Git for Visual Studio 2015

目录

[Git for Visual Studio 2015 1](#_Toc449616974)

[1 快速启动Git 3](#_Toc449616975)

[1.1 从你的Visual Studio团队服务Git库取得代码 3](#_Toc449616976)

[1.2 在Git中保存你的更改 4](#_Toc449616977)

[1.3 为了审查和其他人分享你的更改 6](#_Toc449616978)

[1.4 从其他团队成员得到更改 7](#_Toc449616979)

[2 Git工作流-随Git得到已经做完的工作 8](#_Toc449616980)

[2.1 典型版本控制工作流： 8](#_Toc449616981)

[2.2 Git工作流： 9](#_Toc449616982)

[3 创建一个库-随团队服务创建Git库 9](#_Toc449616983)

[3.1 一个库克隆有： 9](#_Toc449616984)

[3.2 怎样克隆： 10](#_Toc449616985)

[3.3 看它被做的怎样： 11](#_Toc449616986)

[4 用提交保存工作 12](#_Toc449616987)

[4.1 添加一些变更到一个提交 13](#_Toc449616988)

[5 在这些分支里创建工作 15](#_Toc449616989)

[5.1 什么是Git分支？ 15](#_Toc449616990)

[5.2 这些Git分支怎样被创建？ 15](#_Toc449616991)

[5.2.1 Git分支如何帮你完成工作？ 17](#_Toc449616992)

[6 随推送共享代码 17](#_Toc449616993)

[6.1 推送这些更改： 18](#_Toc449616994)

[7 随提取和拉取升级代码 20](#_Toc449616995)

[7.1 Git提取 20](#_Toc449616996)

[7.2 Git合并 21](#_Toc449616997)

[7.3 Git拉取 22](#_Toc449616998)

[8 随拉取请求审查代码 22](#_Toc449616999)

[8.1 一个拉取请求是什么？ 23](#_Toc449617000)

[8.2 创建拉取请求 23](#_Toc449617001)

[8.2.1 谁审查这个拉取请求？ 24](#_Toc449617002)

[8.2.2 代码怎样审查工作？ 24](#_Toc449617003)

[8.3 完成一个拉取请求 25](#_Toc449617004)

[8.3.1 当一个拉取请求被合并发生了什么？ 26](#_Toc449617005)

[9 解决合并冲突 26](#_Toc449617006)

[9.1 为什么Git有合并冲突？ 26](#_Toc449617007)

[9.1.1 Git如何防止合并冲突？ 26](#_Toc449617008)

[9.1.2 你能做什么去阻止合并冲突？ 27](#_Toc449617009)

[9.2 你怎样解决合并冲突？ 27](#_Toc449617010)

[10 积压合并的管理历史—使用积压合并去指明Git历史 29](#_Toc449617011)

[10.1 拉取请求和Git历史 29](#_Toc449617012)

[10.2 一个积压合并是什么？ 29](#_Toc449617013)

[10.3 一个积压合并是怎样有用的？ 30](#_Toc449617014)

[10.4 当积压合并的时候注意事项 30](#_Toc449617015)

[10.5 随积压合并完成拉取请求 31](#_Toc449617016)

[11 取消更改-随重置和恢复在Git中取消更改 31](#_Toc449617017)

[11.1 取消本地文件更改 32](#_Toc449617018)

[11.1.1 自从上次提交取消所有本地更改 32](#_Toc449617019)

[11.2 取消多本地提交 32](#_Toc449617020)

[11.3 在推送提交里取消更改 33](#_Toc449617021)

[12 随变基应用更改-随Git变基管理历史以及合并更改 34](#_Toc449617022)

[12.1 变基怎样工作？ 34](#_Toc449617023)

[12.2 当rebase时小心使用 35](#_Toc449617024)

[12.2.1 这不更改开发的历史吗？ 35](#_Toc449617025)

[12.2.2 在Git里变基： 35](#_Toc449617026)

[12.3 通过变基积压提交 36](#_Toc449617027)

# 快速启动Git

此指南将涵盖下面的规律以使你能够快速地开始运行Git和Visual Studio团队服务。

## 从你的Visual Studio团队服务Git库取得代码

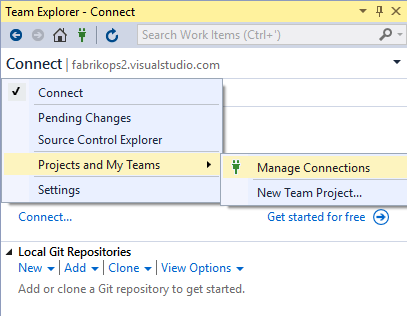
得到一个源代码的备份，你将克隆一个团队服务 Git库。克隆为你创建随工作源代码的一个备份以及所有版本控制信息以使Git能够管理这个源代码。

如果你还没有一个Git库，你能够创建一个使用你自己码并且然后克隆被创建的最新的团队服务库。

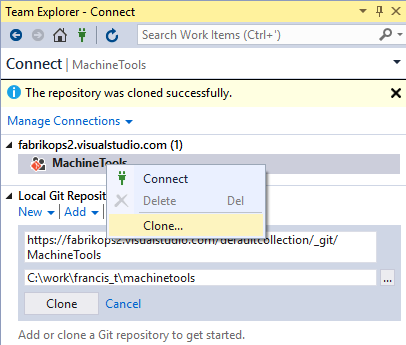
显示步骤：

Visual Studio

1. 在团队资源管理器中，通过选择“项目和我的团队”然后“管理连接”打开连接页



1. 选择“连接”，选择你的团队服务账户，选择你想工作的那些项目，然后点击“连接”。
2. 右键单击该项目并且单击“克隆…”。然后在你的驱动器输入一个本地文件夹去存储已经下载的代码。



