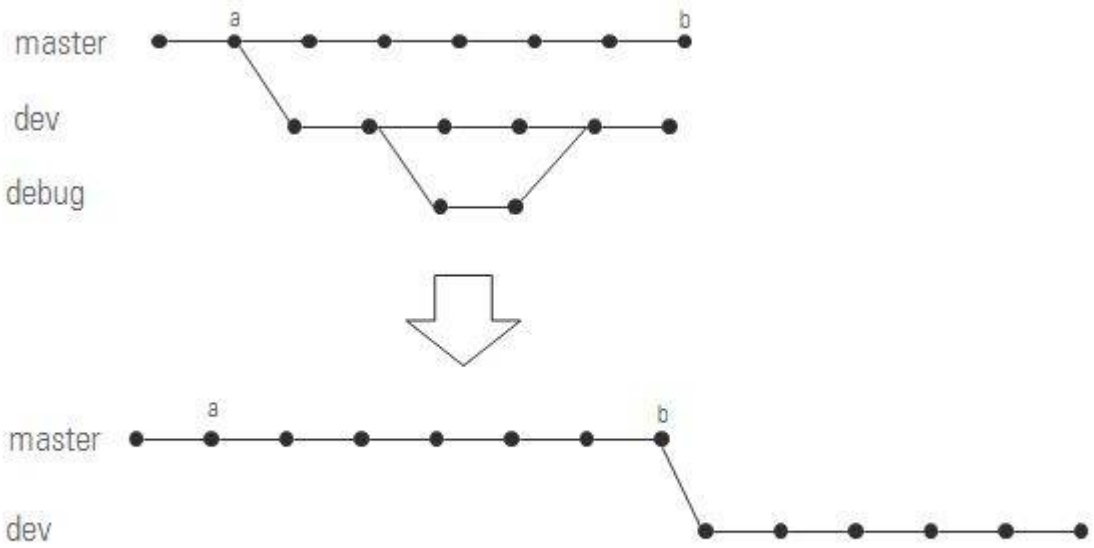
公司git管理中权限控制的比较严格，普通成员仅有pull和push的权限，而且commit的信息必须以jira的ticket号开头，没有merge的权限，一旦当你push的commit中含有两个分支（two parents）的信息，就会提示错误。

而在实际开发中，因为只有leader有在服务器上把其他分支merge进master的权限，所以leader每天都要review和merge来自不同开发者的不同分支的代码。这里面自然就会导致一些conflict的，最简单的情况就是当两个pull request发生的时候，第一个被merge，可能在merge第二个时，就会产生conflict，无法merge。这时leader一般会要求我们重新rebase一下代码，然后提交新的代码，以解决master merge第二个pull request的conflict。

目标

我们的目标是将自己正在开发的分支rebase到develop最新的commit上。从整个代码的提交关系，如图：



这时，我们再push dev branch，master就能merge它了。

rebase和merge的区别

从上面的图中其实可以看出，rebase和merge是两个完全不一样的操作。但是由于它们过程中都会出现conflict的情况，所以容易搞模糊。

rebase是将你当前的分支的起点（base）重新移到另一个节点。比如上图中，就是讲dev分支的起点从master分支的a移到b。实际上，rebase可以在任何节点之间移来移去，只要你能解决好问题。

merge则用debug分支来演示，debug分支是从dev分支分出来的，但是它的目的仅仅是为了解决bug，所以最后dev把debug修改好的代码merge进来。

