九州大学学術情報リポジトリ
Kyushu University Institutional Repository

CVSでファイル管理

池田, 大輔
九州大学情報基盤センター研究部

https://doi.org/10.15017/1470472

出版情報：九州大学情報基盤センター広報 : 全国共同利用版. 3 (1), pp.14-28, 2003-03. 九州大学情報基盤センター
バージョン：
権利関係：

CVSでファイル管理

池田 大輔*

1 はじめに

CVSとはConcurrent Versions Systemの略で、バージョン管理システムの一種です。CVSを使えば、ソースコードや論文などの管理対象となる複数のファイルのバージョンを管理し、任意の時点のファイルを取りだすことができます。さらに、枝と呼ばれるものを作ることで異なる開発や執筆の流れを並行して維持し、独立に作業をすることができます。例えば、ソフトウェアの場合は新機能を追加していく開発版と、機能の追加によりバグやセキュリティホールをなくすことに重点を置いた安定版の開発などが、異なる開発の流れの例です。それぞれの枝で、独立にソースファイルを変更することができます。また、地理的に離れた場所にいる複数の人との共同作業もできます。実際、多くのソフトウェアがCVSを使っていて、世界各地から開発にかかわることを可能にしています。

しかし、CVSは個人や研究室程度の規模でも手軽に使えます。基本的にはテキストファイルであればどのようなファイルであってもCVSで管理できます。したがって、LaTeXのソースファイルや設定ファイル等を管理することも可能です。一人で使う場合も、デスクトップパソコンとノートパソコンをそれぞれ頻繁に使うような場合は、地理的に離れた場所にいる人との共同作業と考えることもできます。どちらで作業をしていても、その変更点を問題なくマージできます。

本稿では、CVSを基本的に1人で利用する場合について説明します。複数のユーザが同時に使ったり、CVSサーバがネットワーク上にある場合の詳しい説明は次回行なう予定です。CVSのバージョン1.11を仮定しますが、基本的な事項のみですので他のバージョンでも問題なく使えます。また、UNIX系のOSから使うことを想定しています。WindowsからCVSを使う場合については、次回で紹介したいと思います。

> ✔ CVS 1.11.4以前のバージョンにセキュリティホールが発見されました。すでにhttp://www.cvshome.org/ではセキュリティホールを塞いだ1.11.5がリリースされていますので、アップデートしましょう。

1.1 CVSの基本概念

CVSでは「モジュール」単位で、ファイルやデータを管理します。モジュールとは、一般に1つのソフトウェアや論文を構成する複数のファイルを含みます。つまり、統合開発環境でいうプロジェクトに相当します。以下では、モジュールの代りにプロジェクトと呼ぶことにします。

*情報基盤センター研究部　mailto:daisuke@cc.kyushu-u.ac.jp

各ファイルには「リビジョン」と呼ばれる番号が自動的につきます。リビジョンは“1.2”や“1.3.2.2”などです。通常、あまりリビジョンを意識する必要はありません。リビジョン番号を確認する方法は3.2節を参照してください。

リビジョン番号は、ソフトウェアなどのバージョンとは別の物です。バージョンに対応するものは、CVSではタグ（4節参照）にあたります。タグは自動的につけられるわけではなく、利用者が明示的にプロジェクトを構成するファイルにつけます。過去のある時点のバージョンを取りだす時などに、タグは重要な働きをします。

すべてのプロジェクトを構成するファイルや、履歴やログなどの情報は「レポジトリ」と呼ばれるディレクトリ（フォルダ）に格納されます。作業の流れは、レポジトリからプロジェクトの作業用コピーを取りだし、これらのファイルに対し編集などを行ない、変更点をレポジトリに格納します。

まだレポジトリがない場合は、どこにファイルを置くかを決めて、以下のようにしてレポジトリを作成します。ここではホームディレクトリの下のCVSというディレクトリに置くことにします。

```
  % cd
  % mkdir CVS
  % cvs -d $HOME/CVS init
  % ls CVS
CVSROOT/
  % ls CVS/CVSROOT
Emptydir/        cvswrappers      modules          taginfo,v
checkoutlist     cvswrappers,v    modules,v        val-tags
checkoutlist,v   editinfo         notify           verifymsg
commitinfo       editinfo,v       notify,v         verifymsg,v
(略)
```

