

Command: cvs edit [-lR] [-a action]... [files]...

准备编辑工作文件 files。cvs 使这些 files 可读写，并且通知请求 edit 监视 files 的用户。

`cvs edit` 命令使用与 `cvs watch add` 同样的选项，并建立用户在 files 上临时的监视。cvs 将在用户使用 unedit 或 commit 命令后删除对这些 files 的监视。如果用户不想得到通知，可以指定 `-a none` 选项。

files 和选项的处理方法与 `cvs watch` 相同。

1.12.10 版本以后的 cvs 客户端和服务器还有两个新增的参数，用于 `cvs edit`，而不用在 `cvs watch`。第一个是 `-c`，如果有人编辑文件，它使 `cvs edit` 命令失败。这也许只对在所有开发人员的.cvsrc 中指定 '`edit -c`' 和 '`commit -c`' 有用。该特性可以被 `-f` 选项覆盖，使得多个用户可以成功编辑。

通常在做了修改后，你会使用 `cvs commit` 命令，这将检入用户的变更并使文件恢复的通常的只读状态。但如果用户决定放弃修改，或者不做任何修改，可以使用 `cvs unedit` 命令。

Command: cvs unedit [-lR] [files]...

放弃对工作文件 files 的修改，并将文件改回的所基于的仓库版本。cvs 对请求了 `cvs watch on` 的 files 修改属性为只读。cvs 通知请求了 unedit 监视 files 的用户。

files 和选项的处理方法与 `cvs watch` 相同。

如果没有使用监视，unedit 命令不会工作，而将文件从仓库恢复的方法是用 `cvs update -C file`[14]。这两种方法不完全相同；后一种还会将用户上次更新后的新的变更加进来。

[13]参阅 Getting the source。
[14]参阅 update。

在 cvs 客户/服务方式下，即使无法连到服务器也可以使用 `cvs edit` 和 `cvs unedit`；而通知将在下次正常使用 cvs 命令时进行。

Watch information

Command: cvs watchers [-lR] [files]...

列出当前监视 files 修改的用户。该报告包括被监视的文件和监视者的邮件地址。

files 和选项的处理方法与 `cvs watch` 相同。

Command: cvs editors [-lR] [files]...

列出当前工作于 files 的用户。报告包括每个用户的邮件地址，用户开始工作的时间，并主机与包含该文件的工作目录路径。

files 和选项的处理方法与 `cvs watch` 相同。

Watches Compatibility

如果用户使用监视特性于仓库，将在仓库里创建 CVS 目录，用来存储该目录的监视信息。如果在 cvs1.6 或者更早的版本中使用该仓库，会得到下列错误信息（全部在一行中）：

```
cvs update: cannot open CVS/Entries for reading:
No such file or directory
```

然后操作中断退出。为了使用监视特性，必需将在本地和服务器上使用该仓库将 cvs 的全部副本都升级。如果无法升级，使用 `watch off` 和 `watch remove` 删除所有的监视，然后将恢复到 cvs1.6 可以正常处理了的状态。

21.2.7 Choosing a model

限制或不限制检出？

限制与非限制检出各自有正反两方面的理由。而且大多只是看法不同的问题，或者同一工作用于不同类型的工作组，这里只是一些问题的概述。组织开发团队有很多种方法。cvs 不会强制采用某种组织结构。它是一个可以用不同方法使用的工具。

限制检出是非常低效的。如果两个人试图编辑一个文件的不同部分，没有理由要阻止他们那样做。还有一种情况很常见，某人打算编辑一个文件，他检出并加锁，但后来忘记了解锁。

习惯限制检出方法的人们，他们转到非限制检出下工作时，经常抱怨冲突发生得太频繁，要解决冲突很麻烦。而对另一些有经验的团队，冲突却很少发生，即使有也很容易解决。

出现严重的冲突非常罕见，除非两个开发者对某个设计的一段代码实现意见不一致；而这种情况首先意味着这个开发团队人员之间没有很好地沟通。不论在任何源码管理机制下，开发者必需遵循系统的概要设计；符合这点，重叠的修改通常很容易合并。

有些情况下，非限制检出明显不适用。如果没有被管理文件（比如字处理文件或计算机辅助设计程序编辑的文件）相应的合并工具，还有不想要合并对使用可合并数据格式的程序的改变，与其解决冲突还不如防止冲突，这时就该使用限制检出。

监视 Watches 特性可以认为是限制检出与非限制检出的中间方案。当你打算编辑一个文件时可以先了解谁正在编辑它。这种方法要好于系统阻止多人去编辑，它告诉你当前的状况，然后让你自己判断会不会出问题。所以，一些开发团队可以考虑监视同时采用限制和非限制检出作为最佳解决方案。

1.12.10 以后版本的 cvs 客户端和服务器，用户可以通过将 '`edit -c`' 和 '`commit -c`' 放入所有开发人员的.cvsrc 文件打开 "advisory locks"。这样，如果其他人在编辑，`cvs edit` 将失败。并且，如果没有通过 `cvs edit` 进行注册，使用 `cvs commit` 也会失败。结合默认只读检出将更有效，这通过将 '`cvs -r`' 放在所有的.cvsrc 中打开监测实现。

21.3 Revision management

读到这里，你可能对 cvs 能做什么有了相当的了解。这章讨论一些使用中仍需考虑的问题。

如果使用 cvs 的仅是自己一个人，可以跳过这一章。本章讨论的是多人使用同一仓库遇到的问题。

21.3.1 When to commit

你的团队应该决定使用何种提交机制。依据你自己 cvs 的经验在多种机制中选择出适合的方式。

如果提交文件太快，你可能甚至提交无法编译的文件。假如你的合作者更新了他的工作源文件而包括了你的错误文件，他也将无法编译该代码。而另一种情况，如果你很少提交，别人又无法得到你对代码所做的改进，并使冲突的几率增大。

通常应该仅在确认可以编译后才可提交。有时要求通过测试工具的检查。可以利用 commitinfo 文件[15]强制实施这种机制，但采用前应慎重考虑。对开发环节控制过多会造成管理死板，反而会降低我们开发出软件这一目标的工作效率。

[15]参阅 commitinfo

21.4 Keyword substitution

只要你在工作目录中编辑源代码，你可以随时使用 '`cvs status`' 和 '`cvs log`' 命令了解文件的状况。一旦你从开发环境中导出了这些文件，将很难识别这些文件的版本。

cvs 提供一种关键字替换（keyword substitution），或者叫作关键字扩展（keyword expansion），机制来帮助识别这些文件。在文章中嵌入 $keyword$ 和 $keyword:...$ 形式的字符串，以后在获得文件新版本时将自动被 $keyword:value$ 字串所替代。

21.4.1 Keyword list

这是关键字列表：

- $Author$
  检入该版本的用户登录名。
- $CVSHeader$
  标准的头部 (类似去掉 CVS 根的 $Header$)。包括 rcs 文件的全路径、版本号、日期 (UTC)、作者、状态、加锁人（如果有锁)。在使用 cvs 中文件通常不用加锁。
  注意，该关键字是新加到 cvs 的，假如旧的文件中有 $CVSHeader$ 并有其他意义，会带来一些问题。可以在 CVSROOT/config 中用 KeywordExpand=eCVSHeader 来排除此关键字[16]。
- $Date$
  该版本被检入的日期与时间（UTC)。
- $Header$
  标准的 header 包括 rcs 文件的全路径、版本号、日期（UTC)、作者、状态、加锁人 (如果有锁)。在使用 cvs 中文件通常不用加锁。
- Id
  除了 rcs 文件不包括路径，其余和 $Header$ 相同。
- $Name$
  检出此文件所用的标签名。该关键字只在检出时显式加上标签时扩展。比如，运行 `cvs co -r first` 命令时，关键字扩展为 '`Name: first`'。
- $Locker$
  锁定版本的用户登录名 (如果没有加锁此项为空，一般就如此，除非使用 cvs admin -l 加锁)。
- Log

[16]参阅 Configuring keyword expansion。

日志信息在提交时提供，前面是一个 header 包括 rcs 文件名、版本号、作者、日期 (UTC)。已有的日志信息不会被替换。相反，新日志信息将插在 $Log:...$ 之后。默认，每一新行前面使用同样的 *Log* 关键字前的字符串, 除非在 CVSROOT/config 里面设置 MaxCommentLeaderLength。
例如，文件文件包含：

```
/* Here is what people have been up to:
 *
 * $Log: frob.c,v $
 * Revision 1.1 1997/01/03 14:23:51 joe
 * Add the superfrobnicate option
 *
 */
```

新增的行扩展 Log 关键字后前面也会带有 ‘ * ’ 字符。与以前版本的 cvs 和 rcs 不同，rcs 文件中的不再使用 comment leader。Log 关键字将会在源文件中累积日志纪录。有些原因可能会造成问题。
如果 Log[17]关键字的前缀超出 MaxCommentLeaderLength 长度，CVS 将跳过关键字扩展，除非在 CVSROOT/config 中设置 UseArchiveCommentLeader，并且有 ‘comment leader’ 设置在 RCS archive 文件，使得 comment leader 替代使用[18]。

- $RCSfile$
  不带路径的 RCS 文件名。
- $Revision$
  该文件的修订版本号。
- $Source$
  RCS 文件的完整路径。
- $State$
  赋予版本的状态。可以通过使用 `cvs admin -s` 命令设置状态[19]。
- Local keyword
  CVSROOT/config 文件中的 LocalKeyword 选项可以被用作其他关键字的别名：Id, $Header$ or $CVSHeader$。例如，在 CVSROOT/config 中包含 LocalKeyword=MYBSD=CVSHeader 这样一行，带有 $MYBSD$ 的文件会像 $CVSHeader$ 一样被扩展。如果 src/frob.c

[17]了解更多默认的 Log 替换配置，参阅 config。
[18]为了了解更多关于 RCS archive 中设置 comment leader，参阅 admin。
[19]参阅 admin options。

文件包含这个关键字，像下面这样：

```
/*
 * $MYBSD: src/frob.c,v 1.1 2003/05/04 09:27:45 john Exp$
 */
```

许多仓库使用“local keyword”这类特性。cvs 的一个旧补丁提供 LocalKeyword 特性，它们使用称为“custom tag”或“local tag”特性的 tag= 选项。它被用于它们称为 tagexpand= 选项。在 cvs 中另一个选项为 KeywordExpand[20]。

知名的项目中有这些例子：$FreeBSD$, $NetBSD$, $OpenBSD$, $XFree86$, $Xorg$.

使用它的优点在于可以使用自己的版本信息而不破坏上面版本 (它们可能使用不同的 local keyword 或标准关键字)。使得 bug 报告可以更恰当地识别第三方的错误源，以及减少导入新版本时产生的冲突数。

除 local keyword 之外所有的关键字都可以在 CVSROOT/config 文件中用 KeywordExpand 选项停用[21]。

21.4.2 Using keywords

在文件中使用关键字只需简单的把相关的文本字符串，比如 Id，放到文件中，然后提交该文件。cvs 会自动[22]扩展字符串作为提交操作的一部分。

通常将 Id 字符串嵌入文件中以致可以穿过生成的文件。比如，你管理的是计算机程序的源代码，你可以初始化一个变量包含这个字符串。或一些 C 编译器可能提供一个 #pragma ident 指令。或者一个文档管理系统可以提供的一个方法将字符串传递给生成的文件。

ident 命令（rcs 包里面有）可以被用来从文件中将关键字和值抽取。它可以处理文本文件，对从二进制文件抽取关键字也很有用。

```
$ ident samp.c
samp.c:
    $Id: samp.c,v 1.5 1993/10/19 14:57:32 ceder Exp $
$ gcc samp.c
$ ident a.out
a.out:
    $Id: samp.c,v 1.5 1993/10/19 14:57:32 ceder Exp $
```

[20]参阅 Configuring keyword expansion。
[21]参阅 Configuring keyword expansion 获得详细信息。
[22]或者，更准确地说，作为更新的一部分在提交之后自动执行。

Sccs 是另一种流行的版本控制系统。它的 what 命令类似于 ident 命令并用于同样的目的。一些场所使用 sccs 而不用 rcs。因为 what 命令查找的是 @(#) 字符，所以很容易包含所有命令检测的关键字。只需将魔术 sccs 前缀加上即可，如：

```
static char *id="@(#) $Id: ab.c,v 1.5 1993/10/19 14:57:32 ceder Exp $";
```

21.4.3 Avoiding substitution

关键字替换也有缺点。有时你需要的是 '$Author$' 字串出现在文件中，而不希望被 cvs 当成关键字来解释并替换成类似 '$Author: ceder$' 的形式。

不幸的是没有一个可选择的关闭关键字替换的方法，只能通过 '`-ko`'[23]来完全关闭。

一些情况下你可以避免源文件中的关键字，但它能在最终产品中出现。比如，本手册的源码里用 '$@asis{}Author$' 来代替最终出现的 '$Author$' 字串。用 nroff 和 troff 嵌入空字符 & 到关键字里面也有同样的效果。

还可以在 CVSROOT/config 文件里面使用 KeywordExpand 选项来指定一个明确的列表来包含或不包含某些关键字[24]。该特性主要用在 LocalKeyword 中，参阅 Keyword list。

21.4.4 Substitution modes

每一个文件保存有一个默认的替换模式，每个文件的工作目录副本也有一个替换模式。前者通过用 '`-k`' 选项的 `cvs add` 和 `cvs admin` 命令设置；后者用 '`-k`' 或 '`-A`' 选项的 `cvs checkout` 或 `cvs update` 命令设置。`cvs diff` 命令也有一个 '-k' 选项。参阅 Binary files 和 Merging and keywords 获得这些例子。

可用的模式为：

- '`-kkv`'
  使用默认形式产生关键字字符串，如。$Revision: 5.7 $ 对于 Revision 关键字。
- '`-kkvl`'
  类似 '`-kkv`'，如果指定的版本现在被加锁，还需要插入加锁者的名字。锁名与使用的 `cvs admin -l` 命令相关。
- '`-kk`'
  在关键字字符串中只生成关键字名；忽略其值。例如，对于关键字 Revision，生成的字符串是 $Revision$ 而不是 $Revision: 5.7 $。这一选项在比较版本差异时忽略关键字替换的影响非常有用 (参阅 Merging and keywords)。

[23]参阅 Substitution modes。
[24]参阅 Configuring keyword expansion。

- '-ko'
  生成旧的关键字字符串，在工作中的文件与检入前一样。例如：对于关键字 Revision，如果检入前字符串是 $Revision: 1.1 $，则生成的字串与以前相同而不会是 $Revision: 5.7 $。
- '-kb'
  类似 '-ko' 选项，并阻止换行字符的转换，这种转换是因为换行符在仓库（只是换行）的形式与客户机使用的操作系统不同。对有些系统，比如 Unix，换行符就是行中止符，这样就和 '-ko' 没什么区别[25]。在 cvs 版本 1.12.2 及之后的 '-kb'，不会被在命令行中使用 '-k' 选项的 `cvs add`、`cvs admin` 或 `cvs import` 跨越。
- '-kv'
  只为关键字符串生成关键字值。例如，对于关键字 Revision，生成的字串是 5.7，而不是 $Revision: 5.7 $。这种方法有助于一些难以从 $Revision: $ 这样的字符串中剥去关键字分界符的编程语言生成文件。但是，一旦删除关键字名，以后将不能执行关键字替换，所以应小心使用该选项。
  通常将 '-kv' 选项与 `cvs export` 命令配合使用—参阅 export。但请注意，它不能正确导出的文件中的二进制文件。

21.4.5 Configuring keyword expansion

在一个包含第三方软件的销售商分支的仓库中，配置 CVS 使用本地的关键字代换 Id 或 $Header$ 很有用处。在一些真正的项目中有实例：$Xorg$、$XFree86$、$FreeBSD$、$NetBSD$、$OpenBSD$，甚至 $dotat$。它的优点在于可以 i 在文件中包含本地的版本信息而不破坏上游版本信息 (它们可能使用不同的 local keyword 或标准关键字)。在这些场合中，希望能只使用配置的本地关键字而停用其它所有的关键字展开。

文件 CVSROOT/config 中的 KeywordExpand 选项可用于明确指定排除或包含某个关键字，或关键字列表。该列表可包含配置的 LocalKeyword。

KeywordExpand 选项后跟随 =，其下一个字符可能是 i 来表示开始一个包含列表或是一个 e 来表示开始一个排除列表。如果以下的行被加到 CVSROOT/config 文件中：

```
# Add a "MyBSD" keyword and restrict keyword
# expansion
LocalKeyword=MyBSD=CVSHeader
KeywordExpand=iMyBSD
```

则只有 $MyBSD$ 会被扩展。可能用列表。此例子：

[25]关于二进制文件的更多信息，参阅 Binary files。

```
# Add a "MyBSD" keyword and restrict keyword
# expansion to the MyBSD, Name and Date keywords.
LocalKeyword=MyBSD=CVSHeader
KeywordExpand=iMyBSD,Name,Date
```

将允许 $MyBSD$、$Name$ 和 $Date$ 被扩展。

也可以用以下的办法配置排除列表：

```
# Do not expand the non-RCS keyword CVSHeader
KeywordExpand=eCVSHeader
```

这让 cvs 忽略新近引入的 $CVSHeader$ 关键字而保留其他的。排除的项还可以包括 RCS 的关键字列表，但会令需要 RCS 关键字扩展的用户费解，所以要恰当设置以该法配置的仓库的用户期望。

如果要不扩展任何 RCS 关键字，并且不用任何地方的 `-ko` 标志，管理员可以在 CVSROOT/config 中添加下面这行禁用全部关键字扩展：

```
# Do not expand any RCS keywords
KeywordExpand=i
```

这会使希望使用 Id 这样的 RCS 关键字扩展的用户迷惑，所以要恰当设置如此配置的仓库的用户期望。

有一点要注意，针对 KeywordExpand 和 LocalKeyword 特性的补丁已经存在很长时间了。但使用 `tag=` 和 `tagexpand=` 关键字的这些特点实现的补丁并不被识别。

21.4.6 Log keyword

对关键字 Log 存在一些争议。即使没有使用 Log 关键字，只要你在开发系统中工作，即使不用 Log 关键字获取相应的信息也是很容易的——执行 `cvs log` 命令即可。一旦导出了文件，历史信息或许没有多大用处。

更严重的认为，cvs 在合并分支到主干时对 Log 条目处理的并不好。因为合并过程中总会有冲突。

人们总是倾向于“更正”文件中的 log 条目（纠正书写失误或者事实错误）。如果完成，`cvs log` 的信息将与文件内部的信息不一致。这也许在实际的文件中不是什么问题。

还有的建议，如果一定要用，Log 关键字应插在文件的最后，而不是文件头。这样长的更新信息就不会干扰每天对源文件的浏览。

21.5 Tracking sources

如果修改了一个程序以适应自己的需要，也许想将这种修改也加到该程序的下一个版本中。cvs 可以帮用户完成这一任务。

在 cvs 术语中，程序的提供者称为vendor。从 vendor 得到的未经修改的发行版检入到它自己的分支，vendor branch。cvs 保留分支 1.1.1 用于此。

当用户修改了源码并提交，版本号将回到主干上。当 vendor 发行了一个新版本，用户将其提交到 vendor 分支，然后把修改部分复制到主干上。

使用 `import` 命令可以创建和更新 vendor 分支。当导入一个新文件时，(通常) vendor 分支创建了 'head' 版本，这时检出该文件的副本得到的是这一版本。如果在本地做了修改并提交，则 'head' 版本移动到主干上。

21.5.1 First import

在第一次提交源码时使用 `import` 命令。使用 import 命令跟踪第三方源码时，vendor tag 和 release tags 很有用处。vendor tag 是分支的符号名（除非使用 '`-b branch`' 标志——参阅 Multiple vendor branches，分支号总是 1.1.1)。release tags 是特定发行版的符号名，比如 'FSF_0_04'。

注意，import 命令不改变用户执行时所处的目录。特别是，它不会将该目录作为 cvs 工作目录；如果想修改这些源码，先导入之再检出到另一个不同目录中[26]。

假设我们有 wdiff 程序的源码，目录为 wdiff-0.04，打算做一些私有的修改，并能用于以后新的发行版本中。将源码导入仓库开始：

```
$ cd wdiff-0.04
$ cvs import -m "Import of FSF v. 0.04" fsf/wdiff FSF_DIST WDIFF_0_04
```

上面的例子中将 vendor tag 命名为 'FSF_DIST'，唯一的 release tag 设为 'WDIFF_0_04'。

21.5.2 Update imports

当新版本的源码到达后，使用我们当初初始化仓库时同样的 import 命令将它们导入的仓库中。此时，差异仅是指定一个不同的发行版标记：

```
$ tar xfz wdiff-0.05.tar.gz
$ cd wdiff-0.05
$ cvs import -m "Import of FSF v. 0.05" fsf/wdiff FSF_DIST WDIFF_0_05
```

[26]参阅 Getting the source。

警告：如果用户使用的发行版标记已经存在于仓库中，import 将删除所有的文件而不经检测。

对没有做过本地修改的文件，新版本成为 head 版本。如果有修改，import 命令将发出警告必需将变化合并到主干上，提示使用 'checkout -j' 命令来完成：

```
$ cvs checkout -jFSF_DIST:yesterday -jFSF_DIST wdiff
```

上面命令检出最新修订版的 'wdiff'，并将从昨天在分支 'FSF_DIST' 上做的修改合并到工作目录。如在合并时有冲突，解决方法跟平时相同[27]，然后才可以提交修改的文件。

但是，最好使用两个发行标签名而不是如上采用的分支上的日期：

```
$ cvs checkout -jWDIFF_0_04 -jWDIFF_0_05 wdiff
```

这种方法更好的原因是上述采用日期的办法假设每天不会导入多个发行版。更重要的是，使用 `release tags` 让 cvs 检测两个 vendor 发行版之间删除的文件并为删除而作标记。因为 import 命令没法检测出删除的文件，应该象这样做合并即使 import 不告诉我们要做。

21.5.3 Reverting local changes

即使做了本地修改，也可以将所有本地的修改恢复，回到最近 vendor 发行版本上，办法是把“head”修订版全部文件改变到 vendor 分支。例如，检出的源码副本放在 /work.d/wdiff，并且你想要使该目录中的全部文件恢复到 vendor 的版本，你可以打字：

```
$ cd ~/work.d/wdiff
$ cvs admin -bFSF_DIST .
```

使用 '-bFSF_DIST' 时在 '-b' 选项后面不能有空格[28]。

21.5.4 Binary files in imports

使用 '-k' 选项告诉 import 哪些文件是二进制的[29]。

21.5.5 Keywords in imports

导入的源码中也许会包含关键字[30]。比如，vendor 也用了 cvs 或其他采用类似关键字展开的系统。如果使用默认的方法导入文件，关键字展开会用用户自己 cvs 副本的而不

[27]参阅 Conflicts example。
[28]参阅 admin options。
[29]参阅 Wrappers。
[30]参阅 Keyword substitution。

是 vendor 的。也许保持 vendor 的关键字展开替换更好些，这样此信息提供从 vendor 导入的源码的信息。

为了维持使用 vendor 的关键字扩展，首次导入文件的时候要用 `cvs import` 命令的 '`-ko`' 选项。这将完全关闭文件中的关键字替换，如果用户需要有选择性的方式，考虑采用 `cvs update` 命令的 '`-k`' 选项，或者用 `cvs admin`。

21.5.6 Multiple vendor branches

到目前为止，所有的例子都假设只从一个 vendor 获得源码。有些情况下，可能会从不同的地方获得源码。例如，一个项目可以有不同的开发人员和组来修改软件。这有种种方法处理，但是在某些情况下，有一大堆源码树摆在周围而你首先想要做的只是把它们全部存放在 cvs 中这样你至少可以使它们放在一处。

为了处理多个 vendor 的情况，需要在 `cvs import` 命令中指定 '`-b`' 选项。它的参数用来说明导入到哪个 vendor 分支。默认是 '`-b 1.1.1`'。

例如，现有两开发组，red 和 blue，给你源码。你想导入 red 组的源码到分支 1.1.1 并用 vendor 标签 RED。用户想导入 blue 组的源码到分支 1.1.3 并用 vendor 标签 BLUE。这样可能使用的命令如下：

```
$ cvs import dir RED RED_1-0
$ cvs import -b 1.1.3 dir BLUE BLUE_1-5
```

注意，如果用户的 vendor tag 与 '-b' 选项不匹配，cvs 不会做检测！例如，

```
$ cvs import -b 1.1.3 dir RED RED_1-0
```

小心；这种不匹配会埋下隐患。我们不认为在这里指定一种机制有什么用处，但是否发现这样的用处了吗。cvs 以后版本会认为这种方式是一种错误。

21.6 Builds

简介中曾提到，cvs 不包含将源代码构建成软件的功能。本节只描述构建系统与 cvs 之间的交互操作。

一个常见的问题，特别是对 rcs 熟悉的人们，是如何得到源代码的最新副本。对 cvs 答案有两方面（tow-fold）。

首先，因为 cvs 可在目录内递归，所以不必修改 Makefile(或其他配置文件) 以使每个文件都更新。所以，先使用 `cvs -q update` 命令，再用 make 或 build 工具中别的命令即可。其次，在完成工作之前，不必去考虑得到其他人做的修改的副本。建议的作法是先

更新源代码然后修改、构建和测试你的修改，然后提交你的源码（如果需要先更新）。通过周期性地（修改之间，如刚才的描述）更新整个源码树，就能保证你的源代码足够新。

还有一个常见的需要是纪录特定构建中源代码的版本。这种功能称为bill of materials或类似叫法。在 cvs 中最好的解决方法是使用 tag 命令来纪录给定构建的版本[31]。

大多数简单使用 cvs 的方式下，每个开发人员会有一份整个源码树的副本用于特定的构建。如果源码树比较小，或者开发人员地理位置分散，这是一种比较合适的作法。对于大的项目，应当将其分成小的可以独立编译的子系统，它们可以内部发布，这样每个开发人员只需检出自己工作的相应的子系统。

另一种方式是创建一种结构，开发人员对部分文件有自己的拷贝，其他文件从中心获得。在许多系统上可以使用符号链接，或者使用 make 的 VPATH 特性。一个专为此设计的 build 工具可以帮你处理，它就是 Odin[32]。

21.7 Special Files

通常，cvs 只处理一般文件。它假定项目中的每个文件都是永久的，文件必须能被打开、阅读、关闭等等。另外，cvs 忽略文件的许可权和所有权，把这些问题留给开发者在安装时解决。换句话说，不能把一个设备“检入”仓库中；如果不能打开设备文件，那么 cvs 将拒绝处理它。在仓库中进行处理时，文件将丧失其所有权和许可权。

[31]参阅 Tags。

[32]参考 ftp://ftp.cs.colorado.edu/pub/distribs/odin。

Chapter 22

CVS Commands

本附录描述了 cvs 命令的整体结构，同时详细地描述了一部分命令[1]。

22.1 Structure

全部的 cvs 命令的格式如下 cvs:

```
cvs [ cvs_options ] cvs_command [ command_options ] [ command_args ]
```

- cvs
  cvs 的程序名。
- cvs_options
  一些作用于所有 cvs 命令的选项。将会在下面具体解释。
- cvs_command
  不同的命令。一些命令可以使用别名替代; 别名将会在那些命令的参考手册部分注明。只有以下两种情况下可以省略 'cvs_command'：'`cvs -H`' 显示一个 CVS 的命令列表，'`cvs -v`' 显示 cvs 的版本信息。
- command_options
  作用于特定命令的选项。
- command_args
  命令的参数。

不幸的是 cvs_options 和 command_options 很容易混淆。当给定一个 cvs_option 的时候, 某些选项只能作用于特定的命令。当给定一个 command_option 它可能有不同的含义，并且多个命令都可以使用相同的选项。换句话说，对待上面的说明不要太认真。要以命

[1] cvs 命令快速参考，参阅 Invoking CVS。

令参考手册的具体说明为依据。

22.2 Exit status

标明 CVS 成功或失败

cvs 通过设置 exit status 可以给调用环境传递成功或失败信息。侦测退出状态的确切方法因操作系统而异。例如在 Unix 中的 shell 脚本变量 '$?'，如果最后命令返回成功退出状态，该变量值为 0，如果该值大于 0，则为失败。

