CVS 사용

김민식

minskim (at) bawi.org

도운이

박용주
문서 형식을 **DocBook**으로 전환

yongjoo (at) kldp.org

$Date: 2002/01/07 23:19:15 $

고친 과정

고침 1.1 2002/01/06고친이 minskim
저장소 설정 방법 보충
고침 1.0.22001/08/30고친이 janghoon (control1 (at) dreamwiz.com)
서버설정법 추가 / 쓰기권한 과 읽기전용권한으로 저장소 관리 추가
고침 1.0.12001/08/21고친이 yongjoo
기존의 LinuxDoc 형식을 DocBook으로 변환
고침 1.0 1999/12/12고친이 minskim
이 글이 첫 선을 보였습니다.

1. 소개

CVS는 각종 파일의 버전을 쉽게 관리할 수 있도록 도와주는 도구이다. 독자들 중에는 아직 버전 관리의 필요성을 크게 느끼지 못하는 사람도 많이 있을 것이다. 하지만 CVS가 제공하는 기능들을 직접 보고 그 편리함을 체험해 본다면, 앞으로는 CVS 없이 프로젝트를 진행한다는 것은 상상할 수도 없는 일이 될 것이다.

예를 들어 간단한 프로그램을 하나 작성한다고 하자. 비교적 간단해 보이는 문제여서 곧바로 코딩을 시작하여 하루만에 작업을 마쳤다. 하지만 실행해보니 의도한 바대로 결과가 나오지 않는다. 원인이 무엇인지 알아내기 위해 이부분 저부분을 뜯어고치다 보면 처음의 모습은 거의 찾아볼 수 없는 복잡한 코드로 변해버린다. 하지만 결국 발견하는 것은 사소한 실수, 이제 그 동안의 작업을 모두 원래대로 돌려놓으려 하지만 이것 역시 버그를 잡는 일 못지 않게 어려운 일이다. 나름대로 바뀌었다고 생각되는 부분을 복구해봐도 제대로 되지 않고, 결국에는 처음부터 다시 코딩해 버린다. 만일 이 때 처음의 코드를 백업해 놓았다면 찾아낸 버그만을 고치고 작업을 계속 진행할 수 있을 것이다. 하지만 그렇다고 해서 매 작업 단계마다 코드

를 백업해 놓는다면 조금만 지나도 수없이 많은 파일들이 생겨날 것이다. 이런 파일들은 공간도 많이 차지할 뿐더러 그 많은 파일들을 관리하는 것 자체가 또다른 문제가 될 것이다. CVS는 바로 이러한 문제를 해결해 준다. CVS를 이용하면 매 작업 단계마다 코드를 저장할 수 있음은 물론, 원하는 단계의 코드를 언제라도 꺼내 볼 수 있다. 게다가 CVS는 각 단계에서 변경된 부분만을 저장하기 때문에 저장 공간도 많이 필요로 하지 않는다. 또한 각 단계마다 작업한 내용을 글로 적어 함께 저장할 수 있으므로 개발 내용을 한눈에 알아볼 수 있게 해 준다.

CVS의 또다른 장점은 여럿이 공동으로 진행하는 프로젝트에서 나타난다. 사실 여러명이 함께 작업하다 보면 각자 자기의 작업 파일을 갖게 되기 때문에 아무리 주의를 하여도 다른 사람이 변경해 놓은 것을 덮어 쓰는 일이 종종 발생하게 된다. 그렇다고 작업을 잘 나누어 각각이 서로 다른 파일만을 갖고 작업하도록 하는 것도 쉽지 않다. 처음에는 각기 다른 파일을 다루도록 일을 나누지만 조금 지나다보면 서로가 상대방의 파일을 고쳐야만 하는 경우가 발생하게 된다. 쉬운 대안은 한 파일을 작업하는 동안 다른 사람은 그 파일을 건드리지 못하게 하는 것이지만, 이럴 경우 파일을 수정하고 싶어도 다른 개발자가 그 파일을 수정하고 있다면 그 작업이 끝날 때까지 기다려야 하기 때문에 전체 작업 효율이 떨어지게 된다. 반면 CVS는 심지어 같은 파일도 여러 사람이 함께 작업할 수 있게 해 준다. 각자 한 파일 내의 다른 부분을 수정한다면 CVS가 그 내용을 하나로 합쳐 줄 것이다. 물론 두 사람이 동시에 같은 줄을 고친다면 CVS가 그것을 합쳐줄 도리는 없다. 하지만 그럴 경우라도 서로 같은 줄을 고쳤음을 개발자에게 알려주고 그 내용도 보여주어서 개발자가 적절한 조치를 취할 수 있도록 해 준다. 뿐만아니라 CVS는 Unix 환경은 물론, Windows나 Macintosh에서도 사용할 수 있어 각 개발자들이 원하는 환경에서 작업할 수 있다.

CVS는 이미 대다수의 공개 프로젝트에서 사용되어 그 효능을 입증하였다. 실제로 우리가 알고 있는 대부분의 공개 프로젝트가 CVS를 사용한다. Apache HTTP server, Mozilla 등이 대표적인 예이다. 이러한 공개 프로젝트들은 대부분 현재 개발 중인 내용을 CVS를 통해 모든 사람들이 받아 볼 수 있도록 하고 있다. 그렇기 때문에 CVS를 프로젝트에 도입하는 경우가 아니라 하더라도 이런 개발 버젼을 받아 보고 싶다면 CVS의 사용법을 익혀 둘 필요가 있다.

2. 동작 방식

2.1. 저장소

여러 명이 함께 작업할 수 있는 환경을 제공하려면 어떠한 요소가 필요할까? 가장 먼저 떠오르는 것은 같이 작업하는 파일을 보관할 장소일 것이다. CVS는 이러한 역할을 하는 곳을 저장소(repository)라 부른다. 그렇다고 특별한 데이터베이스나 다른 저장 매체를 사용하는 것이 아니라 그저 하나의 디렉토리에 불과하다. 이 디렉토리에는 CVS의 전반적인 설정 사항과 각 프로젝트의 파일들(문서, 프로그램 등)은 물론, 각 파일의 버젼 관리에 필요한 정보, 파일별 작업 기록들을 저장하게 된다. 여기에 있는 파일들은 모두 CVS가 관리하므로 사용자는 이 파일들을 직접 건드릴 필요가 전혀 없다.

2.2. 프로젝트 시작

일단 저장소를 만들고 나면 이후로는 여러 프로젝트들이 그 저장소를 이용할 수 있다. CVS를 이용하려는 사용자는 처음 프로젝트를 시작하는데에 필요한 파일 및 기본 디렉토리 구조를 자신의 작업 디렉토리에 만들고, 이를 저장소로 옮길 수 있다. 그러면 저장소에는 새로운 디렉토리가 만들어지고 이 곳은 앞으로 프로젝트 진행을 위한 공간으로 사용된다. 여기까지가 완료되면 프로젝트 시작을 위한 모든 준비가 끝난 것이다. 이후로는 프로젝트에 참여하는 사람들 각각의 몫이다.