これで、初期化が終わりました。ディレクトリCVSの中をみるとCVSROOTというディレクトリがあり、この下に様々なファイルが用意されています。これらはCVSの設定ファイルです。

CVSを使うときには、レポジトリがどこにあるか明示しないといけません。毎回コマンドの引数として指定することも可能ですが、環境変数$CVSROOTを設定しておくとよいでしょう。レポジトリは、ローカルなディスク上でもよいですし、ネットワーク上の計算機のディレクトリを指定しても構いません[1]。ローカルなディスク上の場合は、単にディレクトリ名を書きます。上の例では

```
  % setenv CVSROOT $HOME/CVS
```

とします。

本稿では、CVSを使う時の具体的な例として、本稿のLaTeXファイル群をCVS化する様子をお見せします。プロジェクト名は“Koho/article”とします。プロジェクト名が“/”を含んでいますが、これについては2節で説明します。

CVSを利用するには*cvs*コマンドを、機能ごとに用意されたサブコマンドと一緒に用います。上述した作業用コピーの取りだしと、変更点の格納は別のサブコマンドになっています。これらのサブコマンドは、固有のオプションを持っており、CVSを使うときの一般的な書式は

[1]この説明は次回行ないます。

次のようになります。

```
% cvs [cvs-options] subcommand [sub-options] [args]
```

1.2 研究用システムでの利用

研究用システムのコンピュータ kyu-cc, kyu-ss, kyu-vpp では、サーバとしてもクライアントとしても問題なく CVS が使えます[2]。ここで、サーバとして利用できるとは、上記のコンピュータ上にレポジトリを作り、ネットワークごしにファイルを取得したり、変更点を格納できるという意味です。どのマシンにおいても CVS のバージョンは 1.11.5 です。

これらのコンピュータのいずれかを CVS のサーバとして利用する場合は、まず、上述したようにサブコマンド init でレポジトリを用意します。その後、ネットワークごしに利用する場合には$CVSROOT を、例えば ":ext:*user*@kyu-cc:/home/usr*X*/*user*/CVS" などのようにしてください。ここで、"*user*" は研究用システムにおけるユーザ名です。

2 プロジェクトの登録

まずはプロジェクトをレポジトリに登録します。すでに、このプロジェクトに登録したいファイルがある場合は、これらのファイルがあるディレクトリに移動して

```
% cvs import -m "Imported sources" project-name vender start
```

とします。ここで "*project-name*" がプロジェクト名となり、以後 CVS からプロジェクトを取りだすときの名前になります。"*vender*" はベンダータグと呼ばれるもので、通常自分の名前にしておけばよいでしょう。"*start*" はリリースタグと呼ばれるもので、これも通常そのまま "*start*" としておけばよいでしょう。-m オプションは、ログに書き込むメッセージを指定します。

指定しなくても構いませんが、その場合には$CVSEDITOR または$EDITOR 環境変数で指定してあるエディタが起動して、メッセージを書くように促されます。起動したエディタには

```
CVS: ----------------------------------------------------------------------
CVS: Enter Log.  Lines beginning with 'CVS:' are removed automatically
CVS:
CVS: Committing in .
CVS:
CVS: Modified Files:
CVS:    intro.tex work.tex
CVS: ----------------------------------------------------------------------
```

といった行が含まれていて、これより前に適当なログメッセージを書きます。書き終えたら保存して普通にエディタを終了させます[3]。メッセージを書かずに終了させると

[2]ただし、複数のユーザで使う場合には、CVS の設定ファイルにグループの書き込み権限を与える必要があります。これについては、次回で説明します。

[3]環境設定によっては、意図せずに *vi* が起動することもあるかもしれません。*vi* は "ZZ"(大文字で "z" を 2 回)

```
Log message unchanged or not specified
a)bort, c)ontinue, e)dit, !)reuse this message unchanged for remaining dirs
Action: (continue)
```