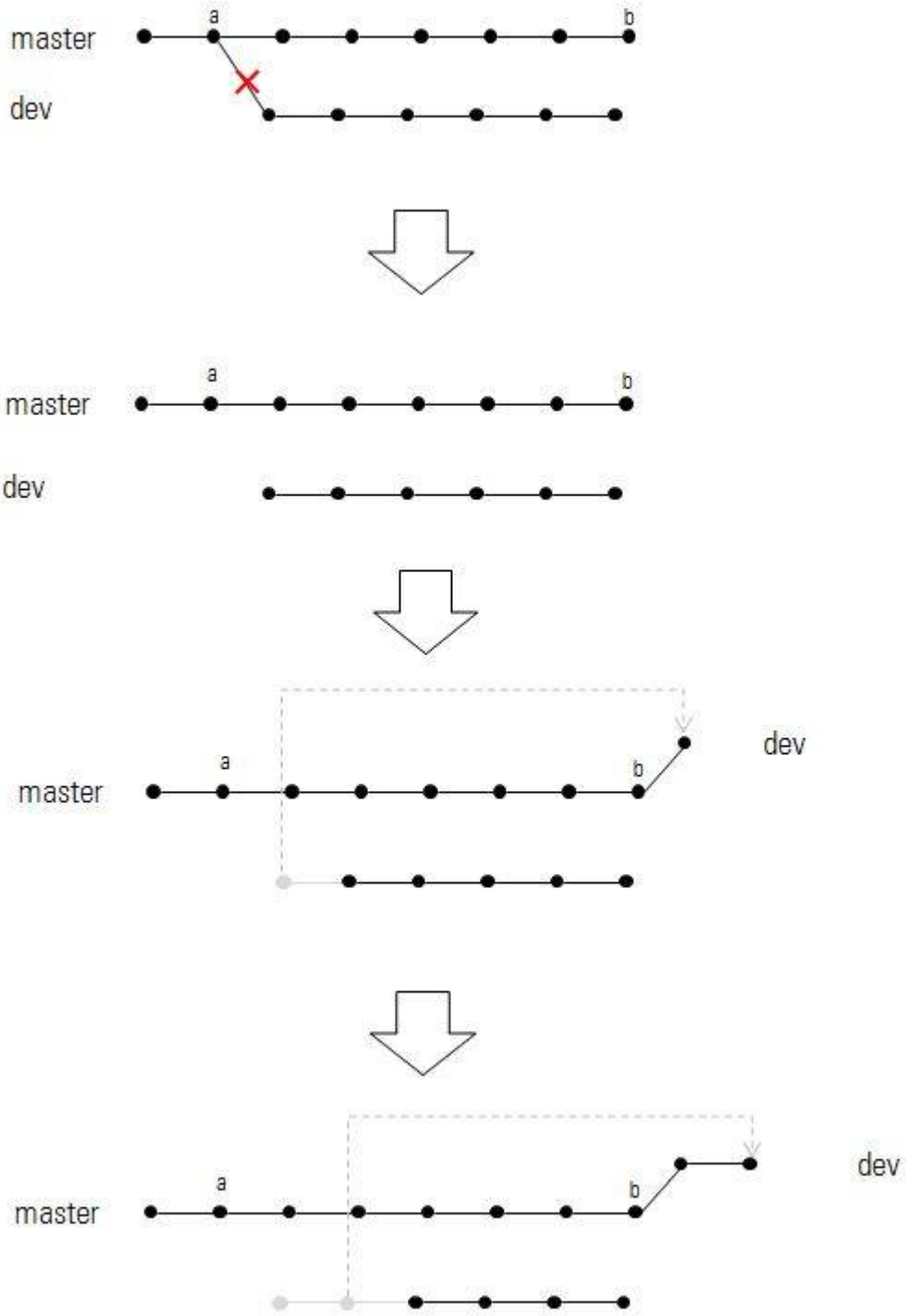
rebase和merge的区别如下：

1. merge是将另外一个分支merge进来，rebase是将自己rebase到另外一个分支的另一个节点。也就是说所站在的分支的视角不同，merge是一个向自己内部的过程，rebase是对外的一个过程。
2. 被merge的分支其实可以结束了，可以被删掉，下次需要的时候再重新创建。rebase的分支并不结束，一般还会进行下一步操作。
3. rebase过程中涉及到merge操作，所以，实际上所有的conflict都是merge产生的。但是这里比较难理解，所以要下文才能解释清楚。

其实rebase跟merge是完全两个不同的操作，只要狠狠抓住这点，理解上都会更进一步。

rebase的运动过程

从上面的图上看，感觉rebase挺简单的，就是把当前所在的分支的base改一下指针就好了。但是实际上完全不是，rebase是commit by commit的一个过程，跟我们想象的“改一下指针”完全不同，可以说rebase的运动是最笨的一种运动。下面来解释上面我们给出的图中rebase要实现的话都是怎样的一个过程。