2.3. 개발자 각자의 프로젝트 진행

저장소에 프로젝트를 위한 공간이 마련되고 나면 실제로 프로젝트에 참여하는 개발자는 어떻게 CVS를 이용하면 될까? 저장소의 내용을 가지고 직접 작업할 수는 없으므로 우선 저장소에 있는 내용을 복사하여 자신의 작업 디렉토리를 만들어야 한다. 이 과정을 CVS에서는 checkout이라고 한다. checkout으로 만들어진 작업 디렉토리는 자신만의 작업 공간이 되며, 개발자는 그 파일들을 마음대로 변경할 수 있다. 이후로는 계속 작업을 진행하면서 CVS의 명령을 이용하여 자신의 작업 결과를 저장소로 옮기고, 다른 사람이 저장소에 올려 놓은 작업 결과를 받아오는 일만을 반복하면 된다. 공동 작업으로 인해 생길 수 있는 문제들의 대부분은 CVS가 해결해 줄 것이다.

2.4. 요약

CVS를 이용하여 프로젝트를 수행하는 절차는 다음과 같다.

1. 저장소를 초기화한다. (CVS 관리자)
2. 프로젝트를 초기화한다. (프로젝트 관리자)
3. 작업 공간을 마련한다. (개발자)
4. 실제 작업 (개발자)
5. 자신의 작업 내용을 저장소에 저장하고 다른 개발자의 작업 내용을 받아온다. (개발자)

1번은 처음 CVS를 설치할 때 한 번만 해 주면 되고, 2번은 새로운 프로젝트를 시작할 때마다 필요한 작업이다. 각 개발자는 3번을 수행한 후 4번과 5번 과정을 되풀이하며 작업을 진행하게 된다.

3. 저장소 설정

3.1. 초기화

CVS를 사용하기 위해서 가장 먼저 할 일은 각 프로젝트의 파일들을 저장할 저장소(repository)의 위치를 정하는 것이다. 저장소를 이용할 프로젝트들의 규모를 고려하여 충분한 공간을 가진 디렉토리를 저장소로 쓰도록 한다. 저장소를 `/home/cvs`에 만들기로 했다면 다음과 같이 초기화할 수 있다.

```
# cvs -d /home/cvs init
```

여기서 `-d`는 저장소의 위치를 나타내며, 마지막의 `init`가 CVS 명령이다. 물론 이 명령을 사용하기 위해서는 `/home`에 쓰기 권한을 갖고 있어야 한다. 그러므로 대개의 경우 저장소의 초기화는 root 권한을 가진 씨스템 관리자가 하게 된다. 명령이 실행되고 나면 `/home`에 CVS란 디렉토리가 생기고, 그 안에는 `CVSROOT`란 디렉토리가 있을 것이다. 이 `CVSROOT` 디렉토리는 CVS의 각종 설정 내용을 담고 있는 디렉토리이다. 저장소(`/home/cvs`)의 파일들을 직접 건드리는 것은 절대로 피해야 한다. 이 파일들을 변경하고 싶다면 **cvs** 명령을 이용해야 한다.

3.2. 권한 부여

저장소를 만들었으면 CVS를 이용할 개발자들에게 저장소를 사용할 수 있는 권한을 주어야 한다. 권한 부여 방법은 개발자들이 저장소가 위치한 씨스템에 계정이 있는지에 따라 두 가지로 나뉜다.

3.2.1. 계정이 있는 경우

개발자들이 계정을 갖고 있는 경우는 개발자들이 모두 저장소가 있는 씨스템에서 작업을 하거나, 각자의 기계에서 작업을 하되 **ssh**이나 **rsh**을 써서 원격으로 CVS를 사용하는 경우이다. 이 방법은 다음에 나올 암호 인증 방식에 비해 설정이 간단하고, 별도의 포트를 열 필요가 없다는 장점이 있다. 개발자들이 모두 계정을 갖고 있으므로 저장소가 위치한 디렉토리에 읽기 및 쓰기 권한을 주기만 하면 되는 것이다. 그러기 위해 개발자들을 모두 같은 그룹(예: cvs)의 구성원으로 만든다. 이는 씨스템별로 존재하는 그룹 관리 도구를 이용하거나 **groupadd**와 같은 명령으로 cvs라는 그룹을 만들고, 그룹 관리 도구를 쓰거나 `/etc/group` 파일을 직접 수정하여 개발자들을 그 그룹에 추가하면 된다. 예를 들어 minskim과 sehkone이라는 사용자를 cvs 그룹에 추가한다면 다음과 같은 줄이 `/etc/group`에 생길 것이다.

```
cvs:*:510:minskim,sehkone
```

이제 다음 명령으로 디렉토리의 권한을 열어 주면 된다.

```
# chgrp -R cvs /home/cvs
# chmod ug+rwx /home/cvs /home/cvs/CVSROOT
```

이후로는 cvs 그룹에 속한 개발자는 이 저장소를 마음대로 이용할 수 있다.

3.2.2. 계정이 없는 경우

개발자들이 씨스템에 계정을 갖고 있지 않다면 CVS의 암호 인증 방식을 이용해서 CVS 써버에 접속할 수 있다. 개발자 각각은 CVS 계정(씨스템 계정과는 다르다)을 부여받게 되며, **inetd**를 통해 정해진 포트로 CVS를 사용하게 된다. 설정은 조금 복잡하지만 개발자들에게 일일이 씨스템 계정을 발급할 필요가 없으므로 씨스템 관리 측면에서는 보다 낳은 방법이라 할 수 있다. 특히 불특정 다수에 대해 CVS로 파일을 받아갈 수 있도록 해야 하는 공개 프로젝트의 경우 대부분이 이 방식을 채택하고 있다. 아파치나 모질라 같은 경우가 대표적인 예가 될 것이다. 반면 개발자들이 씨스템 계정을 갖고 있는 경우라도 **ssh**이나 **rsh**을 통한 접속을 허용하고 싶지 않을 경우는 별도의 CVS 계정을 만들어 암호 인증 방식을 이용할 수도 있다.

