VTT과제 진행을 위한 오픈소스 프로젝트 개발 방법론

제 1 절 오픈소스 소프트웨어 소스코드 관리

소프트웨어 개발에 있어서, 소스코드의 관리는 중요한 포인트 중의 하나이다. 다양한 버전과 변경 관리, 협업을 위해서는 소스코드를 저장 및 관리할 수 있는 시스템이 필요하고, 이를 VCS (Version Control System) 또는 SCM (Source Code Management) System이라고 한다. 이 소스코드 관리 시스템의 주요 기능은 다음과 같다.

- 협업을 위한 코드 공유 - 여러 사람이 협업을 할 경우, 코드를 각 개발자와, 팀 간에 공유할 수 있어야 한다,
- 접근 제한 - 사용자의 권한 등급에 따라 접근을 제한해야 한다.
- 다양한 버전(형상) 관리 - 소프트웨어 개발 버전 또는 릴리즈 (브랜치) 마다, 다른 코드를 저장할 수 있어야 한다. 예를 들어 릴리즈 된 버전이나 마이너 버전에 대한 코드 관리, 패치 코드 관리 등이 그 예가 될 수 있다. 또한 다양한 브렌치 중 두개 이상의 브렌치를 하나의 코드로 병합(merge)할 수 있어야 한다.
- 특정 시점 추적 - 아래 태깅에서 설명
- 변경 추적 - 마지막으로, 각 코드에 대한 변경을 추적할 수 있어야 한다. 누가? 언제? 어떤 이유로 코드를 어떻게 변경을 했는지를 추적하여 문제 발생시 원인 분석을 할 수 있어야 한다.

1. Branch

먼저 소스코드 관리에 대한 개념 이해를 위해서는 Branch의 개념 이해가 필요하다. 소프트웨어 개발을 할 때, 하나의 저장소에 소스코드를 저장해서 개발하고, 개발자간의 공유를 통해 협업을 진행한다.

개발을 진행하다가 특정 목적에 의한 별도의 작업, 예를 들어 영문으로 된 버전을 중국어나 한국어로 제공하는 작업을 위해서는 기존 개발 소스코드의 복사본을 만들어서, 중국어 개발용으로 하나 사용하고, 한국어 개발용으로 하나 사용 한다. 이렇게 새롭게 만들어진 소스코드의 복사본 (중국어버전, 한국어 버전)을 브랜치라고 하고, 원래 소프트웨어를 개발하던 소스코드 저장소를 메인 브랜치(또는 trunk version, head version)이라고 한다.

이렇게 코드의 원본에서 용도와 목적에 따라서 새로운 복사본을 만들어 나가기 때문에, 메인 코드에서 복사본을 나뭇가지(Branch)라고 하고, 이 모양이 마치 나무와 같은 구조가 되기 때문에, Source Code Tree또는 Branch Tree라고 한다.

2. Check Out / Commit & merge

소스코드 저장소를 만들었으면 해야 할 일이 코드를 저장하거나 내려 받는 일이다. 이 중 코드를 내려 받는 행위를 Check Out이라고 하고, 작성된 코드를 저장소에 업로드 하는 행위를 Commit이라고 한다..

VCS의 종류에 따라서 Check Out된 코드의 경우, 다른 사람이 편집을 할 수 없고 read만 할 수 있도록 lock을 걸 수 도 있다.(VCS 종류에 따라 다르다). 이 경우 다른 사람이 내가 편집하고 있는 코드를 편집할 수 없기 때문에, 코드 작성에는 문제가 없지만 반대로 다른 개발자는 lock이 풀리기 까지 기다려야 하기 때문에, 협업에 문제가 발생할 수 있다. 그래서 lock을 걸지 않고 동시에 2인 이상이 코드를 다운받아 편집할 수 있는 방법을 제공하는데, 이 경우 코드를 Commit을 할 때, 한 파일을 다운 받아서 각자 편집을 하고 저장을 시도할 경우, 코드상 Conflict(충돌)이 발생한다.

즉 개발자 A,B가 소스 코드 버전 1을 Check Out 받았다고 할 때, 개발자 A가 소스코드 2번 라인에 “printf(’hello world’)“ 라는 코드를 추가하고 Commit을 해서 코드 버전이 2가 되었고 , B도 역시 2번 라인에 “printf(’hello developer’)“ 라는 코드를 추가하고 Commit을 하려고 하면, A와 B가 편집한 내용이 충돌이 생긴다. 둘다 1번 버전의 코드를 가지고 수정을 하였으나, 공교롭게 개발자 A,B 가 같은 파일을 수정하였기 때문에 VCS 입장에서는 B 개발자가 수정한 내용을 반영하게 되면 A개발자가 반영한 내용이 없어져 버리기 때문에 문제가 될 수 있다. 이런 경우 VCS에서 B가 Commit하려는 코드가 현재 버전인 2 버전에 의해서 편집 된것이 아니라는 것 을 알려주고, 개발자 B에게 선택을 하도록 하는데, 2 버전 코드와 다른 내용을 병합 (merge)하게 하거나, 3버전의 코드를 Overwrite하여 2버전의 코드 변경 내용을 무시할 수 있도록 한다.