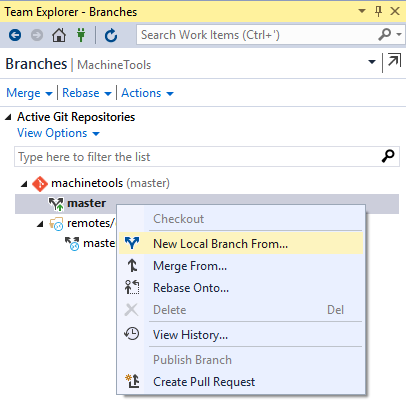
## 在Git中保存你的更改

在这个项目中Git分支把你的更改与其他已经完成的工作隔离。Git工作流被推荐用于每一个功能的一个新的分支或者修复你的工作。在你的本地Git库中提交到保存你的更改到那个分支。

显示步骤：

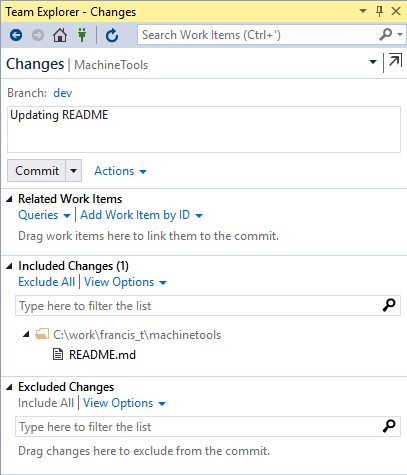
Visual Studio

1. 在团队资源管理器中，点击下拉列表并且选择“分支”。右键单击主分支并且选择“从选定项创建本地分支（N）…”。



为你的工作选择一个描述性分支名称来提醒你和其他人在这个分支里有什么样的工作。

1. 制造更改到该项目中。Git将保持Visual Studio的内外代码制造更改的追踪。
2. 当你满意这些更改时，使用一个提交保存他们到Git中。从团队资源管理器中打开这些“更改”视图。验证这些被包含的更改，添加一个描述这些更改的信息，并且点击“提交”。



## 为了审查和其他人分享你的更改

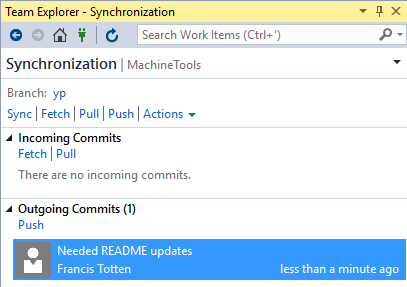
当你准备和这个团队分享你的更改，你“推送”那些更改以便其他人能够得到它们。你添加提交到一个分支后你仅能推送这些更改。

一旦你推送这些更改，你能够创建一个“拉取请求”让其他人知道你想要有一些更改被审查并且添加这些代码的主分支。

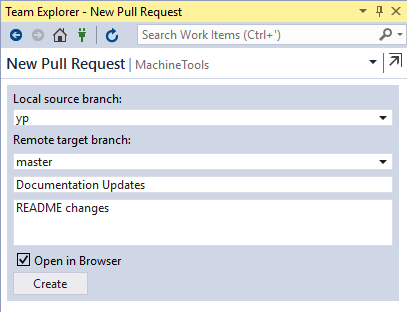
显示步骤：

Visual Studio

1. 在团队资源管理器中打开“同步”视图。你能够看到这些正在传出的提交并且通过点击“推送”分享你正在工作的已经被分享的一个分支，或者“发布”你正工作的一个最新创建的本地分支。



1. 创建一个拉取请求以使其他人能够审核你的更改。在团队资源管理器中打开“拉取请求”，并且点击“新拉取请求”。证实那个远程分支合并进那些更改中，例如“我的配置”。



1. 你能够审查在Visual Studio 团队服务项目页上的一个web浏览器中你的拉取请求制造的评论。通过这个团队一旦所有更改被认可，通过这个web浏览器你完成拉取请求。

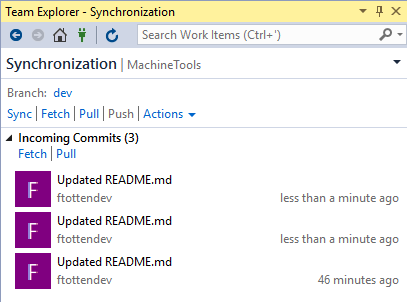
## 从其他团队成员得到更改

从其他人得到更改并且保持你的代码最新，你拉取其他人的提交并且把他们合并到你的分支中。甚至在相同的文件中Git是非常好的关于合并多更改，但有时你必须不得不解决一个合并冲突。当你拉取你的主分支保持它们最新Git工作流运行得最好。这帮助确保你的新分支脱离正在使用最新版本的主分支。

显示步骤：

Visual Studio

1. 在团队资源管理器中打开“同步”视图。你能够下载这些最新的更改到你正在使用的“拉取”分支。

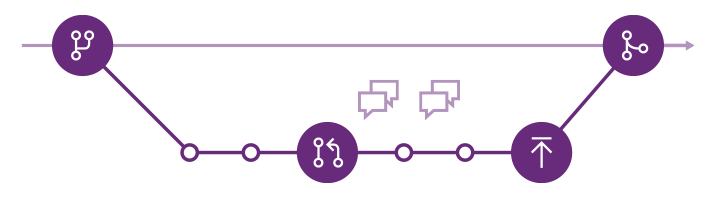


# Git工作流-随Git得到已经做完的工作

开发者使用版本控制为了方便和其他人分享更改并且快速地回到一个已知好的代码套件。当你和多人工作时这是极其有帮助的。它允许每个人制造并且分享更改而无需互相干扰。

## 典型版本控制工作流：

1. 创建一个本地代码备份。
2. 更改代码。
3. 为了审查和其他人共享代码。
4. 为了其他人使用保存代码到一个远程服务器并且添加到这个项目生成。



Git有它自己使用指定独特术语Git工作流的版本。你自始至终将在全部教程学习关于库，分支，提交，和拉取请求。这些条目可能听起来像一个你以前曾经使用过的版本控制系统-像Team Foundation 版本控制或Subversion-但在Git中他们运行大不同地。