그러면 **inetd**로 CVS 접속을 허용하는 방법을 알아보자. 우선 CVS가 사용하는 포트 번호(2401번)를 등록해야 한다. `/etc/services`에 다음과 같은 줄이 있는지 살펴 보자.

```
cvspserver      2401/tcp
```

만일 없다면 위의 내용을 추가하면 된다. 다음은 실제로 해당 포트를 열어줄 차례인데, 이는 씨스템이 **inetd**를 쓰고 있는지, **xinetd**를 쓰고 있는지에 따라 설정 방법이 다르다. 먼저 **inetd**의 경우는 `/etc/inetd.conf`에 다음 내용을 추가한다.

```
cvspserver stream tcp nowait root /usr/bin/cvs cvs
   --allow-root=/home/cvs pserver
```

편의상 두 줄로 나타냈으나, 실제 파일에는 한 줄로 들어가야 한다. 만약 **tcpd**를 사용한다면 위의 줄 대신 다음을 추가한다.

```
cvspserver stream tcp nowait root /usr/sbin/tcpd /usr/bin/cvs
   --allow-root=/home/cvs pserver
```

물론 `/usr/bin/cvs`나 `/usr/sbin/tcpd`는 실제로 이들 명령이 위치하는 절대 경로로 써 주어야 한다.

새로운 설정 내용을 반영하려면 **inetd**를 재시작하여야 한다. **inetd**의 프로세스 ID가 357이라면 다음과 같이 HUP 신호를 보내면 된다.

```
# kill -HUP 357
```

xinetd를 쓴다면 `/etc/xinetd.d`에 `cvspserver`란 이름으로 별도의 파일을 만들어야 한다. 파일 내용은 다음과 같다.

```
# default: on
# description: The cvspsever serves CVS Passowrd Server ses-
sions; it uses \
#          unencrypted username/password pairs for authentication.
service cvspserver
{
        disable         = no
        flags           = REUSE
        socket_type     = stream
        wait            = no
        user            = root
```

```
        server          = /usr/bin/cvs
        server_args     = --allow-root=/home/cvs pserver
        log_on_failure  += USERID
}
```

inetd의 경우와 마찬가지로 `/usr/bin/cvs`는 **cvs** 명령의 절대 경로, `/home/cvs`는 저장소의 위치로 바꿔 준다.

xinetd를 재시작하는 방법도 **inetd**와 동일하다. 우선 **xinetd**의 프로세스 ID(357이라 가정한다)를 알아낸 후, HUP 신호를 보내자.

```
# kill -HUP 357
```

이제 접속 포트는 열어두었으니, 개발자들에게 CVS 계정을 발급하는 일만 남았다. 암호 인증 방식을 이용하는 경우, 계정과 암호는 저장소의 `CVSROOT` 디렉토리 밑에 `passwd`란 이름의 파일에 저장된다. 여기에서는 `/home/cvs/CVSROOT/passwd`가 될 것이다. 하지만 이 파일은 처음에는 존재하지 않는다. 그러므로 직접 만들어주어야 하는데, 먼저 예를 하나 보도록 하자.

```
minskim:YxNPCzaM/WCp2:cvs
sehkone:Yw2najHG5cLfo:cvs
```

각 줄은 한 사용자에 대한 정보를 담고 있다. 줄은 ':'을 경계로 다시 세 부분으로 나뉘는데 첫 부분이 사용자의 CVS 계정 이름(씨스템 계정과는 무관하다), 그 다음은 암호, 그리고 마지막은 씨스템 계정 이름이다. 즉, 이 파일에는 현재 minskim과 sehkone이라는 두 사용자가 등록되어 있고, 이들이 CVS 이용시에는 cvs란 씨스템 계정의 권한을 갖는 것이다. 암호부분은 유닉스 씨스템에서 전통적으로 사용되는 `crypt` 함수를 이용하여 변환된 값이 저장되어 있다. 새로운 사용자를 추가하려면 같은 형식으로 한 줄을 추가해 주면 된다.

마지막으로 필요한 것은 cvs란 씨스템 계정에 저장소에 대한 읽기 및 쓰기 권한을 주는 것이다. 3.2.1절과 일관성을 유지하려면 cvs란 그룹을 만들고 cvs란 사용자를 cvs 그룹에 추가한 후, cvs 그룹에 대한 권한을 같은 방법으로 열어주면 된다.

```
# chgrp -R cvs /home/cvs
# chmod ug+rwx /home/cvs /home/cvs/CVSROOT
```

4. 기본적인 사용 방법

저장소가 초기화되어 있고 사용할 수 있는 권한이 있다면 **cvs** 명령을 이용해서 바로 프로젝트를 시작할 수 있다. 이 절에서는 CVS의 모든 기능을 설명하기보다는 자주 사용되는 기능을 예와 함께 설명한다. 여기에서 설명하는 내용 정도만 가지고도 실제 프로젝트에서 CVS를 사용하는 데에는 무리가 없을 것이다.

CVS는 대부분의 Linux 배포본에 포함되어 있으므로 Linux를 사용하고 있다면 이미 CVS가 설치 되어 있을 것이다. 만일 CVS 명령이 없다면 http://www.cyclic.com/에서 적당한 파일을 받아 설치하면 된다.

CVS를 이용하는 모든 과정은 **cvs** 명령을 통해 이루어진다. **cvs** 명령의 기본 형식은 다음과 같다.

```
$ cvs [cvs 옵션] 명령 [명령 옵션과 인자]
```

명령 부분에는 실제로 **cvs**에게 지시할 명령이 오게 되며, 각 명령마다 요구하는 인자가 있을 수 있다. 실제로 사용되는 명령의 종류와 기능에 대해서는 새로운 명령이 소개될 때마다 언급하도록 하겠다.

4.1. 저장소 이용

모든 CVS 명령은 저장소의 위치를 알아야 수행될 수 있다. 앞에서 본 저장소 초기화의 예에서도 init 명령을 사용하면서 `-d` 옵션으로 저장소의 위치를 알려 주고 있다. CVS의 다른 명령들도 같은 옵션으로 저장소의 위치를 지정할 수 있다. 또다른 방법은 CVSROOT라는 환경변수의 값으로 저장소의 위치를 주는 것이다. 예를 들어 저장소의 위치가 `/home/cvs`인 경우, **sh**이나 **bash**를 쓴다면 다음의 내용을 초기화 파일(`.bash-profile` 등)에 넣어 놓으면 편리할 것이다.

```
CVSROOT=/home/cvs
export CVSROOT
```

csh이나 **tcsh**을 쓴다면 다음과 같이 한다.

```
setenv CVSROOT /home/cvs
```

이렇게 설정을 하고 나면 `-d` 옵션과 저장소의 위치를 매번 줄 필요 없이 바로 **cvs** 뒤에 원하는 명령을 주면 된다.

개발자가 CVS 명령을 이용하려 하는 경우 고려해야 할 사항이 하나 더 있다. 개발자가 어떤 방식으로 저장소가 위치한 기계에 연결하는가 하는 것이다. CVS는 다양한 연결 방식을 지원하는데, 여기에서는 주로 사용되는 세 가지 방식에 대해 저장소의 위치를 지정하는 방법을 알아보도록 하자. 이하의 예에서는 편의상 `-d` 옵션으로 저장소의 위치를 지정하겠다. 물론 `-d` 옵션은 항상 CVSROOT 환경변수로 대치하는 것이 가능하다.

