团队使用Git和Git-Flow手记

May 11, 2015

去年10月份，在我们被产品节奏逼到墙角无路可走的时候，我们在几乎没有准备的情况下，在团队中引入了Git。目前时间已经过去半年，回顾这半年的时间，基本还是运作得比较顺利。当然过程中也少不了踩坑，因此记录一些心得。

Why Git?

如果用一句话来说的话，我们是冲“分支”而来的。背景如下：

团队的固定版本节奏为两周一个版本，一周半的时间开发，半周时间测试发布。如果使用软件工程中的概念来说的话，这是一个比较典型的瀑布式流程，即“需求->设计->开发->测试->发布”，然后周而复始，过程中几乎没有重叠。

伴随着瀑布式的流程，代码也只有一份，“开发->改bug->发布”，周而复始。

直到去年10月，公司做了一场声势浩大的营销活动，灾难开始了：一方面需要以并不确定的研发周期支持各种运营活动（运营时间不等人，必须快速写快速发），一方面是运营活动带来大量客户，与客户相关的流程也陷入频繁修改发布的过程。此外支持各种终端需求也集中爆发，需要web侧快速跟进上线。当然，还有一个东西，就是上面瀑布流程中两周发布一次的“主版本”。当这么多版本交叠在一起时，我们发现必须要引入分支来管理研发过程。于是果断切Git。

困扰

引入Git的过程并非一帆风顺，中间也有不少困扰。

概念和特性

首先是Git的概念繁多，而且很多命令并不直观。尤其是有一些和svn很像的命令，需要一段时间去理解。比如svn的 `commit` 会推送到服务器，而Git不会，这会导致解释Git的 `push` 命令并不是那么容易。

另外Git会在操作可能导致修改丢失时拒绝操作。

比如当前分支上有个文件A，当前状态为A1，我们将它做一点修改，并没提交，此时状态为A2。切换到另一分支的状态B1，如果A1和B1不一致，就需要覆盖当前文件，从A2切到B1，此时就会导致状态A1到A2的改动丢失，Git会拒绝操作。

类似的场景很多，只要Git发现有改动会丢失就会拒绝操作，而如果改动不会丢失，才允许操作。这种是否允许操作的不一致性也会困扰团队成员。解决的方案是引入stash操作，或者鼓励成员多提交。

分支思维

然后是从单线开发转入多线开发的思维转变。

在切Git之前，我们的代码都是只有一个主线的（其实有另外一份代码，但是是以Copy文件夹的形式存在，更多的意义在于“备份”）。而在切换Git之后，一方面概念繁多，一方面还要时刻去关注代码所处的分支状态。甚至会有很多成员在一开始很怀疑分支的有效性，总会担心自己辛苦写的代码一不小心切完就再也找不回来。这个问题也同样需要一段时间来适应。

日常操作

另外就是对于日常操作的困扰。

对比svn，Git的日常代码提交增加了add和push的过程，略显繁琐。但繁琐并不是很大的问题。真正的问题在于Git对版本和文件完整性的要求导致不允许对未提交的文件进行合并。直观的表现就是本机修改了文件无法直接与远程合并，必须要先提交，再合并远程，最后再推送。如果本机有部分文件无法提交的话，还需要增加stash相关的动作，整个流程就变为“add->commit->stash->pull(merge或rebase)->push->stash pop”，而svn的则是“update->commit”，光看看路径的长度就懂了。

这个问题并没有什么好的解决办法，只能是让成员不断练习、练习、实践、实践，然后习惯。

当然也有部分Git客户端会简化这个操作，比如Github客户端（Mac Windows）和SmartGit还有命令行工具Le Git都提供了一个操作叫 `sync` ，就是把上面列的这一长串流程放到一起了。

二进制文件

在从单线开发转到多线并行的开发后，一般情况下可以在各个分支独立处理自己的事情，最后由Git来进行合并。但是如果碰到二进制文件，事情就变得完全不一样，比较典型的案例就是雪碧图。

当多个分支都要修改雪碧图的时候，如果独立在各分支中修改，最后再合并，场景一定很壮观。因为Git没法处理二进制文件的合并。

对这个问题，短期的解决方案是，涉及到二进制文件改动时，在各个分支同步修改。

是的，一点都不优雅，所以长期的方案是，干掉二进制文件合并的场景。比如对于雪碧图来说，如果将图都拆开，在开发阶段不作合并，则代码合并也不会出问题。因为改动都是“增加了icon1.png”“删除了icon2.png”，只有两个分支同时增加或者修改同名文件才会出问题，如果不合并雪碧图，则几乎不存在这样的场景。因此这个问题较好的解决办法是将雪碧图放到构建阶段，由自动化打包工具来完成。

经验

使用Git-Flow

在没有分支管理经验的时候，全盘引入别人的成功经验是可取的。我们一开始也是全盘引入了Git-Flow，虽然并不是100%完美，但很大程度上避免了前期刚切入Git时的混乱期。