대부분의 경우에는 당연히 다른 개발자의 변경 내용을 무시할 수 없기 때문에, merge를 선택하게 되고, merge는 개발자B가 2번 버전과의 로직의 차이를 일일이 확인하면서 수동으로 merge를 하도록 한다.

3. Update

Update의 개념은 개발자가 소스 코드를 Check Out 받아 놨을 때, 현재 작업 버전이 오래되거나, 또는 다른 개발자가 Commit을 먼저 해서, VCS의 소스코드 버전이 올라갔을 때, VCS와 Local의 코드를 동기화시키기 위해서 사용한다.

위에서 설명한 Commit시 발생하는 Conflict(충돌) 시나리오와 마찬가지로, 내가 변경한 코드와 다른 개발자에 의해서 변경된 코드가 Conflict이 발생할 수 있으며, 앞에서 설명한 방법과 같은 방법으로 수동으로 코드의 변경 부분(차이 나는 부분)을 병합하도록 한다.

4. Tagging

다음으로 Tagging이라는 개념을 알아보자, Tagging이란, 코드를 개발 중에, 특정 시점의 이미지에 표시를 해 놓는 것을 의미한다. 예를 들어, 매일 소스코드에 대한Tagging을 달아놓으면, 개발 중에 문제가 생겼을 경우, 특정 날짜의 소스 코드로 다시 돌아갈 수 가 있다.

Tagging은 주로, Build 시마다 하는 경우가 많다(이를 빌드 태그라고 함). 이는 통상 적으로 빌드시 에러가 날 경우 다른 개발자들이 빌드 에러로 인하여 개발을 못하는 경우가 생길 수 있기 때문이다, 이런 경우에는 이전 Build시 태그 해놓은 버전으로 소스코드를 돌려서 다시 개발을 수행하고, 문제가 해결 되면, 새로운 코드를 다시Commit 하는 방식으로 개발을 진행한다.

5. Release Branch

다음으로 Branch중 Release Branch에 대한 설명이다. 패키지 소프트웨어 개발 프로세스를 생각해보자, 예를 들어, 서버 제품을 개발하여, 출시를 했다고 하자. 현재 개발 중인 메인 Branch에서 해당 시점에 릴리즈를 했다. 릴리즈 한 서버의 버전은 6.1이다. 메인 branch 로는 계속해서 신제품 개발을 이어나가고 7.0 개발을 진행중이다. 이때, 6.1을 사용하던 고객에게서 버그 수정 요청이 온다면 어떻게 해야 할까?

6.1 코드에 일부 코드만 수정하여 패치를 발급하면 되지만, 메인 branch의 경우 이미 7.0 버전 개발을 위해서 코드 개발이 많이 진행되었기 때문에 6.1 에 대한 코드는 바뀌어서 찾을 수 가 없고 7.0 버전 역시 개발이 완료되지 않아서 수정이 불가능하다. 이런 상황을 방지하기 위해서 패키지 형태의 소프트웨어 개발은 각 릴리즈 마다 release branch를 발급하는 것이 좋다. 우리가 사용하는 Windows도, Windows XP, 7 등으로 release를 하지만 Service Pack이나 필수 패치 등을 발급하는 것은 각 release 별 branch를 발급한 뒤에, branch 별로 패치나 service pack을 배포하는 것과 같은 원리이다.

이런 release branch는 웹서비스 (face book 등)의 경우, 하나의 코드가 하나의 서비스에만 배포가 되기 때문에, 별도의 release branch를 유지하는 것 보다는 release tagging을 한 후에, 이슈가 있을 경우 이슈를 fix한 최신 버전을 다음release 때 배포 하는 방식 등을 주로 사용한다, 이에 웹서비스의 경우 release branch의 효용성이 패키지 소프트웨어에 비해서 떨어진다.

1. QA branch

다음으로 사용할 수 있는 branch중 하나는 QA branch 이다. production 으로(상품화 하기 위해) 개발팀에서 source code를 freezing 하고 QA 팀에 해당 제품을 넘겨서 테스트를 의뢰할 경우, QA팀에서 계속해서 버그 수정 요청을 해오게 된다. 이 때, 버그를 main branch에서 계속해서 수정을 하게 되면, main 개발에 많은 수정이 가해지기 때문에 개발이 어려워질 수 있다.(merge도 많이 발생하고, 번거로움) 그리고, main branch에서는 개발이 계속 진행되기 때문에, feature 변경이나 기타 수정이 있을 경우 QA에 의해서 reporting된 버그가 제대로 재현이 되지 않는 경우가 있을 수 있다. 그래서 QA에 넘기기 전에 QA branch를 따고, 버그에 대한 fix를 이 QA branch에서 수행 및 반영 한다.

QA가 모두 완료되고 나면, 이 QA branch에 있는 변경 내용을 다시 main branch로 merge하여, bug 수정 내용을 반영하도록 한다.