4.1.1. 저장소와 개발자가 같은 기계를 쓸 경우

앞에서 저장소를 초기화한 것과 같은 경우이다. 같은 기계를 쓰기 때문에 저장소가 있는 디렉토리 이름을 주는 것으로 충분하다. 앞에서 만든 저장소에 CVS 명령을 내리려면 다음과 같이 한다.

```
$ cvs -d /home/cvs 명령
```

4.1.2. 개발자가 ssh이나 rsh을 이용하는 경우

개발자가 저장소와 다른 기계를 사용할 경우 일반적으로 쓰이는 방법이다. 여럿이 진행하는 프로젝트는 대부분 개발자들이 자신의 기계에서 작업을 하게 되는데, 이 방식을 이용하면 그러한 경우에도 쉽게 저장소를 공유할 수 있게 해 준다. 이 방법을 쓰려면 먼저 **ssh**이나 **rsh**을 사용할 수 있어야 한다. 예를 들어 저장소는 cvs.webdox.co.kr이란 기계에 있고, 개발자의 기계는 birch.webdox.co.kr이라고 하자. 개발자는 cvs.webdox.co.kr에 minskim이란 계정을 갖고 있고, birch.webdox.co.kr에 min이란 계정을 갖고 있다. 그러면 다음과 같은 방법으로 **ssh**을 쓸 수 있는지를 시험해 볼 수 있다.

```
ssh -l minskim cvs.webdox.co.kr 'echo $PATH'
```

rsh을 시험해 보려면 명령어만 **rsh**로 바꿔주면 된다.

위의 명령이 수행된다면 **ssh**(혹은 **rsh**)을 쓸 수 있는 것이다. **ssh**이 동작하지 않는다면 cvs.webdox.co.kr의 **sshd** 동작 여부와 설정 파일을 점검하도록 한다.

rsh이 동작하지 않는다면 cvs.webdox.co.kr에 로그인하여 `.rhosts`란 파일을 만들자. `.rhosts`에는 다음과 같은 내용이 있어야 한다.

```
birch.webdox.co.kr min
```

이제 다시 한 번 **rsh** 명령을 시험해보자. 이번에도 안 된다면 cvs.webdox.co.kr에서 **rsh**을 통한 접속을 막아 놓았을 가능성이 크므로 cvs.webdox.co.kr의 네트웍 설정을 점검하도록 한다.

ssh이나 **rsh**을 쓸 수 있다면 저장소 위치에 다음과 같이 계정과 기계 이름을 포함시켜서 CVS 명령을 사용할 수 있다.

```
$ cvs -d :ext:minskim@cvs.webdox.co.kr:/home/cvs 명령
```

여기서 :ext:가 **ssh**이나 **rsh**을 이용한 접속 방식임을 나타내는 구실을 한다. 다만 기본적으로 **cvs**는 :ext:로 시작하는 저장소의 경우 **rsh**을 이용하므로, **ssh**을 쓰려면 CVS_RSH이라는 환경 변수를 **ssh**로 설정해 주어야 한다. 물론 **ssh** 외에도 **rsh**과 같은 방식으로 동작하는 다른 명령이 있다면 그 명령을 사용할 수도 있다. **rsh**은 제삼자가 네트웍을 통해 암호를 가로챌 수 있으므로, 보안이 중요시되는 환경이라면 **ssh**을 사용할 것을 권한다.

4.1.3. 암호 인증 방식 이용

암호 인증 방식은 앞의 두 방식과는 달리 CVS 명령을 사용하기에 앞서 로그인을 해야 한다. CVS로 파일을 제공하는 공개 프로젝트들의 경우 홈페이지에서 'CVSROOT를 ...로 설정하십시오'라는 문구를 볼 수 있을 것이다. 이것이 바로 암호 인증 방식의 저장소 위치이다. cvs.webdox.co.kr이란 기계의 `/home/cvs`에 저장소가 위치하고 minskim이란 CVS 계정이 만들어져 있다면 다음과 같이 로그인할 수 있다.

```
$ cvs -d :pserver:minskim@cvs.webdox.or.kr:/home/cvs login
```

그러면 암호를 물어오는데 암호는 `CVSROOT/passwd`에 CVS 계정을 추가할 때 사용한 것과 같다. 저장소 위치의 :ext:만 :pserver:로 바뀌었을 뿐, 형식은 **ssh**을 이용하는 경우와 동일함을

알 수 있다.

일단 로그인을 하고 나면 인증 정보가 .cvspass란 파일에 저장이 되며, 이후로는 암호 없이 CVS 명령을 이용할 수 있다. 방법은 다음과 같다.

```
$ cvs -d :pserver:minskim@cvs.webdox.or.kr:/home/cvs 명령
```

4.2. 프로젝트 초기화

사용 가능한 CVS 저장소가 있고, 어떻게 저장소를 이용하는지 알고 있다면 프로젝트를 시작할 모든 준비가 끝난 셈이다. 더군다나 기존의 프로젝트에 CVS를 도입하려는 경우라면 이미 디렉토리와 파일들이 있을테니 이를 저장소에 저장하는 방법만 배우면 된다. 하지만 처음 시작하는 프로젝트를 CVS로 관리하려 한다면 먼저 디렉토리 구조를 잘 설계해야 한다. 물론 언제라도 CVS를 이용해서 파일의 위치를 옮기고 새로운 디렉토리를 만드는 등의 일은 가능하다. 하지만 처음부터 디렉토리 구조와 각 디렉토리에 들어갈 파일들을 잘 정해 놓는 것은 전체 프로젝트 관리에도 도움을 주므로 되도록이면 나중에 파일이나 디렉토리의 위치 변경이 생기지 않도록 설계를 하고 프로젝트를 시작하도록 하자.