若 cvs 执行成功，它返回成功状态；如果有错误，它打印错误信息并返回失败状态。但 cvs diff 命令是个例外。如果比较结果相同，将返回成功状态，返回失败状态则表示有不同或者出错。由于这种方式无法提供一个良好的检测错误途径，将来 cvs diff 命令或许会改成与其他 cvs 命令相同的方法。

22.3 /.cvsrc

/.cvsrc 文件的默认选项

有一些 command_options 使用得非常频繁，需要采用别名或其他方式来确保指定这些选项。一种情况（事实上，正是此原因导致使用.cvsrc）) 是人们发现默认的 'diff' 输出难以阅读，相对而言，上下文 diffs 或 unidiffs 都比它容易理解。

/.cvsrc 文件是一种将默认选项加给 cvs_commands 的方法，用来取代别名或其他 shell 脚本的方法。

/.cvsrc 的格式很简单。当 cvs_command 被执行时，该文件按相同的命令名搜索每一行。若匹配，该行剩余部分将被分隔（空格）成各选项，添加到命令行的所有参数之前。

如果一个命令有两个名称（比如，checkout 和 co），其正式名称，而不是命令行中使用的，将被用来进行比较。于是如果用户的一个 /.cvsrc 文件内容为:

```
log -N
diff -uN
rdiff -u
update -Pd
checkout -P
release -d
```

命令 '`cvs checkout foo`' 将有 '`-P`' 选项被添加到参数中，对于 '`cvs co foo`' 也如此。

上面的例子，‘`cvs diff foobar`’命令的输出为 unidiff 格式。‘`cvs diff -c foobar`’提供上下文 diffs。如要获得“旧的” diff 的输出格式，会有些麻烦，因为 diff 并没有一个选项适用于“旧的”格式，所以我们需要用‘`cvs -f diff foobar`’。

还可以使用 cvs 作为命令名来指定全局选项[2]。例如，在.cvsrc 中添加这样一行

```
cvs -z6
```

将迫使 cvs 采用压缩级别 6。

22.4 Global options

cvs_command 左侧给出的选项

有效的‘cvs_options’（作用于左边的‘cvs_command’）有：

- `--allow-root=rootdir`
  可以为多次调用指定一个合法的 cvsroot 目录。也导致 CVS 预先解析每个特定根目录的配置文件，在配置写代理的时候很有用[3]。
- `-a`
  验证服务器和客户端之间的所有通信。只在 cvs 客户端有效。到写此帮助时为止，这个选项只会在使用 GSSAPI 连接时才有效[4]。验证可以防止某些通过截取和篡改 tcp 连接而进行的攻击。
- `-b bindir`
  在 cvs 1.9.18 和更早的版本，这个用来指定 rcs 程序在 bindir 目录中。当前版本的 cvs 不运行 rcs 程序; 为了兼容性这个选项仍然可以使用，但是不会产生任何效果。
- `-T tempdir`
  使用 tempdir 目录作为临时文件的存放目录。将会覆盖 $TMPDIR 环境变量的设置和任何预编译目录。这个参数可以用绝对路径指定[5]。
- `-d cvs_root_directory` 使用 cvs_root_directory 作为仓库的根目录路径名。覆盖 $CVSROOT 环境变量的设置[6]。
- `-e editor` 使用 editor 作为编辑人来记录到修订版日志信息中。覆盖 $CVSEDITOR 和 $EDITOR 环境变量的设置[7]。

[2]参阅 Global options。

[3]参阅 Password authentication server & Write proxies。

[4]参阅 GSSAPI authenticated。

[5]当在客户端或者服务器运行时，‘`-T`’只影响本地处理；在客户端指定‘`-T`’将不会对服务器造成影响，反之亦然。

[6]参阅 Repository。

[7]更多的信息，参阅 Committing your changes。

- `-f`
  不要读取 /.cvsrc 文件。因为 cvs 选项集合的非正交性特性，所以这个选项会被经常使用。例如，'cvs log' 选项 '`-N`'(关闭显示标签名) 没有一个相应的选项来打开显示。所以如果在 /.cvsrc 中为 'log' 指定了 '`-N`' 选项，可能需要使用 '`-f`' 来禁止此选项来显示标签名。
- `-H`
- `--help`
  显示关于特定 'cvs_command' 的使用信息 (但不实际执行此命令)。如果没有指定命令名，'`cvs -H`' 显示全部 cvs 的帮助，包括其他帮助选项的列表。
- `-R`
  打开只读仓库模式。这将允许从一个只读仓库进行检出操作，例如一个匿名 CVS 服务器，或者一个 cd-rom 仓库。就像设置了 CVSREADONLYFS 环境变量一样的效果。使用 '`-R`' 也能加快在 NFS 上面的检出速度。
- `-n`
  不改变任何文件。尝试执行 'cvs_command'，但是只是显示执行结果; 而不删除，更新，或者合并任何存在的文件，或者建立任何新文件。
  注意 cvs 不确定在不使用 '`-n`' 的时候也能得到完全一样的输出信息。有些时候输出的信息会相同，但是有些时候 cvs 会跳过某些处理，而导致输出信息不同。
- `-Q`
  完全关闭命令执行的输出信息显示; 只有当命令出现严重问题的时候才会显示信息。
- `-q`
  关闭部分命令执行的输出信息显示; 普通的消息，比如反馈对子目录的递归的处理的消息将被关闭。
- `-r`
  使新的工作文件成为只读状态。和设置了 $CVSREAD 环境变量一样[8]。默认的工作文件状态是可写的，除非使用了监视方式[9]。
- `-s variable=value`
  设置一个用户变量[10]。
- `-t`
  跟踪程序执行; 显示 cvs 执行步骤的信息。配合 '`-n`' 一起使用对于研究那些不熟悉

[8]参阅 Environment variables。
[9]参阅 Watches。
[10]参阅 Variables。

的命令的执行效果有非常大的帮助。

- `-v`
- `--version`
  显示 cvs 的版本和版权信息。
- `-w`
  使新建立的文件为读写状态。覆盖 $CVSREAD 环境变量的设置。文件默认是以读写状态被创建的，除非 $CVSREAD 被设置或者指定 '`-r`' 选项。
- `-x`
  加密服务器和客户端之间的所有通信。只在 cvs 客户端有效。到写此帮助时为止，这个选项只会在使用 GSSAPI 连接[11]或者 Kerberos 连接[12]时才有效。允许加密意味着这个通信也同时被验证。默认是不支持加密模式的; 必须使用特殊配置选项，`--enable-encryption` 来编译 cvs。
- `-z gzip-level`
  设置压缩级别。有效的级别是 1（高速，低压缩率）到 9（低速，高压缩率），或者 0 来禁止压缩（默认值）。只在 cvs 客户端有效。

22.5 Common options

cvs_command 右侧给出的选项

这一节来说明 'command_options'，它通用于一些 cvs 命令。这些选项总是出现在 'cvs_command' 的右侧。不是所有的命令都支持这些选项，每个选项只用在有意义的命令上。但是，当一个命令具有这些选项的时候，它和其余命令总是起着相同的作用[13]。

注意：'history' 命令是个例外；它支持的许多选项，即使是标准的选项也会有冲突。

- `-D date_spec`
  使用不迟于 date_spec 的最新版本。date_spec 是单一参数，指定一个过去的日期。当使用这种方法生成一个源文件的拷贝时，这是一个 sticky 标签; 也就是说，当使用 '`-D`' 得到的工作文件，cvs 会记住所指定的日期，以后在同一个目录里面更新时，仍会使用这个日期[14]。

  '`-D`' 可用于 annotate，checkout，diff，export，history，ls，rdiff，rls，rtag，tag 和 update 命令[15]。

[11]参阅 GSSAPI authenticated。
[12]参阅 Kerberos authenticated。
[13]单独列在命令中的其他命令选项，如果用于不同的 cvs 命令也许会有不同的意义。
[14]了解更多的粘性标签/日期信息，参阅 Sticky tags。
[15]history 命令使用这个选项时有不同的作用，参阅 history options。

全面了解 cvs 中可以使用的日期格式，参见 Date input formats。
要记住需要将这些 '-D' 的参数用引号标明，避免 shell 将参数分割解释。采用 '-D' 标记的命令如下:

```
$ cvs diff -D "1 hour ago" cvs.texinfo
```

- `-f`
  当为 cvs 命令指定一个日期或标签时，通常会忽略不包括该标签的文件（或者在所要日期之前不存在)。使用 '-f' 选项可以让用户在这些文件不匹配标签或日期时也能取得（使用这些文件的最新版本)。
  注意，即使使用 '-f' 选项，所指定的标签也必须存在（就是说存在于一些文件上，但不必是所有文件)。如果敲错了标签名，cvs 还是会给出错误信息。
  '-f' 可用于：annotate，checkout，export，rdiff，rtag 和 update。
  警告：commit 和 remove 命令也有 '-f' 选项，它们有不同的含义[16]。
- `-k kflag`
  改变不同于 '-kb' 默认处理 RCS 关键字的方式[17]。与 checkout 或 update 命令一起使用这个选项，它是属于粘性的，也就是说，在 checkout 或 update 命令中使用这个选项，cvs 将这些文件与 kflag 关联起来，以后同一个文件上持续使用 kflag，直到换用其他方式。
  '-k' 选项用于 add，checkout，diff，export，import 和 update 命令。
  警告：CVS 1.12.2 之前的版本，'-k' 标志覆盖指定二进制文件的 '-kb' 标志。这会损坏二进制文件[18]。
- `-l`
  只作用于本目录，不递归到子目录。
  用于下列命令：annotate，checkout，commit，diff，edit，editors，export，log，rdiff，remove，rtag，status，tag，unedit，update，watch 和 watchers。
- `-m message`
  使用 message 作为日志信息，而不调出一个文本编辑器。
  适用于下列命令：add，commit 和 import。
- `-n`
  不执行任何 tag 程序。外部程序可在模块数据库[19]里指定，本命令忽略它们)。
  注意：此选项与 'cvs -n' 不同，它可以位于 cvs 命令的左侧!

[16]参阅 commit options 和 Removing files。
[17]KFLAG 的含义参阅 Keyword substitution。
[18]参阅 Merging and keywords，进一步了解。
[19]参阅 modules。

适用于 checkout，commit，export 和 rtag 命令。

- `-P`
  去除空目录。参阅 Removing directories。
- `-p`
  将源码库中取得的结果由管道至标准输出，而不是写到当前工作目录中。适用于 checkout 和 update 命令。
- `-R`
  用递归方式处理目录。除了 ls & rls，cvs 所有的命令都是默认打开。
  适用于下列命令：annotate，checkout，commit，diff，edit，editors，export，ls，rdiff，remove，rls，rtag，status，tag，unedit，update，watch 和 watchers。
- `-r tag`
- `-r tag[:date]`
  使用指定的 tag 的参数 (以及命令接受的 date 参数)，代替默认的 head 作为所需的版本。与 tag 或 rtag 定义的标签一起，有两个特殊的标签存在于源码库中: 'HEAD' 指仓库中最新的版本，'BASE' 指最后检出到工作目录的版本。
  用 checkout 或 update 命令配合这个选项生成的源码拷贝是粘性的: cvs 会记住这个标签，并继续用在以后的更新命令中，除非你特别指定另外一个[20]。
  标签可以是字符代号或者数字[21]，或者是分支名[22]。当 tag 是分支名，一些命令接受可选的 date 参数指定分支上给定日期的修订版。
  全局选项 '`-q`' 经常和命令 '`-r`' 选项一起使用，当 rcs 文件不包括指定标签时，它不显示相应的警告信息。
  注意: 这个选项与全局选项 '`cvs -r`' 不同，那个出现在 cvs 命令的左侧!
  '`-r tag`' 可用于 commit 和 history 命令。
  '`-r tag[:date]`' 可用于：annotate, checkout, diff, export, rdiff, rtag 和 update 命令。
- `-W`
  指定需要被过滤的文件名。可以循环使用这个选项。规格与.cvswrappers 文件中指定的文件名模板相同。
  可用于下列命令：import 和 update。

[20]了解粘性标签/日期的信息，参阅参阅 Sticky tags
[21]参见 Tags
[22]见 Branching and merging

22.6 Date input formats

日期规格允许的格式

首先我们引用一段话:

Our units of temporal measurement, from seconds on up to months, are so complicated, asymmetrical and disjunctive so as to make coherent mental reckoning in time all but impossible. Indeed, had some tyrannical god contrived to enslave our minds to time, to make it all but impossible for us to escape subjection to sodden routines and unpleasant surprises, he could hardly have done better than handing down our present system. It is like a set of trapezoidal building blocks, with no vertical or horizontal surfaces, like a language in which the simplest thought demands ornate constructions, useless particles and lengthy circumlocutions. Unlike the more successful patterns of language and science, which enable us to face experience boldly or at least level-headedly, our system of temporal calculation silently and persistently encourages our terror of time. ...

It is as though architects had to measure length in feet, width in meters and height in ells; as though basic instruction manuals demanded a knowledge of five different languages. It is no wonder then that we often look into our own immediate past or future, last Tuesday or a week from Sunday, with feelings of helpless confusion. ...

—Robert Grudin, Time and the Art of Living.

本节说明 gnu 程序采纳的日期文字表现形式。它们是你作为用户提供给各种程序的字符串参数。这里不说明 C 的接口（通过 get_date 函数)。

22.6.1 General date syntax

date 是一个字符串，可以为空，包含许多用空格分开的项目。空格在没有歧义时可以删除。空字串意味着今天起始时间 (也就是午夜)。项目的次序不是很重要。在日期字串里面含有许多项目:

```
calendar date items
time of day items
time zone items
day of the week items
relative items
pure numbers.
```

下面我们说明每个项目的类型。

某些情况下，一些普通的数字会写成文字形式。常用在指定星期或相对项目（见下）。常见的数字中，‘last’ 表示 -1，‘this’ 表示 0，而 ‘first’ 和 ‘next’ 都指的是 1。因为 ‘second’ 表示时间的单位，以至于 2 没有对应的数字，但习惯上 ‘third’ 还是表示 3，‘fourth’ 表示 4，‘fifth’ 表示 5，‘sixth’ 表示 6，‘seventh’ 表示 7，‘eighth’ 表示 8，‘ninth’ 表示 9，‘tenth’ 表示 10，‘eleventh’ 表示 11 而 ‘twelfth’ 表示 12.

当月份用这种方式书写时，也仍然考虑写成数字，而不是“全拼”；这修改允许的字串。

在当前的实施中，只支持英文单词和缩写，如 ‘AM’, ‘DST’, ‘EST’, ‘first’, ‘January’, ‘Sunday’, ‘tomorrow’ 和 ‘year’。

date 命令并不总是接受日期字串，不仅是因为语言问题，还由于现在还没有像 ‘IST’ 这样时区的标准含义。当用 date 生成的日期字串给以后解析时，指定一个独立于语言的日期格式，并不采用除 ‘UTC’ 和 ‘Z’ 之外的时区。有这样一些方法可以做到:

```
$ LC_ALL=C TZ=UTC0 date
Mon Mar 1 00:21:42 UTC 2004
$ TZ=UTC0 date +'%Y-%m-%d %H:%M:%SZ'
2004-03-01 00:21:42Z
$ date --iso-8601=ns # a GNU extension
2004-02-29T16:21:42,692722128-0800
$ date --rfc-2822 # a GNU extension
Sun, 29 Feb 2004 16:21:42 -0800
$ date +'%Y-%m-%d %H:%M:%S %z' # %z is a GNU extension.
2004-02-29 16:21:42 -0800
$ date +'@%s.%N' # %s and %N are GNU extensions.
@1078100502.692722128
```

日期里面完全忽略字母的大小写。注释写在圆括号内，包括相应的嵌套。不在数字后的连字符被忽略。数字前面的零也忽略不计。

22.6.2 Calendar date items

calendar date item 指出年里面的日。指定的方式因月份使用数字还是文字表示而不同。下面的字串指的都是相同的日历日期:

```
1972-09-24   # iso 8601.
72-9-24      # Assume 19xx for 69 through 99,
```

```
              # 20xx for 00 through 68.
72-09-24      # Leading zeros are ignored.
9/24/72       # Common U.S. writing.
24 September 1972
24 Sept 72    # September has a special abbreviation.
24 Sep 72     # Three-letter abbreviations always allowed.
Sep 24, 1972
24-sep-72
24sep72
```

年份可以省略。这种情况下，使用最后指定的年份，如果没有则是当前年份。例如:

9/24

sep 24

下面是规则。

对于数字月份，允许使用 iso 8601 格式 'year-month-day'，其中 year 是任意正数，month 是 01 到 12 的数字，day 是 01 到 31 的数字。如果数字小于十，前面必须要加零。如果 year 是 68 或更小，则加上 2000; 否则，year 小于 100 时，加 1900。在美国，常见的是 'month/day/year' 结构，也可以使用。还可以是省略年份的 'month/day'。

文字月份可以是全拼: 'January', 'February', 'March', 'April', 'May', 'June', 'July', 'August', 'September', 'October', 'November' 或 'December'。还可以使用前三个字母的缩写形式，后面可以跟点。'September' 也允许写成 'Sept'。

当月份以文字形式书写时，日历日期可以如下:

```
day month year
day month
month day year
day-month-year
```

或者，省略年份:

month day

22.6.3 Time of day items

日期字符串里面的 time of day item 指出给定日的时刻。这里是一些例子，代表相同的时间:

```
20:02:00.000000
```

```
20:02
8:02pm
20:02-0500    # In est (U.S. Eastern Standard Time).
```

通常，时刻可以用 'hour:minute:second' 表示，其中 hour 是 0 到 23 的数字，minute 是 0 到 59 的数字，second 是 0 到 59 的数字，后面可以跟 '.' 或 ',' 以及一位或多位数字。另外 ':second' 可以省略，这时用零替代。

如果时间后面是 'am' 或 'pm' (或者 'a.m.' 或 'p.m.')，hour 缩减为 1 到 12, 同时 ':minute' 可以省略 (用零代替)。'am' 表示上半天，'pm' 表示下半天。这种记法下，12 在 1 之前: 午夜是 '12am'，正午为 '12pm'[23]。

时间还可以通过后面跟上时区校正替代，形式为 'shhmm'，其中 s 是 '+' 或 '-'，hh 是时区的小时，mm 是时区的分钟。当时区以这种方式给出时，时间相对 Coordinated Universal Time (utc) 解释，覆盖以前给出的时区和本地时区。minute 在时区校正里面不应被省略。这是指定少于一小时的时区校正最好的方法。

可以指定 'am'/'pm' 或时区校正的任意一个，但不能同时使用。

22.6.4 Time zone items

A time zone item 同一组字符指定国际时区，例如，'UTC' 或 'Z' 代表 Coordinated Universal Time。忽略含有的周期。在非夏令时时区后面加上分隔的 (就是用空格隔开的) 'DST' 字串，可以指定相应的夏令时。

除了 'UTC' 和 'Z' 之外的时区即将废弃，不建议使用，因为它们含有歧义; 例如，'EST' 在澳大利亚和美国有不同的含义。作为替代，使用无歧义的数字时区校正，如 '-0500' 更好，见前面一节说明。

如果既没有时区项也没有提供时区校正，解释时间戳使用默认时区规则[24]。

22.6.5 Day of week items

指明将来日期 (如必要) 所在的星期。

星期可以使用全拼: 'Sunday', 'Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday' 或 'Saturday'。还可以使用前三个字母缩写，后面可以跟周期。还允许使用 'Tues' 于 'Tuesday', 'Wednes' 于 'Wednesday' 和 'Thur' 或 'Thurs' 于 'Thursday' 的特殊缩写形式。

星期项前面的数字将向后移动数周。最好的表示方法如 'third monday'。在上下文里面，'last day' 或 'next day' 也可以接受; 它们表示指定 day 向前或向后移动一周的日期。

[23]这是零向方法解释 '12am' 和 '12pm'，过去拉丁方法是用 '12m' 表示正午，'12pm' 表示午夜。

[24]参阅 Specifying time zone rules。

星期项后面的逗号被忽略。

22.6.6 Relative items in date strings

Relative items 向前或向后调整指定日期（如不指出日期为当前日期)。相对项的效果是累积的。这里有一些例子:

```
1 year
1 year ago
3 years
2 days
```

时间的单位可以用字串 'year' 或 'month' 替换，表示移动整年或整月。有一些模糊的单位，像 years 和 months 时间段并不一致。精确的单位有 'fortnight' 表示 14 天，'week' 是 7 天，'day' 是 24 小时，'hour' 是 60 分钟，'minute' 或 'min' 是 60 秒，'second' 或 'sec' 是一秒。单位后面可以有后缀 's'，但被忽略。

时间单位前面可以跟乘数，它是一个符号数。没有符号的数字认为是正数。没有给出乘数隐含为 1。相对时间后面的 'ago' 字串等于乘数值为 -1。

字串 'tomorrow' 值为将来的一天 (等于 'day')，字串 'yesterday' 值为过去的一天 (等于 'day ago')。

字串 'now' 或 'today' 相对项等同于用零值替代，如果没有被前面项目修改，这些字串代表当前时间。它们也可以用来强调其他项，如 '12:00 today'。字串 'this' 也代表零值，但常用于 'this thursday' 这样的日期字串里面。

当相对项让日期跨越时钟调整边界时，典型的例子是夏时制，日期和时间的结果也要相应地调节。

模糊的单位会另相对项出现问题。例如，'2003-07-31 -1 month' 可以等于 2003-07-01，这是因为 2003-06-31 是一个无效的日期。为了更可靠地确定上一个月，你可以询问当前月十五号的上一个月，例如:

```
$ date -R
Thu, 31 Jul 2003 13:02:39 -0700
$ date --date='-1 month' +'Last month was %B?'
Last month was July?
$ date --date="$(date +%Y-%m-15) -1 month" +'Last month was %B!'
Last month was June!
```

还有，要小心处理夏令时闰年日期和时间的改变。一些情形下它们增加或减少 24 小时，所以开始日历计算前，最好先将 TZ 环境变量设成 'UTC0'。

22.6.7 Pure numbers in date strings

纯十进制数的精确解释取决于日期字串的上下文。

如果十进制数为 yyyymmdd 格式，并且日期字串前面有其他的日历日期项[25]，那么对于指定的日历日期，yyyy 当作年，mm 当作月份数字，dd 当作月中几号。

如果十进制数为 hhmm 格式，并且日期字串前面没有其他的时刻项，那么对于指定的一天中的时间项，hh 当前小时，mm 当作分钟。mm 也可以省略。

如果日历日期和一天中的时间都出现在日期字串的左侧，并且没有相对项，那么数字覆盖年份。

22.6.8 Seconds since the Epoch

如果数字前面有 '@'，表示这是一个秒数的内部时间戳。数字可以包含内部十进制小数点 ('.' 或者是 ',')；任何超出支持范围的数字将被截掉。这个数字不能与其他日期项组合，它提供的是完整的时间戳。

在计算机内部，时间是用纪元以来的秒数表示 – 一个明确定义的时间点。在 GNU 和 POSIXPOSIX 系统上，纪元为 1970-01-01 00:00:00 UTC，'@0' 即此时间，'@1' 表示的时间是 1970-01-01 00:00:01 UTC，依此类推。GNU 和大多数 POSIX 兼容的系统支持 POSIX 扩展，可使用负数，用 '@-1'-1' 代表 1969-12-31 23:59:59 UTC。

传统的 Unix 系统使用 32 位整数计算秒数，可以表示从 1901-12-13 20:45:52 到 2038-01-19 03:14:07 utc 的时间范围。新的系统使用 64 位可以计算秒和纳秒，能表示精确到 1 纳秒的已知宇宙时间。

大多数系统上，计算时忽略秒的跳跃。例如，大多数系统 '@915148799' 表示 1998-12-31 23:59:59 utc，'@915148800' 表示 1999-01-01 00:00:00 utc，而没有介于其间的 1998-12-31 23:59:60 utc 的表示方法。

22.6.9 Specifying time zone rules

通常，日期的解释采用 TZ 环境变量指定的本地时区规则，如果 TZ 没有设置，则使用系统默认的设置。给一个日期指定不同的时区，在日期前面使用形如 'TZ="rule"' 的字串。日期里面必须使用双引号 ('")，rule 里面的任何引号和反斜杠都要用反斜杠转义符号。

[25]参阅 Calendar date items。

例如，使用 GNU date 命令可以回答像”What time is it in New York when a Paris clock shows 6:30am on October 31, 2004?” 的问题，在下面的转换脚本里面日期的前面加上 ‘TZ=”Europe/Paris”’:

```
$ export TZ="America/New_York"
$ date --date='TZ="Europe/Paris" 2004-10-31 06:30'
Sun Oct 31 01:30:00 EDT 2004
```

这个例子里面，–date 开始使用自己的 TZ 设置，其余的使用 ‘Europe/Paris’ 规则，将字串 ‘2004-10-31 06:30’ 当成位于巴黎来处理。然而，因为 date 命令用系统的时区规则处理，它使用纽约时间[26]。

一个 TZ 值，对应 ‘tz’ database 里面地点名的规则。最新的地点归类见 TWiki Date and Time Gateway。一些非 GNU 主机要求在 TZ 设置的地点名前面加上冒号，例如，‘`TZ=":America/New_York"`’。

‘tz’ 数据库包括从 ‘Arctic/Longyearbyen’ 至 ‘Antarctica/South_Pole’ 的广范的地点范围，但是如果如果你处在海洋有自己的私人时区，或者你使用不支持 ‘tz’ 数据库的非 GNU 主机，你也许要用 POSIX 规则替代。简单的 POSIX 规则如 ‘UTC0’ 指定的时区不使用夏时制; 其他的规则可以指定简单的夏时制[27]。

22.6.10 Authors of get_date

get_date 最初由 Steven M. Bellovin(smb@research.att.com) 在 Chapel Hill 的北卡罗莱纳大学实现，后来在 Usenet 上许多人优化过代码，1990 年 8 月，Rich $alz (rsalz@bbn.com) 和 Jim Berets (jberets@bbn.com) 彻底地检查了一遍。gnu 系统上的多个版本由 David MacKenzie、Jim Meyering、Paul Eggert 等人编写。

本章最初由 François Pinard (pinard@iro.umontreal.ca) 从 getdate.y 源码生成，然后由 K. Berry (kb@cs.umb.edu) 编辑。

22.7 admin

需要：仓库，工作目录。

修改：仓库。

同义词：rcs

[26] 2004 年巴黎通常早于纽约六小时，但这个例子中的五小时是指万圣节期间。

[27] 参阅 Specifying the Time Zone with TZ。

本命令是配合管理功能的 cvs 接口。这其中的一些功能被质疑是否有用，但因为历史的缘故还保留着。将来也许会去掉这些有问题的选项。此命令具备递归特性，所以使用的时候要特别小心。

在 unix 系统中，如果有一个组名为 cvsadmin，则只有该组的成员可以执行 cvs admin 命令，除非在 CVSROOT/config 里面规定 UserAdminOptions 配置选项。指定 UserAdminOptions 选项后，任何用户都能执行此命令[28]。

cvsadmin 组应该存在于服务器端，或者是任何非客户机/服务器的 cvs 上。为了禁止所有用户使用 cvs admin，可创建一个无成员的组。在 NT 系统上，并不存在 cvsadmin 特性，因此所有用户都可以执行 cvs admin 命令。

22.7.1 admin options

这里面的一些选项虽然被质疑是否有用，但由于历史原因仍然保留。其中一些甚至造成 cvs 无法使用，除非你恢复到原状!