と表示され、強制終了するか、そのまま続けるか、編集しなおすか選択を求められます。そのままでよければ、単にリターンキーを押します。

以後、このプロジェクトをレポジトリから取りだした場合、プロジェクト名"*project-name*"が、取りだしたファイルをすべて含んだディレクトリの名前になります。

登録されるのは、カレントディレクトリ以下にあるすべてのファイルです。したがって、作業用に使ったがCVSに登録したくないファイルは、あらかじめ移動するなりしておいてください[4]。

プロジェクト名はレポジトリに格納するときのディレクトリ名にもなります。上の例では、レポジトリの下にproject-nameというディレクトリができます。プロジェクト名に、上述した例"Koho/article"のように"/"を含めることもできます。この場合、Kohoディレクトリの下にarticleという名前のディレクトリができます。したがって、複数のプロジェクトをまとめて1つのディレクトリ下で管理する場合に"/"で区切ったプロジェクト名を使うとよいでしょう。

サンプルプロジェクトの例では、以下のようになります。

```
  % cvs import -m "article of cvs for Koho" Koho/article daisuke start
I Koho/article/MEMO.txt~
I Koho/article/.#cvs.tex
I Koho/article/cvs.tex~
I Koho/article/intro.tex~
I Koho/article/conclusion.tex~
N Koho/article/cvs.tex
N Koho/article/MEMO.txt
N Koho/article/intro.tex
N Koho/article/conclusion.tex

No conflicts created by this import
```

ここで、各ファイル名の前に"I"や"N"が表示されます。"I"はそのファイルが無視されることを、"N"はそのファイルが新規にレポジトリに追加されることを示しています。Emacsが作るバックアップファイルなどは無視されていることが分かります。

> ✔CVSで無視されるファイルに*.objという形式のファイル名を持つものがあります。しかし、UNIXでよく使われるドローツール*tgif*の出力ファイル名も*.objという形式です。この場合、何もしないと作業用コピーに*tgif*のファイルを作っても、無視されてしまいます。デフォルトで無視されるファイル名のパターンは
>
> ```
> RCS       SCCS     CVS      CVS.adm
> RCSLOG    cvslog.*
> tags      TAGS
> .make.state       .nse_depinfo
> ```

と入力すると終了します。

[4]CVSには、あらかじめ無視するファイルのパターンが定義されていて、Emacsの自動バックアップファイルやcoreファイル等は無視されます。

```
*~      #*      .#*     ,*      _$*     *$
*.old   *.bak   *.BAK   *.orig  *.rej   .del-*
*.a     *.olb   *.o     *.obj   *.so    *.exe
*.Z     *.elc   *.ln
core
```

となっています。そこで、*.obj ファイルのあるディレクトリに.cvsignore というファイルを作り、"!" を入れるとパターンがすべて無効になります。その後で CVS に無視して欲しいパターンを記述してもよいでしょう。

常に *.obj を無視しないようにするには、レポジトリに CVSROOT/cvsignore というファイルを作り、上述の.cvsignore と同じ内容を書き込んでおきます。あるいは、ホームディレクトリの下に.cvsignore という名前で保存しても構いません。

ただし、*.dvi ファイルなどはデフォルトで無視されませんので、プロジェクトに登録したくないファイルは消しておいてください。誤って不要なファイルをプロジェクトに登録してしまった場合は 3.5 節を参照してプロジェクトから削除してください。

登録時にファイルを置いていたディレクトリは、CVS の管理下にはありません。以後の編集作業などを行なう場合は、このディレクトリではなく、レポジトリから作業用コピーを取りだして行なってください。取りだし方は 3.1 節を参照してください。

3　日常の作業例

プロジェクトを作ってしまえば、あとは作業用コピーをレポジトリから取りだし、編集などを行ないます。以前のリビジョンとの差分を見たり、他人がレポジトリに変更を加えてないかを確認したりしながら作業は進みます。適当なところで、自分が加えた変更点をレポジトリに格納します。これらを、CVS のサブコマンドごとに説明していきます。

3.1　ファイルの取得

最新の作業用コピーを取得するには

```
% cvs checkout project
```

とします。ここで *project* は取りだしたいプロジェクトの名前です。project という名前のディレクトリが作成され、この下にプロジェクトのファイルが置かれます。