2. 소스 코드 브랜치 관리 전략

앞서 살펴본 바와 같이 사실 branch는 main code(branch)의 복사본이다. 용도에 따라서 얼마든지 만들 수 있고, 필요에 따라 merge를 할 수 도 있다. 그러나 패키지 또는 웹서비스와 같은 소프트웨어의 종류와, 팀의 크기와 구조, 릴리즈 정책 등에 따라서 어떠한 branch를 언제 사용할지가 다르기 때문에, 개발하는 소프트웨어의 형상에 따라서 알맞은 branch 전략을 결정해야 한다.[1)]

제 2 절 오픈소스 소프트웨어 브랜치 관리

1. 일반적인 소프트웨어 Branch 관리 전략

분산형 저장소인 git 사용 이전에 많이 유행하였던, Subversion(이하 SVN) 사용시 많이 사용하는 Branch 구조이다. SVN에서는 크게 세 가지 Branch를 메인으로 하여 코드를 저장하는 구조를 사용한다.

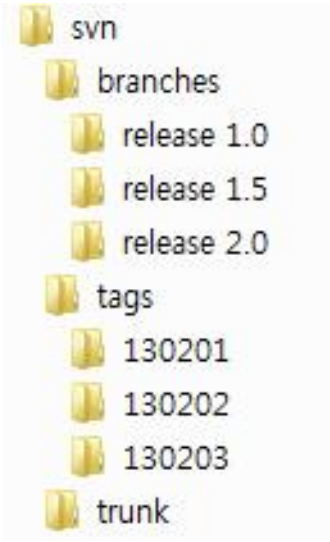

그림 1 SVN 의 Branch 구조

1) 출처 : http://bcho.tistory.com/773

첫 번째 Branch는 trunk 또는 head 라고도 하는데, 이 Branch는 현재 개발이 진행 중인 코드를 저장한다. 이 Branch는 개발 진행 중에, 빌드나 또는 특정 날짜를 기준으로 tag를 따 놓는데, 향후 빌드 실패나, 특정 날짜의 개발 내용으로 돌아가기 위해서 사용하며 이는 tags라는 Branch 아래 들어간다. tag명은 날짜를 사용할 수도 있으며, 빌드 시스템의 빌드 넘버를 사용하기도 한다.

마지막으로 Release Branch이다. Branches는 Release할 때마다, 그 때의 소스코드 형상을 저장해 놓는 구조이다. 이 Branch 전략은 단일팀에서, 서비스나 제품을 Release 주기를 통해서 Release 하는 구조에 적절하며, 버그 수정이나 기능 개선은 다음 버전(Release)에 포함 시키는 형태일 경우 적절하다(별도의 패치를 위한 Branch를 관리하는 전략이 없기 때문이다).

2. 분산 형상 관리 시스템 Git

그러면 최근에 많이 사용되고 있는 VCS 시스템인 Git에 대해서 알아보도록 하자. Git은 기본적으로 분산 소스코드 관리 시스템이다. 모든 개발자가 중앙 저장소에 붙어서 작업을 하는 것이 아니기 때문에 소스 코드가 여러 개의 서버에 다른 복제본으로 존재할 수 있다. 먼저 이 분산 저장소의 개념부터 이해 해보도록 하자

가. 중앙 집중형 저장소 (Centralized Version Control System)

중앙 집중형 저장소는, 코드가 저장 서버 단 한군데만 저장된다. 개발자가, 코드를 받아서 수정하고 저장하면, 그 내용이 바로 중앙 저장소에 반영된다. 즉 서버에는 항상 마스터 버전(최신 버전)의 소스코드가 저장되어 있다.

서버가 다운 되거나, 네트워크에 접속할 수 없다면 당연히 코드를 commit하거나 최신 코드를 내려받을 수 없다.

나. 분산형 저장소 (Distributed Version Control System)

분산 저장소는 말 그대로, 소스코드가 하나의 중앙 서버가 아니라, 여러 개의 서버나 여러 개의 개발자 PC에 저장될 수 있으며, 각각이 소스 코드 저장소 (source repository)가 된다. 각 저장소에 저장되는 소스 코드는 같은 버전의 코드가 아니라 제각기 다른 Branch 코드가 저장된다. 즉 서버 A에는 branch A, B, C 버전이, 서버 B에는 branch A, C, D버전과 같이 다른 Branch 버전을 저장할 수 있다. 즉 각 저장소에 Branch 버전이 모두 틀리고, 소스 코드를 access해서 가지고 오는 장소도 모두 다르기 때문에, 시스템 자체에서는 마스터 버전 (최신 버전이 항상 어느 곳에 저장되어 있는가)의 개념이 없다.