- -Aoldfile
  可能无法用在 cvs。追加 oldfile 的存取列表到 rcs 文件的存取列表。
- -alogins
  可能无法用在 cvs。追加登录名到 rcs 文件存取列表的 logins 列表，由逗号分隔。
- -b[rev]
  设置默认分支为 rev。在 cvs 中，你不应该手动修改默认分支；采用粘性标签[29]决定工作分支是更佳的方式。有一种情况需要使用 `cvs admin -b` 命令：当使用第三方分支时，有需要回复到他们的版本[30]。在 '-b' 和它的参数之间可以没有空格。
- -cstring
  设置注释头为 string。注释头在当前的 cvs 和 rcs 5.7 版本中不在使用。你可以不用理它。参阅 Keyword substitution.
- -e[logins]
  可能无法用在 cvs。从 RCS 文件的存取列表 logins 列表中删除登录名，由逗号分隔。如果不指定 logins，删除整个存取列表。在 '-e' 和它的参数之间可以没有空格。
- -I
  交互式执行，即使标准输入并不是一个终端。该选项于 cvs 客户机/服务器方式下无效，并将在以后的 cvs 中去掉。
- -i

[28]参考 config 了解有关 UserAdminOptions 的详细情况。
[29]参阅 Sticky tags
[30]参阅 Reverting local changes

对 cvs 无用。它用来创建和初始化新的 rcs 文件，并不设置版本。对于 cvs，添加文件使用 cvs add 命令 (参阅 Adding files)。

- -ksubst
  设置默认的关键字替换为 subst。参阅 Keyword substitution. 在 cvs update，cvs export 或 cvs checkout 命令中可以使用 '-k' 选项覆盖此默认值。
- -l[rev]
  锁定修订版号为 rev。如果指定的是分支，则锁定该分支最后的修订版。如果没有 rev，则锁定默认分支的最新修订版本。在 '-l' 和它的参数之间可以没有空格。
  它与 rcslock.pl 脚本配合使用，该脚本位于 cvs 源码发行版的 contrib 目录，用来提供限制检出 (一个文件只允许一个用户同时修改)。查看脚本的注释了解更多信息 (并参阅该目录中 README 关于不支持特性的无责声明)。据注释所说，锁定必须设置为 strict(此为默认)。
- -L
  设置锁定为 strict。strict 锁定意思是说，RCS 文件的主人也不能免除锁定而提交。cvs 使用时，必须设置 strict 锁定；参阅上面的 '-l' 选项所述。
- -mrev:msg
  用 msg 替换日志信息中的 rev。
- -Nname[:[rev]]
  与 '-n' 类似，除了可以覆盖以前设置的 name。了解如何使用魔术分支，参阅 Magic branch numbers，了解魔术分支的使用。
- -nname[:[rev]]
  将符号名 name 与分支或修订版 rev 关联。通常采用 'cvs tag' 或 'cvs rtag' 命令更佳。如果没有 ':' 和 rev，则删除符号名; 否则如果符号名 name 已经存在，将会打印错误信息。如果 rev 是符号名，则扩展之前的关联。rev 由分支号和 '.' 组成，表示当前分支的最新修订版。':' 和空的 rev 表示默认版本的最新修订版，通常指主干。例如，'cvs admin -nname:' 将 name 关联到所有 RCS 文件的当前最新修订版; 对比 'cvs admin -nname:$'，它是将 name 关联到解开的关键字对应的工作文件修订版。
- -orange
  删除 range 指定的 (outdates) 修订版。
  注意，在你明确你在做什么之前 (例如查看了下面的有关 rev1:rev2 语法困惑警告)，该命令是非常危险的。
  如果磁盘空间不够，该选项可以提供帮助。但用之前请三思 – 这将无法依靠备份取消这个命令! 如果你错删了修订版，即使是出错或 (但愿不是) cvs bug 造成的，除

了删除修订版，没有可能修复。或许先在仓库的复制版上进行实验是个好主意。
指定 range 可以采用以下的方式:

1. rev1::rev2
   清除 rev1 和 rev2 之间的所有版本，那么 cvs 将只保存 rev1 和 rev2 之间的差别，而无中间状态。例如，执行 '-o 1.3::1.5' 之后，只能得到 1.3 和 1.5 修订版，或者 1.3 和 1.5 之间的差异，但无法获得 1.4 或 1.3 和 1.4 之间的差异。另一个例子: '-o 1.3::1.4' 和 '-o 1.3::1.3' 没有任何效果，这是因为它们没有中间状态可以删除。
2. ::rev
   清除含 REV 的分支从开始到 rev 之间的修订版。分支点和 rev 则保留。例如，'-o ::1.3.2.6' 删除 1.3.2.1 和 1.3.2.5 修订版和它们之间的所有修订版，但保留 1.3 和 1.3.2.6。
3. rev::
   清除 rev 和包含 rev 分支最后的修订版。rev 修订版保留，但最新修订版删除。
4. rev
   删除 rev 修订版。例如，'-o 1.3' 等于 '-o 1.2::1.4'。
5. rev1:rev2
   删除 rev1 到 rev2 之间的修订版，包括同一个分支。以后将无法获取 rev1 或 rev2 以及之间的修订版。例如，'cvs admin -oR_1_01:R_1_02 .' 很少使用。意思是删除修订版到标签 R_1_02，并含标签。但要当心! 如果文件在 R_1_02 和 R_1_03 之间没有修改，那么它们具有相同的数字修订版号赋于标签 R_1_02 和 R_1_03。于是不仅无法获得 R_1_02; R_1_03 也不得不要从磁带上恢复! 在大多数情况下，我们应该指定 rev1::rev2。
6. :rev
   删除含 rev 的分支从开始到 rev，并包括 rev。
7. rev:
   删除从 rev 修订版，包含 rev 自己，到含 rev 分支的结尾。
   含有分支或锁定的修订版将不会被删除。

如果修订版包含符号名，并且指定 '::' 语法，cvs 将给出错误信息并不删除任何修订版。如果您的确想删除符号名和修订版，首先用 cvs tag -d 删除符号名，然后执行 cvs admin -o。如果指定不含 '::' 的语法，那么 cvs 将删除修订版，但保留含符号名指向不存在的修订版。该特性是为与以前的 cvs 版本兼容，但由于没什么用处，将来可能会改成类似 '::' 方式。

对于因 cvs 不能处理 rev 为分支的符号名[31]。
确保你要处理的过期修订版没有被人检出。如果有人试图编辑，并尝试提交，会出现一些奇怪的事情。正因如此，该选项不适合撤消假提交; 而使用提交新修订版替代撤消假修改[32]。

- -q
  安静地运行；不打印任何调试信息。
- -sstate[:rev]
  在 cvs 下很有用。为 rev 修订版设置状态为 state。如果 rev 是分支号，则假定是该分支的最新版本。如果 rev 省略，假定是默认分支的最新版本。state 可以使用任何标识。常用的有 'Exp'(实验)，'Stab'(稳定)，'Rel'(发行)。新的修订版创建时默认使用 'Exp' 标识。从 cvs log [33]的输出，以及 'Log' 和 '$State$' 关键字[34]中可以看到状态。注意，cvs 内部使用 dead 状态；对文件设置或取消 dead 状态应该采用诸如 cvs remove 和 cvs add 命令，而不是 cvs admin -s。
- -t[file]
  在 cvs 下很有用。将名为 file 的文件内容写到 RCS 文件的描述中，删除以前的文字。file 路径名不能以 '-' 开头。描述文件可以从 'cvs log'[35]的输出中看到。在 '-t' 和它的参数之间可以没有空格。
  如果省略 file，描述文字将从标准输入获得，以 end-of-file 或 '.' 行结束。如果是交互式，则有相关提示；参阅 '-I'。
- -t-string
  与 '-tfile' 类似。将 string 写入 rcs 文件的描述文字，删除已有的文字。在 '-t' 和它的参数之间可以没有空格。
- -U
  设置锁定为 non-strict。non-stric 锁定意味着文件的所有者不必锁定修订版来提交。在 cvs 下使用 strict 锁定必须设置；参阅上面的 '-l' 选项。
- -u[rev]
  参阅上面的 '-l' 选项中关于 cvs 中使用的说明。为 rev 修订版解锁。如果给定的是分支，为分支的最新修订版解锁。如果省略 rev，删除设置人的最新锁。通常，只有加锁的人才能解锁；如果其他人也解锁会打破锁的作用。这将发送一个 commit 通

[31]参阅 Magic branch numbers, 了解详细情况。
[32]参阅 Merging two revisions。
[33]参阅 log。
[34]参阅 Keyword substitution。
[35]参阅 log。

知给加锁的人[36]。在 '-u' 和它的参数之间可以没有空格。

- -Vn 在以前的 cvs 中，这个选项可以将 rcs 版本 n 写到 rcs 文件，但现在已经废止，指定将产生错误。
- -xsuffixes
  在以前的 cvs 中，文档说明使用这种方法指定 rcs 文件名。然而，cvs 要求一直使用以 ',v' 结束的 rcs 文件，所以，这个选项也不再有意义。

22.8 annotate

修订版里面文件的每一行做了哪些修改?

语法: annotate [options] files...

需要: 仓库。

修改: 无。

对 files 中每个文件，打印主干上的最新修订版，以及最后做了哪些修改的信息。

22.8.1 annotate options

这些是 annotate 所支持的标准选项[37]:

- -l
  只作用于本目录，不递归。
- -R
  递归处理目录。
- -f
  如果不指定标签/日期，使用最新修订版。
- -F
  批注二进制文件。
- -r tag[:date]
  批注指定修订版/标签的文件，或者当指定 date 并且 tag 是分支标签时，tag 分支上的版本当作存在于 date 上。见 Common options。
- -D date
  批注指定日期的文件。

[36]参阅 Getting Notified。
[37]参阅 Common options，了解详细信息。

22.8.2 annotate example

例如:

```
$ cvs annotate ssfile
Annotations for ssfile
***************
1.1         (mary    27-Mar-96): ssfile line 1
1.2         (joe     28-Mar-96): ssfile line 2
```

文件 ssfile 当前包含两行。行 ssfile line 1 是 mary 于三月 27 日提交的。接着，三月 28 日，joe 添加了行 ssfile line 2，但没有修改行 ssfile line 1。该报告并不会告诉你有关删除或替换的行; 你需要使用 cvs diff 去了解[38]。

`cvs annotate` 的选项列在 Invoking CVS，可批注选定的文件与修订版[39]。

22.9 checkout

语法：checkout [options] modules...

需要：仓库。

修改：工作目录。

同义词：co, get

创建或更新 modules 指定的工作目录，包含源码副本。使用大多数其他 cvs 命令之前，必须执行 checkout，这是因为它们大部分操作的是工作目录。

modules 可以是源码目录和文件集合的符号名，或者目录路径，或仓库里面的文件。符号名在 'modules' 文件里面定义[40]。

随你指定的模块，checkout 可以递归地创建目录，并将适当的文件放在里面。可以在任意时间编辑这些文件 (不论是否其他的开发人员也在编辑他们自己的副本); 更新它们以包括源码仓库中其他人员提交的修改; 或将修改永久地提交到源码仓库。

注意，checkout 用于创建目录。在 checkout 执行的目录，总会创建顶级目录，而且通常与指定的模块同名。在 module alias 情况下，创建的子目录也许有不同的名字，但可以肯定它是一个子目录，并且 checkout 会在解到你私人工作空间时，显示出每个文件的相对路径 (除非你指定了 '-Q' 全局选项)。

[38]参阅 diff。
[39]详细的选项说明见 Common options。
[40]参阅 modules。

checkout 建立的文件属性是可读写的，除非事先给 CVS 指定 '-r' 选项[41]，或 CVSREAD 环境变量[42]，或者该文件已被监视[43]。

注意，在一个已经由 checkout 建立的目录里面运行 checkout 目录是允许的。这类似给 update 指定 '-d' 选项，使得新目录可以在工作区内创建。但 checkout 使用的是模块名，而 update 使用目录名。还有，checkout 要在顶级目录使用这种方式 (以前使用 checkout 的目录)，所以，在用 checkout 更新已有目录前，别忘了将目录改到顶级目录。

关于 checkout 命令的输出，见 update output。

22.9.1 checkout options

这些是 checkout 支持的标准选项[44]:

- -D date
  使用不迟于 date 的最新修订版。该选项是粘性的，也就是 '-P'[45]。
- -f
  只与 '-D' 或 '-r' 标识一起使用。如果找不到匹配的修订版，将使用最新的修订版 (而不是忽略文件)。
- -k kflag
  根据 kflag 处理关键字，此选项是粘性的，以后在这个工作目录里面更新，还是使用相同的 kflag[46]。status 命令可以看到粘性的选项[47]。
- -l
  只在当前工作目录里面执行。
- -n
  不运行任何检出程序[48]。
- -P
  清除空目录。参阅 Moving directories。
- -p
  管道方式输出到标准输出。
- -R

[41]参阅 Global options
[42]参阅 Environment variables
[43]参阅 Watches
[44]参阅 Common options，了解完整的信息。
[45]参阅 Sticky tags，了解更多粘性标签/日期的信息。
[46]参阅 Keyword substitution。
[47]参阅 Invoking CVS，了解 status 命令的信息。
[48]如同在模块文件里面指定 '-o' 选项，参阅 modules。

递归方式检出。此选项是默认的。

- -r tag[:date]
  使用 tag 修订版，或者当 date 指定，并且 tag 是分支标签，tag 分支上的版本当作存在于 date 上。此选项是粘性的，含 '-P'[49]。

除了这些，你还可以在 checkout 里面使用特殊的命令选项：

- -A
  重置任何的粘性标签，日期，或 '-k' 选项[50]。
- -c
  复制模块文件，排序，输出到标准输出，而不是在工作目录里面创建和修改文件与目录。
- -d dir
  为工作文件创建名为 dir 的目录，而不是使用模块名。通常，使用此标识等同于使用 'mkdir dir; cd dir'，然后是不带 '-d' 标识的检出命令。然而，有一个重要的例外。习惯上，检出单独的项目只输出到一个目录而不会包含中间空的目录。仅在这种情况下，cvs 尽力“缩短”路径名，避免空目录。
  例如，模块 'foo' 包含 'bar.c'.c' 文件，'cvs co -d dir foo' 命令将建立 'dir' 目录并将 'bar.c' 文件放在里面。同样，模块 'bar' 里面有 'baz' 子目录，其中有一个文件 'quux.c'.c'，'cvs co -d dir bar/baz' 将创建 'dir' 目录，并将 'quux.c' 放在里面。
  使用 '-N' 标识将破坏这种行为。使用上面相同的模块，'cvs co -N -d dir foo' 将建立 'dir/foo' 目录并放入 'bar.c' 文件，使用 'cvs co -N -d dir bar/baz' 将建立 'dir/bar/baz' 目录并将 'quux.c' 放在里面。
- -j tag
  用两个 '-j' 选项，合并第一个 '-j' 选项至第二个 '-j' 之间修订版间的修改到工作目录。使用一个 '-j' 选项，合并祖先修订版至 '-j' 选项指定的修订版间的修改到工作目录。祖先修订版是所基于的工作目录和 '-j' 选项指定的修订版的共同祖先。
  另外，每个 -j 选项可以包含可选的日期规格，当与分支使用时，可以限定在指定日期内选择修订版。可选的日期由标签中增加的 (:) 分号指定:'-jSymbolic_Tag:Date_Specifier'。参阅 Branching and merging.
- -N
  只与 '-d dir' 一起使用。采用这个选项，cvs 在检出单独模块时，将不再“缩短”工作目录里面的模块路径[51]。

[49]参阅 Sticky tags，了解更多粘性标签/日期的信息。以及 Common options。
[50]参阅 Sticky tags，了解更多粘性标签/日期的信息。
[51]参阅 '-d' 标识中的例子和讨论。

- -s
  类似 '-c'，但包含所有模块的状态，并按状态字符串排序[52]。

22.9.2 checkout examples

获得模块 'tc' 的副本:

```
$ cvs checkout tc
```

获得一天以前 'tc' 模块的副本:

```
$ cvs checkout -D yesterday tc
```

22.10 commit

语法：commit [-lnRf] [-m 'log_message' | -F file] [-r revision] [files...]
需要：工作目录，仓库。
修改：仓库。
同义词：ci
当需要将工作目录里面的修改合并到源码仓库时，使用 commit 命令。

如果不指定具体的文件，当前工作目录里面的文件，经过检验，都会被提交。commit 会谨慎地在仓库中修改那些真正做了变更的文件。默认情况下（或特别指定了 '-R' 选项），在子目录中的也要检查，如果它们有变更也会被提交；你可以使用 '-l' 选项让 commit 只针对当前的目录。

commit 会校验选择的文件已经更新到源码仓库的当前修订版；如果有文件需要首先使用 update （参阅 update）更新到当前版本，它会通知你，然后不做提交退出。commit 不会为我们调用 update 命令，而是让我们自行处置。

如果就绪，会打开一个编辑器用来输入日志消息，用来写到一个或多个日志程序[53]并将其放到仓库的 rcs 文件。日志消息可以通过 log 命令看到；见 log。也可以在命令行上用 '-m message' 选项指定日志消息，以避免打开编辑器，或者用 '-F file' 来指定包含日志消息的文件。

commit 时，在仓库的 rcs 文件里面会放入唯一的 commitid。同时提交的文件是相同的 commitid。使用 log 和 status 命令可以查询 commitid；见 log, File status。

[52]参阅 modules, 了解 '-s' 选项的信息，它用在模块文件里面设置模块状态。
[53]参阅 modules 和 loginfo。

22.10.1 commit options

commit 支持标准选项[54]:

- -l
  只在当前目录运行。
- -R
  递归方式提交。此为默认。
- -r revision
  提交到 revision。revision 必须是分支，或者是主干上高于任何已有版本号的修订版[55]。不能提交到分支上的一个特定修订版。

commit 还支持这些选项:

- -c
  拒绝提交文件，除非用户已经通过 cvs edit 注册了一个有效的编辑。将 'commit -c' 和 'edit -c' 放在所有的.cvsrc 文件里面最有用。通过 cvs edit 注册可追溯的编辑 (不会丢失文件的变更) 或使用 -f 选项，可以强行提交。要支持 commit -c 要求客户端和服务器的版本为 1.12.10 或更高。
- -F file
  从 file 里面读取日志消息，而不是打开编辑器输入。
- -f
  注意，这不是在 Common options 里面定义的 '-f' 选项的标准行为。
  即使没有修改文件也要强制 cvs 提交到一个新修订版。对于 cvs 1.12.10 版本，它使 -c 选项被忽略。假如当前的 file 修订版是 1.7，那么下面的命令是相等的:

```
$ cvs commit -f file
$ cvs commit -r 1.8 file
```

  '-f' 选项禁止了递归 (如同使用 '-l')。要让 cvs 提交所有子目录里面的所有文件，必须用 '-f -R'。
- -m message
  用 message 作为日志消息，而不是打开编辑器输入。

[54]参阅 Common options，了解完整说明。
[55]参阅 Assigning revisions。

22.10.2 commit examples

可以用 '-r' 选项提交到分支修订版 (小数点是偶数)。要创建分支修订版，需要用 rtag 或 tag 命令里面的 '-b' 选项[56]。之后，checkout 或 update 就基于新创建的分支。从这一刻起，这些工作源码的修改 commit 都自动添加到分支修订版，也就是不会干扰主线上的开发。例如，我们给产品的 1.2 版做了个补丁，即使已经开发到了 2.0 版，也可以:

```
$ cvs rtag -b -r FCS1_2 FCS1_2_Patch product_module
$ cvs checkout -r FCS1_2_Patch product_module
$ cd product_module
[[ hack away ]]
$ cvs commit
```

因为 '-r' 选项是粘性的，所有工作都自动进行。

当用户在进行试验性软件的开发，工作是基于上周检出的修订版。如果小组里面的其他人要一同参与这个软件，但又不想干扰主线上的开发，就可以将修改提交到新的分支上。其他人可以检出这项试验性的工作，并利用 cvs 解决冲突的特性。场景如下:

```
[[ hacked sources are present ]]
$ cvs tag -b EXPR1
$ cvs update -r EXPR1
$ cvs commit
```

update 命令可以在所有的文件加上粘性的 '-r EXPR1' 选项。注意，这里对文件的修改将不会因 update 命令删除。由于 '-r' 是粘性的，commit 将自动提交到正确的分支。我们也许会:

```
[[ hacked sources are present ]]
$ cvs tag -b EXPR1
$ cvs commit -r EXPR1
```

但这样做，只有这些修改过的文件有 '-r EXPR1' 粘性标识。如果你继续开发，提交时没有指定 '-r EXPR1' 标识，一些文件可能会提交到主干。

其他人要跟我们一同进行试验性开发，只需

```
$ cvs checkout -r EXPR1 whatever_module
```

[56]参阅 Branching and merging。

22.11 diff

语法：diff [-lR] [-k kflag] [format_options] [(-r rev1[:date1] | -D date1) [-r rev2[:date2] | -D date2]] [files...]

需要：工作目录，仓库。

修改：无。

diff 命令用于比较文件的不同修订版。默认是比较工作目录文件与其所基于的修订版，然后报告所发现的差异。

如果给定文件名，则只比较这些文件。如果给定目录，则会比较目录下所有的文件。

diff 的退出状态与其他 cvs 命令不同，详细情况见 Exit status。

22.11.1 diff options

diff 支持标准选项[57]：

- -D date 使用不迟于 date 的最新修订版。见 '-r' 了解它是如何影响比较的。-k kflag 根据 kflag 处理关键字。参阅 Keyword substitution。-l 只在当前目录运行。-R 递归方式检验。此为默认。-r tag[:date] 比较指定的 tag 修订版，或者当 date 指定，并且 tag 是分支标签，分支 tag 上的版本可以当作是在 date 上。可以没有，有一个或两个 '-r' 选项。没有 '-r' 选项时，工作文件将与它所基于的修订版进行比较。有一个 '-r' 选项时，指定的修订版与当前工作文件进行比较。两个 '-r' 选项时，将比较这两个修订版 (同时你的工作文件不会影响输出结果)。一个或所有的 '-r' 选项都能用上面提到的 '-D date' 选项替代。

下面的选项指定输出的格式。他们与 GNU diff 有相同的意思。许多选项有两个相等的名字，一个是 '-' 后面的单个字母，另一个是 '–' 后面的长名字。

- '-lines'
  显示上下文 lines (一个整数) 行。此选项不指定输出的格式; 如不与 '-c' 或 '-u' 一起使用，没有任何作用。该选项已经废弃。对适当操作，patch 通常至少要两行内容。
- '-a'
  所有的文件都视为文本文件来逐行比较，甚至他们似乎不是文本文件。
- '-b'
  忽略空格引起的变化，并认为一个或多个空格都相同。
- '-B'
  忽略插入删除空行引起的变化。

[57]参阅 Common options，了解完整说明

- '–binary'
  以二进制模式读写数据。
- '–brief'
  仅报告文件是否相异，不在乎差别的细节。
- '-c'
  使用上下文输出格式。
- '-C lines'
- '–context[=lines]'
  使用上下文输出格式，显示以指定 lines (一个整数)，或者当 lines 没有给出时是三行。对于正确的操作，patch 需要上下文至少要有两行。
- '–changed-group-format=format'
  使用 format 输出一组包含两个文件的不同处的行，其格式是 if-then-else。参阅 Line group formats.
- '-d'
  改变算法也许发现变化的一个更小的集合。这会使 diff 变慢 (有时很慢)。
- '-e'
- '–ed'
  输出为一个有效的 ed 脚本。
- '–expand-tabs'
  在输出时扩展制表符为空格，保持输入文件的制表符对齐方式。
- '-f'
  产生一个类似 ed 脚本的输出，但是改变他们在文件出现的顺序。
- '-F regexp'
  在上下文和统一格式中，对于每一大块的不同，显示出匹配 regexp 的一些前面的行。
- '–forward-ed'
  产生象 ed 脚本的输出，但是它们在文件出现的顺序有改变。
- '-H'
  使用启发规则加速操作那些有许多离散的小差异的大文件。
- '–horizon-lines=lines'
  比较给定 lines 的有共同前缀的最后行，和有共同或缀的最前 lines 行。
- '-i'
  忽略大小写; 认为大小写字母是相同的。

- '-I regexp'
  忽略因匹配 regexp 而插入，删除行带来的改变。
- '–ifdef=name'
  合并使用 name 的 if-then-else 格式输出。
- '–ignore-all-space'
  在比较行的时候忽略空格。
- '–ignore-blank-lines'
  忽略插入和删除空行。
- '–ignore-case'
  忽略大小写; 认为大小写字母是相同的。
- '–ignore-matching-lines=regexp'
  忽略因匹配 regexp 而插入，删除行带来的改变。
- '–ignore-space-change'
  忽略后面的空格，并认为所有的单个与多个空格是相同的。
- '–initial-tab'
  无论是常规的或者格式化的前后文关系，在文本行前输出制表符代替空格。使制表符对齐方式看上去象是常规的一样。
- '-L label'
  使用 label 给出的字符替代文件头里面上下文和统一格式的文件名。
- '–label=label'
  使用 label 给出的字符替代文件头里面上下文和统一格式的文件名。
- '–left-column'
  以并列方式印出两公共行的左边。
- '–line-format=format'
  使用 format 输出 if-then-else 格式所有的行。参阅 Line formats.
- '–minimal'
  改变算法也许发现变更的一个更小的集合。这会使 diff 变慢 (有时很慢)。
- '-n'
  输出 RCS 格式的比较; 除了每条指令指定的行数受影响外像 '-f' 一样。
- '-N'
- '–new-file'
  在目录比较中，如果那个文件只在其中的一个目录中找到，那么它被视为在另一个目录中是一个空文件。

- ‘–new-group-format=format’
  使用 format 以 if-then-else 格式输出只在第二个文件中取出的一个行组。参阅 Line group formats.
- ‘–new-line-format=format’
  使用 format 以 if-then-else 格式输出只在第二个文件中取出的一行。参阅 Line formats.
- ‘–old-group-format=format’
  使用 format 以 if-then-else 格式输出只在第一个文件中取出的一个行组。参阅 Line group formats.
- ‘–old-line-format=format’
  使用 format 以 if-then-else 格式输出只在第一个文件中取出的一行。参阅 Line formats.
- ‘-p’
  显示带有 C 函数的改变。
- ‘–rcs’
  输出 RCS 格式的比较; 除了每条指令指定的行数受影响外像 ‘-f’ 一样。
- ‘–report-identical-files’
- ‘-s’
  当两个文件相同时报告。
- ‘–show-c-function’
  显示带有 C 函数的改变。
- ‘–show-function-line=regexp’
  在上下文和统一的格式，对于每一大块的差别，显示出匹配 regexp 的一些前面的行。
- ‘–side-by-side’
  使用并列的输出格式。
- ‘–speed-large-files’
  使用启发规则加速操作那些有许多离散的小差异的大文件。
- ‘–suppress-common-lines’
  在并列格式中不印出公共行。
- ‘-t’
  在输出时扩展制表符为空格，保护输入文件的制表符对齐方式。
- ‘-T’
  无论是常规的或者格式化的前后文关系，在文本行前输出制表符代替空格。使得制表符对齐方式看上去象是常规的一样。

- '–text'
  所有的文件都视为文本文件来逐行比较，甚至他们似乎不是文本文件。
- '-u'
  使用统一的输出格式。
- '–unchanged-group-format=format'
  使用 format 输出两个文件的公共行组，其格式是 if-then-else。参阅 Line group formats.
- '–unchanged-line-format=format'
  使用 format 输出两个文件的公共行，其格式是 if-then-else。参阅 Line formats.
- '-U lines'
- '–unified[=lines]'
  使用统一输出，显示以指定 lines (一个整数), 或者当 lines 没有给出时是三行。对于正确的操作，patch 典型地至少要有两行。
- '-w'
  在比较行时忽略空格。
- '-W columns'
- '–width=columns'
  在并列格式输出时，使用指定的 columns。
- '-y'
  使用并列格式输出。
  - Line group formats: 行组格式
  - Line formats: 行格式

22.11.2 diff examples

下面的行产生 backend.c 文件 1.14 和 1.19 修订版间的 Unidiff ('-u' 标识)。因为使用 '-kk' 标识，没有关键字会被替换，所以差异是忽略了关键字替换。

```
$ cvs diff -kk -u -r 1.14 -r 1.19 backend.c
```

假设试验分支 EXPR1 基于 RELEASE_1_0 标签的一组文件。要查看分支上的状态，可以试验下面命令:

```
$ cvs diff -r RELEASE_1_0 -r EXPR1
```

类似这样的命令可以产生两个发行版的不同内容:

```
$ cvs diff -c -r RELEASE_1_0 -r RELEASE_1_1 > diffs
```

如果你维护着 ChangeLog，提交前使用如下命令，可以帮助你撰写 ChangeLog 条目。将打印出本地尚未提交的修改。

```
$ cvs diff -u | less
```

22.12 export

从 CVS 导出源码，类似检出

语法：export [-flNnR] (-r rev[:date] | -D date) [-k subst] [-d dir] module...