## Git工作流：

1. 为了你计划制造的那些变更创建一个分支并且给它一个名字，例如fix-bug-3214或者cool-feature-x。
2. 提交更改到你的分支。为了一个bug修复或者配置人们经常有多个提交。
3. 推送你的分支到那个远程库。
4. 创建一个拉取请求以使其他人能够审查你的更改。包含反馈，你可能需要制造更多的提交和推送更多的更改。
5. 完成你的拉取请求并且解决一些从你创建你的分支以后其他人生成的更改的合并冲突。

这是一个最常见的Git工作流的大纲，而你更改它最好地反应了你的团队工作方式。如果你是刚接触Git，使用这个工作流直到你发现你需要更改的任何东西。

# 创建一个库-随团队服务创建Git库

一个库，经常被叫做一个repo，存储你的项目的文件和文件夹。Git存储在repo中每一个的完整历史以方便看到整个repo或者仅选则的那些文件的历史。如果你的项目有一个现存的Git库你能够用Git的“克隆”功能创建一个备份。如果你正启动一个新项目，你能够在Visual Studio团队服务创建你的库并且克隆它去得到启动。你也能够使用Git的“初始化”命令从一个本地文件夹创建一个本地库并且然后推送这个repo到团队服务中。

## 一个库克隆有：

* 在这个库里那些“分支”的一个列表和他们的更改。
* 被克隆库和被托管库之间的一个链接。远程库得到“原始”的一个Git别名。

你克隆以后，你有你需要开始工作的每一件事：一个最新的代码的版本，在这个库里的所有分支以使你能够看到在进展中的开发工作，以及一个配置到你将共享你的更改的远程库。

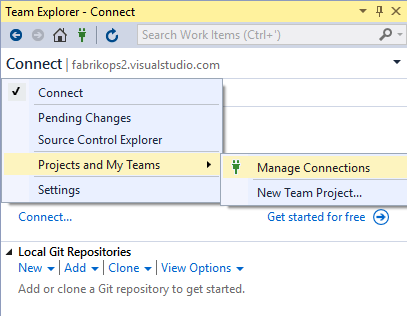
我需要克隆一个库执行什么操作呢？

* 你想创建本地库的一个文件夹。如果你没有指明一个路径，Git将为你已经克隆的库创建一个文件夹。你能够移动这个文件夹到你想要克隆完成之后的任何地方。
* 一个URL到你正在克隆的那个远程库。

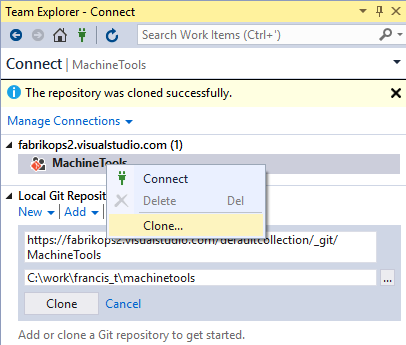
## 怎样克隆：

Visual Studio

连接Visual Studio到你的Visual Studio团队服务项目。在团队资源管理器中，从“项目”打开“连接”视图然后从上下文菜单的“管理连接”。



这将弹出一个你能够从下拉列表选择你的团队的账号的对话框并且然后选择项目去连接到Visual Studio。选择项目之后点击“连接”并且你能够为了通过右键点击这个项目并且选择“克隆…”。你将被提供一个URL以及提示一个为Git存储本地库的文件夹。



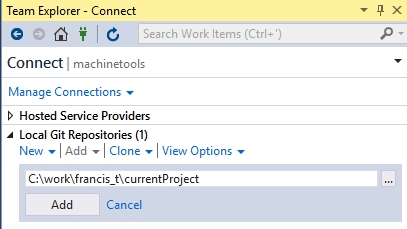
我已经没有关于在像Visual Studio团队服务的一个Git服务的代码应如何？

Git能够把在你的电脑上的一个现存文件夹用“init”命令把它变为一个Git库。你能够连接这个Git库到一个托管Git库之后发布你的工作。

## 看它被做的怎样：

Visual Studio

在团队资源管理器中，通过选择“项目”打开“连接”视图然后从上下文菜单“管理连接”。在“本地Git库”下，选择“添加”并且给一个库将被通常地随代码创建的一个现存目录的文件夹名，而假如你从零开始也能够指向一个新的目录。



连接本地库到一个托管Git库去共享你的工作，你将需要使用以下命令：

一旦你准备共享你的代码，为了“克隆“使用这些步骤得到你想连接的那个库的克隆URL，并且然后创建一个链接到创建一个为了使用Git的”远程“命令的你的本地Git库的“来源”：

>git remote add origin <https://myaccount.visualstudio.com/DefaultCollection/myproject/projectname/_git/reponame>

>git fetch origin

“remote add“创建一个在本地库上相同名字的一些分支与在”来源“库上的那些远程分支之间的关系之后你运行”git fetch origin“命令。

这些步骤的一个替代方法是在Visual Studio团队服务上的你的项目里创建一个新空Git库并且然后克隆那个库。这给你一个新空本地库去开始进行为这个项目相关联的一个原始库的一些更改以使它更容易推送以后的一些更改。