✔CVS の多くのサブコマンドは、別名を持っています。例えば、この節で説明した checkout の代わりに co または get を使用して構いません。マニュアル [1] の各サブコマンドのページには、別名がある場合は明記されています。

サンプルプロジェクトの場合、以下のようにします。

```
  % cvs checkout Koho/article
cvs server: Updating Koho/article
U Koho/article/MEMO.txt
U Koho/article/conclusion.tex
U Koho/article/cvs.tex
U Koho/article/intro.tex
  % ls Koho
CVS/  article/
  % ls Koho/article
CVS/  MEMO.txt  conclusion.tex  cvs.tex  intro.tex
```

"U"は、ファイルが変更されてないことを示します。また、単にレポジトリに登録したファイルだけでなく、各サブディレクトリにCVSという名前のディレクトリがあります。ここにレポジトリの場所、プロジェクトを構成しているファイル名などの管理情報が書いてあります。

特に指定しない限り、最新のリビジョンが取りだされます。CVSでは、過去の任意の時点のバージョンも取りだすことができます。まず、日付を指定するには`-D date`コマンドを使います。これにより、その日付より前で最も新しいリビジョンからなるファイルを取りだします。CVSはかなり多くの日付の書式を理解します。例えば"1972-09-24"や"24 Sep 1972 20:05"などです。

また、後述するタグを指定して取りだすこともできます。この場合`-r tag`とします。タグ名の調べ方は3.2節を、タグそのものの意味やつけ方については4節を参照してください。

`-d dir`で、取りだしたときのディレクトリの名前を変更します。通常はプロジェクト名がディレクトリ名になってしまうので、いくつかのバージョンの作業用コピーを同時に編集する場合は名前を変更しなければなりません。

$CVSROOTに設定されているレポジトリではないレポジトリを一時的に利用したい場合は、*cvs*のオプションとして`-d repository`を指定してください。例えば、普段は研究室のレポジトリを利用しているが、今回は研究用システムのkyu-ccに置いたレポジトリを利用したい場合は

```
% cvs -d :ext:user@kyu-cc:/home/usrX/user/CVS checkout project
```

などとしてください。このようにして取りだした作業用コピーでは、その後の作業には`-d`によるレポジトリの指定は不要です。

3.2 ファイルの状態確認

CVSでは、作業用コピーの各ファイルに対しいくつかの状態があります。例えば、自分で変更をした場合は"Locally Modified"となり、他の人物[5]が変更している場合は"Needs Patch"となります。

他に、以下のような状態があります。

Up-to-date 最新リビジョンと同じ

[5]別のコンピュータから自分が変更しても同様です。

Locally Added add コマンドによりプロジェクトにファイルを加えたが、まだその内容を格納してない

Locally Removed remove コマンドによりプロジェクトからファイルを削除したが、まだその変更を格納してない

Needs Checkout 他の人物が新しいファイルをリポジトリに格納した

Needs Merge 他の人物が新しいリビジョンをリポジトリに格納したが、作業ファイルを自分で修正している

File had conflicts on merge update コマンドの結果、変更点の衝突が発見された

add と remove については 3.5 節を、衝突とその解消の仕方については 3.4 節を参照してください。

ファイルの状態を確認するには status サブコマンドを使います。

```
% cvs status [files]
```

ファイル名を指定しなかったら、すべてのファイルの状態を表示します。-v オプションを指定すれば、4 節で説明するタグの情報も出力されます。

サンプルプロジェクトの例では、以下のようになります。

```
% cvs status
? Auto
? cvs.aux
? cvs.dvi
? cvs.log
===================================================================
(中略)
===================================================================
File: cvs.tex           Status: Locally Modified

   Working revision:    1.2
   Repository revision: 1.2     /XXX/CVS/Koho/article/cvs.tex,v
   Sticky Tag:          (none)
   Sticky Date:         (none)
   Sticky Options:      (none)

===================================================================
File: intro.tex         Status: Up-to-date

   Working revision:    1.2
   Repository revision: 1.2     /XXX/CVS/Koho/article/intro.tex,v
   Sticky Tag:          (none)
   Sticky Date:         (none)
   Sticky Options:      (none)
```