예를 들어보자. 개발자 Terry는 Server A에서 코드를 내려 받아서, 데이터베이스 관련 모듈을 개발하고 있고, 개발자 Michelle은 Server B에서 UI 관련 모듈을 개발하고 있다. 각자는 개발을 진행하면서, 수시로 각자의 Server A와 Server B에 Commit을 하고 있다고 가정하자. Server A는 전체 시스템에서 데이터베이스 모듈 부분은 가장 최신일 것이고, Server B에는 UI 모듈의 가장 최신 버전의 코드가 들어가 있을 것이다. 대신 각 모듈의 개발이 끝나면, Server A와 Server B의 코드를 Merge하여, 개발 내용을 병합할 수 있다. 즉 전체 시스템의 최신 소스 코드가 명시적으로 어느 한곳에 저장되어 있지 않는 구조이다.

이러한 형태로, 코드의 여러 버전을 여러 저장소에 분산해서 저장할 수 있기 때문에, 몇 가지 장점이 있을 수 있는데, 팀 단위나 기능 단위로 저장소를 분리해서 개발하거나 Release 버전 단위로 저장소를 분리해서 개발할 수 있는 등 소스코드 버전 관리에 많은 유연성을 가지고 있다.

또한 중앙 저장소의 개념이 없기 때문에, 특정 VCS 시스템이 장애가 나더라도, 내가 사용하고 있는 VCS만 문제가 없다면 개발을 계속할 수 있고, 앞에서 언급한 바와 같이 개발자의 Local PC에 VCS를 설치하여 네트워크 연결이 없는 상태에도 개발을 지속할 수 있다.

그리고 소스코드가 중앙 서버만이 아니라 여러 서버와 PC에 분산 되서 저장되기 때문에 서버 장애로 저장소가 손상된다고 해도, 다른 서버나 다른 PC에서 소스코드와 History들을 모두 저장하고 있기 때문에, 중앙 서버

방식에 비해서 복구가 쉽다.

제 3 절 커뮤니티 중심의 오픈소스 소프트웨어 개발방법론

소프트웨어 개발방법론이란 소프트웨어 개발을 위한 절차와 각각의 절차에서 수행해야하는 일의 내용, 그리고 수행하는 일에 따른 결과물에 대한 내용을 모두 합한 것을 의미한다. 따라서 오픈소스 소프트웨어의 개발방법론을 위해서 오픈소스 소프트웨어를 개발하는 절차와 해당 절차에서 하는 일의 내용, 그리고 결과물에 대해서 살펴보기로 한다.

1. 오픈소스 소프트웨어 개발 절차

외부에 제공되는 오픈소스 소프트웨어 개발을 위한 경우 필요한 거버넌스 체계와 활동 요소를 이전 장의 내용에서 제시하였다. 이러한 거버넌스를 조직에 적합하게 준비한 후에는, 실제 소프트웨어를 개발하기 위한 Branch 전략을 수립해야 한다.

예전에는 Subversion, CVS 등의 SCM을 주로 사용하였으나, 다른 SCM에 비해 쉽게 분기가 가능한 Branch 모델을 가지고 있는 Git이 출시된 이후 대부분의 오픈소스 소프트웨어 프로젝트는 Git을 SCM으로 사용하는 추세이다.

Git을 사용하는 경우, 오픈소스 소프트웨어의 소스코드를 개발하기 위한 Branch 전략은 조직의 상황에 따라 다음과 같은 다양한 구조를 사용할 수 있다.

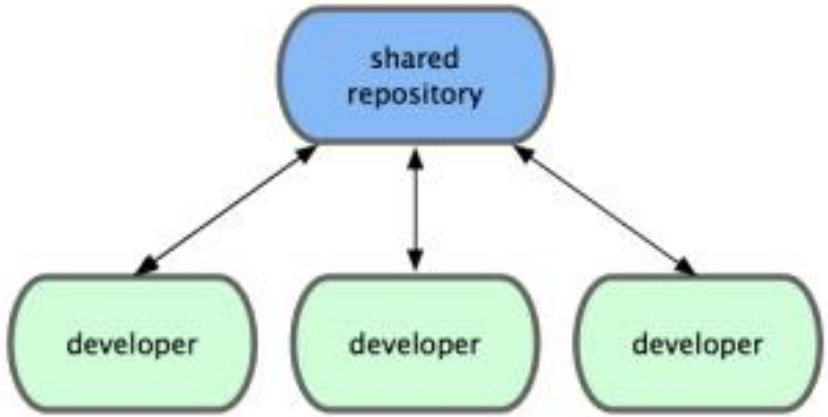

그림 2 중앙 집중식 Workflow

가. 중앙집중식 Workflow

- 예전의 Subversion에 익숙한 사용자들의 방식
- 예로 2명이 작업하고, 한 명이 서버로 Update코드 Push하면 다른 한 명이 서버에서 Update코드를 Pull하여 Merge 수행
- 모든 사용자에게 서버에 push 권한을 부여해야 함
- 그러나 중앙 서버 저장소로A코드가 다른 이에 의해 Push 되었고, 다른 사람이 A코드를 수정하여 Push해도 Git은 이것을 방지해 준다. 즉, 다시 로컬로 Fetch 또는 Pull 하지 않으면 중앙 서버로 Push 할 수 없게 막아준다.