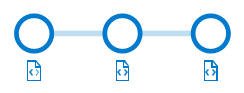
# 用提交保存工作

用Git提交保存你的工作

一旦你“克隆“你的repo或者创建一个本地Git库，你能够用你最喜欢的工具工作。一旦你达到一个你想保存你的工作的点，在Git中你将需要记录那些更改使你能够容易地回到这个项目的当前版本。Git通过使用”add“已暂存使用文件更改并且保存这些已暂存更改到Git来保存工作进一个”提交“。提交总是与你的本地Git库冲突，因此你不必担心这个提交完美或者准备与其他人共享。当你继续工作时你能够生成更多的提交并且当你准备这样做时分享这些代码。

Git提交包含下面的项目：

* 描述一个提交的一条信息。当你创建这个提交你给这个信息到Git。它是一个好主意去尽量保持这个信息简短和要点。
* 这些文件更改包含进这个提交。Git记录一些提交之间的文件差异，这些不完整的文件有助于性能以及考虑到的智能合并。
* 对父提交的引用。这些提交之间Git使用这些引用来维护一个开发的历史。



Git提交保留一个上次提交的一个历史，如图所示。从一个历史的多个提交，随在这个图片的右边最新的提交。Git能够自始至终对这些提交通过从左到右建立一个按时间顺序排列的开发历史记录。Git使用这个历史去决定这些文件更改是怎样被应用到它能够智能合并更改到一起。

Git提交历史还充当一个开发记录，允许你通过看提交历史找出关于一个文件变更的信息。

## 添加一些变更到一个提交

当你创建一个提交Git不自动地找并且添加被更改的那些文件。为列入你下次生成的一个提交你必须告诉Git哪些更改将被暂存。

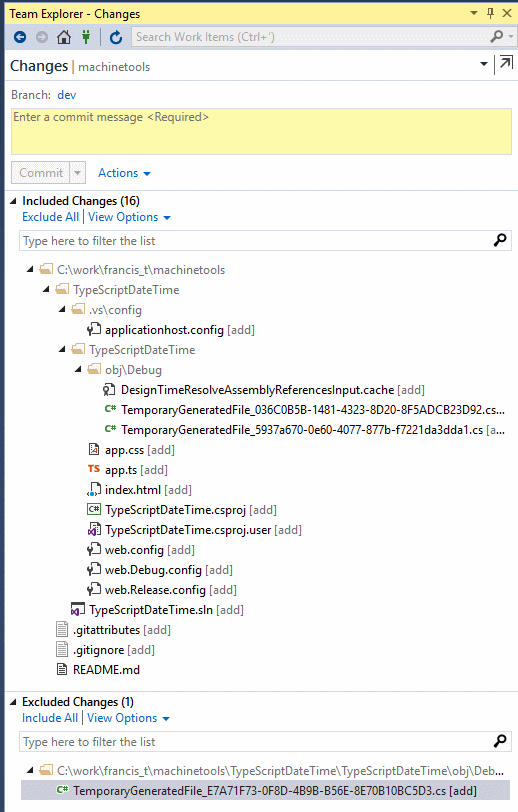
你包含文件到使用“add“命令的那个提交。

看怎样创建那些提交：

Visual Studio

当你做你的工作时Visual Studio将自动地保持那些更改的追踪到你的项目。当你准备生成一个提交时，在团队资源管理器中打开“更改“视图。在包含与不包含更改之间你能够拖放被更改的文件到仅仅提交某些已修改文件。通过输入一个提交信息并且单击”提交“按钮然后你能够提交这些更改。

注意在Visual Studio中当你提交你也能够立刻地“推送“这个提交并且”拉取“一些更改到你的项目还没有的那个分支。在团队资源管理器里的”提交“按钮上的下拉列表里这些选项是可用的。



# 在这些分支里创建工作

在这些Git分支里创建工作

## 什么是Git分支？

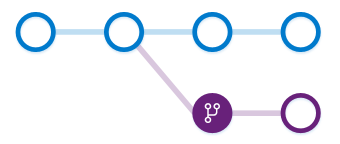
Git分支是在这个库里代码的一个特定版本上你能够生成更改的单独工作区域。正在提交的更改到一个分支里将不影响其他的分支。这些分支隔离那些更改，允许用户彼此共享代码而无须合并这些更改进主分支中。

Git分支不比一个保留提交的一个确切历史的一个小指针更多，因此Git分支是非常轻量地去创建。因此这些分支是这么的轻量，分支之间的切换时快的并且容易。当和使用这些文件更改的分支工作Git不创建源码的多备份并且当你在其上开始工作时历史信息存储进提交里去重新创建在一个分支上的一些文件。

为了管理更改你的“Git工作流“应该频繁创建并且使用这些分支。这个Git工作流的剩余部分，例如分享代码并且随通过这些分支拉取请求所有工作审查代码以及那些非个人提交。在这些分支里隔离工作也使它非常简单去更改你正在通过简单更改你的当前本地分支之上的工作。

## 这些Git分支怎样被创建？

使用“branch“命令Git创建这些分支。为了新的分支在Git里”branch”创建一个引用并且通过下面的这些命令一个指针回到父提交以使Git能够保留这些文件更改的一个历史。当你正在和某个其他人共享一个分支工作，Git保留一个上游追踪关系去将在本地库的分支与在远程库上的相应分支联系起来。



在这个图里，一个新分支从主分支被创建。在两个分支和提交之上继续工作是被添加到两个分支。

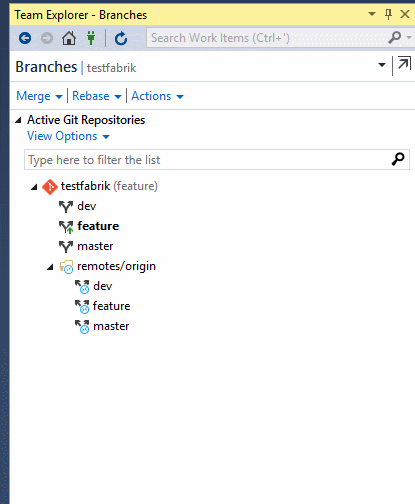
Git将总是添加新提交到当前本地分支。它总是一个好主意去知道进行一些文件更改之前你正在工作在什么分支之上以便你不提交这些更改到错误的分支。使用Git的”checkout”命令你在本地分支之间移动。当在这个分支里你的工作已经准备与该团队的其他人去分享，你推送这些更改并且升级那个远程分支。