最初のほうの "?" で始まる行は、CVS が関知しないファイルです。その後、各ファイルごとに状態が "Status: Up-to-date" などのように表示されます。各ファイルの "Working revision" は作

業用コピーにあるファイルのリビジョン、"Repository revision" はレポジトリにあるファイルのリビジョンです。"Sticky Tag" など "Sticky"（貼りついた）で始まる行については、次回説明します。

他人が変更したものを作業用コピーに取り込むには、update を使い作業用コピーを最新の状態へ更新します（3.4 節参照）。

3.3 差分の確認

作業用コピーを編集していきますが、どこを変更したのか確認したい場合があります。このような時は

```
% cvs diff [files]
```

とします。これで、最も新しいリビジョンと現在の作業用コピーとの差分を見ることができます。

特定のリビジョン間の差分を見たい場合は-r でリビジョン番号を指定します。-r オプションは2つまで指定できます。1つの時は現在の作業用コピーと指定したリビジョン間の差分、2つの時は指定した2つのリビジョン間の差分を表示します。

CVS の diff は、通常の *diff* コマンドと同じオプションを指定できます。例えば-i オプションで大文字小文字を同一視し、-b オプションで空白文字の違いを無視します。

3.4 最新の状態へ更新

CVS では、作業用コピーに対して編集などの作業を行ないます。しかし、変更点は 3.6 節で説明する commit でレポジトリに格納しない限り、他の人は利用できません。逆に、作業中に他の人がその人独自の変更点をレポジトリに格納することがあります。この場合、作業用コピーに新たな変更を取り込む必要があります。これを行なうのが update サブコマンドです。

```
% cvs update [files]
```

ファイルを指定しなければ、プロジェクト中のすべてのファイルが update の対象になります。

サンプルプロジェクトにおいて、ローカルのコンピュータで conclusion.tex を編集し、cvs.tex と intro.tex を他のユーザ[6]が変更したとしましょう。この場合、update すると以下のようになります。

```
% cvs update
cvs server: Updating .
M conclusion.tex
P cvs.tex
P intro.tex
```

"M" は作業用コピーが変更されていることを、"P" はレポジトリの変更点がパッチとして送られたことを示しています。特に問題なく update は完了しました。

[6]実際には、別のコンピュータの筆者です。

他のユーザがファイルを変更して、その変更がレポジトリに格納された時、自分の変更との競合の有無によりupdateの出力やその後の対応の仕方が変わってきます。さきほどの例では、作業用コピーのファイルのうち、未編集のファイル、つまりcheckoutした時と同じファイルを他人が変更した場合でしたが、この時は問題ありません。自分と他人が同じファイルを変更した場合でも、変更した所が離れていれば問題ありません。

問題は、同じファイルのほぼ同じ箇所を変更した場合[7]で、こういうファイルをupdateすると**衝突** *(conflict)* が起こります。このとき、updateの出力の最初に"C"が表示されます。衝突が起きたファイルは、2つのリビジョンをマージしようとした結果に置き換えられ、元のファイルは".#file.revision"という名前で保存されます。ここでrevisionは、ファイルの修正を開始した時点でのリビジョンです。

サンプルプロジェクトで意図的に同じ箇所を編集して衝突を起こした例を示します。作業用コピーのconclusion.texは

```
\section{\label{sec:conclusion}おわりに}

ここを変更してみました。
```

となっていて、レポジトリにおける同ファイルは

```
\section{\label{sec:conclusion}おわりに}

衝突を起こすための例です。
```

となっています。この変更は、作業用コピーをレポジトリから取り出した後に格納されたものです。

この作業用ファイルに、レポジトリの変更点を取り込むと以下のようになります。

```
  % cvs update conclusion.tex
RCS file:  /XXX/CVS/Koho/article/conclusion.tex,v
retrieving revision 1.1.1.1
retrieving revision 1.2
Merging differences between 1.1.1.1 and 1.2 into conclusion.tex
rcsmerge:  warning:  conflicts during merge
cvs server:  conflicts found in conclusion.tex
C conclusion.tex
```