需要：仓库。

修改：当前目录。

此命令是 checkout 的变体; 用它可以获得没有 cvs 管理文件目录的模块源码。例如，你可以使用 export 准备出货的源码。这个命令需要你指定日期或标签 (用 '-D' 或 '-r')，来复制你要发给别人的代码 (而且它总会删除空的目录)。

人们常常使用在 cvs export 中使用 '-kv'。它让任何关键字都被扩展，使得其他地方导入代码不会丢失关键字修订版信息。但要小心，它不能正确处理导出的二进制文件。还需注意，使用了 '-kv'，就不能在用 ident 命令 (它属于 rcs 套件——见 ident(1)) 查询关键字字串。如果你打算使用 ident，就不要用 '-kv'。

22.12.1 export options

export 支持这些标准选项[58]:

- -D date
  使用不迟于 date 的最新修订版。
- -f
  如果没有匹配的修订版，获取最新的修订版 (而不是忽略这些文件)。
- -l
  只在当前目录运行。
- -n
  不运行任何检查程序。
- -R
  递归方式导出。此为默认。
- -r tag[:date]

[58]参阅 Common options，了解完整说明。

导出指定的 tag 修订版，或者当 date 指定，并且 tag 是分支标签，分支 tag 上的版本可以当作是在 date 上。见 Common options。另外还支持以下选项 (checkout 和 export 通用):

- -d dir
  为工作文件创建 dir 目录，而不是使用模块名。参阅 checkout options, 了解 cvs 处理此标识的详细信息。
- -k subst
  设置关键字扩展模式 (参阅 Substitution modes)。
- -N
  只与 '-d dir' 一起使用。参阅 checkout options, 了解 cvs 处理此标识的详细信息。

22.13 history

显示文件和用户的状态

语法：history [-report] [-flags] [-options args] [files...]

需要：$CVSROOT/CVSROOT/history 文件

修改：无。

cvs 用一个历史文件来跟踪 checkout, commit, rtag, update 和 release 命令的使用。你可以用 history 以各种格式显示此信息。

创建 $CVSROOT/CVSROOT/history 文件必须要打开日志功能。

注意：history 使用的 '-f', '-l', '-n' 和 '-p' 的方式与普通 cvs 的方式冲突 (参阅 Common options)。

22.13.1 history options

有一些选项可以控制报告的生成 (如 '-report' 显示):

- -c
  报告每次提交 (即，仓库的每次修改)。
- -e
  任何事情 (所有的记录类型)。等同于给所有的记录类型指定 '-x'。当然，'-e' 还包括以后 cvsCVS 将要包含的类型; 如果你要在脚本里面只处理特定的记录类型，需要使用 '-x''-x' 来指定。
- -m module
  报告特定的模块。(可以在命令行上多次使用使用 '-m'。)

- -o
  报告检出的模块。此为默认的报告类型。
- -T
  报告所有的标签。
- -x type
  从 cvs 历史中取出特定 type 类型的记录。类型用单独的字母表示，你也可以组合起来指定。

一些命令有一个单独的记录类型：

- F
  release
- O
  checkout
- E
  export
- T
- rtag
  update 产生的记录类型:
- C
  合并后有冲突发生 (需要手动合并)。
- G
  合并成功。
- U
  从仓库复制了工作文件。
- P
  为工作文件打补丁，与仓库中相配。
- W
  更新期间删除了工作副本 (因为在仓库里面已经删除)。

commit 产生的三种记录类型：

- A
  首次增加文件。
- M
  修改了文件。
- R

删除了文件。

显示为 '-flags' 方式的选项强迫或展开报告而不需要参数：

- -a
  显示所有用户的数据 (默认只显示执行 history 的用户数据)。
- -l
  只显示最后的修改。
- -w
  只显示 history 所执行目录的最后修改。

显示为 '-options args' 的选项强迫报告基于一个参数：

- -b str
  显示在模块名，文件名，或记录路径中包含字符串 str 的记录。
- -D date
  显示自从 date 开始的数据。这与 '-D date' 有一些不同，那是选择 date 日期之前的最新修订版。
- -f file
  显示指定文件的数据 (你可以在同一命令行上指定多个 '-f' 选项)。这与在命令行上指定文件相同。
- -n module
  显示指定模块的数据 (你可以在同一命令行上指定多个 '-n' 选项)。
- -p repository
  显示指定源码仓库的数据 (你可以在同一命令行上指定多个 '-p' 选项)。
- -r rev
  显示名为 rev 的单独 rcs 文件修订版或标签名的记录。每个 rcs 文件都被搜索。
- -t tag
  显示最后添加到历史文件的 tag 标签。与上面 '-r' 标识不同的是仅从历史文件读取，而不是 rcs 文件，所以更快。
- -u name
  显示用户 name 的记录。
- -z timezone
  使用指定的时区而非 UTC 显示所选记录的时间。

22.14 import

语法：import [-options] repository vendortag releasetag...

要求：仓库，源码目录。

修改：仓库。

使用 import 从外部 (例如，源码提供商) 将整个源码分发并入你的源码仓库目录。用户可以用这个命令初始化建立一个仓库，和从外部源码大批更新模块[59]。

repository 参数给出仓库中 cvs 根目录下的目录名 (或者目录的路径); 如此目录不存在，import 将创建一个。

当使用 import 更新已经仓库里面修改过的源码时 (从上次 import)，它会提示在两个开发分支上有冲突的文件，import 会指示用户使用 'checkout -j' 去处理。

如果 cvs 决定忽略某个文件 (参阅 cvsignore)，它不会导入该文件并在文件名前打印 'I ' [60]。

如果 $CVSROOT/CVSROOT/cvswrappers 文件存在，文件名匹配该文件中相应规格的文件将以包对待，并在导入前对此文件/目录执行相应的过滤程序。参阅 Wrappers.

外部的源码保存到第一级分支，默认为 1.1.1。以后更新也在这个分支上; 例如，首次导入源码的修订版为 1.1.1.1，更新后是 1.1.1.2，依此类推。

最少需要三个参数。repository 用于辨识源码集合。vendortag 是分支 (如，1.1.1) 的标签。你还需指定 releasetag 用来识别每次执行 import 建立的文件。releasetag 应为新建的，而非已经存在的标签，以便唯一确定导入的发行。

注意 import 不会修改执行时的目录。特别要指出的，是它不会建立一个目录作为 cvs 的工作目录; 如果你打算修改导入的这些文件，要先将它们导出到另外一个目录[61]。

22.14.1 import options

import 支持的标准选项[62]:

- -m message
  使用 message 作为日志消息，而不是打开编辑器。
  另外还有下列额外的选项。
- -b branch
  见 Multiple vendor branches.
- -k subst
  指出所需的关键字替换模式。此项设定会应用到导入的所有文件，但不影响仓库

[59]参阅 Tracking sources，了解有关此话题的讨论。
[60]参阅 import output，了解输出的完整说明。
[61]参阅 Getting the source。
[62]参阅 Common options，了解完整说明。

中已存在的文件[63]。

- -I name
  指定导入时忽略的文件名。你可以重复使用此选项。要避免某些文件被忽略 (即使是默认忽略的文件)，指定为 '-I !'。
  name 可以是文件名模板，与.cvsignore 文件中使用的类型相同[64]。
- -W spec
  指定导入时需要过滤的文件名。你可以重复使用此选项。
  spec 可以是文件名模板，与.cvswrappers 文件中使用的类型相同[65]。
- -X
  修改 cvs 的机制，让新导入的文件不会立即出现在主干上。
  特别是，此标识让 cvs 标注新文件如同它们在主干上已被删除，除了正常的导入之外，还需为每个文件进行下面的步骤: 在主干上创建新版本，标识新文件是 dead，重置新文件的默认分支，将文件放到 Attic[66]目录。

要强制在仓库范围内使用此选项，在 CVSROOT/config[67]文件设置 'ImportNewFilesToVendorBranchOnly' 选项。

22.14.2 import output

import 过程中会为每个文件打印一行进度信息，行头用一个子母标记文件状态:

- U file
  在仓库里面已经存在，并且本地没有修改; (如果需要) 会创建新的修订版。
- N file
  此为新文件，已经添加到仓库。
- C file
  仓库里面已经存在，并且本地有修改; 你需要合并变更。
- I file 文件被忽略 (参阅 cvsignore)。
- L file
  文件为符号链接; cvs import 忽略符号链接。经常有人建议修改这项行为，但改成什么样，还没有明确。(在 modules 文件里面有很多选项可以检出，更新等操作中重建符号链接; 参阅 modules)

[63]见 Substitution modes，了解有效的 '-k' 设置列表。
[64]参阅 cvsignore。
[65]参阅 Wrappers。
[66]参阅 Attic。
[67]参阅 config。

22.14.3 import examples

参阅 Tracking sources 和 From files。

22.15 log

语法：log [options] [files...]

需要：仓库，工作目录。

修改：无。

显示文件的日志信息。log 过去调用 rcs 的 rlog 工具。虽然现在不再使用，这段历史也会影响输出的格式和选项，使它跟其余的 cvs 命令风格都不相同。

输出信息包括 rcs 文件的位置，head 修订版 (主干上的最新修订版)，所有的符号名 (标签)，以及其他的字串。对于每个修订版，将打印版本号，日期，作者，添加/删除的行数，commitid 和日志消息。日期以本地的时间显示。通常在 $TZ 环境变量里面指定，可以设置以决定 log 如何显示日期。

注意：log 使用 '-R' 的方式与其他 cvs 不同 (参阅 Common options)。

22.15.1 log options

默认情况下，log 打印所有有效的信息。其他选项用来限制输出。注意，修订版选择选项 (-d, -r, -s 和 -w) 无效，其他可能搜索 Attic 目录里的文件，除非指定 -S 选项，否则联合使用此选项只限制输出 log 头字段 (-b, -h, -R, and -t)。

- -b
  打印默认分支上的修订版信息，通常是主干上最高的分支。
- -d dates
  打印修订版提交的日期，用分号分隔的日期列表指定范围。日期格式同 cvs 其他命令中的 '-D' 选项 (参阅 Common options)。日期可以用下面方式组合：
  - d1<d2
    d2>d1
    选择 d1 和 d2 之间的修订版。
  - <d d>
    选择 d 或之前的所有修订版。
  - d<
    >d
    选择日期 d 或之后的所有修订版。

  - d
    选择日期 d 或早期的单个最新的修订版。

  '>' 或 '<' 字符可以跟随 '=' 用来指明包括到范围之内，注意分隔符是分号 (;)。

- -h
  只打印 rcs 文件名，文件的名字是在工作目录，头，默认分支，访问列表，锁，符号名和后缀。
- -l
  只运行在当前工作目录。(默认是递归运行)。
- -N
  不打印文件的标签列表。当你有众多的标签时，这个选项很好用，日志信息将不打印标签，而不是让你用"more" 来看 3 页以上的标签信息。
- -R
  只打印 rcs 文件的名字。
- -rrevisions
  打印由逗号分隔的 revisions 列表指定范围内的修订版。下面解释可用的格式:
  - rev1:rev2
    修订版 rev1 至 rev2 (必须在同一个分支上)。
  - rev1::rev2
    同上，但不包含 rev1。
  - :rev ::rev 从分支开始到 REV 的修订版，包括 rev。
  - rev:
    从 rev 开始到包含 rev 分支的最后修订版。
  - rev:: 从 rev 之后开始到含 rev 的分支的最后修订版。
  - branch
    参数是分支表示此分支上的所有修订版。
    branch1:branch2
    branch1::branch2
    分支范围表示在此范围内的所有修订版。
    branch.
    branch 分支上的最新修订版。
    没有修订版的单独 '-r' 意思是默认分支上的最新修订版，通常是主干。在 '-r' 选项和其参数之间没有空格。
- -S

如没有选择修订版抑制。

- -s states
  打印匹配状态列表的修订版，列表 states 由逗号分隔。
- -t
  同 '-h'，外加说明文字。
- -wlogins
  打印列表指定用户提交的修订版，logins 列表用逗号分隔。如 logins 被忽略，假设是当前用户。在 '-w' 选项和其参数之间没有空格。log 打印 '-d', '-s' 和 '-w' 选项的修订版交集，'-b' 和 '-r' 选项修订版的并集。

22.15.2 log examples

即使 log 显示的是本地日期，你也可以显示 Coordinated Universal Time (UTC) 或其他时区。可以在 cvs 之前通过设置 $TZ 环境变量实现:

```
$ TZ=UTC cvs log foo.c
$ TZ=EST cvs log bar.c
```

(如果你用的是 csh 风格的 shell，比如 tcsh，你需要在上面例子中指定 env。)

22.16 ls & rls

ls [-e | -l] [-RP] [-r tag[:date]] [-D date] [path...]

需要：'rls' 需要仓库，'ls' 需要仓库和工作目。ls.

修改：无。

同义词：dir & list 是 ls 的同义词，rdir 和 rlist 是 rls 的同义词。

ls 和 rls 命令用来列出仓库里的文件和目录。

默认情况下，ls 列出属于工作目录里面的文件和目录，也就是 update 命令之后可以得到的。

rls 默认列出仓库中顶级目录里面的文件和目录。

这两个命令都接受可选的文件和目录名列表，ls 相对于工作目录，rls 相对于仓库顶级目录。默认都不递归操作。

22.16.1 ls & rls options

ls & rls 支持的标准选项:

- -d
  显示死亡的修订版 (可使用指定的标签)。
- -e
  显示 CVS/Entries 格式。此格式打算为自动化保留易解析性。
- -l
  显示所有的细节。
- -P
  递归时不列出空目录。
- -R
  递归执行。
- -r tag[:date]
  显示文件的指定的 tag，或者当 date 指定，并且 tag 是分支标签，分支 tag 上的版本可以当作是在 date 上。见 Common options。
- -D date
  显示指定日期的文件。

22.16.2 rls examples

```
$ cvs rls
cvs rls: Listing module: `.'
CVSROOT
first-dir
$ cvs rls CVSROOT
cvs rls: Listing module: `CVSROOT'
checkoutlist
commitinfo
config
cvswrappers
loginfo
modules
notify
rcsinfo
taginfo
verifymsg
```

22.17 rdiff

rdiff [-flags] [-V vn] (-r tag1[:date1] | -D date1) [-r tag2[:date2] | -D date2] modules...

需要：仓库。

修改：无。

同义词：patch

创建两个发行版之间的 Larry Wall 格式的 patch(1) 文件，它可以直接用于 patch 文件，让旧发行版更新到新发行版。(此为仅有的直接操作仓库的 cvs 命令之一，而不需要事先检出。) diff 的输出发送到标准输出设备上。

可以指定 (采用标准的 '-r' 和 '-D' 选项) 任何一个、两个修订版或日期的组合。如果指定一个修订版或日期，patch 反映的是此修订版或日期与当前 rcs 文件中最新修订版之间的差异。

注意，如果软件发行版影响多个目录，那么在给旧源码打补丁的时候，需要给 patch 命令指定 '-p' 选项，让 patch 找到处于其他目录中的文件。

22.17.1 rdiff options

rdiff 支持的标准选项[68]：

- -D date
  使用不迟于 date 的最新修订版。
- -f
  如果没有匹配的修订版，获取最新的修订版 (而不是忽略这些文件)。
- -l
  只在当前目录，不传递到子目录。
- -R
  递归方式检查。此为默认选项。
- -r tag
  使用 tag 修订版，或者当 date 指定，并且 tag 是分支标签，分支 tag 上的版本可以当作是在 date 上。见 Common options。

另外还支持以下选项：

- -c
  使用上下文 diff 格式。此为默认。
- -s

[68]参阅 Common options，了解完整说明。

创建更新总结报告，而不是补丁。总结中包括两个发行版间文件修改和添加的信息。它会被发送到标准输出设备。它的用处在于，例如，找出两个日期或发行版间有哪些文件被改动。

- -t
  两个修订版的顶端的差异，发送到标准输出设备。这对于查看文件最后的修改很有用。
- -u
  上下文差异中使用 unidiff 格式。记住，旧版本的 patch 程序不能处理 unidiff 格式，所以，如果打算将补丁发到网上，不该使用 '-u'。
- -V vn
  按照当前 rcs 版本 vn 的规则 (扩展格式在 rcs 版本 5 做了修改) 扩展关键字。注意此选项不再使用。cvs 只按 rcs 版本 5 的方式扩展关键字。

22.17.2 rdiff examples

假设你收到 foo@example.net 发来的邮件，索取 tc 编译器从 1.2 到 1.4 发行版的更新。你手头上没有这个补丁，但使用 cvs 可以用一个命令很容易得到:

```
$ cvs rdiff -c -r FOO1_2 -r FOO1_4 tc | \
$$ Mail -s 'The patches you asked for' foo@example.net
```

假如你生成了发行版 1.3，并为修复 bug 建立了 'R_1_3fix' 分支。以前还为发行版 1.3.1 建立 'R_1_3_1'。现在，你想了解分支上做了多少开发工作。可以用这个命令:

```
$ cvs patch -s -r R_1_3_1 -r R_1_3fix module-name
cvs rdiff: Diffing module-name
File ChangeLog,v changed from revision 1.52.2.5 to 1.52.2.6
File foo.c,v changed from revision 1.52.2.3 to 1.52.2.4
File bar.h,v changed from revision 1.29.2.1 to 1.2
```

22.18 release

release [-d] directories...

需要：工作目录。

修改：工作目录，历史日志。

这个命令用来安全地撤消 'cvs checkout' 的影响。因为 cvs 不锁文件，所以没必要使用这个命令。如果你原意，可以只是删除工作目录; 但风险是你可能忘记里面还有改动的地方，并且丢弃了检出，在 cvs 历史文件里面没有跟踪记录 (参阅 history file)。

使用 'cvs release' 可避免这些问题。该命令检测当前没有未提交的更改; 在 cvs 工作目录上层执行; 仓库记录的文件与模块数据库定义的相同。

如果所以条件成立，'cvs release' 在 cvs 历史日志里面留下执行的记录 (证明你的确要放弃检出的文件)。

22.18.1 release options

release 命令支持一个命令选项:

- -d
  如果成功，删除工作目录中文件的副本。如果没有给出此标识，你的文件仍然保留在工作目录里面。
  警告：release 命令递归地删除所有的目录和文件。有一个严重的副作用，在此目录里面创建的任何目录，它们没有添加到仓库 (使用 add 命令; 参阅 Adding files)，也将被没有提示地删除 – 即使不是空目录!

22.18.2 release output

在 release 释放源码之前，它会为没有更新的文件打印一行信息。

- U file
- P file
  仓库里面存有此文件的新修订版，本地副本没有修改 ('U' 和 'P' 是相同意思)。
- A file
  文件已经被添加，但没有提交到仓库。如果你删除，该文件将丢失。
- R file
  文件已经被删除，但没有提交到仓库，所以也没有从仓库删除。参阅 commit.
- M file
  文件已经在当前目录里面修改。这也许应该是仓库里面的一个新修订版。
- ? file
  file 处于工作目录，但是仓库里面没有对应的文件，它们也不是 cvs 忽略的文件[69]。
  如果删除，该文件将丢失。

[69]参见 '-I' 选项的说明，以及参阅 cvsignore。

22.18.3 release examples

释放 tc 目录，并且删除工作目录里面的文件副本。

```
$ cd ..       # You must stand immediately above the
              # sources when you issue `cvs release'.
$ cvs release -d tc
You have [0] altered files in this repository.
Are you sure you want to release (and delete) directory `tc': y
$
```

22.19 update

update [-ACdflPpR] [-I name] [-j rev [-j rev]] [-k kflag] [-r tag[:date] | -D date] [-W spec] files...
需要：仓库，工作目录。
修改: 工作目录。

在你从公共仓库检出创建自己的源码副本之后，其他的开发人员会继续修改中央源码。随着时间的推移，在开发进程中，需要的时候，你可以在工作目录中使用 update 命令，将上次检出或更新后仓库里面变更的修订版，并入到你的工作目录。在没有使用 -C 选项时，update 将合并本地副本和 -r，-D 或 -A 指定的修订版的差异。

22.19.1 update options

update 支持下面标准的选项[70]:

- -D date
  使用不迟于 date 的最新修订版。这是一个粘性的选项，含有 '-P'。见 Sticky tags，了解更多粘性标签/日期的信息。
- -f
  只与 '-D' 或 '-r' 标识一起使用。如果没有匹配的修订版，获取最新的修订版 (而不是忽略这些文件)。
- -k kflag
  根据 kflag 处理关键字。参阅 Keyword substitution。此选项是粘性的; 以后在这个工作目录里面更新，还是使用相同的 kflag。status 命令可以看到粘性的选项。参阅 Invoking CVS，了解 status 命令的信息。

[70]参阅 Common options，了解完整说明。

- -l
  只在当前工作目录里面执行。参阅 Recursive behavior.
- -P
  清除空目录。参阅 Moving directories.
- -p
  管道方式输出到标准输出。
- -R
  递归方式检出 (默认)。参阅 Recursive behavior.
- -r tag[:date]
  获得 rev 修订版/标签，或者当 date 指定，并且 tag 是分支标签，分支 tag 上的版本可以当作是在 date 上。此选项是粘性的，含 '-P'。参阅 Sticky tags 和 Common options 了解更多粘性标签/日期的信息。

update 还有一些特殊的选项。

- -A
  重置任何的粘性标签，日期，或 '-k' 选项。参阅 Sticky tags，了解更多粘性标签/日期的信息。
- -C
  用仓库里面干净的副本覆盖本地的修改 (但修改过的文件另存为.#file.revision)。
- -d
  创建仓库里面存在而工作目录里面没有的目录。通常，update 只作用于你工作目录里面已经存在的文件和目录。
  此选项通常用来更新最初检出创建的目录; 但也有不好的副作用。如果你在建立工作目录时，刻意避免仓库里面的某些目录 (通过模块名，或在命令行上明确指定所需的文件和目录)，用 '-d' 选项更新将创建这些你不想要的目录。
- -I name
  更新时忽略匹配 name 的文件 (在工作目录里面)。你可以在命令行上多次使用 '-I' 指定多个要忽略的文件。'-I !' 可以避免忽略任何文件。参阅 cvsignore, 了解 cvs 忽略文件的其他方式。
- -Wspec
  指定更新时需要过滤的文件名。你可以重复使用此选项。
  spec 可以是文件名模板，与.cvswrappers 文件里面的类型相同。参阅 Wrappers.
- -jrevision
  通过两个 '-j' 选项，合并第一个 '-j' 选项指定的修订版至第二个 'j' 选项修订版的变

更到工作目录。

使用一个 '-j' 选项，合并最初的修订版至 '-j' 选项指定修订版的变更到工作目录。最初的修订版是工作目录文件基于的修订版和 '-j' 选项指定修订版共同的祖先。

注意使用 '-j tagname' 选项而不是 '-j branchname'，合并的通常不是从分支上删除文件的变更[71]。

另外，每个 '-j' 选项可以包含可选的日期规格，当用于分支时，可以限制修订版处于指定日期之内[72]。可选的日期通过分号 (:) 加在标签里面: '-jSymbolic_Tag:Date_Specifier'。

22.19.2 update output

update 和 checkout 在执行中，会为每个文件打印一行提示信息，文件的状态通过前面的单个字符指明:

- U file
  文件按要求从仓库得到更新。用在那些仓库里面有但你的工作目录没有的文件，以及工作目录里面没有修改过，但旧于仓库的文件。
- P file
  类似 'U'，但是 cvs 服务器发送的是补丁而不是整个文件。完成与 'U' 同样的工作，但降低带宽的使用。
- A file
  添加到你的私人副本中，当你使用 commit 后会加到仓库。这可以提醒你需要提交文件。
- R file
  从你的私人副本中删除，当你执行 commit 命令后会从仓库清除。这可以提醒你文件需要提交。
- M file
  在你的工作目录中，文件已经修改。
  'M' 可以标明你工作的文件的两种状态: 同样的文件仓库里面没有修改，你的文件仍保持原样; 或者仓库里面的文件也有修改，但在工作目录里成功合并，没有冲突发生。
  如果合并，cvs 将打印一些信息，并建立工作文件的备份 (与 update 执行前相同)。update 运行时会打印相应的名字。
- C file

[71]参阅 Merging adds and removals, 了解详细情况。
[72]参阅 Branching and merging。

合并你与仓库中修改到 file 时检测到冲突。file (用户工作目录里面的副本) 是合并两个修订版的结果; 工作目录里面还有未修改文件的副本，名为.#file.revision，其中 revision 是你修改的文件所基于的修订版[73]。(注意，在有些系统里面，如果一段日子没有访问.# 起头的文件，系统会自动清除。如果你需要保留这些原始的文件副本，最好将其改名) vms 系统中，文件名以 __ 开始，而不是.#。

- ? file
  file 处于工作目录，但是仓库里面没有对应的文件，它们也不是 cvs 忽略的文件 (参见 '-I' 选项的说明，以及参阅 cvsignore)。

[73]解决冲突见 Conflicts example 说明。

Chapter 23

Invoking CVS

本附录说明如何执行 cvs，以及每个命令和特性详细的参考。可以运行 cvs –help 命令了解其他的参考。

一个 cvs 命令像这样:

cvs [global_options] command [command_options] [command_args]

全局选项:

- –allow-root=rootdir
  指定合法的 cvsroot 目录 (只用于服务器)(cvs 1.9 和更早版本无效)。见 Password authentication server。
- -a
  认证所有的通讯 (只用于客户端)(CVS 1.9 和更早版本无效)。见 Global options.
- -b
  指定 RCS 位置 (cvs 1.9 和更早版本)。见 Global options。
- -d root
  指定 cvsroot。见 Repository。
- -e editor
  使用 editor 编辑消息。见 Committing your changes。
- -f
  不读取 /.cvsrc 文件。见 Global options。
- -H
- –help
  打印帮助讯息。见 Global options。
- -n

不修改任何文件。见 Global options。

- -Q
  完全沉默。见 Global options。
- -q
  一定程度上沉默。见 Global options。
- -r
  让新工作文件只读。见 Global options。
- -s variable=value
  设置用户变量。见 Variables。
- -T tempdir
  将临时文件放到 tempdir。见 Global options。
- -t
  跟踪 cvs 执行。见 Global options。
- -v
- –version
  显示 cvs 的版本和版权信息。
- -w
  让新工作文件可读写。见 Global options。
- -x
  加密全部的通讯 (只用于客户端)。见 Global options。
- -z gzip-level
  设置压缩等级 (只用于客户端)。见 Global options。
  关键字扩展模式 (参阅 Substitution modes):

```
-kkv $Id: file1,v 1.1 1993/12/09 03:21:13 joe Exp $
-kkvl $Id: file1,v 1.1 1993/12/09 03:21:13 joe Exp harry $
-kk  $Id$
-kv  file1,v 1.1 1993/12/09 03:21:13 joe Exp
-ko  no expansion
-kb  no expansion, file is binary
```

关键字 (参阅 Keyword list):

```
$Author: joe $
$Date: 1993/12/09 03:21:13 $
$CVSHeader: files/file1,v 1.1 1993/12/09 03:21:13 joe Exp harry $
```

```
$Header: /home/files/file1,v 1.1 1993/12/09 03:21:13 joe Exp harry $
$Id: file1,v 1.1 1993/12/09 03:21:13 joe Exp harry $
$Locker: harry $
$Name: snapshot_1_14 $
$RCSfile: file1,v $
$Revision: 1.1 $
$Source: /home/files/file1,v $
$State: Exp $
$Log: file1,v $
Revision 1.1 1993/12/09 03:30:17 joe
Initial revision
```

命令，命令选项，以及命令参数:

add [options] [files...]

添加新文件/目录。见 Adding files。

- -k kflag
  设定关键字扩展。
- -m msg
  设定文件说明。

admin [options] [files...]