一个非常普通的错误是生成一些更改并且“提交”他们，发生你是在一个错误的分支上，然后“签出”到那个正确的分支。自从这些分支工作在隔离Git里你的最近更改将不再是在文件系统之上将使这些文件的状态恢复回之前提交的你置换进的那个分支上，不是你生成的那些更改的那个早期分支。你将需要从那个分支挑选那些提交或者合并那些更改到正确的分支里。那能够证明有问题的，因此它通常是一个好的实践去总让你是在生成那些提交之前的那个分支上了解。

看它是怎样做的

Visual Studio

打开团队资源管理器并且到“分支”视图。创建一个新的本地分支，右键单击父分支（通常master）以你的更改作为根据并且选择“从选定项创建本地分支”。在被需要的字段里你提供一个分支名并且点击“创建分支”。Visual Studio将自动地“签出”到新地被创建的分支。



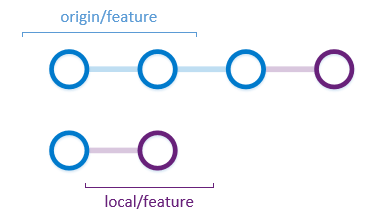
### Git分支如何帮你完成工作？

Git保持你正在工作的哪个分支的追踪，并且确保当你在文件系统之上的那些文件签出到一个分支与在给分支之上的上次提交匹配。这允许你通过使用签出转换那些分支在相同本地Git库里随源代码的多个版本工作。你告诉Git通过“签出”你想工作在哪一个分支之上，并且Git为你照顾本地文件系统。

分支的一个好处是你从来不需要更多在你系统之上的代码的一个备份。你建立你的开发环境和本地测试/生成一次工具并且然后使用Git分支去管理你需要做的不同工作。甚至你从来没有和其他人分享这个工作，在这些分支里有被正在隔离工作让你保持不影响你想去分享工作的那些更改。

# 随推送共享代码

通过推送命令你共享代码更改生成在这些分支上的Git提交里。这些本地分支被推送到“来源”库，在“来源”上Git需要这些提交并且把它们添加到一个现存分支或者在“来源”上随相同名字创建一个新分支以及提交历史作为本地分支。



在这个图里，在紫色的这些本地提交是被推送到远程库并且“origin/feature”分支现在有从“local/feature”分支的那些提交以及他们自己的那些提交。Git管理那些提交历史并且确保他们是符合你的更改以便其他人能够拉取你的那些提交并且合并它们到他们自己本地分支里。

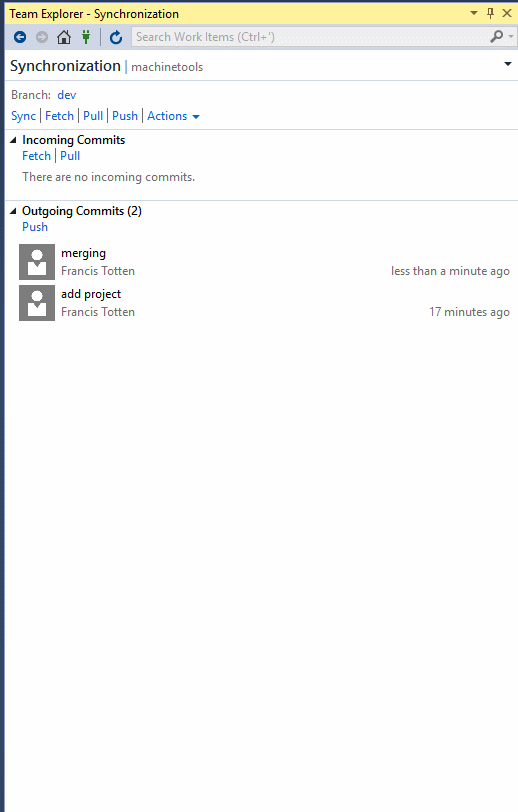
在一个分支上Git工作流用户们推送分享这些更改以使其他人能够得到这些更改。这些推送分支已经被完成了被审查工作并且通过一个拉取请求合并到代码的“master”分支。

## 推送这些更改：

Visual Studio

在团队浏览器里，打开同步视图。这个视图列出正在出口的这些提交以及即将提供你一个或者两个链接去推送它们：

1. 当在原始库上没有一个分支时Visual Studio有“发布”链接与当前本地分支相关。在原始库上这将随相同名字创建一个分支并且给它推送这些提交。之后你“推送”，由于在原始上的分支与现存本地分支之间的关系当使用这个分支不发布更改。
2. 当本地分支和在原始上的一个相关追踪之间有一个关系Visual Studio有“推送”链接。点击这个将推送你的更改到远程分支。



在那个远程分支上如果随这些提交有合并冲突，随那些调和的更改在一个新提交里你本地地解决那些冲突。然后你能够完成“推送”到远程分支。

# 随提取和拉取升级代码

从其他人Git提供三个命令去得到更改并且应用它们到你的本地库。它们是：

* 提取，从哪一些下载的更改但不适用它们到你的代码。
* 合并，在你的本地库哪一些应用更改取自“提取”到那些分支。
* 拉取，一个“提取”和然后一个“合并”他们哪一个是一个简写方式。