レポジトリの変更点と作業用コピーの変更点をマージしようとして、衝突が見つかったことを示しています。.#conclusion.tex.1.1.1.1というファイルに作業用コピーが保存され、新たなconclusion.texの該当箇所は以下のようになっています。

[7]CVSでは変更点、つまり差分をレポジトリに格納します。このときの動作はUNIXの*diff*を基本にしています。*diff*からみて「同じ」箇所が同時に変更している場合ということです。

```
\section{\label{sec:conclusion}おわりに}

<<<<<<< conclusion.tex
ここを変更してみました。

=======
衝突を起こすための例です。

>>>>>>> 1.2
```

"<<<<<<<"から"======="までが作業用コピーにあった変更点で、そこから">>>>>>>"までがレポジトリの変更点です。

この時の状態(Status)は

```
  % cvs status conclusion.tex
===================================================================
File: conclusion.tex    Status: File had conflicts on merge

   Working revision:    1.2
   Repository revision: 1.2     /XXX/CVS/Koho/article/conclusion.tex,v
   Sticky Tag:          (none)
   Sticky Date:         (none)
   Sticky Options:      (none)
```

となっていて、衝突していることが分かります。これらは手作業でマージして、衝突を解消する必要があります。適当に編集してから、3.6節を参考に変更をレポジトリに格納してください。その際、"<<<<<<<", "=======", ">>>>>>>"が残っていても警告を出すだけでレポジトリに格納されます。

3.5 ファイルの追加と削除

プロジェクトにファイルを追加するにはaddを、削除するにはremoveを使います。

```
  % cvs add files
  % cvs remove files
```

どちらもcommitして初めてレポジトリに反映されます。addで追加したいファイルは、すでに存在している必要があります。removeで削除したいファイルは、*rm*などであらかじめ削除しておく必要があります。

サンプルプロジェクトへファイルを追加すると以下のようになります。

```
  % cvs add project.tex tag.tex work.tex
cvs server: scheduling file 'project.tex' for addition
cvs server: scheduling file 'tag.tex' for addition
cvs server: scheduling file 'work.tex' for addition
cvs server: use 'cvs commit' to add these files permanently
```

出力メッセージにより、commitするまでレポジトリに格納されないことが分かります。

新しいサブディレクトリ以下のファイルを追加する場合は、まずディレクトリを add してから subdir/file を追加します。

3.6 変更の格納

作業用コピーを変更したら、適当な間隔で変更点をレポジトリに格納しましょう。格納するのは commit コマンドです。

```
% cvs commit [files]
```

とします。ファイルを指定しなければ、作業用コピー全体が対象となります。

サンプルプロジェクトの例は以下のようになります。

```
 % cvs commit -m "test"
cvs commit: Examining .
（中略）
Checking in MEMO.txt;
/XXX/CVS/Koho/article/MEMO.txt,v  <--  MEMO.txt
new revision: 1.2; previous revision: 1.1
done
Checking in cvs.tex;
/XXX/CVS/Koho/article/cvs.tex,v  <--  cvs.tex
new revision: 1.3; previous revision: 1.2
done
```

ここでは、あらかじめオプションでログのメッセージを指定します。いくつか、CVS の関知しないファイルが表示されたあと、変更のあったファイルのリビジョンが上がったことが表示されます。

> ✔ commit するタイミングはいつがよいかは難しい問題かもしれません。マニュアル [1] の 11.1 節「いつ格納すべきか? 」では、この問題について述べられています。ソフトウェアの開発の場合、最低限コンパイルは通る状態で commit したほうがよいでしょう。ただ、あまり長いこと変更点を作業用コピーにためておいたら、バージョン管理している意味が半減します。個人で計算機実験用に開発しているソフトウェアの場合、バグを含んでいてもこまめに commit するようにしています。その変わり、ChangeLog ファイルなどにバグを含んでいることを明記するようにしています。

4 タグによるバージョン管理

CVS に限らず、バージョン管理システムなるものは、過去の任意の時点でのプロジェクトのソースファイルなどを取りだすことができます。この過去の任意の時点をどうやって指定するのでしょうか？