먼저 할 일은 프로젝트의 이름을 정하는 것이다. 이 이름은 저장소에서 이 프로젝트 관련 파일들을 저장하는 디렉토리의 이름으로 쓰인다. 여기에서는 myprj란 이름으로 프로젝트를 만들어 보기로 하겠다. 이후에 나올 모든 예는 환경변수 CVSROOT가 제대로 설정되어 있다고 가정한다. CVSROOT를 어떻게 설정해야 하는가에 대해서는 4.1절의 내용을 참고하기 바란다.

myprj란 디렉토리가 다음과 같이 되어 있다고 하자.

```
$ cd myprj
$ ls
Hello.java      README
```

저장소에 새 프로젝트를 만들고 이 두 파일을 저장하기 위해서는 import 명령을 사용한다. import 명령의 사용법은 다음과 같다.

```
$ cvs import -m "메시지" 프로젝트이름 vender_tag release_tag
```

'메시지'는 프로젝트를 시작하면서 저장소에 기록하고 싶은 내용을 적어주면 된다. CVS는 파일을 저장할 때마다 메시지를 적도록 하고 있다. 이 내용을 원하는 때에 다시 볼 수 있으므로 개발자가 어떤 작업을 했는지를 적어 놓으면 다른 개발자들이 파일의 변경 내용, 작업 진척 상황등을 파악하는데에 도움을 줄 수 있다. 뒤의 두 태그는 지금 단계에서는 별 의미가 없으므로 적당한 말을 써 주면 된다. myprj란 프로젝트를 저장소에 만들기 위해서는 다음 명령을 사용한다.

```
$ cvs import -m "프로젝트 시작" myprj webdox start
N myprj/Hello.java
N myprj/README
```

```
No conflicts created by this import
```

앞에 말했듯이 뒤의 두 단어는 별 의미가 없으므로 크게 신경쓰지 말기 바란다. 이 명령을 실행하면 CVS는 저장소에 `myprj`란 디렉토리를 만들고, 거기에 `README`와 `Hello.java`를 저장한다. 이 외에도 옵션으로 준 메시지와 각 파일의 부가적인 정보를 기록하게 된다. 출력되는 메시지를 보면 각 파일 이름 앞에 N이란 글자가 있는 것을 알 수 있다. CVS는 특정 명령 수행 시에 파일별로 수행 결과를 나타낸다. N은 새로운 파일이 추가된 것을 의미한다. 이후의 예에서 다른 경우들도 보게 될 것이다. 마지막 줄에서 conflict란 현재 디렉토리에 있는 파일을 저장소에 저장하게 되면 어떤 이유로 이미 저장소에 있는 파일과 충돌을 일으키는 경우를 뜻한다. 다른 사람이 작업해서 저장소에 저장해 놓은 것을 덮어쓰려 한다거나 하는 경우인데, 자세한 것은 뒤에서 conflict가 발생하는 경우를 예로 들어 설명하겠다. 여기에서는 저장소에 아무 파일도 없으므로 충돌(conflict)이 생기지 않는 것이 당연하다.

import 명령으로 프로젝트를 초기화하고 나면 더 이상 현재 디렉토리에 갖고 있는 것들을 사용하면 안 된다. 현재 디렉토리에는 CVS에 관련된 정보가 아무 것도 없기 때문이다. 대신 저장소에 모든 내용이 들어 있으므로 `myprj` 디렉토리 자체를 완전히 지워버려도 괜찮다. 다음 절에서 소개할 명령들을 이용해서 언제라도 다시 불러올 수가 있다. 이상으로 프로젝트 초기화는 모두 끝난 것이며, 이후로는 각 개발자들이 CVS를 이용하여 프로젝트를 진행시키는 일만이 남아 있다.

4.3. 프로젝트 진행

이제 프로젝트에 참여하는 개발자의 입장에서 프로젝트를 바라보자. 모든 필요한 파일들은 저장소에 갖추어져 있고, 프로젝트 관리자로부터 저장소의 위치와 프로젝트 이름도 통보받았다. 그러면 개발자는 앞으로의 작업을 위해 저장소의 위치를 환경변수 CVSROOT에 저장하고, 작업을 시작하면 된다. 단, CVS는 파일의 변경 시각을 참조하는 경우가 많으므로 각 개발자들이 사용하는 기계의 시간을 서로 맞추어 놓을 필요가 있다.

4.3.1. 작업 공간 생성 (checkout)

작업을 시작하기 위해서는 파일을 마음대로 변경하고 저장하여 테스트해 볼 수 있는 작업 공간이 필요하다. 저장소에 있는 파일들을 불러와 나만의 작업 공간을 만드는 명령이 checkout이다. 앞에서 만든 myprj를 checkout 명령으로 불러 오자.

```
$ cvs checkout myprj
cvs checkout: Updating myprj
U myprj/Hello.java
U myprj/README
```

checkout 대신 co만 써 주어도 된다. 앞에 붙은 U는 파일이 갱신(update)되었음을 의미한다. 명령을 수행하고 나면 현재 디렉토리에 myprj란 디렉토리가 생긴다. 이 디렉토리가 바로 작업 공간이다. 이 안에서 필요에 따라 파일들을 수정하고, 이를 다시 저장소로 저장하면 되는 것이다. 그러면 디렉토리 안에는 어떤 파일들이 들어 있을까?

```
$ cd myprj
```

```
$ ls
CVS    Hello.java    README
```

프로젝트를 초기화할 때 있던 파일들이 그대로 들어 있는 것을 볼 수 있다. 하지만 하나 달라진 것이 있다. CVS란 디렉토리가 바로 그것이다. 바로 이 디렉토리에 CVS가 파일들을 관리하는데에 필요한 정보가 기록된다. 예를 들면 각 파일들의 버전, 최종 수정 시각, 저장소의 위치 등이 이에 속한다. 앞으로 여러 CVS 명령들을 사용하게 되는데, 그때마다 CVS는 이 디렉토리의 정보를 참고하여 각 명령을 수행한다. 하지만 개발자가 이 디렉토리의 내용에 신경쓸 필요는 전혀 없다. 관심이 있다면 살펴보는 것은 좋지만, 내용을 임의로 변경한다거나 해서는 안된다.

4.3.2. 작업 내용의 저장 (commit)

만들어진 작업 공간에서 작업을 시작해보자. `Hello.java`의 내용이 다음과 같다고 하자.

```
public class Hello {
    public static void main(String args[]) {
    }
}
```

여기에 인사말을 출력하는 부분을 추가해보자. 아무 편집기로나 파일을 열어 작업하면 된다. CVS를 사용한다고 해도 특별한 에디터나 통합 개발 환경을 필요로 하지는 않는다. `Hello.java`에 '안녕하세요?'라고 출력하는 문장을 추가했다고 하자.

```
public class Hello {
    public static void main(String args[]) {
        System.out.println("안녕하세요?");
    }
}
```

컴파일을 해서 실행을 해 보고 예상대로 실행된다면 이제 작업 내용을 저장소에 저장해야 한다. 그래야 다른 개발자들도 인사말이 추가되었다는 것을 알게 될 것이다. 파일의 변동 사항을 저장소에 저장하는 명령은 commit이다.

```
$ cvs commit -m "인사말 추가" Hello.java
Checking in Hello.java;
/home/cvs/myprj/Hello.java,v  <--  Hello.java
new revision: 1.2; previous revision: 1.1
done
```

어떤 명령인지 쉽게 이해할 수 있을 것이다. `-m`은 앞의 예와 마찬가지로 메시지를 의미한다. 이 명령으로 `Hello.java`는 '인사말 추가'란 메시지와 함께 저장소에 저장된다. 그렇다고 이전의 `Hello.java`가 없어지는 것은 아니다. CVS는 각 버전의 변경 내용을 파악하여 언제라도 개발자가 원하는 버전을 꺼내 줄 수 있도록 파일들을 저장한다. 출력 결과를 보면 원래 있던 `Hello.java`는 1.1이고, 새로 저장된 `Hello.java`는 1.2임을 알 수 있다. 이 번호는 CVS가 자동으로 붙이는 것이며 변경 사항이 저장될 때마다 올라가므로 각 파일마다 다를 수 있다. 나중에 특정 버전이 필요하다면 이 번호를 이용해서 불러 오면 된다.