## Git提取

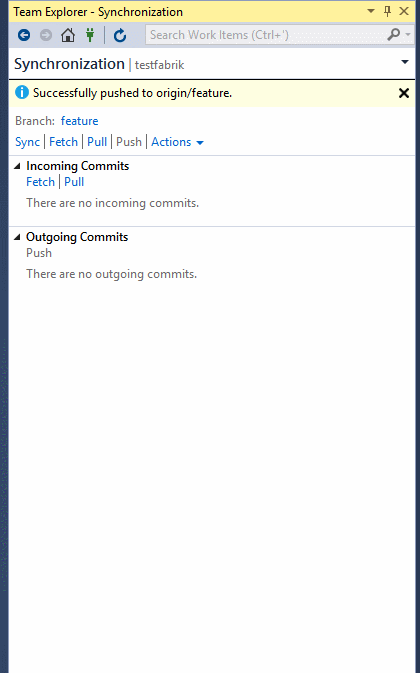
通过“提取”从“来源”你下载更改到你的本地分支。提取要求远程库为了其他人已经生成而你在你的本地库没有的所有提取。“提取”下载这些提交并且添加它们到本地库。

一个重要的内容要注意是提取不使你的本地分支镜像那个远程分支。它从远程分支得到更改，而在你的Git库里不应用它们到那些文件。为了你正在随工作的所有分支提取也将收集所有那些待定提交和他们的更改，并且不仅仅当前分支。

看怎样提取：

Visual Studio

在团队资源管理器中Visual Studio使用同步视图去提取更改，并且通过默认在后台每几分钟将提取。因为通过提取更改下载不被应用到这个分支直到你合并，在你的本地项目文件上没有更改被生成直到你拉取或者同步这些更改。



## Git合并

通过”提取“使用“合并”命令你能够应用更改下载。从提取“合并”将采取检索这些提交并且尝试添加它们到你的本地分支。这个合并将保持你的本地更改的提交历史以使当你随“推送”分享你的分支Git将知道怎样其他人能够合并你的更改。

随合并的挑战是当一个提交取自在你的分支上随一个存在的未推送的提交的冲突。这被叫做一个合并冲突。Git是通常非常自动地智能地解决合并冲突。如果Git不能自动地解决这些冲突，你必须手动地解决这些合并冲突以及随一个新合并提交完成这个合并。

看怎样合并：

Visual Studio

当你做一个“拉取”或者一个“同步”Visual Studio的团队资源管理器执行这些合并。“同步”是一个拉取远程更改的联合操作并且推送本地更改去同步在本地和远程分支上的提交。

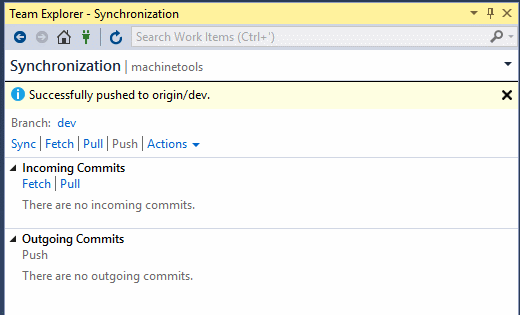
## Git拉取

“提取”和“合并”经常充分地一起运行在两个结合在一起的命令，“拉取”，被创建。“拉取”是一个“提取”并且然后一个“合并”去添加已经下载的那些提交到你的本地分支里。

看使用拉取怎样升级代码：

Visual Studio

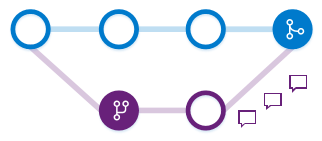
打开团队资源管理器并且打开同步视图。然后点击“传入提交”下面的“拉取”链接去“拉取”远程更改以及合并它们到你的本地分支里。这将在你的本地项目里升级那些文件，因此如果在你的工作里有未被保存的更改Git将阻止你拉取。



# 随拉取请求审查代码

## 一个拉取请求是什么？

Git工作流显示开发在隔离不同持续开发工作的那些分支里如何开发工作的。Git使用拉取请求到包括在一个分支里的已完成的工作到另一个代码的分支，例如“master”。一个拉取请求让这个团队的其他人知道这个分支工作是已经做了并且要求这个团队去审查和批准这些变更。



在这张图里，一旦工作完成紫色分支是被合并回蓝色分支里。在紫色分支上一旦上一次提交被添加，你创建一个拉取请求使那个分支被合并进那个蓝色分支里。这个拉取请求之上的批准，有在紫色更改现在被包括到蓝色分支里的一个合并提交（见于那个被填充蓝色提交）。

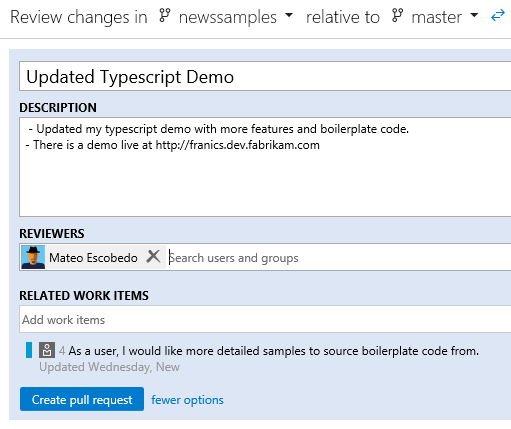
## 创建拉取请求

在团队服务里创建一个拉取请求：

* 在你的本地分支上“推送”那些提交到一个远程分支。
* 一旦那个远程分支升级，在团队服务里为那个分支你将有一个提示去创建一个拉取请求。



* 创建那个拉取请求。你应该为了包括那个正在更改的拉取请求给一个明确的标题。在那个描述字段里给出一个怎样更改与一些可以帮助审阅者理解那些更改的源一起被执行的明确解释。当审查你的更改你能够包括团队服务工作项和超链接去允许其他人有尽可能多的上下文。
* 添加一些你喜欢的团队成员去审查那些更改。