管理仓库里面的历史文件。见 admin。

- -b[rev]
  设定默认分支。见 Reverting local changes。
- -cstring
  设定注释头。
- -ksubst
  设定关键字替换。见 Keyword substitution。
- -l[rev]
  给修订版 REV，或最新修订版加锁。
- -mrev:msg
  用 msg 替换修订版 rev 的日志消息。
- -orange
  删除仓库里面的修订版。见 admin options。
- -q

安静地运行; 不打印任何诊断消息。

- -sstate[:rev]
  设定状态。
- -t
  从标准输入设定文件说明。
- -tfile
  从 file 设定文件说明。
- -t-string
  从 string 设定文件说明。
- -u[rev]
  给修订版 rev，或最新修订版解锁。

  annotate [options] [files...]
  列出最新修订版修改的每一行。见 annotate。
- -D date
  批注不迟于 date 的最新修订版。见 Common options。
- -F
  批注二进制文件。(没有此选项，二进制文件显示消息后跳过。)
- -f
  如果没有 tag/date，使用最新修订版。见 Common options。
- -l
  只运行在当前目录。参阅 Recursive behavior.
- -R
  递归方式处理 (默认)。参阅 Recursive behavior.
- -r tag[:date]
  批注修订版 tag。或者当 date 指定，并且 tag 是分支标签，分支 tag 上的版本可以当作是在 date 上。见 Common options。

  checkout [options] modules...
  获得源码的副本。见 checkout。
- -A
  重置任何粘性的 tags/date/options。见 Sticky tags 和 Keyword substitution。
- -c
  输出模块数据库。见 checkout options。
- -D date

检出 date (粘性的) 的修订版。见 Common options。

- -d dir
  检出到 dir。见 checkout options。
- -f
  如果 tag/date 找不到，使用 head 修订版。见 Common options。
- -j tag[:date]
  合并指定的 tag 修改。或者当 date 指定，并且 tag 是分支标签，分支 tag 上的版本可以当作是在 date 上。见 checkout options。
- -k kflag
  使用 kflag 关键字扩展。见 Substitution modes。
- -l
  只在当前工作目录里执行。参阅 Recursive behavior.
- -N
  如果指定 -d，不“缩短”模块路径。见 checkout options。
- -n
  不执行模块程序 (如有)。见 checkout options。
- -P
  删除空目录。见 Moving directories。
- -p
  检出文件到标准输出 (避免粘著)。见 checkout options。
- -R
  递归执行 (默认)。参阅 Recursive behavior.
- -r tag[:date]
  检出 tag 修订版 (粘性的)。或者当 date 指定，并且 tag 是分支标签，分支 tag 上的版本可以当作是在 date 上。见 Common options。
- -s
  类似 -c，但是包含模块状态。见 checkout options。
  commit [options] [files...]
  将修改提交到仓库。见 commit。
- -c
  在提交之前，检验有效的修改。要求 cvs 客户端和服务器都是 1.12.10 或更新版本。
- -F file
  从 file 读取日志消息。见 commit options。

- -f
  强制文件提交，禁止递归。见 commit options。
- -l
  只在当前目录运行。见 Recursive behavior。
- -m msg
  用 msg 作为日志消息。见 commit options。
- -n
  不执行模块程序 (如有)。见 commit options。
- -R
  递归执行 (默认)。参阅 Recursive behavior.
- -r rev
  提交到 rev。见 commit options。

  diff [options] [files...]
  显示修订版间的差异。见 diff。除了下面这些选项，它还可以使用大量输出控制格式选项，例如用于比较内容的 '-c'。
- -D date1
  比较此日期修订版与工作文件。见 diff options。
- -D date2
  比较 rev1/date1 和 date2。见 diff options。
- -l
  只运行在当前工作目录。见 Recursive behavior。
- -N
  比较包括添加和删除的文件。见 diff options。
- -R
  递归执行 (默认)。见参阅 Recursive behavior.
- -r tag1[:date1]
  比较 tag1 修订版和工作文件，或者当 date1 指定，并且 tag1 是分支标签，分支 tag1 上的版本可以当作是在 date1 上。见 diff options 和 Common options。
- -r tag2[:date2]
  比较 rev1/date1 与 tag2，或者当 date2 指定，并且 tag2 是分支标签，分支 tag2 上的版本可以当作是在 date2 上。见 diff options 和 Common options。

  edit [options] [files...]
  准备编辑被监视的文件。见 Editing files。

- -a actions
  指定临时监视的动作，其中 actions 是 edit, unedit, commit, all 或 none。见 Editing files。
- -c
  检验编辑: 如果有人已经编辑了文件，edit 将失败。需要 cvs 客户端和服务器的版本都在 1.12.10 或更高。
- -f
  强行编辑; 忽略其他的编辑。1.12.10 版添加。
- -l
  只在当前工作目录执行。见 Recursive behavior。
- -R
  递归执行 (默认)。参阅 Recursive behavior.
  editors [options] [files...]
  查看有谁在监视文件。见 Watch information。
- -l
  只在当前工作目录执行。见 Recursive behavior。
- -R
  递归执行 (默认)。参阅 Recursive behavior.
  export [options] modules...
  从 cvs 导出文件。见 export。
- -D date
  检出 date 的修订版。见 Common options。
- -d dir
  检出到 dir。见 export options。
- -f
  如有标签/日期，使用最新的修订版。见 Common options。
- -k kflag
  使用 kflag 关键字扩展。见 Substitution modes。
- -l
  只在当前工作目录执行。参阅 Recursive behavior.
- -N
  如果指定 -d，不“缩短”模块路径。见 export options。
- -n
  不执行模块程序 (如有)。见 export options。

- -R
  递归执行 (默认)。参阅 Recursive behavior.
- -r tag[:date]
  检出 tag 修订版，或者当 date 指定，并且 tag 是分支标签，分支 tag 上的版本可以当作是在 date 上。见 Common options。

  history [options] [files...]
  显示访问仓库的历史。见 history。
- -a
  所有用户 (默认只是自己)。见 history options。
- -b str
  显示在模块名/文件名/仓库中包含字符串 str 的记录。见 history options。
- -c
  报告提交 (修改) 的文件。见 history options。
- -D date
  从 date 开始。见 history options。
- -e
  所有的记录类型。见 history options。
- -l
  最后的修改 (提交或修改报告)。见 history options。
- -m module
  报告特定的 module (可重复)。见 history options。
- -n module
  在 module 里面。见 history options。
- -o
  报告检出的模块。见 history options。
- -p repository
  在 repository。见 history options。
- -r rev
  从 rev 修订版开始。见 history options。
- -T
  对所有的 TAG 生成报告。见 history options。
- -t tag
  自从标签报告放入历史文件 (任何人)。见 history options。

- -u user
  对用户 user (可重复)。见 history options。
- -w
  必须匹配工作目录。见 history options。
- -x types
  报告 types，TOEFWUPCGMAR 中的一个或多个。见 history options。
- -z zone
  输出时区 zone。见 history options。

  import [options] repository vendor-tag release-tags...
  导入文件至 cvs，使用供应商分支。见 import。
- -b bra
  导入到供应商 bra 分支。见 Multiple vendor branches。
- -d
  使用文件的修改时间做完导入时间。见 import options。
- -k kflag
  设置默认的关键字替换模式。见 import options。
- -m msg
  使用 msg 作为日志消息。见 import options。
- -I ign
  更多的忽略文件 (! 重置)。见 import options。
- -W spec
  更多的封装。见 import options。
- init
  如果没有 cvs 仓库，创建它。见 Creating a repository。
- kserver
  Kerberos 认证服务器。见 Kerberos authenticated。

  log [options] [files...]
  打印文件的历史信息。见 log。
- -b
  只列出默认分支的修订版。见 log options。
- -d dates
  指定日期 (d1<d2 是范围，d 是最新版之前)。见 log options。
- -h

只打印前面。见 log options。

- -l
  只在当前目录里面运行。见 Recursive behavior。
- -N
  不列出标签。见 log options。
- -R
  只打印 RCS 文件名。见 log options。
- -rrevs
  只列出 revs 修订版。见 log options。
- -s states
  只列出指定状态的修订版。见 log options。
- -t
  只打印前面和描述文字。见 log options。
- -wlogins
  只列出指定登录用户提交的修订版。见 log options。

  login
  提示输入认证服务器的密码。见 Password authentication client。

  logout
  删除储存的认证服务器密码。见 Password authentication client。

  pserver
  密码认证服务器。见 Password authentication server。

  rannotate [options] [modules...]
  显示最后修订版中所做的每一行修改。见 annotate。
- -D date
  批注不迟于 date 的最新修订版。见 Common options。
- -F
  强制批注二进制文件。(无此选项，跳过二进制文件，只显示信息。)
- -f
  如果没有找到标签/日期，使用最新修订版。见 Common options。
- -l
  只运行在当前工作目录。参阅 Recursive behavior.
- -R
  递归执行 (默认)。参阅 Recursive behavior.

- -r tag[:date]
  批注 tag 修订版，或者当 date 指定，并且 tag 是分支标签，分支 tag 上的版本可以当作是在 date 上。见 Common options。
  rdiff [options] modules...
  显示发行版间的差异。见 rdiff。
- -c
  上下文 diff 格式 (默认)。见 rdiff options。
- -D date
  基于 date 的修订版。见 Common options。
- -f
  如果没有匹配的修订版，使用最新的修订版。见 Common options。
- -l
  只运行在当前目录。Recursive behavior。
- -R
  递归方式操作 (默认)。参阅 Recursive behavior.
- -r tag[:date]
  基于 tag 的修订版，或者当 date 指定，并且 tag 是分支标签，分支 tag 上的版本可以当作是在 date 上。见 diff options 和 Common options。
- -s
  短 patch - 每个文件一行。见 rdiff options。
- -t
  diff 顶端 - 文件最后的修改。见 diff options。
- -u
  unidiff 输出格式。见 rdiff options。
- -V vers
  使用 RCS 版本 vers 的关键字扩展 (荒废)。见 rdiff options。
  release [options] directory
  从仓库里面删除条目。见 release。
- -d
  删除给出的目录。见 release options。
  remove [options] [files...]
  从仓库里面删除条目。见 Removing files。
- -f

在 remove 之前先删除文件。见 Removing files。

- -l
  只在当前工作目录里面运行。见 Recursive behavior。
- -R
  递归执行 (默认)。参阅 Recursive behavior.

rlog [options] [files...]
打印模块的历史信息。见 log。

- -b
  只列出默认分支的修订版。见 log options。
- -d dates
  指定日期 (d1<d2 是范围，d 是最新版之前)。见 log options。
- -h
  只打印前面。见 log options。
- -l
  只在当前目录里面运行。见 Recursive behavior。
- -N
  不列出标签。见 log options。
- -R
  只打印 RCS 文件名。见 log options。
- -rrevs
  只列出 revs 修订版。见 log options。
- -s states
  只列出指定状态的修订版。见 log options。
- -t
  只打印前面和描述文字。见 log options。
- -wlogins
  只列出指定登录用户提交的修订版。见 log options。

rtag [options] tag modules...
给模块加符号标签。见 Revisions 和 Branching and merging。

- -a
  清除删除文件上不需要的标签。见 Tagging add/remove。
- -b
  创建名为 tag 的分支。见 Branching and merging。

- -B
  联合 -F 或 -d 使用，允许移动和删除分支。使用时要特别小心。
- -D date
  给 date 的修订版打标签。见 Tagging by date/tag。
- -d
  删除 tag。见 Modifying tags。
- -F
  如果存在，移动 tag。见 Modifying tags。
- -f
  如果没有标签/日期，使用最新的修订版。见 Tagging by date/tag。
- -l
  只在当前工作目录运行。见 Recursive behavior。
- -n
  不执行 tag 程序。见 Common options。
- -R
  递归执行 (默认)。参阅 Recursive behavior.
- -r tag[:date]
  给已有的 tag 打标签，或者当 date 指定，并且 tag 是分支标签，分支 tag 上的版本可以当作是在 date 上。见 Tagging by date/tag 和 Common options。
- server
  rsh 服务器。见 Connecting via rsh。

  status [options] files...
  显示工作目录里的状态信息。见 File status。
- -l
  只在当前目录里面运行。见 Recursive behavior。
- -R
  递归执行 (默认)。参阅 Recursive behavior.
- -v
  包括文件的标签信息。见 Tags。

  tag [options] tag [files...]
  给检出的文件增加符号标签。见 Revisions 和 Branching and merging。
- -b
  创建名为 tag 的分支。见 Branching and merging。

- -c
  检验未修改的工作文件。见 Tagging the working directory。
- -D date
  给 date 的修订版打标签。见 Tagging by date/tag。
- -d
  删除 tag。见 Modifying tags。
- -F
  如果存在，移动 tag。见 Modifying tags。
- -f
  如果没有标签/日期，使用最新的修订版。见 Tagging by date/tag。
- -l
  只在当前工作目录运行。见 Recursive behavior。
- -R
  递归执行 (默认)。参阅 Recursive behavior.
- -r tag[:date]
  给已有的 tag 打标签，或者当 date 指定，并且 tag 是分支标签，分支 tag 上的版本可以当作是在 date 上。见 Tagging by date/tag 和 Common options。

  unedit [options] [files...]
  撤消编辑命令。见 Editing files。
- -l
  只在当前工作目录运行。见 Recursive behavior。
- -R
  递归执行 (默认)。参阅 Recursive behavior.

  update [options] [files...]
  同步工作目录与源码。见 update。
- -A
  重置所有的粘性标签/日期/选项。见 Sticky tags 和 Keyword substitution。
- -C
  用仓库里面干净的副本覆盖本地的修改 (但修改过的文件另存为.#file.revision)。
- -D date
  检出 date 的修订版 (粘性)。见 Common options。
- -d
  创建目录。见 update options。

- -f
  如没有标签/日期，使用最新修订版。见 Common options。
- -I ign
  忽略更多的文件 (! 重置)。见 import options。
- -j tag[:date]
  合并 tag 指定的修改，或者当 date 指定，并且 tag 是分支标签，分支 tag 上的版本可以当作是在 date 上。见 update options。
- -k kflag
  使用 kflag 关键字扩展。见 Substitution modes。
- -l
  只在当前工作目录里面运行。参阅 Recursive behavior.
- -P
  清除空目录。见 Moving directories。
- -p
  检出到标准输出 (避免粘性)。见 update options.
- -R
  递归操作 (默认)。参阅 Recursive behavior.
- -r tag[:date]
  检出 tag 修订版，或者当 date 指定，并且 tag 是分支标签，分支 tag 上的版本可以当作是在 date 上。见 Common options。
- -W spec
  更多封装。见 import options。
  version
  显示 cvs 使用的版本。如果使用远程仓库，同时显示客户端和服务器的版本。
  watch [on|off|add|remove] [options] [files...]
  on/off: 打开/关闭检出文件的只读。见 Setting a watch。
  add/remove: 添加或删除操作的提示。见 Getting Notified。
- -a actions
  指定临时监视的动作，其中 actions 为 edit, unedit, commit, all 或 none。见 Editing files。
- -l
  只在当前工作目录里面执行。见 Recursive behavior。
- -R
  递归方式检出 (默认)。参阅 Recursive behavior.

watchers [options] [files...]
查看谁在监视文件。见 Watch information。

- -l
  递归方式检出 (默认)。Recursive behavior。
- -R
  递归方式检出 (默认)。参阅 Recursive behavior.

Chapter 24

CVS Administrative Files

在 cvs 仓库之中，有一个 $CVSROOT/CVSROOT 目录，里面放着一些辅助文件[1]。如果你只使用 cvs 有限的特性，可以不需要这些文件，但当你能娴熟地运用这些辅助文件，生活将变得更美好。

这些文件中最重要的一个文件是 modules，它用于定义仓库中的模块。

24.1 modules

modules 文件记录用户为源代码集所定义的名称。如果用户使用 cvs 更新模块文件(比如 add，commit 这类常用的命令)，cvs 可以使用这些定义。

modules 文件中与模块定义一起，还可以包含空行和注释（以 '#' 号开始的行)。过长的行可以在行尾指定反斜杠（'\'）表示与下一行连接。

一共有三种模块：alias 模块，regular 模块和 ampersand 模块。它们之间的不同在于仓库和工作目录之间的映射方式。在接下来的例子中，仓库根目录里面有一个 first-dir，它包含两个文件，file1 和 file2，以及一个目录 sdir。first-dir/sdir 里面有一个 sfile 文件。

24.2 Wrappers

封装 (Wrappers) 指的是 cvs 的一种特性，它可以让你控制基于被操作文件的文件名的设定。设定中 '-k' 用于二进制文件，'-m' 用于不可合并的文本文件。

'-m' 选项指定非二进制文件更新时应当采用的合并方法。MERGE 是 cvs 通常的行为: 尝试合并文件。COPY 是让 cvs update 拒绝合并文件，像用 '-kb' 指定为二进制文件那样

[1]关于如何编辑这些文件，参阅 Intro administrative files。

(但对指定为二进制的文件，没有必要用 '-m 'COPY")。cvs 将提供给用户文件的两个版本，让用户使用 cvs 之外的机制来插入任何必要的修改。

警告: 不要在 cvs 1.9 之前的版本中使用 COPY - 那些版本的 cvs 将复制一个版本的文件覆盖另外一个，清除以前的内容。使用 '-m' 封装选项只作用于更新时的合并行为；它不影响文件如何存储。参阅 Binary files，了解二进制文件更多信息。

cvswrappers 的基本格式为:

wildcard [option value][option value]...

其中 option 为下列之一

- -m update methodology value: MERGE or COPY
- -k keyword expansion value: expansion mode

value 用单引号指明。

例如，下列命令导入一个目录，将其中的 '.exe' 结尾文件当作二进制文件:

```
cvs import -I ! -W "*.exe -k 'b'" first-dir vendortag reltag
```

24.3 Trigger Scripts

在 cvs 命令执行过程中，有许多管理文件支持触发事件，或者在特定事件发生之前或之后，加载外部脚本或程序脚本。这些 hook 可以用来阻止某些动作，记录日志，并/或做一些你认为需要的维护。

在客户机 -服务器模式，所有的触发脚本在服务器上都在用户提交的沙盘副本里面加载。在本地模式下，脚本直接从提交的用户沙盘里面加载。对于大多数用途，相同的脚本可以不经修改地用在两种方式下。

24.4 rcsinfo

rcsinfo 文件用来指定提交日志填充的样式。rcsinfo 的语法与 verifymsg, commitinfo and loginfo 文件类似。参阅 syntax. 与其他文件不同，它的第二部分不是命令行模板。而在正则表达式后面，是一个包含日志消息模板的文件的完整路径。

如果仓库名不与这个文件里面的正则表达式匹配，将使用指定的 'DEFAULT' 行。

除了第一个匹配正则表达式或者 'DEFAULT' 之外，其他都用 'ALL' 作为正则表达式。

这个日志消息模板将用作默认的日志消息。如果你用 'cvs commit -m message' 或 'cvs commit -f file' 指明一个日志消息，它将覆盖模板。

参阅 verifymsg, 关于 rcsinfo 文件例子。

在 cvs 存取远程仓库过程中，当目录首次检出时，rcsinfo 的内容将指定一个模板。该模板会在 'cvs update' 命令下更新。它也会在 'cvs add new-directory' 命令执行时添加到新的目录。在 cvs 1.12 以前的版本中，CVS/Template 文件不会更新。如果 cvs 服务器的版本等于 1.12，或高于旧的客户端，CVS/Template 可能会从服务器上更新。

24.5 cvsignore

在工作目录中经常会有一些文件，但我们却不想将它们置于 cvs 控制之下。比如那些编译源码产生的目标文件。通常在执行 'cvs update' 命令后，会为每个不认识的文件打印一行信息 (参阅 update output)。

cvs 有一个文件列表 (或 sh(1) 文件名模板)，将在执行 update，import 和 release 时忽略它们。这个文件列表由以下方式构成。

这个文件列表初始包括这样的文件名模板：用于 cvs 管理，或是其他的源码控制系统的文件名称；补丁文件，目标文件，存档文件，和编辑备份文件的名称；以及一些相关工具产生的文件的名称。当前，默认的忽略文件模板为：

```
RCS     SCCS    CVS     CVS.adm
RCSLOG  cvslog.*
tags    TAGS
.make.state     .nse_depinfo
*~      #*      .#*     ,*      _$*     *$
*.old   *.bak   *.BAK   *.orig  *.rej   .del-*
*.a     *.olb   *.o     *.obj   *.so    *.exe
*.Z     *.elc   *.ln
core
```

如果每个源码库中有 $CVSROOT/CVSROOT/cvsignore 文件存在，它将附加在这个列表中。

如果每个用户的 home 目录中有.cvsignore 文件，它将附加在这个列表中。

环境变量 $CVSIGNORE 所指也将附加在这个列表中。

任何 cvs 命令中的 '-I' 选项也附加在其中。

当 cvs 遍历目录，目录中的.cvsignore 将添加到该列表中。.cvsignore 中的模板仅作用于包含此文件的目录，不影响其他子目录。在以上所列的 5 个地方，使用惊叹号 ('!') 可以清除忽略列表。用于保存通常被 cvs 忽略的文件。

给 cvs import 命令指定 '-I !' 将导入所有文件，一般用于导入一些来自原始出处或者

认为源码里面没有多余文件的情况下。然而，检查上述规则，将会发现美中不足之处；如果发行文件中包括.cvsignore，即使使用 '-I !'，CVS 也会按照该模板的规则处理。唯有删除.cvsignore 文件才能按照最初目的导入文件。因为这是一个缺点，将来 '-I !' !' 可能会覆盖每个目录中的.cvsignore。

注意，忽略文件的语法中包含很多行，每行为空格分开的文件名列表。这造成没有一个简单的方法用于包含空格的文件名，但我们可以用 foo?bar 来匹配 foo bar (当然它也能匹配 fooxbar)。还要注意当前不支持注释。

24.6 checkoutlist

用 cvs 也能帮助我们维护 CVSROOT 目录中自己的文件。例如，设想有一个 logcommit.pl 脚本文件，执行它，要在 commitinfo 管理文件中包含：

```
ALL  $CVSROOT/CVSROOT/logcommit.pl %r/%p %s
```

要用 CVS 维护 logcommit.pl，需要将下面一行添加到 checkoutlist 文件：

```
logcommit.pl
```

checkoutlist 文件的格式是，需要 cvs 维护的文件名子单独放在一行，给定文件名，接着是可选的空格和提交后该文件无法检出到 CVSROOT 时打印的错误消息:

```
logcommit.pl    Could not update CVSROOT/logcommit.pl.
```

按照上述式样设置好 checkoutlist，列于其中的文件将得到与 cvs 内置管理文件相当的功能。例如，导入其中的一个文件，会得到以下信息：

```
cvs commit: Rebuilding administrative file database
```

接着 CVSROOT 目录中的文件也得到更新。

注意，出于安全考虑，请不要将 passwd(参阅 Password authentication server) 列在 checkoutlist 文件里面。

关于不使用 checkoutlist 保持导出副本的通用形式，请参考 Keeping a checked out copy。

24.7 history file

$CVSROOT/CVSROOT/history 用来记录 history 命令的信息 (参阅 history)。这个文件创建以后才能打开日志功能。如果你用 cvs init 命令建立仓库 (参阅 Creating a repository)，它会自动创建。

history 文件格式仅在 cvs 源码注释中说明，一般程序应该使用 cvs history 访问，防止将来 cvs 发行版会改变格式。

24.8 Variables

在写管理文件的时候，你想让该文件可以知道 cvs 运行环境的一些情况。

寻找运行 cvs 用户的 home 目录 (从 HOME 环境变量)，使用 ‘ ’，紧跟着是 ‘/’，或者行尾。同样对应 user 的 home 目录，使用 ‘ user’。这些变量在服务器上展开，但如果使用 pserver (参阅 Password authenticated) 不会得到任何合理的扩展; 因此定制用户执行 cvs 时的行为，采用用户变量 (见下) 可能是较好的选择。

有人可能想了解 cvs 内部的各个部分信息。cvs 内部变量使用 ${variable} 语法，其中 variable 以字母开头，并由字目数字和 ‘_’ 组成。如果 variable 后面的字符是非字母数字及 ‘_’，符号 ‘{’ 和 ‘}’ 将被忽略。cvs 内部变量有:

- CVSROOT
  它是当前 cvs 根目录的绝对路径。参阅 Repository, 了解指定它的各种方式，但要注意，内部变量只包含目录，但不包括任何访问方式。
- RCSBIN
  cvs 1.9.18 及以前，它指定 cvs 要寻找的 rcs 程序目录。因为 cvs 不再需要运行 rcs 程序，现在指定该内部变量会出错。
- CVSEDITOR
- EDITOR
- VISUAL
  它们扩充是相同值，即 cvs 使用的编辑器。参阅 Global options, 了解如何指定。
- USER
  当前运行 cvs 的用户名 (在 cvs 服务器上)。当使用 pserver 时，它是仓库里面指定的用户，可以与服务器上运行的不同 (参阅 Password authentication server)。不要因为使用了同样的名字，将它与环境变量混淆。
  如果要传递一个值到管理文件, 并由运行 cvs 的用户指定，请使用用户变量。为了扩展用户变量，管理文件要包含 ${=variable}。为了设置用户变量，请在 cvs 里面指定 ‘-s’ 全局选项，然后使用 variable=value 参数。把它在.cvsrc (参阅 /.cvsrc) 里面设置特别有用。
  例如，如果你想在管理文件里面指定一个测试目录，可以创建用户变量 TESTDIR。然后启动 cvs

```
cvs -s TESTDIR=/work/local/tests
```

管理文件里面包含的 sh ${=TESTDIR}/runtests，将被扩展为 sh /work/local/tests/runtests。

其余包含 '$' 的字符串将保留; 由于没有引用 '$' 字符的方式，所以它还保持原样。

传递到管理文件的环境变量有:

- CVS_USER
  如果能提供 (当前只是用于 pserver 访问方式)，是用户提供的 cvs 指定的用户名，其他方式为空串。(当使用 $CVSROOT/CVSROOT/passwd 把 cvs 用户名映射到系统用户名时，CVS_USER 和 USER 会不一致)。
- LOGNAME
  系统用户的用户名。
- USER
  与 LOGNAME 相同。请勿将它与内部变量因为同名而混淆。

24.9 config

config 管理文件包含影响 cvs 行为的各种设定。其语法与其他的管理文件略有不同。不会扩展变量。以 '#' 开头的行是注释。其他的行以关键字，'=' 和值组成。要留意语法要求很严格。额外的空格与制表符都不允许。

现在定义的关键字有:

- RCSBIN=bindir
  对 cvs 1.9.12 到 1.9.18 之间的版本，该设置告诉 cvs 用于在 bindir 目录里面搜索 rcs 程序。当前的 cvs 不再使用 rcs 程序；为了兼容该设置仍被保留，但不起任何作用。
- SystemAuth=value
  如果 value 是 'yes'，pserver 将在 CVSROOT/passwd 没有对应用户的情况下检查系统用户。如果设为 'no'，那么所有的 pserver 用户都必须在 CVSROOT/passwd 里面存在。默认使用 'yes'。更多 pserver 的信息，参阅 Password authenticated。
- LocalKeyword=value
  指定标准关键字的本地别名。例如，'LocalKeyword=MYCVS=CVSHeader'。更多本地关键字的信息，参阅 Keyword substitution。
- KeywordExpand=value
  指定 'i' 开始的可扩展的关键字列表 (例如，'KeywordExpand=iMYCVS,Name,Date')，或者 'e' 开始的不可扩展的关键字列表 (例如，'KeywordExpand=eCVSHeader')。更多