上图示意了rebase的运动过程，但是别急，有地方需要阐述一下。上图的第一个、第二个状态说的是，当rebase发生的时候，首先实际上是把当前分支的所有commit先捡出来，然后按照one by one的顺序进行暂时放置。

第三个状态是说，rebase实际上是以rebase的对象分支，也就是master分支指定的commit节点（默认是最新节点）作为base，重新创建一个同名分支，也就是dev，这个时候，实际上老的dev分支已经跟整个分支体系脱节了，只属于索引结构中的暂存信息。新的dev分支其实相当于checkout master && checkout -b dev，但是rebase并不会停在这个步骤，新的dev分支跟master是一模一样的。接下来就是把老的dev分支上的commit一个一个转移到新的dev分支上。所以git从已经脱节的dev老分支中取出第一个commit，并再次在新的dev分支上执行commit的全套过程。

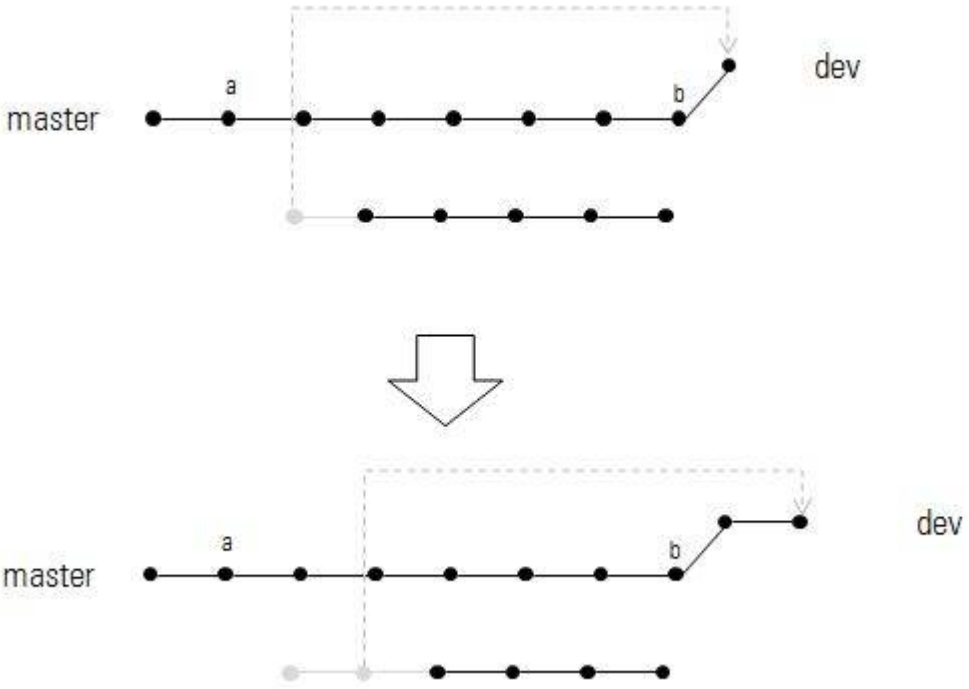
第四个状态和状态三种commit的过程是一样一样的，后面的所有commit都会按照这个方式一个一个复制到新的dev分支中。注意，这里的“复制”不是改一下指针这么简单，这里的复制其实是一个非常复杂的过程，git从老的暂存的dev分支里面取出一个commit之后，同时还要把代码的改动也提交进来，这就会在上一个commit的基础上（上一个commit是指rebase过程中commit到新的dev分支）再次进行commit。如果不发生冲突，那么实际上这个过程是一个merge的过程，就是站在新的dev分支上，merge老的dev分支上对应的那个commit的改动，同时还要把commit的信息也复制过来（而且这个commit信息也可以不复制，而是自己重写过，具体可以参看 这篇文章 ）。如此一个一个commit下去，直到把所有的老的dev分支上的commit转移到新的dev分支上。