期间我们发现Git-Flow并没有对测试介绍做出指导（何时在哪个分支做测试），导致 我们唯一的测试服务器上经常出现版本混乱，于是我们尝试加入了tests分支，但事实证明这个分支并不能很好地与其它分支协作，于是作罢，仍然基本采用Git-Flow的流程来做。（回想起来，当时测试服务器上出的问题并不是Git-Flow带来的。测试应该是在release分支拉出后再做，而如果是单独测试特性，则直接在feature分支做。如果需要同时测多个东西，则需要多台测试服务器，于是后来我们也增加了很多测试服务器来解决这个问题。）

公司内也有其它团队使用Git，但是不采用Git-Flow，结果就会经常为拉分支和合并分支的事情而困扰。

有关release分支

在一开始看到Git-Flow的时候，我以为release分支是一个存活期非常短的分支，只是拉出来打打版本号，然后立马就消失了。但是后来逐渐认识到，release分支其实相当于对develop分支的一个冻结副本，release拉出来的时候就意味着上面的需求都已经冻结，在release上唯一可以继续做的改动只有这些需求的bug修复。而release一旦拉出后，develop上就可以继续执行新功能开发，这样新功能开发和版本测试发布可以并行，所以测试的介入的理想节点是release版本。

如果是瀑布式模型，测试和新功能开发不重合，则可以不需要release版本。目前我们团队内还没有使用到release分支。

有关develop分支

在理想情况下，一个版本开始的时候develop和master是完全一样的。此时开始在develop上做开发，相当于这是一个大版本，上面拉出来的各种feature分支都是在做这个大版本开发的一部分（它们最后也合回develop，和直接在develop上开发的效果是一样的）。而如果一个大版本正在开发，则时需要再来一个并行的大版本（比如碰到了长线需求），也即正在开发的独立大版本不止一个，则理论上需要并行的多个develop分支。这是Git-Flow的各种介绍文章中均没有提到的问题。

所以，当碰到有多个并行大版本的需求时，如果要准备开发第二个大版本的需求（也可以简化为“独立发布的需求”），则必须**不能**使用Git-Flow从develop拉出feature分支。此时应该使用hotfix分支（相当于另一个develop分支）。

有关hotfix

在看过上面一段后，应该能明白，hotfix的地位和develop其实是一样的，只是develop存在的时间更久一些（“大”版本嘛），还会拉出功能分支，接受功能分支合并。简单说，他们的区别就是develop更“重”一些，hotfix更轻量一些，本质上都是从master来，回master去。

有关master

在有紧急bug的时候，Git-Flow要求拉出一个hotfix分支来处理，而不能直接改动master。但从我们实践的经验来看，master上并不是不能做修改，有时候为简单起见，也可以在master上直接改，只是做完修改需要合并回develop，合并完之后和拉hotfix分支再完成等效。

一些习惯

rebase和merge

建议本机和远程分支（相同分支）同步必须用rebase，因为不用rebase就会使用merge，这样会导致（逻辑上的）同一分支在版本记录中变成分叉的多个分支（然后这些分支合并的消息全是 `merge xxx branch of http://remote.server/xxx.git` ），不利于追踪版本记录。

但是，从实践的经验来看，rebase在发生冲突时解决方案并不是那么友好。首先是此时的“my version”和“theirs version”并不和想象的一样。（详情可参考rebase文档，很可能“theirs”其实是自己的代码。）然后，在极端情况下rebase会导致代码丢失（目前原因未知）。

上面说的是本机和远程的相同分支使用rebase。如果是不同分支的合并则必须用merge，否则会导致历史记录不可读，因为再也找不到合并之前各个分支到底是在哪里。

另外还有一个需要注意的点，假设有这样的场景：现在有两个分支A和B，本机A的状态为A1，远程的比A1新，为A2，此时需要将B合并到A。按照上方所说，本机和远程的A分支同步使用rebase，B合并到A使用merge。

如果先将B merge到A，则本机的状态为A1，然后是B。此时再拉远程的A2，会导致B合并过来的记录丢失（在分支图中看不到合并的痕迹。）因此碰到这种情况，既需要和远程同步，又需要合并其它分支的情况，一定要先rebase再merge，否则会丢失merge记录。

commit和stash

如前文所述，svn的习惯（update->commit）行不通，因此需要先stash或者commit。鼓励团队成员多commit，即使没写完，也可以在下次提交时使用 `--amend` 将两次提交进行合并。

之所以鼓励commit而不是stash，是因为stash并不在版本记录中，理解起来并不那么容易，对比和合并也不是特别方便。

此外就是前文说的，Git sync是个好功能，但是从实践的情况来看，sync时并不一定会选择rebase，而merge会导致版本记录可读性变差。

有关tags

如果有需要从Git中提取代码的情况，则通过tags来操作是非常好的，因为可以很方便地追踪代码版本情况。如果不打tags，也可以使用提交的hash值，也可以追踪，只是人眼看起来没有那么直观了。而如果使

用分支名，技术上也可行，但是会导致追踪困难。（试想，你能很轻易地找到前天晚上master在哪里么？）

而如果有发布系统来记录每一次发布的情况，则可以不用tags，因为发布系统可以记录当时提交的hash值，追踪起来也很方便。

小结

坑都是要自己踩的，认真踩下来，结果还不错。谨以此文献给正被Git团队协作困扰的团队，希望能提供一些参考。

PREV NEXT