关键字扩展的信息，参阅 Configuring keyword expansion。

- TopLevelAdmin=value
  修改 'checkout' 命令，除了在检出目录里面，还在新工作目录顶级创建 'CVS' 目录。默认值为 'no'。
  如果你要在工作目录的顶级，而不是检出的子目录执行许多命令，该选项就很有用。在那里创建 CVS 目录意味着你不用为每个命令指定 CVSROOT。它还为 CVS/Template 文件提供了一个场所 (参阅 Working directory storage)。
- LockDir=directory
  将 cvs lock 文件置于 directory 目录，而不是仓库中的目录。这对于需要给用户仓库访问权限，可以只给它们 directory 的写权限，而不是仓库。还可以将 lock 放到快速的使用内存的文件系统，以加速仓库的加锁和解锁。你需要自己创建 directory，cvs 将在需要的时候创建 directory 下的子目录。关于 cvs 锁，参阅 Concurrency。
  在启用 LockDir 选项之前，确保你查找并删除了 cvs 1.9 或早期版本。这些版本不支持 LockDir，并不会给出不支持的错误信息。结果将会是一些 cvs 用户将锁放在这个地方，其他用户放在另外的地方，仓库自然就被破坏了。cvs 1.10 仍不支持 LockDir，但是假如运行的仓库启用了 LockDir，它会打印一个警告。
- LogHistory=value
  控制哪些将记录到 CVSROOT/history 文件 (参阅 history)。默认是 'TOEFWUPCG-MAR' (或简写 'all'。任何默认子集都合法。(例如，只记录对 *,v 文件的修改，使用 'LogHistory=TMAR'。)
- RereadLogAfterVerify=value
  修改 'commit' 命令，使得 CVS 在执行完 verifymsg 里面指定的程序之后，重读日志消息。当 VALUE 是 'yes' 或 'always' 时，指明总是去重读日志消息; 'no' 或 'never'，指明不会去重读; 或者 value 是 'stat'，指明文件应该在重读之前通过文件系统 'stat()' 函数检测，判断是否已经修改 (参阅下面警告)。默认值为 "always'。
  注意: 'stat' 模式可能会造成 CVS 在提交每个目录时暂停数秒。这也可能不消耗多少 IO 和 CPU，但还是不建议在大型的仓库里面使用
  参阅 verifymsg, 以了解更多关于如何使用 verifymsg 的信息。
- UserAdminOptions=value
  控制哪些 cvs admin 命令 (参阅 admin)) 选项可以被不属于 cvsadmin 组里面的用户使用。value 字符串是允许的选项列表。不属于 cvsadmin 组的用户，如果执行 cvs admin 的选项没有列在其中，将获得选项受限的错误消息。
  如果服务器上没有 cvsadmin 组存在，cvs 将忽略 UserAdminOptions 关键字 (参阅

admin)。

没有指定时，UserAdminOptions 默认为 'k'。也就是说，默认只允许 cvsadmin 组以外的用户通过 cvs admin admin' 命令修改文件的关键字扩展模式。

例如，让 cvsadmin 组以外的用户可以使用 cvs admin 命令修改默认的关键字替换模式，锁住修订版，开锁修订版和替换日志消息，设定为 'UserAdminOptions=klum'。

- UseNewInfoFmtStrings=value
  指定 cvs 在提交支持文件 (参阅 commit files) 时，是支持新的还是旧的命令行模板模型。此配置变量准备放弃，现在只是给用户一定时间来更新旧仓库，在删除旧语法之前可以使用新的格式化字符串。关于更新仓库以支持新模型，请参考 Updating Commit Files。

  注意，新的仓库 (使用 cvs init 命令创建) 将此值设为 'yes'，但默认值是 'no'。
- ImportNewFilesToVendorBranchOnly=value
  指定 cvs import 是否总是执行如命令行上使用 '-X' 标识的行为。value 可以是 'yes' 或 'no'。如果设为 'yes'，所有的用户执行 cvs import 的结果，就像 '-X' 已经设置。默认值为 'no'。
- PrimaryServer=CVSROOT
  指定后，如果 CVSROOT 指定的仓库与当前访问的不同，那么服务器转换为 CVSROOT 的透明代理用于写请求。作为 CVSROOT 一部分的 hostname，必须与当前工作中的主服务器 uname 命令返回的字串相同。域名解析是通过组合的 named，/etc/hosts 的行，以及 Network Information Service (NIS 或 YP)，这取决于系统的配置。当前只支持 ':ext:' 方式 (实际上，':fork:' 也支持，但只用于测试 - 如果你发现通过 ':fork:' 方式有其他的用途，请发一个提示到 bug-cvs@gnu.org)。参阅 Write proxies 了解有关配置和使用写代理的情况。
- MaxCommentLeaderLength=length
  设置一些尺寸，以字节，或者 'k', 'M', 'G', 'T' 让前面的数字分别解释为千字节，兆字节，亿字节，或 T 字节，使 Log 关键字 (参阅 Keyword substitution)，在一行里大于指定 length 字节时被忽略 (或者退到 RCS 档案文件里设置的 comment leader - 参见下面的 UseArchiveCommentLeader)。默认在检出时只处理 20 字节，是防止用户疏忽，没有将二进制文件标明时，二进制文件里面含有 Log 关键字的情况。
- UseArchiveCommentLeader=value
  设为 true，如果前面文本里有 Log 关键字，超过 MaxCommentLeaderLength 设置的字节，如果有，会被替代。如果在档案文件里面没有设置 comment leader，或者 value 设为 'false'，关键字将不会被替换 (参阅 Keyword list)。为了强制使用 RCS 档案

文件里面的 comment leader(并且档案文件里面没有设置 comment leader，文件里面的 Log 扩展会跳过)，设置 value 且 MaxCommentLeaderLength 设为 0。

Chapter 25

CVS Environment Variables

这里是影响 cvs 的全部环境变量的完整列表。

- $CVSIGNORE
  cvs 应该忽略的文件名列表模板，用空格分开。参阅 cvsignore.
- $CVSWRAPPERS
  CVS 应该当作封装的文件名列表，用空格分开。参阅 Wrappers.
- $CVSREAD
  如果设置，checkout 和 update 将更改你工作目录下的文件为只读。如果未设，默认的行为允许修改工作目录下的文件。
- $CVSREADONLYFS
  打开只读仓库模式。允许从只读仓库检出，例如从匿名服务器，或 cd-rom 仓库。在命令行上使用 '-R' 选项与它有相同的效果。这也允许使用只读的 NFS 仓库。
- $CVSUMASK
  控制仓库中文件的权限。参见 File permissions。
- $CVSROOT
  应包含指向 cvs 源码仓库 (rcs 文件保存的地方) 根目录的完整路径。大多数 cvs 命令要有该信息存在才能执行; 如果 $CVSROOT 没有设置，或者你希望一次实施中覆盖它，可以在命令行里面提供: 'cvs -d cvsroot cvs_command...' 一旦你检出到工作目录，cvs 会保存适当的 root (在文件 CVS/Root 里面)，因此通常你只在最初检出到工作目录时关心它。
- $CVSEDITOR
- $EDITOR
- $VISUAL

指定提交时记录日志消息用的程序。$CVSEDITOR 覆盖 $EDITOR，而它又覆盖 $VISUAL。参见 Committing your changes 了解更多信息或 Global options 如何用其他方法指定日志编辑器。

- $PATH
  如果没有设置 $RCSBIN，并没有指定路径编译进 cvs，它将使用 $PATH 寻找所有使用的程序。
- $HOME
- $HOMEPATH
- $HOMEDRIVE
  用来确定.cvsrc 或类似文件的搜索目录。在 Unix 上，cvs 只检查 HOME。在 Windows NT 上，系统设置 HOMEDRIVE，例如设为 'd:'，以及 HOMEPATH 设为 \joe。在 Windows 95 上，你需要自己设置 HOMEDRIVE 和 HOMEPATH。
- $CVS_RSH
  当访问模式涉为:ext: 时，指定 cvs 连接时使用的外部程序。参阅 Connecting via rsh.
- $CVS_SERVER
  使用 rsh 访问远程仓库时，用于客户机 -服务器模式。当访问模式使用:ext:, :fork: 或:server: 时，指定服务器端启动的程序名 (以及一些必要的参数)。对于:ext: 和:server:，默认值是 cvs; 而:fork: 的默认值与运行在客户端的相同。参阅 Connecting via rsh
- $CVS_PASSFILE
  使用 cvs login server 时，用于客户机 -服务器模式。默认值为 $HOME/.cvspass。参阅 Password authentication client。
- $CVS_CLIENT_PORT
  当通过 Kerberos，GSSAPI 或 cvs 的密码认证协议访问服务器时，如果在 CVSROOT 里面没有指定端口，用于客户机 -服务器设置端口。参阅 Remote repositories
- $CVS_PROXY_PORT
  当通过 web 代理访问服务器时，如果在 CVSROOT 里面没有指定端口，用于客户机 -服务器模式设置端口。与 GSSAPI 和密码验证协议一同工作。参阅 Remote repositories
- $CVS_RCMD_PORT
  用于客户机 -服务器模式。如果设置，当访问服务器端的 rcmd 守护进程时，指定端口号。(现在不用于 Unix 客户端)。
- $CVS_CLIENT_LOG

在客户机 -服务器模式下，用来调试。如果设置，发到服务器的任何信息都记录到 $CVS_CLIENT_LOG.in，从服务器发来的任何信息都记录到 $CVS_CLIENT_LOG.out。

- $CVS_SERVER_SLEEP
  在客户机 -服务器模式下，用来调试。如果设置，启动服务器子进程时会延迟数秒，以便附加调试器。
- $CVS_IGNORE_REMOTE_ROOT
  对于 cvs 1.10 和更早的版本，设置该变量后，可以阻止 cvs 用指定 '-d' 全局选项覆盖 CVS/Root 文件。新的 cvs 版本不再改写 CVS/Root，所以 CVS_IGNORE_REMOTE_ROOT 已经无效。
- $CVS_LOCAL_BRANCH_NUM
  设置该变量，允许一些控制可以越过分配的版本号。它特别是支持 CVSup 本地提交的特性。如果设置 CVS_LOCAL_BRANCH_NUM 为 (声明) 1000，那么就建立本地仓库分支，修订版号就会像 1.66.1000.xx。可以确定没有与版本号冲突。
- $COMSPEC
  只在 OS/2 下使用。它指定命令解释器的名字，默认为 cmd.exe。
- $TMPDIR
- $TMP
- $TEMP
  临时文件保存的目录。cvs 服务器使用 TMPDIR。参阅 Global options, 了解如何指定。cvs 的某些部分总是使用 /tmp(通过系统提供的 tmpnam 函数)。
  在 Windows NT 上，使用 TMP(通过系统提供的 _tempnam 函数)。
  cvs 客户端的 patch 程序使用 TMPDIR，如果没有设置，将使用 /tmp(至少是 GNU patch 2.1)。注意，如果你的服务器和客户机都运行的是 cvs 1.9.10 或更新的版本，cvs 将不再调用外部的 patch 程序。
- $CVS_PID
  它是 cvs 进程的进程识别符 (也程 pid)。常常用于程序和/或 commitinfo, verifymsg, loginfo 指定的脚本之中。

Chapter 26

CVS Compatibility

仓库格式的兼容性可以回溯到 cvs 1.3。如果你有 cvs 1.6 或更早的副本，并且你使用了开发人员社区的可选特性，但请参考 Watches Compatibility。

工作目录格式的兼容性可以回溯到 cvs 1.5。在 1.3 到 1.5 之间做了一些变动。如果你从 cvs 1.3 检出的工作目录运行 cvs 1.5 或较新的版本，cvs 将转换它，但要复原到 cvs 1.3，你需要用 cvs 1.3 检出到一个新的工作目录。

远程访问协议的互操作性可以回溯到 cvs 1.5，但到此为止 (1.5 是使用远程协议的第一个正式发布版，一些更旧的版本也许能行)。大多少情况下，你需要同时升级客户端和服务器，以获得新特性的优点，及修正的错误。

Chapter 27

CVS Troubleshooting

如果你在使用 cvs 时遇到故障，本附录会有一定帮助。如果看到错误信息，你可以根据字母顺序查找消息。如果没有，你可以在提到该错误的相关章节查看。

27.1 Error messages

这里是你从 cvs 看到的部分错误消息列表。它不是一个完整的列表——cvs 可以打印很多很多的错误消息，其中一部分通常由操作系统提供，这里是要列出通用和/或易误解的错误消息。

这些消息按字母顺序排列，但是指导文字如 'cvs update: ' 不按这种次序。

某些情况下列表包含了旧版本 cvs (部分原因是因为用户在特定时刻无法确认所使用的 cvs 版本) 打印的消息。

file:line: Assertion 'text' failed

准确的消息格式会因不同的系统而变。它指出 cvs 里面有 bug，参见 BUGS 了解如何处理。

cvs command: authorization failed: server host rejected access

连接 pserver 服务器，认证失败时的一般的响应。检查用户名与密码是否正确，在 inetd.conf 的 '–allow-root' 是否允许 CVSROOT。参见 Password authenticated。

cvs command: conflict: removed file was modified by second party

此消息显示你删除了一个文件，但同时另外有人还在修改它。要解决此冲突，首先要运行 'cvs add file'。如果需要，查看别人的修改以决定是否仍旧删除。如果不再删除，结束。如果仍要删除，继续执行 'cvs remove file' 命令并提交。

cannot change permissions on temporary directory

Operation not permitted

当我们在 Red Hat Linux 3.0.3 和 4.1 上测试客户机/服务器测试套件时，此消息以不重现和偶然的方式出现。我们不清楚造成的原因，也无法知道是否仅在 Linux (甚至是仅在某个特定的机器) 上出现。如果问题出现在其他 Unix 机器上，'Operation not permitted' 可能是读了 'Not owner' 或者是有问题系统使用了 unix EPERM 错误。如果你有其他信息补充，请参考 BUGS 然后让我们知道。使用 cvs 时，你碰到这个错误，再执行操作应该可以工作良好。

cvs [server aborted]: Cannot check out files into the repository itself

产生这个消息明显的原因 (特别是非客户机/服务器 cvs)，是由于 cvs 的根为，比如，/usr/local/cvsroot，但你要检出到它的子目录，像 /usr/local/cvsroot/test。但还有一种微妙的原因，是将服务器上的临时目录设为根的子目录 (这也是不允许的)。如果这就是原因，将临时目录改成其他地方，例如 /var/tmp; 参考 Environment variables 里面的 TMPDIR，了解如何设置临时目录。cannot commit files as 'root'

参阅 ''root' is not allowed to commit files'.

cannot open CVS/Entries for reading: No such file or directory

通常这是一个 cvs 内部错误，需要通过解决其他 cvs bug 处理 (参阅 BUGS)。一般情况下有一个工作环境 – 性质决定于状态，但希望能指出。

cvs [init aborted]: cannot open CVS/Root: No such file or directory

此消息是无害的。无其他错误时，操作可以成功执行。此消息不会出现在现在的 cvs 版本里面，写在这里是为 cvs 1.9 和之前的版本。

cvs server: cannot open /root/.cvsignore: Permission denied

cvs [server aborted]: can't chdir(/root): Permission denied

参阅 Connection.

cvs [checkout aborted]: cannot rename file file to CVS/„file: Invalid argument

该消息在 Solaris 2.5 的 cvs 1.9 上偶尔会出现。原因不明; 如果你知道造成的原因请让我们知道，见 BUGS。

cvs [command aborted]: cannot start server via rcmd

很不幸，如果你运行的客户端是 cvs 1.9，这是相对非特异性的错误消息，并在连接服务器时有错误。当前版本的 cvs 应该显示更多特定的错误消息。如果你看到这个消息，并没有打算作为客户端，你应该指定:local:，参考 Repository。

ci: file,v: bad diff output line: Binary files - and /tmp/T2a22651 differ

在 cvs 1.9 和更早版本中，如果 rcs 没有正确安装，提交二进制文件，将显示此消息。请再读一次 rcs 发布版的指示以及 cvs 发布版的 install 文件。或者，升级到当前的 cvs 版

本，它不会使用 rcs，而是自己提交文件。

cvs checkout: could not check out file

在 cvs 1.9 中，这意味着 co 程序 (rcs 的一部分) 返回错误。在此之前应该还有其他的错误消息，但没看到其他错误消息会让这个信息难以理解。在当前版本的 cvs 中，因为不再使用 co，如果没有伴随其他错误消息，可以说是 cvs 的 bug (参阅 BUGS)。

cvs [login aborted]: could not find out home directory

意思是你需要设置环境变量，使得 cvs 可以找到 home 目录。参见 Environment variables 里面的 HOME，HOMEDRIVE 和 HOMEPATH 讨论。

cvs update: could not merge revision rev of file: No such file or directory

在 cvs 1.9 和更早的版本中，如果搜索 rcsmerge 程序有问题，将显示此消息。请确保它在你的 PATH 之中，或者更新到现在的 cvs 版本，因为它不再使用外部的 rcsmerge 程序。

cvs [update aborted]: could not patch file: No such file or directory

意思是搜索 patch 程序遇到问题。请确保它在你的 PATH 之中。注意，尽管显示的消息不是指是否可以找到 file。如果客户端和服务器都运行当前版本的 cvs，它们不使用外部的 patch 程序，所以也不会有这样的消息。但如果客户端或服务器运行的是 cvs 1.9，那么你就要使用 patch 程序。

cvs update: could not patch file; will refetch

这意味着客户端无论如何也不会打服务器发来的补丁。因为不能打补丁，只会降低速度，对 cvs 也没有影响，所以消息是无关紧要的。

dying gasps from server unexpected

在 cvs 1.9.18 和更早的版本中，这是一个已知的 bug。我使用 '-t' 全局选项时会重现。如果有人想了解情况，它已经被 Andy Piper 于 1997 年 11 月 14 日更改 src/filesubr.c 而修复。如果你见到这个消息，可在执行一次失败的操作，并让我们知道，见 BUGS。

end of file from server (consult above messages if any)

通常造成的原因是你使用了外部 rsh 程序，并返回一个错误。在这种情况下，rsh 程序应该在上面消息前面打印一个错误消息。了解建立 CVS 客户机和服务器的更多信息，见 Remote repositories。

cvs [update aborted]: EOF in key in RCS file file,v

cvs [checkout aborted]: EOF while looking for end of string in RCS file file,v

这意味着在 rcs 文件里面有语法错误。注意，即使 rcs 可以正确读取这个文件; cvs 要对 rcs 文件做更多的错误检测。这也是从 cvs 1.9 升级到 1.10 时看到这个消息的原因。可能是硬件、操作系统或者类似的因素造成错误。当然，如果你发现是 cvs 造成的，请报

告 (参阅 BUGS)。这个错误消息会因 cvs 在 rcs 文件里面发现的不同语法错误，有其他一些变化。

cvs commit: Executing 'mkmodules'

这个消息意味着你的参考是在 cvs 1.8 之前的版本下建立的。当使用 cvs 1.8 以后的版本，这个消息前会有

cvs commit: Rebuilding administrative file database

如果你看到所有的两条消息，数据库重建了两次，虽没有必要，但是无害。如果你希望避免重复，并且没有使用 cvs 1.7 和更早的版本，对每个出现的 modules 文件，用删除 -i mkmodules。modules 文件的更多信息，见 modules。

missing author

当你创建 RCS 文件时将用户名设为空，这条消息将显示。cvs 将假装创建一个作者字段没有值的非法的 RCS。解决方法是确保用户名非空并重新建立 RCS 文件。

cvs [checkout aborted]: no such tag tag

此消息的意思是 cvs 不知道 tag 标签。经常是由于你打错了标签名。偶尔也会因创建的标签的用户没有权限写 CVSROOT/val-tags 文件 (参考参阅 File permissions 了解更多情况)。

在 cvs 1.12.10 之前的版本里面，有时是一些隐含的原因造成，其中标签是在仓库里面的档案文件中创建，但是 cvs 要求用户试用其他的涉及此标签的 cvs 命令，直到发现让 cvs 更新 val-tags 文件的命令，然后最初失败的命令才能工作。同样可以修理因前面权限造成 val-tags 过期的问题。每个标签只需要更新一次 - 一旦标签列入 val-tags 文件，它就留在那里。

注意使用 'tag -f' 不要求标签匹配，也不覆盖此检查 (参阅 Common options)。

PANIC administration files missing

一般这是因为名为 cvs 的目录下并没有包含 cvs 使用的管理文件。如果这个问题是因为该 cvs 目录是由非 cvs 程序创建，那么简单的处理方法是将它改成其他名字。如果不是上述原因，这意味着是 cvs 的 bug (参阅 BUGS)。

rcs error: Unknown option: -x,v/

这条消息之后会紧跟着 rcs 的使用方法。这意味着你使用的是旧版本的 rcs (也许是你的操作系统提供的)，和旧版本的 cvs。在 cvs 1.9.18 和更早的版本只与 rcs 5 及以后的版本一起工作; 当前版本的 cvs 不再需要运行 rcs 程序。

cvs [server aborted]: received broken pipe signal

这条消息是因为 loginfo 程序从标准输入读取所有的日志信息失败造成。如果你发现它在其他环境下产生，请让我们知道，见 BUGS。

'root' is not allowed to commit files

当提交一个永久性的更改，cvs 会为提交修改的人建立一个日志条目。如果你的提交被当作"root" (不是使用"su" 或者其他具有 root 授权的程序) 记录，cvs 将无法判断是谁做了真正的修改。正因为此，CVS 默认不允许登录为"root" 来提交。(你可以在 configure 里面加上 –enable-rootcommit 选项并重新编译来禁止此选项。在有些系统上面需要在编译 cvs 前修改对应的 config.h 文件)。

Too many arguments!

一般该消息是 log.pl 脚本打印，此脚本位于 cvs 源码发布版的 contrib 目录下面。在某些版本的 cvs 中，log.pl 是 cvs 默认安装的一部分。log.pl 脚本从 loginfo 管理文件里面调用。请检查传递给 loginfo 的参数是否匹配你所用的 log.pl 脚本。特别是 cvs 1.3 和更早的版本里面的 log.pl 用日志文件作为参数，而 cvs 1.5 和更新的版本的 log.pl 需要用 '-f' 来指定日志文件。当然，如果你不需要 log.pl，请从 loginfo 里面将其注释掉。

cvs [update aborted]: unexpected EOF reading file,v

参见 'EOF in key in RCS file'.

cvs [login aborted]: unrecognized auth response from server

这个消息一般是指服务器没有正确建立。例如，inetd.conf 指向了一个不存在的 cvs 执行文件。为了进一步调试，请查找 inetd 的日志文件 (/var/log/messages 或者其他你系统里面 inetd 所采用的)。详细情况，参阅 Connection，和 Password authentication server。

cvs commit: Up-to-date check failed for 'file'

它的意思是自从你上次 cvs update 以后，有人提交了这个文件的修改。所以，在进行 cvs commit 之前需要先 cvs update。cvs 将合并你与他人所做的修改。如果没有检测到冲突，将提示 'M file'，你可以进行 cvs commit。如果发现冲突，将打印 'C file'，你需要手动解决冲突。要了解更详细的处理过程，见 Conflicts example。

Usage: diff3 [-exEX3 [-i | -m] [-L label1 -L label3]] file1 file2 file3

Only one of [exEX3] allowed

这指明了安装 diff3 和 rcsmerge 的问题。特别是 rcsmerge 编译需要寻找 GNU diff3，但由搜索到的 unix diff3 替代。确切的消息内容会根据不同的系统变化。最简单的解决方法是升级当前的 cvs 版本，它不再需要使用外部的 rcsmerge 或 diff3 程序。

warning: unrecognized response 'text' from cvs server

假如 text 包含有意义的回答 (像 'ok')，跟随着一个额外的回车符 (在某些系统上，这会使第二部分的消息覆盖第一部分)，那么也许是你用 ':ext:' 访问方式所采用的 rsh 程序，如非 unix 的 rsh 版本，它们默认不提供传输数据报文。在这种情况下，你也许需要采用 ':server:' 替代 ':ext:'。如果 text 是其他内容，它预示着你的 cvs 服务器有问题。请再次确认

安装是否符合 cvs 服务器架设的要求。

cvs commit: [time] waiting for user's lock in directory

这是一条普通消息，不是错误。参见 Concurrency 了解详细情况。

cvs commit: warning: editor session failed

这是指 cvs 使用的编辑器返回了非零状态。一些版本的 vi 即使编辑正常也会这样。那么将 CVSEDITOR 环境变量指向一个小的脚本，如：

```
#!/bin/sh
vi $*
exit 0
```

cvs [server aborted]: Secondary out of sync with primary!

这通常出现是 cvs 运行在一个第二服务器上面，并且第一服务器 (参阅 Write proxies) 与其不是同一个。如果客户端支持重定向，将不会发生。

这里版本号并不重要，但是支持列表需要服务器提供给客户端。因此，如果第二服务器编译的有 GSSAPI 支持，然而第一服务器没有，那么两个服务器提供的支持列表会不同，并且第二服务器将不能作为第一服务器的传输代理。反之，如果两个服务器都提供相同的客户端请求支持，一个的版本可能是 1.12.10，另一个可能是 1.12.11。

27.2 Connection

本节关注的是连接 cvs 服务器问题的解决方法。如果你在 Windows 上运行 cvs 命令行客户端，首先要更新客户端到 cvs 1.9.12 或之后的版本。较早版本的错误报告只能对问题提供很少的信息。如果客户端是在非 Windows 上，cvs 1.9 就可以。

如果错误信息不足以追踪错误，下一步取决于你使用的访问方式。

:ext:

在命令行上运行 rsh 程序。例如: "rsh servername cvs -v" 会打印 cvs 的版本。如果不能正常输出，在担忧 cvs 问题之前应先解决它。

:server:

使用这种访问方式不需要使用命令行的 rsh 程序，但如果你有 rsh 程序，它可以用作测试工具。见:ext: 相关指引。

:pserver:

一般对于"connection refused" 错误，是因为 inetd 没有监听 2401 端口。而像"connection reset by peer"，"received broken pipe signal"，"recv() from server: EOF" 或"end of file from server" 这种典型的问题，是因为 inetd 监听了连接但无法启动 cvs (常常因为 inetd.conf 使用了不

正确的路径或防火墙阻止了连接)。”unrecognized auth response” 错误是 inetd.conf 中错误的命令行造成，像无效的选项或忘记将 ‘pserver’ 命令置于行末。另外一种可能的原因是编辑器添加了不可见的控制字符，而没有给出提示。

一个很好用的调试工具是”telnet servername 2401”。连接以后，发任意一个文本 (例如”foo” 并回车)。如果 cvs 工作正常，将回显

cvs [pserver aborted]: bad auth protocol start: foo

如果是看到:

Usage: cvs [cvs-options] command [command-options-and-arguments] ...

那么应该是忘记在 inetd.conf 行末加上 ‘pserver’ 命令; 检验一下，确保整个命令处于一行并且完整。

同样，如果你得到:

Unknown command: ‘pserved’

CVS commands are: add Add a new file/directory to the repository ...

也就是在某处拼写错了 ‘pserver’。如果不那么明显，检查一下 inetd.conf 里面的非显示控制字符 (特别是回车)。

如果完全不工作，那么先确保 inetd 可以正常使用。修改 inetd.conf，用 echo 替换里面的 cvs 调用。例如:

2401 stream tcp nowait root /bin/echo echo hello

修改之后让 inetd 重新读取配置文件，”telnet servername 2401” 应该回显 hello，然后服务器关闭连接。如果也不工作，你应该在考虑 cvs 之前先解决这个问题。

在 AIX 系统上，2401 端口会被系统自己的程序使用。这是 AIX 的问题，因为 2401 是 cvs 注册使用的端口。我听说有一个 AIX 补丁可以解决这个问题。

另外一个很好的调试工具是在 inetd 上使用 ‘-d’ (调试) 选项。研究一下系统文件掌握如何使用。

如果好像已经连接，但得到如下的错误:

cvs server: cannot open /root/.cvsignore: Permission denied

cvs [server aborted]: can’t chdir(/root): Permission denied

那么你也许没有在 inetd.conf 里面指定 ‘-f’。(在 cvs 1.11.1 以前的发行版，这个问题是因为环境变量 $HOME$ □□*inetd*□□□□□□□□□□□□□□,□□□□*inetd*□□□□□□□,□□□*unset*HOME，再运行 cvs，或者使用 env 在原始环境下运行 cvs。)

如果你开始可以成功连接，然后有不行了，可能是达到了 inetd 的限制。(如果 inetd 短期内同一个服务接收了过多的请求，它会认为出错，并暂时禁止服务。) 查看一下 inetd 的文档，找到如果调节限制 (有些版本的 inetd 只有一个限制可以调，另外的可以为每个

服务单独设置。)

27.3 Other problems

这里列出不属于上面类别的问题。它们也不按什么次序。

在 Windows 上，运行 cvs 命令时如果有超过 30 秒的延时，这可能是因为你将 home 目录设为 C:/ 这样 (参阅 Environment variables 里面的 HOMEDRIVE 和 HOMEPATH)。CVS 要求 home 目录结尾没有斜扛，例如 C: 或 C:\cvs。

如果你运行的是 cvs 1.9.18 或更早的版本，执行 cvs update 发现有冲突并合并，像 Conflicts example 那样，但没有提示你有冲突发生，你也许是有一个旧版本的 rcs。最简单的解决方法是升级到最新的 cvs，因为它不再需要外部的 rcs 程序。

Chapter 28

CVS Manual Credits

在 Cygnus 的支持下，Roland Pesch <roland@wrs.com> 为 cvs 1.3 配写了一个使用手册，本文档的很多内容都是直接复制这个手册。他也阅读了本文档早期的一些草稿，提出了很多建议，以及一些更正。

info-cvs 邮件列表时常提供很多知识。我引用了下列人员所发邮件中的信息: David G. Grubbs <dgg@think.com>。

某些内容摘录自 rcs 使用手册。

David G. Grubbs 提供的 cvs faq 是一份很有价值的资料。虽然该 faq 已不再维护，但它仍是新手们寻求各种问题答案的首选 (至少在介绍如何使用 cvs 方面)。

除此之外，下面的这些人员曾经指出了我犯过的一些错误:

Roxanne Brunskill <rbrunski@datap.ca>,

Kathy Dyer <dyer@phoenix.ocf.llnl.gov>,

Karl Pingle <pingle@acuson.com>,

Thomas A Peterson <tap@src.honeywell.com>,

Inge Wallin <ingwa@signum.se>,

Dirk Koschuetzki <koschuet@fmi.uni-passau.de>

and Michael Brown <brown@wi.extrel.com>.