当所有的commit复制完之后，新的dev分支其实是全新的一个分支，和老的dev只是代码改动和commit上相同，但就git的分支体系而言，是两个完全独立的，这个时候从资源上看，老的dev分支应该还是存在于git的索引中，直到整个rebase完成之后，git自动将老的dev分支的一切都清除。所以，**当rebase完成之后，你无法恢复到rebase之前的状态**，也就是第一张图片，你无法恢复到第一个状态。因此，rebase提供了abort功能，也就是rebase到一半的时候放弃，恢复原样。另外一个办法就是，rebase的时候先以要rebase的分支创建一个分支，这个分支跟rebase之前完全相同，如果rebase后后悔，可以切换到那个分支。

rebase过程中发生conflict

这是最恐怖的一件事，特别是当你的commit特别多的时候发生conflict，你会崩溃。首先，conflict是怎么发生的？其实是在上面提到的merge时发生的。

再回到上面那张图。



当复制commit的过程发生时，实际上也发生着merge。很明显，从a到b，master分支上也发生着变化，这个变化可能会导致merge原来的dev分支的代码时，产生冲突。比如在老的dev分支上改了的某一

段代码，在master分支上也发生了变化。这就是我文章一开头的时候遇到的问题，也就是前面merge了一个程序员的代码，现在merge你的代码时，conflict产生了。

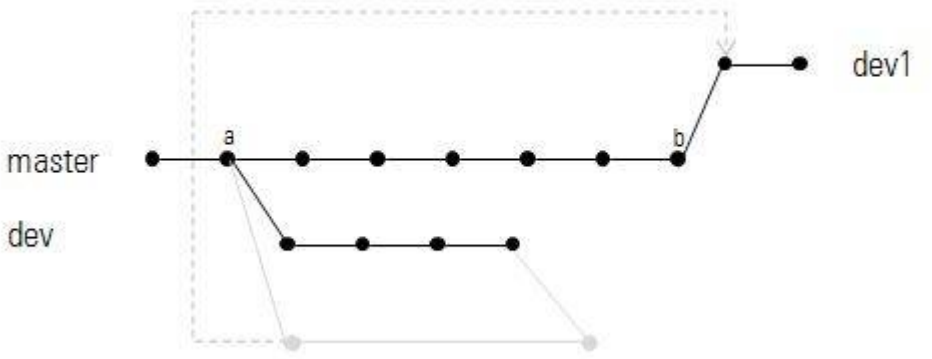
更恐怖的是，rebase每复制一个commit都会去merge一次，这是因为对于新的dev而言，它的base已经改变了，每一个commit其实是从原来的dev中将代码合并过来，所以很有可能新的dev中的代码和要合并的代码会产生冲突。

这也是为什么我们每一次git rebase --continue都有可能产生conflict的原因。

更有可能，你的commit中前一个加入了某串代码，后一个commit又把它删除了，那么你在rebase的时候如果发生conflict的话，你不得不先解决第一个commit的问题，然后解决第二个commit的问题，这种蛋疼的问题，都是无可救药的rebase造成的。所以，不在万不得已的时候，不要rebase啊！

因此，如果你要rebase的分支原来的base节点越老、commit次数越多，那么出现conflict的次数也就可能越多。反过来说，如果你要rebase的分支只有一个commit，那么即使出现conflict，也只需要你解决一次就可以完成rebase。

让我们来看下曲线救国的方法：



就是用一个新分支先把dev的代码整合过来（注意不是merge，merge会把所有的commit都包含）然后用这个新分支去rebase，这样只需要解决一次conflict，当然，这样会损失所有的commit，看情况是不是采取这样的方案。怎么整合代码呢？就是先把dev merge进dev1，merge完之后dev1包含了所有的dev的commit，这个时候使用reset --soft，把所有的commit取消，但是merge的时候解决好的conflict的代码还在，所以只需要再commit一次，这样就实际上拥有了dev的最新代码，完成之后再rebase到b。

小结

其实本文最核心的是要抓住rebase的运动过程，一旦知道rebase是一个一个commit重新提交的一个全新的分支，就知道为什么rebase的时候会出现那么多状况了。