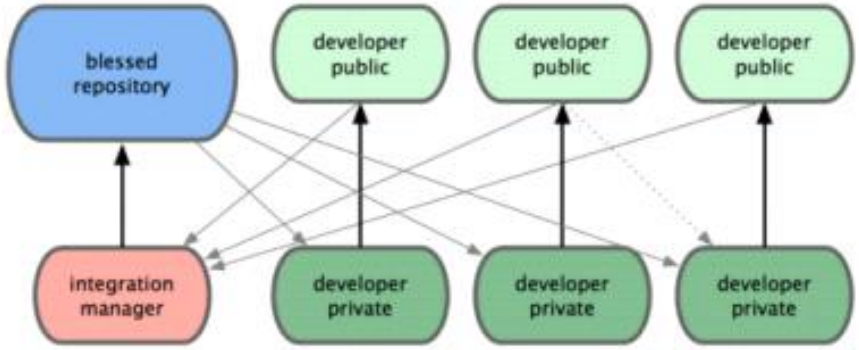

그림 3 통합관리 Workflow

나. 통합관리(Integration-Manager) Workflow

- 여러 Remote저장소를 두고 한 개는 R/W 가능, 다른 한 개는 R만 가능하게 설정 가능하다.
- 이것은 GitHub의 운영방식이다. Pull Request 수행하면 통합관리자가 이것을 보고 기여자의 저장소에서 원본 저장소로 Pull하여 Merge 하게 된다.

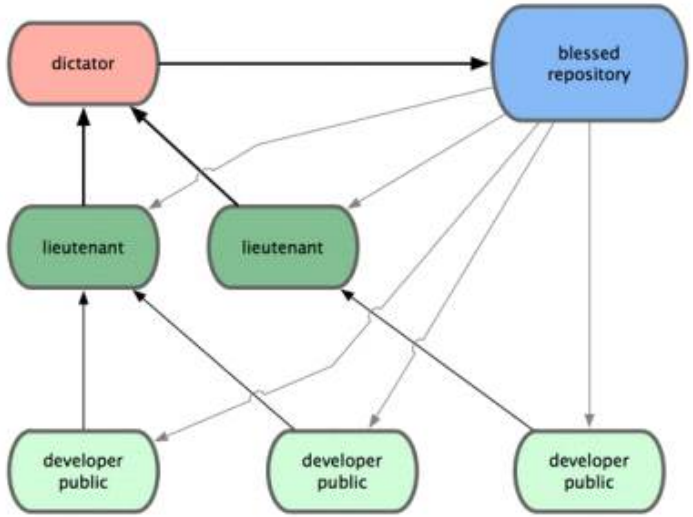

그림 4 독재자 보조관 Workflow

2. 오픈소스 소프트웨어 개발 단계별 내용

Github에서는 Develop Branch를 두는 대신 Pull Request라는 과정을 두어 리뷰를 거친 후, 안전한 코드만 배포를 하게 한다. 또한 Chatroom/Hubot을 통해 Merge, Pull Request를 IRC상에서 수행함으로써 구성원들이 실시간으로 과정을 지켜볼 수 있고 장애가 발생했을 때 빠른 대응이 가능하다. 하지만 일반적인 조직에서는 이러한 공통의 소통 창구가 없기 때문에 기민한 대응이 어렵다.

따라서, 안전한 오픈소스 소프트웨어 소스코드의 관리를 위해서는 다음과 같은 소스코드 브랜칭 모델을 고려하는 것이 필요하다.[2] 이 모델은 Git-flow를 개발한 Vincent Driessen 이 오픈소스 소프트웨어를 Git 으로 관리하는 과정에서 사용한 브랜칭 모델이며, 여러 팀이 분산된 구조의 조직이 개발을 하는 과정에서 소스코드에 대한 각각의 기능 개발과 통합이 체계적으로 이루어지기 위한 좋은 모델을 제시하고 있다.

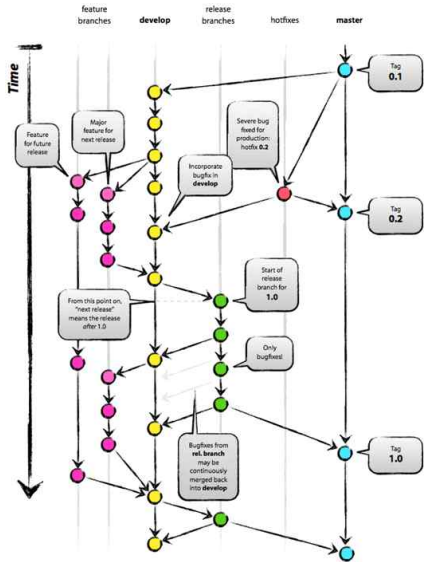

그림 5 A Successful Git branching model

가. 주요 브랜치

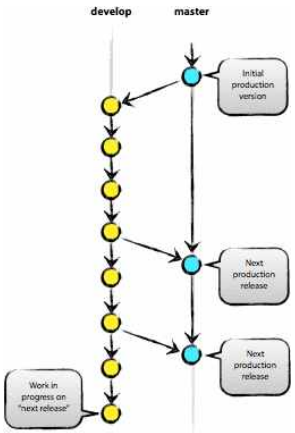

그림 6 master / develop

이 개발 모델은 전혀 새롭지 않다. 핵심은 기존에 있던 개념들이다. 주요 브랜치 두 개는 중앙 저장소에 영원히 유지한다:

- master
- develop

Git 사용자라면 누구나 익숙한 master 브랜치와 develop 브랜치를 병행으로 유지한다. 먼저 배포했거나 곧 배포할(production-ready) 코드는 origin/master에 두고 관리한다. 그리고 다음에 배포할 것을 개발하는 코드는 origin/develop에 두고 관리한다. 혹자는 이 브랜치를 "통합 브랜치(integration branch)"라고 부르기도 하는데, 이 브랜치를 자동으로 매일 빌드하는데 사용한다.

develop branch의 코드가 안정되고 배포할 준비가 되면 곧 master로 merge하고 배포 버전으로 태그를 단다. 즉, 정의한 대로 master로 merge하는 것은 새 버전을 배포하는 것을 의미한다. 우리는 이것을 매우 엄격하게 지킬 것이다. 그래서 master 브랜치에 커밋될 때마다 Git hook 스크립트로 자동으로 빌드하고 말아서 운영 서버로 배포할 수 있다.

2) Vincent Driessen, http://nvie.com/posts/a-successful-git-branching-model/

나. 보조 브랜치

master와 develop 브랜치말고 다른 브랜치도 필요하다. 기능을 구현하고, 배포를 준비하고, 이미 배포한 제품이나 서비스의 버그를 빠르게 해결해야 한다. 이 모든 것을 동시에 진행해야 하기 때문에 다양한 브랜치가 필요하다. 사용할 브랜치의 종류는 다음과 같다.

- feature 브랜치
- release 브랜치
- hotfix 브랜치

각 브랜치마다 만든 목적이 있고 어떤 브랜치에서 갈라져 나왔는지, 어떤 브랜치에 merge할지에 따라 꼭 지켜야 하는 규칙도 있다. 이 분류는 어떻게 사용할지에 따라 나누었다. 하지만 기술적으로는 모두 같은 브랜치다. Git의 다른 브랜치와도 똑같다.

1). feature 브랜치

- 갈라져 나온 브랜치: develop
- 다시 merge할 브랜치: develop
- 브랜치 이름 규칙: master, develop, release-*, hotfix-*를 제외한 것

feature 브랜치(토픽 브랜치라고도 부른다)는 다음, 아니면 다다음, 어쨌든 조만간에 배포할 기능을 개발하는 브랜치다. 기능을 개발하기 시작할 때에는 사실 언제 배포할 수 있는지 알 수 없다. feature 브랜치는 그 기능을 다 완성할 때까지 유지하고 다 완성되면 develop 브랜치로 merge한다. 다음 배포에 확실히 넣을 거라고 판단될 때 merge하고 결과가 실망스러우면 아예 버린다.

feature 브랜치는 보통 개발자 저장소에만 있는 브랜치고 origin에는 push하지 않는다.

가). feature 브랜치 만들기

feature 브랜치를 develop 브랜치에서(base) 새로 만든다.

```
$ git checkout -b myfeature develop
Switched to a new branch "myfeature"
```

어떤 기능이 다 완성돼 다음 배포에 넣기로 했다면 develop 브랜치에 merge한다.

```
$ git checkout develop
Switched to branch 'develop'
$ git merge --no-ff myfeature
Updating ea1b82a..05e9557
(Summary of changes)
$ git branch -d myfeature
Deleted branch myfeature (was 05e9557).
$ git push origin develop
```

'--no-ff' 옵션을 주면 항상 merge 커밋을 만들어 merge한다. fast-forward로 merge할 수 있어도 fast-forward하지 않는다. 그러면 feature 브랜치에 추가된 모든 커밋이 merge되고 feature 브랜치에서 merge했다는 기록이 커밋 히스토리에 남는다. 그 둘을 비교해보자.