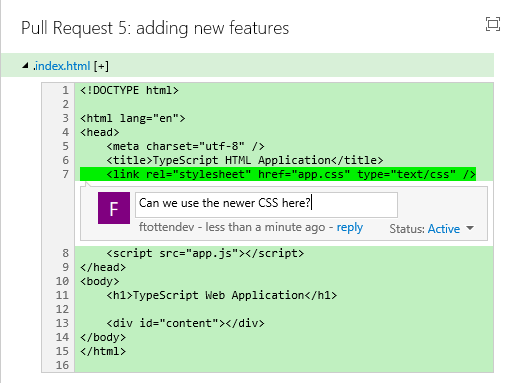


### 谁审查这个拉取请求？

当你创建那个拉取请求，你能够添加需要审查你的更改的其他人。如果审查的范围需要扩展它被创建之后你能够添加使用者和组到那个拉取请求。在团队服务里你也能够将一个任务与这个拉取请求关联让其他人知道更改准备审查随任务一起工作。

### 代码怎样审查工作？

拉取请求审阅者们将看到在两个分支之间文件不同的形式里那个被建议升级到的分支。审阅者们能够添加关于那些更改的一些评论以及包括为其他团队成员回答一个问题或者给出其它反馈的通知。你能够生成更改并且推送提交去解决在反馈里提出的问题以及这些更改被立即地反应在拉取请求中。



如果这些更改需要更多的开发去完成，你能够放弃这个拉取请求。你能够以后打开一个新拉取请求去重新考虑这些更改和超链接到发生在被放弃的拉取请求里的对话。

你也能够在你的代码的一个非常早版本上打开一个拉取请求去寻求从其他人的反馈，甚至这个代码还没有准备合并。一旦你得到这个团队的反馈，你能够保持打开这个拉取请求去继续这个对话或者放弃这个拉取请求直到你的代码再一次准备被共享。

## 完成一个拉取请求



一旦所有的审阅者被批准这些更改，你现在能够完成这个拉取请求。通过这个拉取请求你将被提示为一个描述性的被建议更改的信息。当这个拉取请求完成时你能够选择删除这个拉取请求分支。这个拉取请求完成之后在主分支里Git保留这个提交历史，因此除非在这个分支里你打算做更多的工作，它被安全地删除。

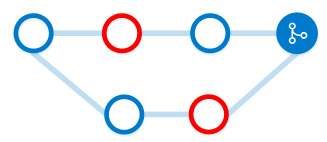
### 当一个拉取请求被合并发生了什么？

在拉取请求分支和目标分支之间你必须解决一些合并冲突。Git添加一个新提交（合并提交）到主分支的末尾。这个合并提交链接主分支和为作为拉取请求的一部分的被合并的分支提交的早期历史。

# 解决合并冲突

## 为什么Git有合并冲突？

当你“推送”或者“拉取”提交时，在分支历史里作为其他更新的提交能够有相同文件的更改。如果通过它自己Git不能解决这些更改，它告诉你有一个合并冲突。通过指明确切地你如何想合并这个文件内容你负责解决这个合并冲突。



在这个图里，在不同分支里红色提交更改一个文件的相同行。当本地分支试图合并回到父分支里，Git创建合并提交（这里见于深蓝色）把这些更改生成进两个红色提交中以及添加所有不冲突的那些提交。

### Git如何防止合并冲突？

Git使用提交历史与那些文件一起列入这些提交去看哪一个更改在冲突的那个时刻已经存在。Git使用文件差异决定通过提交历史去通常地在它自己上合并更改而无需为了手动合并。

### 你能做什么去阻止合并冲突？

它不必积极地避免合并冲突。它是经常比解决它们更多工作和更多风险去采取阻止合并冲突。几个例子说明这一点：

1. 一个分支已经在开发了一些时间并且在其它分支上积极开发下生成这些更改到编码。有在这个分支里的代码将不随最新版本升级的一个风险以及简直不工作。在结束时在这个情况里一个好的实践是不避免那个“巨大的合并”，而从开发分支创建一个本地分支并且偶尔地合并一个最新master到那个分支里。

当继续开发，从原始分支你能够合并到更新里并且从master确保结果不符合这个项目的剩余部分。如果它是，当它发生时你快速地抓住以使你能够随你的团队解决重大更改。

1. 在一个源文件里相同部分里频繁的小更改也能够驱动合并冲突。这些是经常频繁文本更新与代码逻辑不相关。移动这个文本去格式化属性文件有利于更容易更改，特别地如果一个相应的属性文件已经存在于这个项目里。

## 你怎样解决合并冲突？

Git解决合并冲突通过以下步骤：

1. Git暂停当前的推送或者拉取有合并冲突的命令。
2. Git生成有冲突的那些文件的那些行的更改。这些标记看起来像：

<<<<<<< HEAD

console.log("Writing changes to dev console");

debug("Writing changes to debug module);

>>>>>>> dev-updates

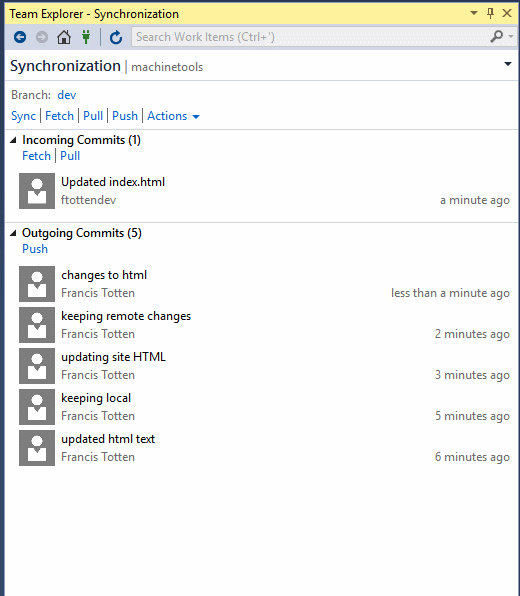
发生了这是两个人有一个好的想法去添加一些日志记录到代码的一个特定部分，尽在不同的方式。Git标记这些使用<<<<<<<HEAD部分的更改去注意你的更改，=======作为这些更改之间的一个分割，以及>>>>>>><branchname>对于这些更改不是你的冲突。