通常、1 つのプロジェクトは複数のファイルで構成されます。ファイルごとに編集する回数は違うでしょうから、各ファイルのリビジョン番号は異なっているのが普通です。あるファイルの以前のリビジョンが欲しいと思っても、他の依存関係のあるファイルのリビジョンに影響を与えるので、任意の時点におけるほとんどすべてのファイルのリビジョンを把握しておかな

ければならず、現実的ではありません。そこで、CVSでは主にタグと呼ばれるユーザが任意につけることができる文字列でバージョンを管理します。

まず、ある時点でのプロジェクトの全ファイルに対してタグをつけます。タグをつけておけば、checkoutコマンドの-rオプションでタグを指定することもできます。また、後述の枝を作るときもタグが重要な役割を果たします。

4.1 枝の必要性

枝は、あるプロジェクトにおいて複数の異なる目的の開発を並行して実現するためのもので、一つの枝が一つの開発の流れに相当します。例えば、LinuxやApacheなどのソフトウェアは、開発版と安定版といった異なるバージョンが並行して存在します。どちらの枝にも独立に変更を加えることができます。

「大学の研究室程度では、そのようなオオゲサな仕組みはいらないや」と思われる方もいらっしゃるかもしれませんが、枝の機能は必須です。例えば、研究室で計算機実験用のソフトウェアを開発しているとしましょう。うまく実験が進み、結果を国際会議に提出しました。もちろん、会議終了後も改良を重ねていくでしょう。さて、ここでこのソフトウェアに結果の信頼性にかかわるような大きなバグが発見されたとします。

まずは、いつこのバグが混入したかを知る必要があります。このためには、日付やタグによるcheckoutで古いバージョンを取りだし、実験をやりなおせば確認できます。確認したら、会議に提出したバージョンにもバグが混入していたとしましょう。すぐに修正することは可能ですが、この修正をレポジトリに格納しないといけません。なぜなら、再度以前のバージョンにバグが見つかるかもしれず、その時また最初のバグから直していくのは面倒だからです。

では、この修正はどこに（どのリビジョンとして）格納すべきでしょうか。当然ですが、最新版に格納してしまうと、会議以後に新たに取り組んだ新機能や改良点を捨てないといけません。こういうときに、会議に提出した版のバグフィックス用の枝を作り、その枝に格納します。こうすることで、最新版のある幹とは独立になりますので、新機能や改良点は活かしたままで、古いバージョンのバグフィックスができます。もし、会議に提出したバージョンにあらたにバグが見つかったとしても、この枝の最新版ではこれまで見つかったバグは潰してあるので、新たなバグのみに専念できます。

また、新機能の追加をする場合、しばらくソフトウェアが不安定になることが予想されます。ソフトウェアが満足に動かないと、従来の機能による実験にも支障をきたします。そこで、枝を作り従来の機能の分は確実に動けるようにした上で、新機能の開発に取り組むことができます。新たに追加した機能が十分安定し、結果も満足のいくものであれば、従来版にマージすることもできます。

> ✔幹と枝に本質的には違いはなく、CVSのサブコマンドの多くがデフォルトで幹を対象にしているというだけです。例えば、checkoutコマンドは-rオプションで幹の名前を指定しなければ、幹における最新のリビジョンを作業用コピーとして取りだします。
>
> では、新機能を追加する場合は、どちらを枝にしてどちらを幹にするべきでしょうか。明確な基準はありませんが、私は採用するかどうか迷っている新機能であれば、新機能開発のための枝を作ることにしています。一方、開発の途中で不安定になろうとも、最終的にその機能を必ず使いたいのであれば、従来版の枝を作り、新機能は幹で開発しています。

4.2 タグを付ける

タグ付けは tag コマンドで行ないます。

```
% cvs tag tagname [files]
```

ファイル名を指定して、一部のファイルにのみタグをつけることも可能です。

> ✔ タグ付けは、作業用コピーに名前をつけるわけではなく、いきなりレポジトリに変更を加えます。したがって、作業用コピーに修正を加えてまだ commit していないファイルがあっても、修正する前のリビジョンに対して名前がつくことに注意してください。つまり、作業用コピーは単にリビジョンを参照するためだけに使われるだけです。