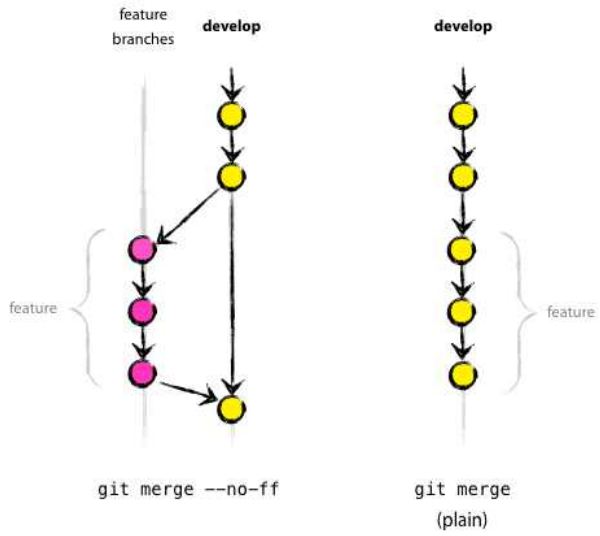

그림 7 커밋 히스토리 비교

후자처럼 fast-forward merge하면 나중에 커밋 히스토리를 다시 확인할 때 어떤 커밋이 어떤 기능(feature)을 구현한 것인지 확인하기 어렵다. 그래서 히스토리에 있는 커밋 메시지를 하나하나 눈으로 찾아야 한다. 추가한 feature를 되돌려야(revert) 할 때 feature와 관련된 모든 커밋을 되돌려야 하는데 merge 커밋이 없으면 욕이 절로 나올 것이다. --no-ff 옵션을 주고 merge했다면 되돌리기 쉽다.

2). release 브랜치

- 갈라져 나온 브랜치: develop
- 다시 merge할 브랜치: develop, master
- 브랜치 이름 규칙: release-*

release 브랜치는 제품 배포를 준비하는 브랜치이다. 이 브랜치가 화룡이 승천할 수 있도록 점정하는 곳이다. 배포하는 데 필요한 버전 넘버, 빌드 일정 등의 메타데이터를 준비하고 사소한 버그도 잡는다. 이런 일을 release 브랜치에서 함으로써 develop 브랜치는 다음에 배포할 때 추가할 기능에 집중할 수 있다.

develop 브랜치가 배포할 수 있는 상태에 다다랐을 때 release 브랜치를 만드는 것이 중요하다. 이때, 배포해야 하는 기능이 모두 develop 브랜치에 merge돼 있어야 하고 이번에 배포하지 않을 기능은 release 브랜치를 만들 때까지 기다려야 한다.

release 브랜치를 만든다는 것은 이제 배포 버전을 부여하겠다는 것을 의미한다. 그때까지 develop 브랜치가 다음 배포가 어떤 모습일지 보여주지만, 아직 깨끗하게 정리된 상태가 아니다. 최종적으로 release 브랜치를 만들어 '0.1', '0.3' 같은 버전 넘버 붙을 때까지는 "진짜" 배포라고 할 수 없다. 그러니까 release 브랜치를 만들기로 하는 것이 버전 넘버를 새로 부여하기로 하는 것을 의미한다. 이것은 규칙이다.

가). release 브랜치 만들기

release 브랜치는 develop 브랜치에서 만든다. 예를 들어 배포할 수 있을 정도로 develop 브랜치가 준비돼 이제 곧 새 버전을 배포할 것이라고 하자. 그리고 현재 배포된 버전이 '1.1.5'이고 새 버전은 '1.1.6'이나 '2.0'이 아니라 '1.2' 버전으로 배포하기로 했다. 그럼 다음과 같은 이름으로 release 브랜치를 만든다.

```
$ git checkout -b release-1.2 develop
Switched to a new branch "release-1.2"
$ ./bump-version.sh 1.2
Files modified successfully, version bumped to 1.2.
$ git commit -a -m "Bumped version number to 1.2"
[release-1.2 74d9424] Bumped version number to 1.2
1 files changed, 1 insertions(+), 1 deletions(-)
```

브랜치를 새로 만들고 버전 넘버를 생성했다. bump-version.sh는 버전 넘버가 들어 있는 파일을 전부 수정하는 가상의 쉘 스크립트다. 손으로 직접 수정해도 된다. 중요한 것은 이 시점에 파일을 수정한다는 점이다. 그리고 수정한 파일을 커밋한다.

새로 만든 release 브랜치는 잘 말아서 진짜로 배포할 때까지 유지한다. 그동안 발견한 버그는 develop 브랜치가 아니라 이 브랜치에서 해결하고 새 기능은 이 브랜치에 추가하지 않는다. 그런 기능은 develop 브랜치에 merge하고 다음 배포로 미뤄야 한다.

나). release 브랜치 마치기

release 브랜치가 진짜 배포할 상태가 되면 배포한다. master 브랜치에 있는 것을 배포하는 것으로 정의했으므로 먼저 release 브랜치를 master로 merge한다. 그리고 나중에 이 버전을 찾기 쉽도록 태그를 만들어 지금 master가 가리키는 커밋을 가리키게 한다. 그리고 release 브랜치를 develop 브랜치에 merge하고 다음에 배포할 때 release 브랜치에서 해결한 버그가 적용되도록 한다.