在此列出的有贡献者名单并不完整; 如果你想看一个更完整的名单，可以查阅 cvs 源码包中的 doc/ChangeLog 文件。

Chapter 29

CVS Bugs

cvs 和本手册都不是完美的，将来也不会是。如果你使用 cvs 中遇到麻烦，或者认为找到了一个 bug，这里有一些处理方法。如果认为本手册解释得不够清楚，这可以被认为是手册的一个 bug，解决这些问题与解决 cvs 中的问题同样有价值。

如果需要别人帮助你修补遇到的 bug，有一些公司可以提供有偿服务。其中的一家公司为：

Ximbiot

319 S. River St.

Harrisburg, PA 17104-1657

USA

Email: info@ximbiot.com

Phone: (717) 579-6168

Fax: (717) 234-3125

http://ximbiot.com/

如果你从分发商那里得到的 cvs，比如操作系统提供商或自由软件 cd-rom 提供商，你可以去了解一下他们所提供的支持。通常他们不提供或提供很少的支持，这取决于分发商。

如果你有能力和时间，可以尝试自己来修改 bug。如果你希望能在以后的 cvs 发布版中包含你的补丁，请参考 cvs 源码中的 hacking 文件。那里包含了关于提交补丁足够的信息。

互联网上有很多可提供帮助的资源。其中一个很好的起点是：

http://www.cvshome.org

如果你有什么灵感，将它们公布到互联网上是最好的方法。例如，在 cvs 官方版本

支援 Windows 95 之前，有一个网页包含说明和运行在 Windows 95 上的补丁，在官方 cvs 支持之前，许多人在邮件列表和新闻组中寻求帮助时都得益于此网页。

另外还可以提交 bug 报告至 bug-cvs@gnu.org。如果你需要一个关于手册中某选项的解决方案，其他人可能会也可能不会响应你的 bug 报告。人们一般会处理特别严重或者容易处理的 bug。为了你的报告有人关注，请尽量描述 bug 的本质和提供相关信息。提交 bug 报告的方法是发送邮件到 bug-cvs@gnu.org。请注意，任何提交到 bug-cvs@gnu.org 的内容将会被置于 gnu Public License 之下，如果你不原意，请不要提交。还有直接发送邮件给维护人员并不比发送到 bug-cvs@gnu.org 妥当；这些维护人员希望从 bug-cvs@gnu.org 读取 bug 报告。另请注意，发送 bug 报告到其他邮件列表或新闻组不会转到 bug-cvs@gnu.org。在你喜欢的论坛讨论 cvs bug 也不错，但维护人员只从 bug-cvs@gnu.org 中获取 bug 报告。

人们总是寻问是否有一个列表列出已知 bug 或者该 bug 是否已经被发现。在 cvs 分发的源代码中有 bugs 文件记录已知的 bug，但是它并不包括所有 bug。将来也可能不会有一个完整的、详细的已知 bug 列表。

Part III

Version Control with Subversionn

Ben Collins-Sussman

Brian W. Fitzpatrick

C. Michael Pilato

Subversion is a free/open source version control system (VCS). That is, Subversion manages files and directories, and the changes made to them, over time. This allows you to recover older versions of your data or examine the history of how your data changed. In this regard, many people think of a version control system as a sort of "time machine."

Apache Subversion（简称 SVN，svn），是一个开放源代码的版本控制系统，相对于的 RCS、CVS，采用了分支管理系统，它的设计目标就是取代 CVS。互联网上越来越多的控制服务从 CVS 转移到 Subversion。

Subversion 是一个自由/开源的版本控制系统。也就是说, 在 Subversion 管理下, 文件和目录可以超越时空。也就是 Subversion 允许你数据恢复到早期版本，或者是检查数据修改的历史。正因为如此, 许多人将版本控制系统当作一种神奇的"时间机器"。

Subversion 的版本库可以通过网络访问，从而使用户可以在不同的电脑上进行操作。从某种程度上来说，允许用户在各自的空间里修改和管理同一组数据可以促进团队协作。因为修改不再是单线进行，开发速度会更快。此外，由于所有的工作都已版本化，也就不必担心由于错误的更改而影响软件质量——如果出现不正确的更改，只要撤销那一次更改操作即可。

Some version control systems are also software configuration management (SCM) systems. These systems are specifically tailored to manage trees of source code and have many features that are specific to software development—such as natively understanding programming languages, or supplying tools for building software. Subversion, however, is not one of these systems. It is a general system that can be used to manage any collection of files. For you, those files might be source code—for others, anything from grocery shopping lists to digital video mixdowns and beyond.

某些版本控制系统本身也是软件配置管理（SCM）系统，这种系统经过精巧的设计，专门用来管理源代码树，并且具备许多与软件开发有关的特性——比如，对编程语言的支持，或者提供程序构建工具。不过 Subversion 并不是这样的系统。它是一个通用系统，可以管理任何类型的文件集。对你来说, 这些文件这可能是源程序—而对别人，则可能

是一个货物清单或者是数字电影。

如果你是一个考虑如何使用 Subversion 的用户或管理员，你要问自己的第一件事就是：”这是这项工作的正确工具吗？”，Subversion 是一个梦幻般的锤子，但要小心不要把任何问题当作钉子。

如果你希望归档文件和目录旧版本，有可能要恢复或需要查看日志获得其修改的历史，那么 Subversion 是你需要的工具。如果你需要和别人协作文档（通常通过网络）并跟踪所做的修改，那么 Subversion 也适合。这是 Subversion 为什么使用在软件开发环境——编程是天生的社会活动，Subversion 使得与其他程序员的交互变得简单。当然，使用 Subversion 也有代价：管理负担。用户会需要管理一个存放所有历史的数据版本库，并需要经常的备份。而在日常的工作中，用户不能像往常一样复制, 移动, 重命名或删除文件，相反，需要通过 Subversion 完成这些工作。

Assuming you're fine with the extra workflow, you should still make sure you're not using Subversion to solve a problem that other tools solve better. For example, because Subversion replicates data to all the collaborators involved, a common misuse is to treat it as a generic distribution system. People will sometimes use Subversion to distribute huge collections of photos, digital music, or software packages. The problem is that this sort of data usually isn't changing at all. The collection itself grows over time, but the individual files within the collection aren't being changed. In this case, using Subversion is "overkill." There are simpler tools that efficiently replicate data without the overhead of tracking changes, such as rsync or unison.

假定你能够接受额外的工作流程，你一定要确定不要使用 Subversion 来解决其他工具能够完成的很好的工作。例如，因为 Subversion 会复制所有的数据到参与者，一个常见的误用是将其作为普通的分布式系统。问题是此类数据通常很少修改，在这种情况下，使用 Subversion 有点“过了”。有一些可以复制数据更简单的工具, 没有必要过度的跟踪变更，例如 rsync 或 unison。

In early 2000, CollabNet, Inc. (http://www.collab.net) began seeking developers to write a replacement for CVS. CollabNet offered a collaboration software suite called CollabNet Enterprise Edition (CEE), of which one component was version control. Although CEE used CVS as its initial version control system, CVS's limitations were obvious from the beginning, and CollabNet knew it would eventually have to find something better. Unfortunately, CVS had become the de facto standard in the open source world largely because there wasn't anything better, at least not under a free license. So CollabNet determined to write a new version control system from scratch, retaining the basic ideas of CVS, but without the bugs and misfeatures.

早在 2000 年，CollabNet, Inc. (http://www.collab.net) 就开始寻找 CVS 替代产品的开

发人员。CollabNet 提供了一个名为 CollabNet 企业版 (CEE) 的协作软件套件。这个软件套件的一个组成部分就是版本控制系统。尽管 CEE 在最初采用了 CVS 作为其版本控制系统，但是 CVS 的局限性从一开始就很明显，CollabNet 知道，迟早要找到一个更好的替代品。遗憾的是，CVS 已经成为开源世界事实上的标准，很大程度上是因为没有更好的替代品，至少是没有可以自由使用的替代品。所以 CollabNet 决定从头编写一个新的版本控制系统，这个系统保留 CVS 的基本思想，但是要修正其中错误和不合理的特性。

In February 2000, they contacted Karl Fogel, the author of Open Source Development with CVS (Coriolis, 1999), and asked if he'd like to work on this new project. Coincidentally, at the time Karl was already discussing a design for a new version control system with his friend Jim Blandy. In 1995, the two had started Cyclic Software, a company providing CVS support contracts, and although they later sold the business, they still used CVS every day at their jobs. Their frustration with CVS had led Jim to think carefully about better ways to manage versioned data, and he'd already come up with not only the Subversion name, but also the basic design of the Subversion data store. When CollabNet called, Karl immediately agreed to work on the project, and Jim got his employer, Red Hat Software, to essentially donate him to the project for an indefinite period of time. CollabNet hired Karl and Ben Collins-Sussman, and detailed design work began in May 2000. With the help of some well-placed prods from Brian Behlendorf and Jason Robbins of CollabNet, and from Greg Stein (at the time an independent developer active in the WebDAV/DeltaV specification process), Subversion quickly attracted a community of active developers. It turned out that many people had encountered the same frustrating experiences with CVS and welcomed the chance to finally do something about it.

最初的设计小组设定了简单的开发目标。他们不想在版本控制方法学中开垦处女地，他们只是希望修正 CVS。他们决定 Subversion 应符合 CVS 的特性，并保留相同的开发模型，但不再重复 CVS 的一些显著缺陷。尽管 Subversion 并不需要成为 CVS 的完全替代品，但它应该与 CVS 保持足够的相似性，以使 CVS 用户可以轻松的转移到 Subversion 上。

经过 14 个月的编码，2001 年 8 月 31 日，Subversion 能够“自己管理自己”了，开发者停止使用 CVS 保存 Subversion 的代码，而使用 Subversion 本身。

While CollabNet started the project, and still funds a large chunk of the work (it pays the salaries of a few full-time Subversion developers), Subversion is run like most open source projects, governed by a loose, transparent set of rules that encourage meritocracy. In 2009, CollabNet worked with the Subversion developers towards the goal of integrating the Subversion project into the Apache Software Foundation (ASF), one of the most well-known collectives of open source projects in the world.

Subversion's technical roots, community priorities, and development practices were a perfect fit for the ASF, many of whose members were already active Subversion contributors. In early 2010, Subversion was fully adopted into the ASF's family of top-level projects, moved its project web presence to http://subversion.apache.org, and was rechristened "Apache Subversion".

虽然 CollabNet 启动了这个项目，并且一直提供了大量的工作支持（它为一些全职的 Subversion 开发者提供薪水），但 Subversion 像其它许多开源项目一样，被松散的、透明的规则管理着，这样的规则激励着知识界的精英们。CollabNet 的版权许可证完全符合 Debian 的自由软件方针。也就是说，任何人都可以根据自己的意愿自由的下载、修改和重新发布 Subversion，不需要 CollabNet 或其他人的授权。

Chapter 30

Subversion Architecture

Subversion 设计总体上的“俯视图”如下：

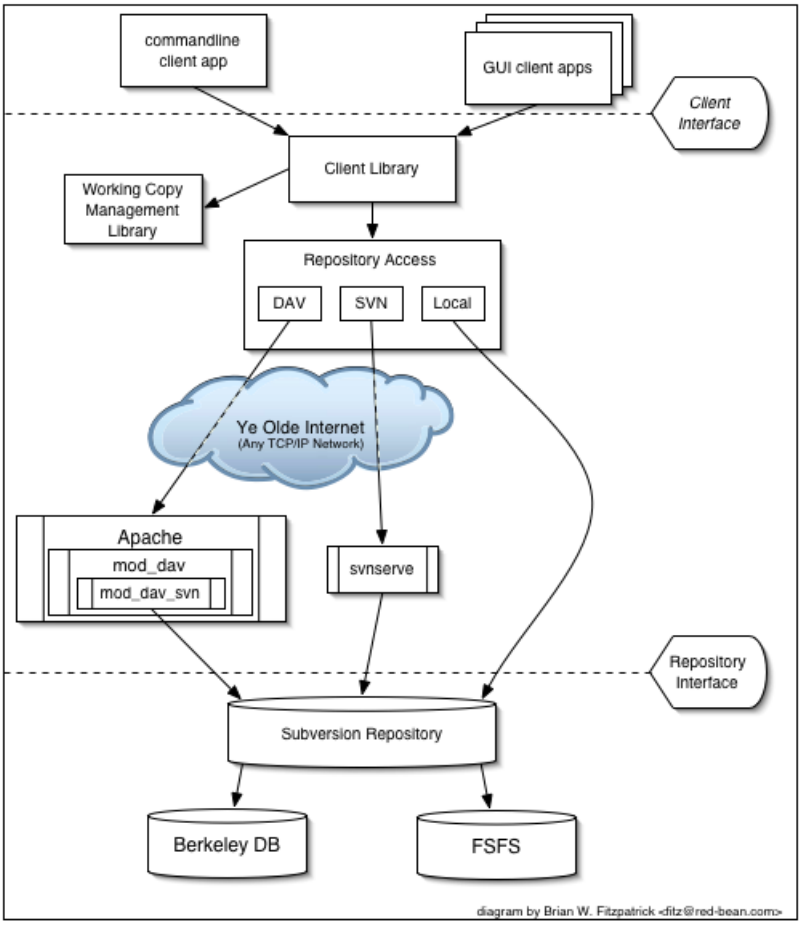


Figure 30.1: Subversion architecture

On one end is a Subversion repository that holds all of your versioned data. On the other end is your Subversion client program, which manages local reflections of portions of that versioned data. Between these extremes are multiple routes through a Repository Access (RA) layer, some of which go across computer networks and through network servers which then access the repository, others of which bypass the network altogether and access the repository directly.

图中的一端是保存所有版本数据的 Subversion 版本库，另一端是 Subvesion 的客户程序，管理着所有版本数据的本地影射 (称为“工作拷贝”)，在这两极之间是各种各样的版本库访问 (RA) 层，某些使用电脑网络通过网络服务器访问版本库，某些则绕过网络服务器直接访问版本库。

Chapter 31

Subversion Components

Subversion 由如下部分组成：

- svn
  命令行客户端程序。
  While Subversion runs on a number of different operating systems, its primary user interface is command-line-based (svn). The svn program also runs on non-Unix platforms such as Microsoft Windows. With a few minor exceptions, such as the use of backward slashes (\) instead of forward slashes (/) for path separators, the input to and output from this tool when run on Windows are identical to that of its Unix counterpart.
  svn 程序也可以在如 Microsoft Windows 这样的非 Unix 平台上运行，除了一些微小的不同，如使用反斜线（\）代替正斜线 (/) 作为路径分隔符，在 Windows 上运行 svn 程序的输入和输出与在 Unix 平台上运行完全一致。
- svnversion
  此工具用来显示工作副本的状态[1]。
- svnlook
  直接查看 Subversion 版本库的工具
- svnadmin
  建立，调整和修复 Subversion 版本库的工具
- mod_dav_svn
  Apache HTTP 服务器的一个插件，使版本库可以通过网络访问
- svnserve
  A custom standalone server program, runnable as a daemon process or invokable by SSH;

[1]用术语来说，就是当前项目的修订版本。

another way to make your repository available to others over a network

- svndumpfilter
  过滤 Subversion 版本库转储数据流的工具
- svnsync
  一个通过网络增量镜像版本库的程序

Chapter 32

Subversion History

- Subversion 1.1 (2004 年 9 月)
  1.1 版本引入了 FSFS，纯文件的版本库存储选项。虽然 Berkeley DB 后端被广泛的使用，但因为 FSFS 其较低的门槛和较小的管理需要，FSFS 还是成为新建版本库的缺省的选项。另外这个版本能够将符号链纳入版本控制，能够自动封装 URL，还有本地化的用户界面。
- Subversion 1.2 (2005 年 5 月)
  1.2 版本引入了文件在服务器端锁定的功能，实现对特定资源的顺序访问。虽然 Subversion 一直基本上是一个并行版本控制系统，特定类型的的二进制文件（例如艺术作品）不能合并在一起，锁定特性填补了对此类资源的版本化保护。随着锁定也引入了一个完整的 WebDAV 自动版本实现，允许 Subversion 版本库作为网络文件夹加载。最后，Subversion 1.2 开始使用新的，更快的二进制差异算法来压缩和检索文件的旧版本。
- Subversion 1.3 (2005 年 12 月)
  1.3 版本为 svnserve 服务器引入路径为基础的授权控制，与 Apache 服务器对应的特性匹配。Apache 服务器自己也获得了新的日志特性，Subversion 和其它语言的 API 绑定也取得了巨大的进步。
- Subversion 1.4(2006 年 9 月)
  1.4 版本引入了完全的新工具—svnsync—用来通过网络完成单向的版本库复制。一个重要的部分是工作副本元数据得到修补，不再使用 XML（获得客户端的速度改善），而 Berkeley DB 版本库后端获得了在发生崩溃时自动恢复的能力。
- Subversion 1.5 (2008 年 6 月)
  Release 1.5 took much longer to finish than prior releases, but the headliner feature was gi-

gantic: semi-automated tracking of branching and merging. This was a huge boon for users, and pushed Subversion far beyond the abilities of CVS and into the ranks of commercial competitors such as Perforce and ClearCase. Subversion 1.5 also introduced a bevy of other user-focused features, such as interactive resolution of file conflicts, sparse checkouts, client-side management of changelists, powerful new syntax for externals definitions, and SASL authentication support for the svnserve server.

- Subversion 1.6 (2009 年 3 月)

  Release 1.6 continued to make branching and merging more robust by introducing tree conflicts, and offered improvements to several other existing features: more interactive conflict resolution options; de-telescoping and outright exclusion support for sparse checkouts; file-based externals definitions; and operational logging support for svnserve similar to what mod_dav_svn offered. Also, the command-line client introduced a new shortcut syntax for referring to Subversion repository URLs.

虽然 Subversion 最普遍的用途还是跟踪代码变化，但是 Subversion 可以用来管理任何类型的数据——图像、音乐、数据库、文档等等。对于 Subversion，数据就是数据而已。

Chapter 33

Subversion Fundamental Concepts

A version control system (or revision control system) is a system that tracks incremental versions (or revisions) of files and, in some cases, directories over time. Of course, merely tracking the various versions of a user's (or group of users') files and directories isn't very interesting in itself. What makes a version control system useful is the fact that it allows you to explore the changes which resulted in each of those versions and facilitates the arbitrary recall of the same.

In this section, we'll introduce some fairly high-level version control system components and concepts. We'll limit our discussion to modern version control systems—in today's interconnected world, there is very little point in acknowledging version control systems which cannot operate across wide-area networks.

33.1 Subversion Repository

We've mentioned already that Subversion is a modern, network-aware version control system. A repository serves as the core storage mechanism for Subversion's versioned data, and it's via working copies that users and their software programs interact with that data.

Subversion implements the concept of a version control repository much as any other modern version control system would. Unlike a working copy, a Subversion repository is an abstract entity, able to be operated upon almost exclusively by Subversion's own libraries and tools. As most of a user's Subversion interactions involve the use of the Subversion client and occur in the context of a working copy, we spend the majority of this book discussing the Subversion working copy and how to manipulate it.

In Subversion, the client-side object which every user of the system has—the directory of ver-

sioned files, along with metadata that enables the system to track them and communicate with the server—is called a working copy. Although other version control systems use the term “repository” for the client-side object, it is both incorrect and a common source of confusion to use the term in that way in the context of Subversion.

Subversion 是一个“集中式”的信息共享系统。版本库是 Subversion 的核心部分，是数据的中央仓库。版本库以典型的文件和目录结构形式文件系统树来保存信息。任意数量的客户端连接到 Subversion 版本库，读取、修改这些文件。客户端通过写数据将信息分享给其他人，通过读取数据获取别人共享的信息。

事实上，Subversion 的版本库的确是一种文件服务器，但不是“一般”的文件服务器。Subversion 版本库的特别之处在于，它会记录每一次改变：每个文件的改变，甚至是目录树本身的改变，例如文件和目录的添加、删除和重新组织。

33.2 Subversion Revisions

A Subversion client commits (that is, communicates the changes made to) any number of files and directories as a single atomic transaction. By atomic transaction, we mean simply this: either all of the changes are accepted into the repository, or none of them is. Subversion tries to retain this atomicity in the face of program crashes, system crashes, network problems, and other users' actions.

Each time the repository accepts a commit, this creates a new state of the filesystem tree, called a revision. Each revision is assigned a unique natural number, one greater than the number assigned to the previous revision. The initial revision of a freshly created repository is numbered 0 and consists of nothing but an empty root directory.

The figure below illustrates a nice way to visualize the repository. Imagine an array of revision numbers, starting at 0, stretching from left to right. Each revision number has a filesystem tree hanging below it, and each tree is a “snapshot” of the way the repository looked after a commit.

上图可以更形象的描述版本库，想象有一组修订号，从 0 开始，从左到右，每一个修订号有一个目录树挂在它下面，每一个树好像是一次提交后的版本库“快照”。

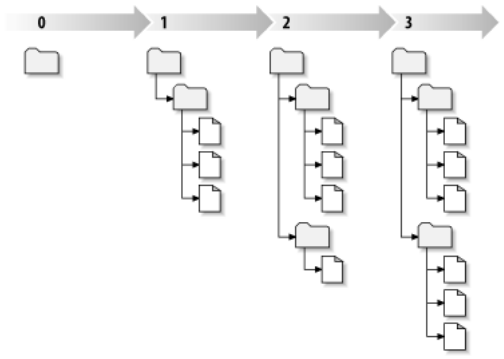


Figure 33.1: Tree changes over time

全局版本号（Global Revision Numbers）

Unlike most version control systems, Subversion's revision numbers apply to the entire repository tree, not individual files. Each revision number selects an entire tree, a particular state of the repository after some committed change. Another way to think about it is that revision N represents the state of the repository filesystem after the Nth commit. When Subversion users talk about "revision 5 of foo.c," they really mean "foo.c as it appears in revision 5." Notice that in general, revisions N and M of a file do not necessarily differ! Many other version control systems use per-file revision numbers, so this concept may seem unusual at first.

33.3 Subversion Repository URL

Subversion client programs use URLs to identify versioned files and directories in Subversion repositories. For the most part, these URLs use the standard syntax, allowing for server names and port numbers to be specified as part of the URL.

- http://svn.example.com/svn/project
- http://svn.example.com:9834/repos

Subversion repository URLs aren't limited to only the http:// variety. Because Subversion offers several different ways for its clients to communicate with its servers, the URLs used to address the repository differ subtly depending on which repository access mechanism is employed. The table below describes how different URL schemes map to the available repository access methods.

Subversion's handling of URLs has some notable nuances. For example, URLs containing the file:// access method (used for local repositories) must, in accordance with convention, have either a server name of localhost or no server name at all:

模式	访问方法
file:///	直接版本库访问 (本地磁盘)
http://	通过配置 Subversion 的 Apache 服务器的 WebDAV 协议
https://	Same as http://, but with SSL encryption
svn://	通过定制的协议访问 svnserve 服务器
svn+ssh://	Same as svn://, but through an SSH tunnel

- file:///var/svn/repos
- file://localhost/var/svn/repos

同样，在 Windows 平台下使用 file:// 模式时需要使用一个非正式的“标准”语法来访问本机上不在同一个磁盘分区中的版本库。下面的任意一个 URL 路径语法都可以工作，其中的 X 表示版本库所在的磁盘分区：

- file:///X:/var/svn/repos
- file:///X|/var/svn/repos

Note that a URL uses forward slashes even though the native (non-URL) form of a path on Windows uses backslashes. Also note that when using the file:///X|/ form at the command line, you need to quote the URL (wrap it in quotation marks) so that the vertical bar character is not interpreted as a pipe.

也必须意识到 Subversion 的 file:// URL 不能在普通的 web 服务器中工作。当你尝试在 web 服务器查看一个 file:// URL 时，它会通过直接检测文件系统读取和显示那个位置的文件内容，但是 Subversion 的资源存在于虚拟文件系统中，你的浏览器不会理解怎样读取这个文件系统。

The Subversion client will automatically encode URLs as necessary, just like a web browser does. For example, the URL http://host/path with space/project/españa —which contains both spaces and upper-ASCII characters —will be automatically interpreted by Subversion as if you'd provided http://host/path%20with%20space/project/espa%C3%B1a. If the URL contains spaces, be sure to place it within quotation marks at the command line so that your shell treats the whole thing as a single argument to the program.

There is one notable exception to Subversion's handling of URLs which also applies to its handling of local paths in many contexts, too. If the final path component of your URL or local path contains an at sign (@), you need to use a special syntax in order to make Subversion properly address that resource.

In Subversion 1.6, a new caret (^) notation was introduced as a shorthand for “the URL of the repository’s root directory”. For example, you can use the ^/tags/bigsandwich/ to refer to the URL of the /tags/bigsandwich directory in the root of the repository. Note that this URL syntax works only when your current working directory is a working copy—the command-line client knows the repository’s root URL by looking at the working copy’s metadata. Also note that when you wish to refer precisely to the root directory of the repository, you must do so using ^/ (with the trailing slash character), not merely ^.

33.4 Subversion Working Copies

A Subversion working copy[1] is an ordinary directory tree on your local system, containing a collection of files. You can edit these files however you wish, and if they’re source code files, you can compile your program from them in the usual way. Your working copy is your own private work area: Subversion will never incorporate other people’s changes, nor make your own changes available to others, until you explicitly tell it to do so. You can even have multiple working copies of the same project.

After you’ve made some changes to the files in your working copy and verified that they work properly, Subversion provides you with commands to “publish” your changes (by writing to the repository), thereby making them available to the other people working with you on your project. If other people publish their own changes, Subversion provides you with commands to merge those changes into your own working copy (by reading from the repository). Notice that the central repository is the broker for everybody’s changes in Subversion—changes aren’t passed directly from working copy to working copy in the typical workflow.

A working copy also contains some extra files, created and maintained by Subversion, to help it carry out these commands. In particular, each working copy contains a subdirectory named .svn, also known as the working copy’s administrative directory. The files in the administrative directory help Subversion recognize which of your versioned files contain unpublished changes, and which files are out of date with respect to others’ work.

一个 Subversion 工作副本是你本地机器上的一个普通目录，保存着一些文件，你可以任意的编辑文件，而且如果是源代码文件，你可以像平常一样编译，你的工作副本是你的私有工作区，在你明确的做了特定操作之前，Subversion 不会把你的修改与其他人

[1] The term “working copy” can be generally applied to any one file version’s local instance. When most folks use the term, though, they are referring to a whole directory tree containing files and subdirectories managed by the version control system. No repository means no working copy.