맨 끝의 파일 이름은 생략할 수 있다. 파일 이름을 주지 않으면 CVS가 변경된 파일을 모두 찾아 저장소에 저장한다. 이때 저장되는 모든 파일에는 같은 메시지가 붙게 된다. 하지만 파일마다 다른 메시지를 붙이고 싶다거나 특정 파일의 변경 내용만을 저장하고 싶다면 위의 예처럼 파일 이름을 명시해 주어야 한다.

commit을 할 때 주의할 점은 반드시 제대로 동작하는가를 확인하고 commit을 해야 한다는 것이다. 일단 commit을 하면 다른 개발자들도 나의 작업 결과를 받아갈 수 있다. 그러므로 내가 제대로 컴파일조차 되지 않는 파일을 commit으로 저장소에 저장하면 이 파일을 받아간 다른 개발자들도 컴파일이 안 되는 결과를 가져올 것이다. 특히나 여러 파일을 변경하고 일부 파일만을 commit하는 경우라면 더욱 주의해야 한다. 공동 작업을 하는 경우라면 이렇게 자신의 부주의로 다른 개발자들에게 피해를 입히는 일은 절대로 삼가야 한다. 대개의 경우 변경 내용 전체가 서로 관련이 있는 경우가 많으므로 특정한 인자를 주지 않고 commit 명령을 사용하여 작업 공간 전체의 변경 내용이 한꺼번에 저장소에 저장되도록 하는 것이 좋다.

4.3.3. 저장소의 파일 받아오기 **(update)**

공동 작업에서 자신의 작업 결과를 저장하는 일 못지 않게 중요한 일은 다른 사람의 작업 결과를 받아오는 일이다. 자신이 작업한 내용을 모두 저장한 상태라면 작업 디렉토리를 지워버리고 checkout 명령으로 새로운 작업 공간을 만드는 것도 한 방법이 될 것이다. 하지만 이는 모든 파일을 새로 받아오므로 저장소와의 연결이 느릴 경우에는 상당한 시간을 기다려야 할 수도 있다. 보다 좋은 방법은 update 명령을 사용하는 것이다. 작업중이던 디렉토리에서 이 명령을 쓰면 CVS에 저장된 파일들 중 내가 받아온 이후로 변경된 것들만을 다시 받아 온다.

```
$ cvs update
cvs update: Updating .
```

현재 작업 공간에 있는 파일들이 저장소에 있는 파일과 모두 동일하면 위와 같이 별다른 출력 없이 끝날 것이다. 그렇다면 작업 공간에 있는 파일을 수정하고 commit을 하지 않은 상태라면 어떤 결과가 나오는지 보기 위해 `Hello.java`를 다음과 같이 수정해보자.

```
public class Hello {
    public static void main(String args[]) {
        System.out.println("안녕하세요?");
System.out.println("반갑습니다.");
    }
}
```

원래의 파일에 "반갑습니다."를 출력하는 문장을 추가하였다. 이제 update 명령을 실행하면 다음과 같이 나올 것이다.

```
$ cvs update
cvs update: Updating .
M Hello.java
```

파일 이름 앞에 있는 영문자(이 경우는 M)가 각 파일별 상태를 알려준다. M은 파일이 변경된(modified) 상태임을 의미한다. 즉, 파일의 내용이 원래 저장소에 있는 것과 달라졌을 때 M을 써서 표시하게 된다.

저장소의 파일과 작업 공간의 파일이 달라질 수 있는 또 다른 경우는 다른 사람이 파일을 변경하여 commit 명령으로 저장소에 저장한 경우이다. 예를 들어 원래의 README 파일이 다음과 같았다고 하자.

```
$ cat README
이 파일은 매우 중요한 내용을 담고 있습니다.
```

그런데 다른 개발자가 README 파일을 다음과 같이 변경하여 저장소에 저장하였다.

```
$ cat README
이 파일은 매우 중요한 내용을 담고 있습니다.
1999년 12월 11일
$ cvs commit -m "날짜 추가" README
Checking in README;
/home/cvs/myprj/README,v  <--  README
new revision: 1.2; previous revision: 1.1
done
```

이제 내가 update를 하면 다음과 같은 결과를 볼 수 있다.

```
$ cat README
이 파일은 매우 중요한 내용을 담고 있습니다.
$ cvs update
cvs update: Updating .
M Hello.java
U README
```

README 앞에 붙은 U는 파일이 갱신된(updated) 것을 의미한다. 이제 내 작업 공간의 README가 저장소에 저장된 내용과 같게 갱신된 것을 볼 수 있다.

```
$ cat README
이 파일은 매우 중요한 내용을 담고 있습니다.
1999년 12월 11일
```

그렇다면 이 두 경우가 혼합된 경우는 어떤 일이 발생할까? 즉, 다른 개발자가 수정하여 저장소에 저장한 파일을 나 역시 수정한 후 update를 하는 경우이다. 다른 개발자가 `Hello.java`를 다음과 같이 `main` 함수 앞에 주석을 한 줄 추가하여 commit을 했다고 생각해보자.

```
$ cat Hello.java
public class Hello {
    // main 함수
    public static void main(String args[]) {
        System.out.println("안녕하세요?");
    }
}
$ cvs commit -m "주석 추가" Hello.java
Checking in Hello.java;
/home/cvs/myprj/Hello.java,v  <--  Hello.java
new revision: 1.3; previous revision: 1.2
```

```
done
```

그런 다음 내가 update를 하면 다음과 같이 된다.

```
$ cat Hello.java
public class Hello {
    public static void main(String args[]) {
        System.out.println("안녕하세요?");
System.out.println("반갑습니다.");
    }
}
$ cvs update
cvs update: Updating .
RCS file: /home/cvs/myprj/Hello.java,v
retrieving revision 1.2
retrieving revision 1.3
Merging differences between 1.2 and 1.3 into Hello.java
M Hello.java
```