먼저 처음 두 단계, master에 merge하고 tag를 단다.

```
$ git checkout master
Switched to branch 'master'
$ git merge --no-ff release-1.2
Merge made by recursive.
(Summary of changes)
$ git tag -a 1.2
```

release 브랜치로 해야 할 일을 끝냈고 미래를 위해 tag도 달았다. tag를 달 때 -s나 -u 옵션을 주고 암호화 알고리즘을 이용해서 서명할 수도 있다. 그리고 develop 브랜치에 다시 merge해서 release 브랜치에서 수정한 것이 앞으로도 계속 유지되게 한다:

```
$ git checkout develop
Switched to branch 'develop'
$ git merge --no-ff release-1.2
Merge made by recursive.
(Summary of changes)
```

버전 넘버를 수정했기 때문에 여기서 merge할 때에는 충돌이 날 확률이 높다. 충돌이 나면 수정해서 커밋한다.

이제 진짜로 배포했기 때문에 release 브랜치는 더는 필요 없다. 삭제한다:

```
$ git branch -d release-1.2

Deleted branch release-1.2 (was ff452fe).
```

3). - hotfix 브랜치

- 갈라져 나온 브랜치: master
- 다시 merge할 브랜치: develop, master
- 브랜치 이름 규칙: hotfix-*

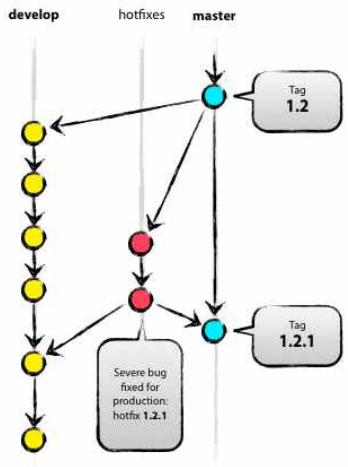

그림 8 hotfix 브랜치 생성 구조

미리 계획을 세워두지 않는다는 점만 빼면 hotfix 브랜치도 새로운 배포를 준비하는 것이기 때문에 release 브랜치와 비슷하다. 이것은 이미 배포한 운영 버전에 생긴 문제를 해결하기 위해 만든다. 운영 버전에 생긴 치명적인 버그는 즉시 해결해야하기 때문에 문제가 생기면 master 브랜치에 만들어 둔 tag로부터 hotfix 브랜치를 만든다. 그리고 버그를 잡는 사람이 일하는 동안에도 다른 사람들은 develop 브랜치에서 하던 일을 계속할 수 있다.

가). hotfix 브랜치 만들기

hotfix 브랜치는 master 브랜치에서 만든다. 예를 들어 현재 운영 버전이 1.2이고 심각한 버그가 발견됐다. develop 브랜치는 아직 불안정하기 때문에 hotfix 브랜치를 만들고 거기서 버그를 잡는다:

```
$ git checkout -b hotfix-1.2.1 master

Switched to a new branch "hotfix-1.2.1"

$ ./bump-version.sh 1.2.1

Files modified successfully, version bumped to 1.2.1.

$ git commit -a -m "Bumped version number to 1.2.1"

[hotfix-1.2.1 41e61bb] Bumped version number to 1.2.1

1 files changed, 1 insertions(+), 1 deletions(-)
```

브랜치를 만들고 버전 넘버를 바꾸는 것을 잊으면 안 된다. 버그를 해결하고 나서 커밋한다. 한, 두 개의 커밋으로 해결한다:

```
$ git commit -m "Fixed severe production problem"

[hotfix-1.2.1 abbe5d6] Fixed severe production problem

5 files changed, 32 insertions(+), 17 deletions(-)
```

나). hotfix 브랜치 마치기

버그를 잡았으면 다시 master에 merge하고 다시 develop 브랜치에도 merge해야 한다. 그래야 다음에 배포할 때도 포함된다. release 브랜치를 마치는 방법과 같다.

먼저 master에 merge하고 tag를 단다.

```
$ git checkout master

Switched to branch 'master'

$ git merge --no-ff hotfix-1.2.1

Merge made by recursive.

(Summary of changes)

$ git tag -a 1.2.1
```

이때에도 -s나 -u 옵션으로 tag에 서명할 수 있다.

그리고 develop에도 merge한다:

```
$ git checkout develop

Switched to branch 'develop'

$ git merge --no-ff hotfix-1.2.1

erge made by recursive.

(Summary of changes)
```

만약 아직 release 브랜치가 삭제되지 않고 있다면 develop 브랜치가 아니라 release 브랜치에 merge한다. release 브랜치가 완료되면 결국 develop 브랜치에 merge될 것이다. 그런데 develop 브랜치도 즉시 해결해야 하면 release 브랜치가 끝날 때까지 기다리지 말고 develop 브랜치에 즉시 merge한다. 문제가 생기지 않도록 조심스럽게 merge한다.

이제 이 임시 브랜치를 삭제한다:

```
$ git branch -d hotfix-1.2.1

Deleted branch hotfix-1.2.1 (was abbe5d6).
```

다. 결론

이 모델은 전혀 새로운 게 아니다. 이 모델이 제시하는 그림은 프로젝트를 진행 할 때 정말 유용하다. 팀원 모두 머릿속에 같은 그림을 그리고 일할 수 있다. 이 모델은 팀이 브랜치와 배포 프로세스를 이해하고 공유 개발할 수 있도록 도와준다.