的合并，也不会把你的修改展示给别人，你甚至可以拥有同一个项目的多个工作副本。

当你在工作副本作了一些修改并且确认它们工作正常之后，Subversion 提供了一个命令可以“发布”你的修改给项目中的其他人 (通过写到版本库)，如果别人发布了各自的修改，Subversion 提供了手段可以把这些修改与你的工作目录进行合并 (通过读取版本库)。

工作副本也包括一些由 Subversion 创建并维护的额外文件，用来协助执行命令。通常情况下，你的工作副本的每个文件夹都有一个以.svn 为名的文件夹，也被叫做工作副本的管理目录，这个目录里的文件能够帮助 Subversion 识别哪些文件做过修改，哪些文件相对于别人的工作已经过期。

> While .svn is the de facto name of the Subversion administrative directory, Windows users may run into problems with the ASP.NET Web application framework disallowing access to directories whose names begin with a dot (.). As a special consideration to users in such situations, Subversion will instead use _svn as the administrative directory name if it finds a variable named SVN_ASP_DOT_NET_HACK in its operating environment. Throughout this book, any reference you find to .svn applies also to _svn when this “ASP.NET hack” is in use.

33.4.1 Working copy operation

For each file in a working directory, Subversion records (among other things) two essential pieces of information:

- 作为工作文件基准的版本 (叫做文件的工作版本)
- 本地副本最近一次被版本库更新的时间戳。

给定这些信息，通过与版本库通讯，Subversion 可以告诉我们工作文件是处于如下四种状态的那一种：

1. 未修改且是当前的
   文件在工作目录里没有修改，在工作版本之后没有修改提交到版本库。svn commit 操作不做任何事情，svn update 不做任何事情。
2. 本地已修改且是当前的
   在工作目录已经修改，从基本修订版本之后没有修改提交到版本库。本地修改没有提交，因此 svn commit 会成功提交，svn update 不做任何事情。
3. 本地未修改，已过时
   这个文件在工作目录没有修改，但在版本库中已经修改了。这个文件最终将更新

到最新版本，成为当时的公共修订版本。svn commit 不做任何事情，svn update 将会取得最新的版本到工作副本。

4. 本地已修改，已过时
   这个文件在工作目录和版本库都得到修改。一个 svn commit 将会失败，这个文件必须首先更新，svn update 命令会合并公共和本地修改，如果 Subversion 不可以自动完成，将会让用户解决冲突。

33.4.2 Fundamental working copy interactions

一个典型的 Subversion 版本库经常包含许多项目的文件 (或者说源代码)，通常每一个项目都是版本库的子目录，在这种布局下，一个用户的工作副本往往对应版本库的的一个子目录。

举一个例子，你的版本库包含两个软件项目，paint 和 calc。每个项目在它们各自的顶级子目录下，见下图：

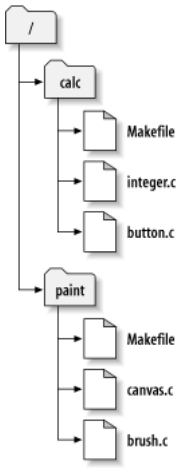


Figure 33.2: The repository's filesystem

To get a working copy, you must check out some subtree of the repository. (The term check out may sound like it has something to do with locking or reserving resources, but it doesn't; it simply creates a working copy of the project for you.) For example, if you check out /calc, you will get a working copy like this:

```
$ svn checkout http://svn.example.com/repos/calc
A    calc/Makefile
A    calc/integer.c
```

```
A    calc/button.c
Checked out revision 56.
$ ls -A calc
Makefile button.c integer.c .svn/
$
```

列表中的 A 表示 Subversion 增加了一些条目到工作副本，你现在有了一个 /calc 的个人拷贝，有一个附加的目录—.svn—保存着前面提及的 Subversion 需要的额外信息。

假定你修改了 button.c，因为.svn 目录记录着文件的修改日期和原始内容，Subversion 可以告诉你已经修改了文件，然而，在你明确告诉它之前，Subversion 不会将你的改变公开，将改变公开的操作被叫做提交 (committing，或者是检入) 修改到版本库。

将你的修改发布给别人，你可以使用 Subversion 的 commit 命令：

```
$ svn commit button.c -m "Fixed a typo in button.c."
Sending      button.c
Transmitting file data .
Committed revision 57.
$
```

这时你对 button.c 的修改已经提交到了版本库，其中包含了关于此次提交的日志信息 (例如是修改了拼写错误)。如果其他人取出了 /calc 的一个工作副本，他们会看到这个文件最新的版本。

假设你有个合作者 Sally，她和你同时取出了 /calc 的一个工作拷贝，你提交了对 button.c 的修改，Sally 的工作副本并没有改变，Subversion 只在用户要求的时候才改变工作副本。

要使项目最新，Sally 可以通过使用 svn update 命令，要求 Subversion 更新她的工作副本。这将结合你和所有其他人在她上次更新之后的改变到她的工作副本。

```
$ pwd
/home/sally/calc
$ ls -A
Makefile button.c integer.c .svn/
$ svn update
U   button.c
Updated to revision 57.
$
```

svn update 命令的输出表明 Subversion 更新了 button.c 的内容，注意，Sally 不必指定要更新的文件，subversion 利用.svn 以及版本库的进一步信息决定哪些文件需要更新。

33.4.3 Mixed-revision working copies

As a general principle, Subversion tries to be as flexible as possible. One special kind of flexibility is the ability to have a working copy containing files and directories with a mix of different working revision numbers. Subversion working copies do not always correspond to any single revision in the repository; they may contain files from several different revisions. For example, suppose you check out a working copy from a repository whose most recent revision is 4:

```
calc/
       Makefile:4
       integer.c:4
       button.c:4
```

此刻，工作目录与版本库的修订版本 4 完全对应，然而，你修改了 button.c 并且提交之后，假设没有别的提交出现，你的提交会在版本库建立修订版本 5，你的工作副本会是这个样子的：

```
calc/
       Makefile:4
       integer.c:4
       button.c:5
```

假设此刻，Sally 提交了对 integer.c 的修改，建立修订版本 6，如果你使用 svn update 来更新你的工作副本，你会看到：

```
calc/
       Makefile:6
       integer.c:6
       button.c:6
```

Sally 对 integer.c 的改变会出现在你的工作副本，你对 button.c 的改变还在，在这个例子里，Makefile 在 4, 5, 6 的修订版本都是一样的，但是 Subversion 会把他的 Makefile 的修订号设为 6 来表明它是最新的，所以你在工作副本顶级目录作一次干净的更新，会使得所有内容对应版本库的同一修订版本。

Updates and commits are separate

One of the fundamental rules of Subversion is that a “push” action does not cause a “pull” nor vice versa. Just because you’re ready to submit new changes to the repository doesn’t mean you’re ready to receive changes from other people. And if you have new changes still in progress, svn update should gracefully merge repository changes into your own, rather than forcing you to publish them.

这个规则的主要副作用就是，工作副本需要记录额外的信息来追踪混合修订版本，并且也需要能容忍这种混合，当目录本身也是版本化的时候情况更加复杂。

举个例子，假定你有一个工作副本，修订版本号是 10。你修改了 foo.html，然后执行 svn commit，在版本库里创建了修订版本 15。当成功提交之后，许多用户希望工作副本完全变成修订版本 15，但是事实并非如此。修订版本从 10 到 15 会发生任何修改，可是客户端在运行 svn update 之前不知道版本库发生了怎样的改变，svn commit 不会拖出任何新的修改。另一方面，如果 svn commit 会自动下载最新的修改，可以使得整个工作副本成为修订版本 15—但是，那样我们会打破“推”和“拉”完全分开的原则。因此，Subversion 客户端最安全的方式是标记一个文件—foo.html —为修订版本 15，工作副本余下的部分还是修订版本 10。只有运行 svn update 才会下载最新的修改，整个工作副本被标记为修订版本 15。

Mixed revisions are normal

事实上，每次运行 svn commit，你的工作拷贝都会进入混合多个修订版本的状态，刚刚提交的文件会比其他文件有更高的修订版本号。经过多次提交 (其间没有更新)，你的工作副本会完全是混合的修订版本。即使只有你一个人使用版本库，你依然会见到这个现象。为了检查混合工作修订版本，可以使用 svn status 命令的选项 –verbose。

通常，新用户对于工作副本的混合修订版本一无所知，这会让人糊涂，因为许多客户端命令对于所检验条目的修订版本很敏感。例如 svn log 命令显示一个文件或目录的历史修改信息，当用户对一个工作副本对象调用这个命令，他们希望看到这个对象的整个历史信息。但是如果这个对象的修订版本已经相当老了 (通常因为很长时间没有运行 svn update)，此时会显示比这个对象更老的历史。

Mixed revisions are useful

如果你的项目十分复杂，有时候你会发现强制工作副本的一部分“回溯”到过去非常有用 (或者更新到过去的某个修订版本)。或许你很希望测试某一子目录下某一子模块的早期版本，又或是要测试一个 bug 什么时候发生，这是版本控制系统像“时间机器”的一个方面—这个特性允许工作副本的任何一个部分在历史中前进或后退。

Mixed revisions have limitations

无论你如何在工作副本中利用混合修订版本，这种灵活性还是有限制的。

首先，你不可以提交一个不是完全最新的文件或目录，如果有个新的版本存在于版本库，你的删除操作会被拒绝，这防止你不小心破坏你没有见到的东西。

第二，如果目录已经不是最新的了，你不能提交一个目录的元数据更改。一个目录的工作修订版本定义了许多条目和属性，因而对一个过期的版本提交属性会破坏一些你没有见到的属性。

Chapter 34

Subversion Basic Usage

34.1 Subversion Help

Conveniently, though, the Subversion command-line is self-documenting, alleviating the need to grab a book off the shelf (wooden, virtual, or otherwise). The svn help command is your gateway to that built-in documentation:

```
$ svn help
Subversion command-line client, version 1.7.13.
Type 'svn help <subcommand>' for help on a specific subcommand.
Type 'svn --version' to see the program version and RA modules
  or 'svn --version --quiet' to see just the version number.

Most subcommands take file and/or directory arguments, recursing
on the directories. If no arguments are supplied to such a
command, it recurses on the current directory (inclusive) by default.

Available subcommands:
   add
   blame (praise, annotate, ann)
   cat
   changelist (cl)
   checkout (co)
   cleanup
```

```
  commit (ci)
  copy (cp)
  delete (del, remove, rm)
  diff (di)
  export
  help (?, h)
  import
  info
  list (ls)
  ...
```

As described in the previous output, you can ask for help on a particular subcommand by running svn help SUBCOMMAND. Subversion will respond with the full usage message for that subcommand, including its syntax, options, and behavior:

```
$ svn help help
help (?, h): Describe the usage of this program or its subcommands.
usage: help [SUBCOMMAND...]

Global options:
  --username ARG        : specify a username ARG
  --password ARG        : specify a password ARG
...
```

选项 (Options), 开关 (Switches) 和标志 (Flags)

The Subversion command-line client has numerous command modifiers. Some folks refer to such things as “switches” or “flags”—in this book, we’ll call them “options”. You’ll find the options supported by a given svn subcommand, plus a set of options which are globally supported by all subcommands, listed near the bottom of the built-in usage message for that subcommand.

Subversion’s options have two distinct forms: short options are a single hyphen followed by a single letter, and long options consist of two hyphens followed by several letters and hyphens (e.g., -s and –this-is-a-long-option, respectively). Every option has at least one long format. Some, such as the –changelist option, feature an abbreviated long-format alias (–cl, in this case). Only certain options—generally the most-used ones—have an additional short format. To maintain clarity in this book, we usually use the long form in code examples, but when describing options, if there’s a short form, we’ll provide the long form (to improve clarity) and the short form (to make it easier to remember). Use the form you’re more comfortable with when executing your own Subversion commands.

Many Unix-based distributions of Subversion include manual pages of the sort that can be invoked using the man program, but those tend to carry only pointers to other sources of real help, such as the project’s website and to the website which hosts this book. Also, several companies offer Subversion help and support, too, usually via a mixture of web-based discussion forums and fee-based consulting. And of course, the Internet holds a decade’s worth of Subversion-related discussions just begging to be located by your favorite search engine. Subversion help is never too far away.

34.2 Getting Data into Repository

有两种方法可以将新文件引入 Subversion 版本库：svn import 和 svn add。

34.2.1 Importing Files and Directories

svn import 是将未版本化文件导入版本库的最快方法，会根据需要创建中介目录。svn import 不需要一个工作副本，你的文件会直接提交到版本库，这通常用在你希望将一组文件加入到 Subversion 版本库时，例如：

```
$ svn import /path/to/mytree \
```

```
            http://svn.example.com/svn/repo/some/project \
            -m "Initial import"
Adding       mytree/foo.c
Adding       mytree/bar.c
Adding       mytree/subdir
Adding       mytree/subdir/quux.h

Committed revision 1.
$
```

The previous example copied the contents of the local directory mytree into the directory some/project in the repository. Note that you didn't have to create that new directory first—svn import does that for you. Immediately after the commit, you can see your data in the repository:

```
$ svn list http://svn.example.com/svn/repo/some/project
bar.c
foo.c
subdir/
$
```

Note that after the import is finished, the original local directory is not converted into a working copy. To begin working on that data in a versioned fashion, you still need to create a fresh working copy of that tree.

34.2.2 Recommended Repository Layout

Subversion provides the ultimate flexibility in terms of how you arrange your data. Because it simply versions directories and files, and because it ascribes no particular meaning to any of those objects, you may arrange the data in your repository in any way that you choose. Unfortunately, this flexibility also means that it's easy to find yourself "lost without a roadmap" as you attempt to navigate different Subversion repositories which may carry completely different and unpredictable arrangements of the data within them.

To counteract this confusion, we recommend that you follow a repository layout convention (established long ago, in the nascency of the Subversion project itself) in which a handful of strategically named Subversion repository directories convey valuable meaning about the data they hold. Most projects have a recognizable "main line", or trunk, of development; some branches, which

are divergent copies of development lines; and some tags, which are named, stable snapshots of a particular line of development. So we first recommend that each project have a recognizable project root in the repository, a directory under which all of the versioned information for that project—and only that project—lives. Secondly, we suggest that each project root contain a trunk subdirectory for the main development line, a branches subdirectory in which specific branches (or collections of branches) will be created, and a tags subdirectory in which specific tags (or collections of tags) will be created. Of course, if a repository houses only a single project, the root of the repository can serve as the project root, too.

以下是一些例子:

```
$ svn list file:///var/svn/single-project-repo
trunk/
branches/
tags/
$ svn list file:///var/svn/multi-project-repo
project-A/
project-B/
$ svn list file:///var/svn/multi-project-repo/project-A
trunk/
branches/
tags/
$
```

34.2.3 Subversion Instance Name

Subversion 努力不限制版本控制的数据类型。文件的内容和属性值都是按照二进制数据存储和传递，并且 Subversion 对于特定文件“文本化的”操作是没有意义的，也有一些地方，Subversion 对存放的信息有限制。

Subversion 内部使用二进制处理数据—例如，属性名称，路径名和日志信息—UTF-8 编码的 Unicode，这并不意味着与 Subversion 的交互必须完全使用 UTF-8。作为一个惯例，Subversion 的客户端能够透明的转化 UTF-8 和你所使用系统的编码，前提是可以进行有意义的转换 (当然是大多数目前常见的编码)。

此外，路径名称在 WebDAV 交换中会作为 XML 属性值，就像 Subversion 的管理文件。这意味着路径名称只能包含合法的 XML(1.0) 字符，Subversion 也会禁止路径名称中出现 TAB, CR 或 LF 字符，所以它们才不会在区别程序或如 svn log 和 svn status (stat, st) 的

输出命令中断掉。

虽然看起来要记住很多事情，但在实践中这些限制很少会成为问题。只要你的本地设置兼容 UTF-8，也不在路径名称中使用控制字符，与 Subversion 的通讯就不会有问题。命令行客户端会添加一些额外的帮助字节—自动将你输入的 URL 路径字符转化为“合法正确的”内部用版本。

Of course, when it comes to choosing valid path names, Subversion isn't the only limiting factor. Teams using multiple operating systems need to consider the limitations placed on path names by those operating systems, too. For example, while Windows disallows the use of colon characters in file names, a user on a Linux system can very easily add such a file to version control, resulting in a dataset that can no longer be checked out on Windows. Adding multiple files to a directory whose names differ only in their letter casing will likewise cause problems for users checking out working copies onto case-insensitive filesystems. So, some broad awareness of the various limitations introduced by different operating systems and filesystems, then, is recommended.

34.3 Creating a Working Copy

Most of the time, you will start using a Subversion repository by performing a checkout of your project. Checking out a directory from a repository creates a working copy of that directory on your local machine. Unless otherwise specified, this copy contains the youngest (that is, most recently created or modified) versions of the directory and its children found in the Subversion repository:

```
$ svn checkout http://svn.example.com/svn/repo/trunk
A    trunk/README
A    trunk/INSTALL
A    trunk/src/main.c
A    trunk/src/header.h
...
Checked out revision 8810.
$
```

Although the preceding example checks out the trunk directory, you can just as easily check out a deeper subdirectory of a repository by specifying that subdirectory's URL as the checkout URL:

```
$ svn checkout http://svn.example.com/svn/repo/trunk/src
A    src/main.c
A    src/header.h
```

```
A    src/lib/helpers.c
...
Checked out revision 8810.
$
```

Since Subversion uses a copy-modify-merge model instead of lock-modify-unlock, you can immediately make changes to the files and directories in your working copy. Your working copy is just like any other collection of files and directories on your system. You can edit the files inside it, rename it, even delete the entire working copy and forget about it.

因为你的工作副本“同你系统上的文件和目录没有任何区别”，你可以随意修改文件，但是你必须告诉 Subversion 你做的其他任何事。例如，你希望拷贝或移动工作副本的一个文件，你应该使用 svn copy 或者 svn move，而不要使用操作系统的拷贝移动命令。

除非你准备好了提交一个新文件或目录，或改变了已存在的，否则没有必要通知 Subversion 你做了什么。

Every directory in a working copy contains an administrative area—a subdirectory named .svn. Usually, directory listing commands won't show this subdirectory, but it is nevertheless an important directory. Whatever you do, don't delete or change anything in the administrative area! Subversion uses that directory and its contents to manage your working copy.

如果你不小心删除了子目录.svn，最简单的解决办法是删除包含的目录 (普通的文件系统删除，而不是 svn delete)，然后在父目录运行 svn update，Subversion 客户端会重新下载你删除的目录，并包含新的.svn。

Notice that in the previous pair of examples, Subversion chose to create a working copy in a directory named for the final component of the checkout URL. This occurs only as a convenience to the user when the checkout URL is the only bit of information provided to the svn checkout command. Subversion's command-line client gives you additional flexibility, though, allowing you to optionally specify the local directory name that Subversion should use for the working copy it creates. For example:

```
$ svn checkout http://svn.example.com/svn/repo/trunk my-working-copy
A    my-working-copy/README
A    my-working-copy/INSTALL
A    my-working-copy/src/main.c
A    my-working-copy/src/header.h
...
Checked out revision 8810.
```

```
$
```

If the local directory you specify doesn't yet exist, that's okay—svn checkout will create it for you.

34.4 Basic Work Cycle

Subversion 有许多特性, 选项和华而不实的高级功能，但日常的工作中你只使用其中的一小部分，典型的工作周期是这样的:

1. Update your working copy. This involves the use of the svn update command.
2. Make your changes. The most common changes that you'll make are edits to the contents of your existing files. But sometimes you need to add, remove, copy and move files and directories —the svn add, svn delete, svn copy, and svn move commands handle those sorts of structural changes within the working copy.
3. Review your changes. The svn status and svn diff commands are critical to reviewing the changes you've made in your working copy.
4. Fix your mistakes. Nobody's perfect, so as you review your changes, you may spot something that's not quite right. Sometimes the easiest way to fix a mistake is start all over again from scratch. The svn revert command restores a file or directory to its unmodified state.
5. Resolve any conflicts (merge others' changes). In the time it takes you to make and review your changes, others might have made and published changes, too. You'll want to integrate their changes into your working copy to avoid the potential out-of-dateness scenarios when you attempt to publish your own. Again, the svn update command is the way to do this. If this results in local conflicts, you'll need to resolve those using the svn resolve command.
6. Publish (commit) your changes. The svn commit command transmits your changes to the repository where, if they are accepted, they create the newest versions of all the things you modified. Now others can see your work, too!

34.4.1 Update Working Copy

When working on a project that is being modified via multiple working copies, you'll want to update your working copy to receive any changes committed from other working copies since your last update. These might be changes that other members of your project team have made, or they might simply be changes you've made yourself from a different computer. To protect your data, Subversion won't allow you commit new changes to out-of-date files and directories, so it's best to

have the latest versions of all your project's files and directories before making new changes of your own.

Use svn update to bring your working copy into sync with the latest revision in the repository:

```
$ svn update
U   foo.c
U   bar.c
Updated to revision 2.
$
```

这种情况下, 其他人在你上次更新之后提交了对 foo.c 和 bar.c 的修改, 因此 Subversion 更新你的工作副本来引入这些更改。

When the server sends changes to your working copy via svn update, a letter code is displayed next to each item to let you know what actions Subversion performed to bring your working copy up to date. To find out what these letters mean, run svn help update or see svn update (up).

34.4.2 Make Changes

Now you can get to work and make changes in your working copy. You can make two kinds of changes to your working copy: file changes and tree changes. You don't need to tell Subversion that you intend to change a file; just make your changes using your text editor, word processor, graphics program, or whatever tool you would normally use. Subversion automatically detects which files have been changed, and in addition, it handles binary files just as easily as it handles text files—and just as efficiently, too. Tree changes are different, and involve changes to a directory's structure. Such changes include adding and removing files, renaming files or directories, and copying files or directories to new locations. For tree changes, you use Subversion operations to "schedule" files and directories for removal, addition, copying, or moving. These changes may take place immediately in your working copy, but no additions or removals will happen in the repository until you commit them.

On non-Windows platforms, Subversion is able to version files of the special type symbolic link (or "symlink"). A symlink is a file that acts as a sort of transparent reference to some other object in the filesystem, allowing programs to read and write to those objects indirectly by performing operations on the symlink itself.

When a symlink is committed into a Subversion repository, Subversion remembers that the file was in fact a symlink, as well as the object to which the symlink "points." When that symlink is checked out to another working copy on a non-Windows system, Subversion reconstructs a real

filesystem-level symbolic link from the versioned symlink. But that doesn't in any way limit the usability of working copies on systems such as Windows that do not support symlinks. On such systems, Subversion simply creates a regular text file whose contents are the path to which the original symlink pointed. While that file can't be used as a symlink on a Windows system, it also won't prevent Windows users from performing their other Subversion-related activities.

下面是 Subversion 用来修改目录树结构的五个最常用的子命令。

- svn add FOO

  Use this to schedule the file, directory, or symbolic link FOO to be added to the repository. When you next commit, FOO will become a child of its parent directory. Note that if FOO is a directory, everything underneath FOO will be scheduled for addition. If you want only to add FOO itself, pass the –depth=empty option.
- svn delete FOO

  Use this to schedule the file, directory, or symbolic link FOO to be deleted from the repository. If FOO is a file or link, it is immediately deleted from your working copy. If FOO is a directory, it is not deleted, but Subversion schedules it for deletion. When you commit your changes, FOO will be entirely removed from your working copy and the repository.[6]
- svn copy FOO BAR

  Create a new item BAR as a duplicate of FOO and automatically schedule BAR for addition. When BAR is added to the repository on the next commit, its copy history is recorded (as having originally come from FOO). svn copy does not create intermediate directories unless you pass the –parents option.
- svn move FOO BAR

  This command is exactly the same as running svn copy FOO BAR; svn delete FOO. That is, BAR is scheduled for addition as a copy of FOO, and FOO is scheduled for removal. svn move does not create intermediate directories unless you pass the –parents option.
- svn mkdir FOO

  This command is exactly the same as running mkdir FOO; svn add FOO. That is, a new directory named FOO is created and scheduled for addition.

Subversion does offer ways to immediately commit tree changes to the repository without an explicit commit action. In particular, specific uses of svn mkdir, svn copy, svn move, and svn delete can operate directly on repository URLs as well as on working copy paths. Of course, as previously mentioned, svn import always makes direct changes to the repository.

There are pros and cons to performing URL-based operations. One obvious advantage to doing

so is speed: sometimes, checking out a working copy that you don't already have solely to perform some seemingly simple action is an overbearing cost. A disadvantage is that you are generally limited to a single, or single type of, operation at a time when operating directly on URLs. Finally, the primary advantage of a working copy is in its utility as a sort of "staging area" for changes. You can make sure that the changes you are about to commit make sense in the larger scope of your project before committing them. And, of course, these staged changes can be as complex or as a simple as they need to be, yet result in but a single new revision when committed.

34.4.3 Review Changes

Once you've finished making changes, you need to commit them to the repository, but before you do so, it's usually a good idea to take a look at exactly what you've changed. By examining your changes before you commit, you can compose a more accurate log message (a human-readable description of the committed changes stored alongside those changes in the repository). You may also discover that you've inadvertently changed a file, and that you need to undo that change before committing. Additionally, this is a good opportunity to review and scrutinize changes before publishing them. You can see an overview of the changes you've made by using the svn status command, and you can dig into the details of those changes by using the svn diff command.

You can use the commands svn status, svn diff, and svn revert without any network access even if your repository is across the network. This makes it easy to manage and review your changes-in-progress when you are working offline or are otherwise unable to contact your repository over the network.

Subversion does this by keeping private caches of pristine, unmodified versions of each versioned file inside its working copy administrative areas. This allows Subversion to report—and revert—local modifications to those files without network access. This cache (called the text-base) also allows Subversion to send the user's local modifications during a commit to the server as a compressed delta (or "difference") against the pristine version. Having this cache is a tremendous benefit—even if you have a fast Internet connection, it's generally much faster to send only a file's changes rather than the whole file to the server.

Check overview of changes

To get an overview of your changes, use the svn status command. You'll probably use svn status more than any other Subversion command.

Because the cvs status command's output was so noisy, and because cvs update not only performs

an update, but also reports the status of your local changes, most CVS users have grown accustomed to using cvs update to report their changes. In Subversion, the update and status reporting facilities are completely separate.

If you run svn status at the top of your working copy with no additional arguments, it will detect and report all file and tree changes you've made.

```
$ svn status
?      scratch.c
A      stuff/loot
A      stuff/loot/new.c
D      stuff/old.c
M      bar.c
$
```

In its default output mode, svn status prints seven columns of characters, followed by several whitespace characters, followed by a file or directory name. The first column tells the status of a file or directory and/or its contents. Some of the most common codes that svn status displays are:

- ? item
  The file, directory, or symbolic link item is not under version control.
- A item
  预定加入到版本库的文件, 目录或符号链的 item。
- C item
  文件 item 发生了冲突。从服务器收到的修改与工作副本的本地修改发生交迭 (在更新期间不会被解决)。在你提交到版本库前，必须手工解决冲突。
- D item
  文件, 目录或是符号链 item 预定从版本库中删除。
- M item
  文件 item 的内容被修改了。

如果你传递一个路径给 svn status，它只给你这个项目的信息：

Chapter 35

Advanced Topics

35.1 Revision Specifiers

35.1.1 Revision Keywords

35.1.2 Revision Dates

35.2 Peg and Operative Revisions

35.3 Properties

35.3.1 Why Properties?

35.3.2 Manipulating Properties

35.3.3 Properties and the Subversion Workflow

35.3.4 Inherited Properties

35.3.5 Automatic Property Setting

35.3.6 Subversion Reserved Properties

Versioned properties

Unversioned properties

35.4 File Portability

35.4.1 File Content Type

35.4.2 File Executability

35.4.3 End-of-Line Character Sequences

35.5 Ignoring Unversioned Items

35.6 Keyword Substitution

35.7 Sparse Directories

Chapter 36

Branching and Merging

Part IV

Version Control with Git

Git 是一个分布式版本控制/软件配置管理软件，原來是 linux 内核開發者林纳斯·托瓦兹（Linus Torvalds）为了更好地管理 linux 内核开发而创立的。需要注意的是和 GNU Interactive Tools，一个类似 Norton Commander 界面的文件管理器有所不同。