여기서 update 명령은 현재 디렉토리의 `Hello.java`는 1.2에서 수정된 상태이고, 저장소에 있는 `Hello.java`는 1.3이기 때문에 저장소의 내용을 가지고 1.2와 1.3의 차이를 파악하게 된다. 그 차이가 한 줄이 추가된 것임을 발견하고 나면 그 차이를 현재 디렉토리의 `Hello.java`에 반영한다. 하지만 `Hello.java`에는 "반갑습니다."를 출력하는 줄도 들어 있기 때문에 여전히 저장소에 있는 파일과는 다른 상태이다. 따라서 저장소에 있는 내용에서 변경되었다는 의미의 M을 출력하는 것이다. 실제로 `Hello.java`의 내용을 보면 다음과 같이 바뀐 것을 볼 수 있다.

```
$ cat Hello.java
public class Hello {
   // main 함수
   public static void main(String args[]) {
      System.out.println("안녕하세요?");
      System.out.println("반갑습니다.");
   }
}
```

다른 개발자가 추가한 주석이 들어 있으면서 내가 추가한 줄도 그대로 있음을 볼 수 있다. 이것이 CVS의 강력한 장점이다. 즉, 서로 다른 두 개발자가 같은 파일을 수정하였는데도, 서로의 작업을 하나로 합쳐주는 것이다. 한가지 주의할 점은, 만일 update를 수행할 때 `Hello.java`를 에디터에서 불러 작업중이었다면, 계속 에디터로 작업을 한 후 저장할 때 update에 의해 변경된 내용을 덮어 쓸 수가 있다는 것이다. 대부분의 에디터는 작업하고 있는 파일이 다른 프로그램에 의해 변경되면 이 사실을 사용자에게 알리고, 변경된 파일을 다시 불러올 수 있도록 하지만, 간혹 그렇지 못한 에디터도 있을 수 있으므로 조심하기 바란다.

4.3.4. 충돌의 해결

비록 CVS가 두 개발자의 작업 내용을 합쳐 주기는 하지만 여기에도 한계는 있다. 두 개발자가 같은 부분을 수정한다면 CVS는 누구의 변경 내용을 택해야 하는지 알 도리가 없다. 이런 경우를 충돌(conflict)이라 하며, CVS는 이 사실을 개발자에게 알려 개발자가 적절한 조치를 취할 수 있도록 한다.

위의 예에서 다른 개발자가 작업을 계속하여 "환영합니다."를 출력하도록 만들고, 이를 저장소에 저장했다고 하자. 그러면 저장소에는 다음과 같은 내용이 들어 있을 것이다.

```
public class Hello {
   // main 함수
   public static void main(String args[]) {
      System.out.println("안녕하세요?");
      System.out.println("환영합니다.");
   }
}
```

이 때 내가 update를 하면 충돌이 발생한다.

```
$ cvs update
cvs update: Updating .
RCS file: /home/cvs/myprj/Hello.java,v
retrieving revision 1.3
retrieving revision 1.4
Merging differences between 1.3 and 1.4 into Hello.java
rcsmerge: warning: conflicts during merge
cvs update: conflicts found in Hello.java
C Hello.java
```

현재 디렉토리의 `Hello.java`는 1.3에서 변경된 상태("반갑습니다." 출력 추가)이다. 저장소에 저장되어 있는 것은 1.4이므로 CVS는 저장소에 있는 1.3과 1.4의 차이를 살핀다. 둘의 차이는 다른 개발자에 의해 "환영합니다."를 출력하는 줄이 추가된 것이다. 이를 현재디렉토리의 `Hello.java`에 추가하려고 시도하지만, 그 위치에 이미 다른 내용("반갑습니다." 출력)이 들어가 있음을 알게 된다. 결국 CVS는 충돌(C로 표시)이 생겼음을 알리고 수행을 마친다. `Hello.java`의 내용을 보면 어디에서 어떤 충돌이 발생했는지 CVS가 표시해 놓은 것을 발견할 수 있다.

```
$ cat Hello.java
public class Hello {
   // main 함수
   public static void main(String args[]) {
      System.out.println("안녕하세요?");
<<<<<<< Hello.java
      System.out.println("반갑습니다.");
=======
      System.out.println("환영합니다.");
>>>>>>> 1.4
   }
}
```

<<<<<<<와 >>>>>>> 사이가 충돌이 일어난 부분이다. 그 부분은 다시 두 부분으로 나뉘는데, ======= 이전까지가 현재 디렉토리에 있는 파일의 내용이고, 그 이후가 저장소에 있는 파일의 내용이다. 개발자는 이걸 보고 어느 한 쪽을 없애거나 두 내용을 적절히 합친 후 다시 commit을 해 주면 된다. 이 경우 "반갑습니다."가 더 마음에 들어 "환영합니다."를 삭제하기로 했다면 다음과 같이 고치고 commit을 한다.

```
$ cat Hello.java
public class Hello {
   // main 함수
   public static void main(String args[]) {
      System.out.println("안녕하세요?");
      System.out.println("반갑습니다.");
   }
}
$ cvs commit -m "인사말 변경" Hello.java
Checking in Hello.java;
/home/cvs/myprj/Hello.java,v  <--  Hello.java
new revision: 1.5; previous revision: 1.4
done
```

하지만 다른 개발자가 수정한 내용을 그대로 놔 두고 내가 고친 부분을 없애기로 했다면 따로 commit을 할 필요가 없다. 이미 저장소에 그 내용이 들어 있기 때문이다.

충돌은 commit 시에도 발생할 수 있다. 앞의 예에서 다른 개발자가 "환영합니다." 출력 문장을 추가하여 commit을 한 후, 내가 update 대신 commit을 하려 했다면, CVS는 다음과 같이 충돌이 발생했음을 알려 줄 것이다.

```
$ cvs commit -m "반갑습니다 추가" Hello.java
cvs commit: Up-to-date check failed for `Hello.java'
cvs [commit aborted]: correct above errors first!
```

이런 메시지를 보게 되면 update 명령으로 어디에서 충돌이 발생했는지를 확인하면 된다. 그 결과와 수정 방법은 앞에서 나온 것과 동일하다.