サンプルプロジェクトでは、以下のようになります。

```
% cvs tag SNAP20030116

cvs server: Tagging .
T MEMO.txt
T conclusion.tex
T cvs.tex
T intro.tex
T project.tex
T tag.tex
T work.tex
```

"T" によってタグがつけられているファイルが分かります。ここでは、単に日付をベースにしたタグをつけました[8]。

作業用コピーのリビジョンではなく、日付などを指定してタグをつけることもできます。この場合は、rtag コマンドを使います。指定の仕方は checkout と同様に、日付の指定は-D *date* で、タグの指定は-r *tag* で行ないます。

とにかく、無駄と思えるくらいタグをつけていきましょう。論文を CVS で書く場合は、投稿やカメラレディのときはもちろんですが、大幅な修正をほどこすときにもタグをつけましょう。さらに、ある程度の間隔で日付けなどをもとにしたタグ名などをつけておくとよいでしょう。

4.3 枝を作る

作業用コピーにあるリビジョンから枝を作る場合は tag -b を使って、以下のようにします。

```
% cvs tag -b branch
```

これで *branch* という名前の枝ができました。枝もタグと同様にレポジトリを直接書きかえて、作業用コピーに影響は与えません。

枝で新たな開発を行なう場合、さきほどの作業用コピーとは別に checkout する必要があります。

```
% cvs checkout -r branch project
```

[8] タグはアルファベットで始まる必要があります。

すでに、幹の作業用コピーのディレクトリがprojectという名前であると思います。その場合は、作業用コピーは別のディレクトリにcheckoutするかcheckout -d *dir*でディレクトリの名前を変えます。また、最初にあったディレクトリ*project*の名前を変更しても構いません。

また、単にさきほどの作業用コピーを枝に切り替えるのであれば、

```
% cvs update -r branch project
```

で切り替えることができます。

枝から取りだした作業用コピーで作業（commitやupdateなど）する場合は、特に指定しなくても枝にのみ限定されます。例えば、commitしたらこの枝の最新版として格納され、幹や他の枝に影響は与えません。したがって、複数の枝や幹で同時に作業する場合は、それぞれの作業用コピーのあるディレクトリに移動してから作業をすればよいことになります。

ある枝に対して行なった作業を、幹に対しても行ないたい場合は、幹の作業用コピーにおいて

```
% cvs update -j branch [files]
```

とします。ここで*branch*は変更を取り込みたい枝の名前です。これで、枝に分岐して以降の変更点が作業用コピーに取り込まれます[9]。このようにして変更点をマージした時には、*branch*枝にタグを打っておくようにしましょう。

さらにさきほどの*branch*枝で開発を続け、再度この枝の変更点を作業用コピーに取り込みたいとします。この場合、さきほどと同じようにすると、再度分岐点からの変更を取り込もうとするので、おかしな結果になります。こういう場合は、さきほど打ったタグを明示して

```
% cvs update -j tag -j branch [files]
```

とします。ここで*branch*を枝の名前ではなく、枝の先頭のタグ名と考えることができ、*tag*から枝の先頭までに行なった変更点を取り込むという意味になります。順番は重要で、必ず古いタグ名から先に指定してください。逆にしてしまうと、変更したところを元に戻すことになります。

最初にマージしたときにタグ*tag*を打っていないと、個々のファイルごとにリビジョン番号を指定するか、マージした日付を指定する必要がありますので注意してください。

5 おわりに

バージョン管理システムCVSについて説明しました。他にまだ説明が必要な項目があり、本文中での述べましたが、いくつかは次回に書く予定です。

CVSを使うまでは、なんとなく個人で使うにはしきいが高いと感じていましたが、一度使うとこれなしでは論文やプログラムが書けなくなります。複数の人間で作業をするときだけでなく、個人で使うにも手軽に使えます。基本的にテキストファイルであれば、どんなものでもCVSで管理できます。私は、論文とプログラム、プレゼンテーション資料[10]、いくつかの設定

[9]衝突が起こる可能性もあります。

[10]MagicPointやProsperのLaTeXファイルです。

ファイルなどをCVSで管理しています。

参考文献

[1] CVS–Concurrent Versions System (in Japanese) http://www.sodan.org/%7Epenny/vc/cvs-ja.html