4.3.5. 파일의 추가/삭제 (add/delete)

지금까지는 저장소에 있는 파일을 수정하고, 그 결과를 다시 저장하는 방법을 알아보았다. 하지만 작업을 하다 보면 새로운 파일을 만들거나 기존의 파일을 지워야 할 경우가 생긴다. 이런 경우에 사용하는 명령이 add와 delete이다. 사용 방법은 매우 간단하다. `Test.java`란 파일을 새로 만들려면 먼저 현재 디렉토리에 `Test.java`를 생성한 후 다음과 같이 하면 된다.

```
$ cvs add Test.java
cvs add: scheduling file `Test.java' for addition
cvs add: use 'cvs commit' to add this file permanently
```

메시지에서 알 수 있듯이 add 명령만으로는 저장소에 `Test.java`가 생기지 않는다. 저장소를 변경하는 것은 commit 명령뿐이다. add 명령은 단지 commit 명령시에 `Test.java`를 추가해야 한다는 것을 기록해 놓을 뿐이다. 이 사실은 update로 확인해 볼 수도 있다.

```
$ cvs update
cvs update: Updating .
A Test.java
```

앞의 A는 추가될(added) 파일임을 의미한다. 그러므로 Test.java는 나중에 작업 내용 전체를 commit할 때 저장소에 추가될 것이다. 아니면 미리 `Test.java`를 인자로 commit을 하여 바로 추가되도록 할 수도 있다.

```
$ cvs commit -m "새 파일" Test.java
RCS file: /home/cvs/myprj/Test.java,v
done
Checking in Test.java;
/home/cvs/myprj/Test.java,v  <--  Test.java
initial revision: 1.1
done
```

추가된 파일은 다른 개발자가 update를 할 때 그 개발자의 작업 공간에도 생겨나며, 파일 내용이 갱신되는 것과 마찬가지로 U를 써서 표시한다.

```
$ cvs update
cvs update: Updating .
U Test.java
```

파일을 지우는 것도 마찬가지이다. `Test.java`를 지우려면 다음과 같이 한다.

```
$ rm Test.java
$ cvs delete Test.java
cvs remove: scheduling `Test.java' for removal
cvs remove: use 'cvs commit' to remove this file permanently
```

delete 명령을 사용하기 전에 반드시 먼저 파일을 작업 공간에서 삭제하여야 한다. 이제 commit 명령을 사용하면 `Test.java`는 저장소에서 삭제될 것이다. 이것도 update로 확인해 본다면 다음과 같이 나올 것이다.

```
$ cvs update
cvs update: Updating .
R Test.java
```

R은 삭제될(removed) 파일임을 나타낸다. 이제 commit으로 완전히 삭제해 보자.

```
$ cvs commmit -m "삭제" Test.java
Removing Test.java;
/home/cvs/myprj/Test.java,v  <--  Test.java
new revision: delete; previous revision: 1.1
done
```

이렇게 삭제된 파일은 이후로 다른 개발자가 update를 하게 되면 다음과 같은 메시지를 출력하면서 그 개발자의 작업 디렉토리에서도 삭제된다.

```
$ cvs update
cvs update: Updating .
cvs update: warning: Test.java is not (any longer) pertinent
```

4.3.6. 작업 기록 열람 (log)

그 동안의 예에서 본 바와 같이 CVS는 작업 내용을 저장소에 저장할 때마다 메시지를 적도록 하고 있다. 이 내용은 저장소에 함께 저장되어 log 명령을 이용하면 언제라도 꺼내 볼 수 있다. `Hello.java`를 예로 들어 보자.

```
$ cvs log Hello.java

RCS file: /home/cvs/myprj/Hello.java,v
Working file: Hello.java
head: 1.5
branch:
locks: strict
access list:
symbolic names:
        start: 1.1.1.1
        webdox: 1.1.1
keyword substitution: kv
total revisions: 6;    selected revisions: 6
description:
----------------------------
revision 1.5
date: 1999/12/12 04:04:23;  author: minskim;  state: Exp;  lines: +1 -0
인사말 변경
----------------------------
revision 1.4
date: 1999/12/12 04:03:50;  author: sehkone;  state: Exp;  lines: +0 -1
환영합니다 추가
----------------------------
revision 1.3
date: 1999/12/12 04:01:40;  author: sehkone;  state: Exp;  lines: +2 -1
주석 추가
----------------------------
revision 1.2
date: 1999/12/12 03:51:36;  author: minskim;  state: Exp;  lines: +1 -0
인사말 추가
----------------------------
revision 1.1
date: 1999/12/12 03:50:48;  author: minskim;  state: Exp;
branches:  1.1.1;
```

```
Initial revision
----------------------------
revision 1.1.1.1
date: 1999/12/12 03:50:48;  author: minskim;  state: Exp;  lines: +0 -0
프로젝트 시작
=============================================================================
```

매번 commit을 할 때마다 날짜와 시각, 저장한 사람, 메시지 등이 기록되는 것을 알 수 있다. 이렇게 log 명령을 쓰면 언제 누가 어떤 작업을 했는지를 한눈에 알 수 있기 때문에 그간의 작업 내용을 쉽게 파악할 수 있다. 이 기능을 잘 활용하기 위해서는 무엇보다도 매번 commit을 할 때 알기 쉽고 작업 내용을 잘 반영할 수 있는 메시지를 적는 것이 중요하다.

5. 저장소 관리

5.1. 암호 인증 방식의 접근 제어

가장 널리 쓰이는 암호 인증 방식의 예는 읽기 전용 사용자가 필요할 때일 것이다. 많은 공개 프로젝트들이 이러한 읽기 전용 계정을 제공하여 누구나 파일들을 받아갈 수 있도록 하고 있다. 그렇다면 저장소에 대한 접근 권한을 제어할 수 있다는 방법이 있다는 얘기인데, 이러한 것은 어디에서 설정해 줄 수 있을까? CVS는 접근 제어를 위해 두 가지 방법을 제공한다. `CVSROOT` 디렉토리에 위치한 `readers`와 `writers`란 파일이 바로 그것이다. 이 파일들은 `passwd`와 마찬가지로 처음에는 존재하지 않는다. 기능이 필요하다면 직접 만들어 주어야 하는 파일들인 것이다. 형식은 매우 간단해서, CVS 계정 이름을 한 줄에 하나씩 나열하기만 하면 된다.

읽기 전용 계정을 만들고 싶다면 `readers`를 이용한다. sehkone과 chang이란 CVS 계정을 읽기 전용으로 하고 싶다면 `CVSROOT` 디렉토리에 다음과 같이 `readers`를 만든다.

```
sehkone
chang
```

마지막 줄 끝에 개행문자를 꼭 넣어 주어야 함을 유념하기 바란다. 이렇게 하면 sehkone과 chang이란 사용자는 읽기만이 허용되며, 나머지 모든 사용자는 읽기와 쓰기가 모두 가능하다.

하지만 `writers` 파일이 있다면 사정이 전혀 달라진다. `writers`에 이름이 올라 있는 계정에 한해서만 읽기와 쓰기가 허용되며, 나머지는 모두 읽기 전용 계정이 된다. 그렇다면 `readers`와 `writers`가 모두 있는 경우는 어떻게 될까? 이 경우 `readers`에는 이름이 없고 `writers`에만 이름이 있는 계정은 일기와 쓰기 권한을, 나머지 모든 계정은 읽기 전용 권한만을 얻는다.