1. 你编辑这些文件以使它们看上去精确地它们应该怎样，删除置于它们中的已添加格式Git。
2. 添加已更改文件到下一次提交。对于具冲突的所有文件重复。
3. 提交这些更改。这将解决这个合并冲突。

看它怎样工作：

Visual Studio

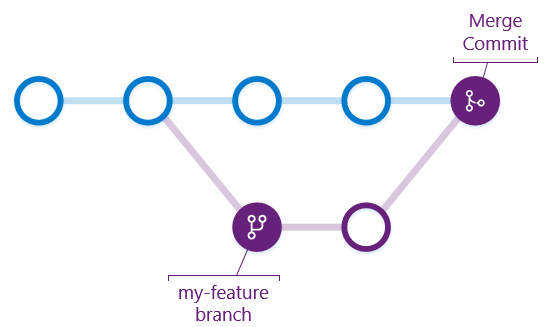
1. 从这个远程库当你做一个“推送”去得到这些提交时你将被告知这些合并冲突。甚至当“推送”时，在Visual Studio里你能够推送本地更改到一个分支之前你必须降低远程更改。
2. 这个冲突通知显示并且将有一个链接解决这些冲突。
3. 这将随冲突提出一个文件的列表。你能够快速地接受远程或者保持本地更改，或者点击“合并”去手动合并这些文件。
4. 你能够使用已经修改的那些行旁边的那些复选框去完全地选择远程和本地更改，或者通过选择任何复选框直接地编辑文件，当完成保存。
5. 当完成更改，点击“接受合并”。对于所有冲突文件重复这个。
6. 在这些更改视图里提交你的更改去解决并且删除这个合并冲突。



# 积压合并的管理历史—使用积压合并去指明Git历史

## 拉取请求和Git历史

在团队服务里当你完成一个“拉取请求”时，你合并主题分支到你的默认分支里，通常地master。这个合并添加主题分支的这些提交去你的主分支并且在默认和主题分支之间创建一个合并提交去调和一些冲突。在拉取请求里评论和讨论给出对于在主题分支里生成的那些更改的额外上下文。

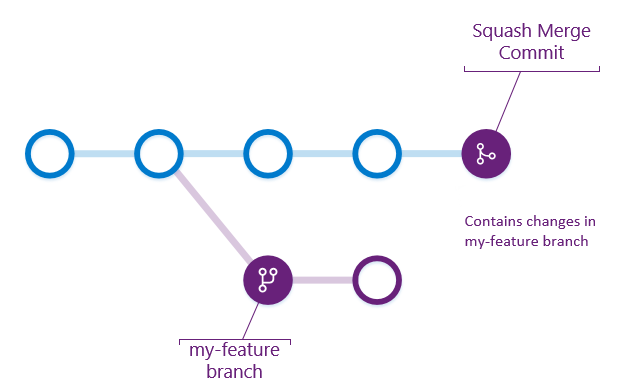


因相关主题分支历史在你的主分支上提交历史（或者其他默认分支）不遵循一条直线。当这个项目变得更大，主题分支的数目对在同一时间的增长有效，使默认分支历史越来越多的困难去遵循。

这个默认分支是每个主题分支的历史的一个准确代表，而难以用回答更广泛的关于你的项目的开发问题。

## 一个积压合并是什么？

积压合并是一种合并选项当你完成一个拉取请求时允许你浓缩主题分支的Git历史。而不是在这个主题分支上的每一个提交被添加到默认分支历史，一个积压合并需要所有文件更改并且在默认分支上添加它们到一个单独新提交。



一个简单方式思考这是积压合并给你只是文件更改，并且一个定期合并给你文件更改以及提交历史。

## 一个积压合并是怎样有用的？

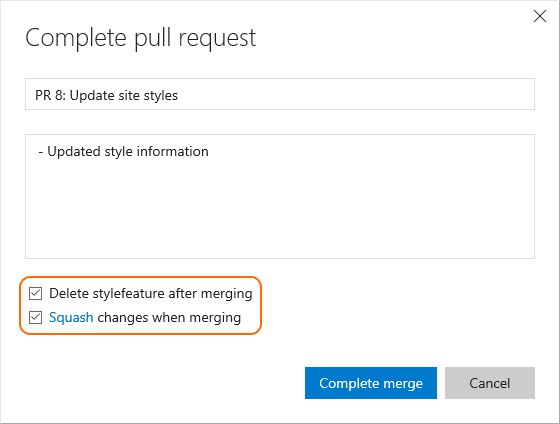
积压合并保持你的默认分支历史干净并且易于跟随而无须在你的团队上要求一些工作流更改。它们想在主题分支里怎样派遣到主题分支工作，并且这些默认分支通过积压合并的使用保持一个线性历史。一个master分支的提交历史随积压合并的升级将有一个为每一个合并分支的提交。当工作已经进行的时候你能够通过提交发现确切地这个历史提交单步调试。

## 当积压合并的时候注意事项

在你的默认分支里积压合并注意更改的历史，因此当你应该积压合并的时候以及当你想去保持一个主题分支的全部提交历史的时候它随你的团队去决定工作是重要的。当积压合并时，它是一个好的实践去删除源分支。这阻止造成混乱由于主题分支自己没有将一个提交合并到默认分支里。

## 随积压合并完成拉取请求

在团队服务里当完成一个拉取请求时你能够选择去积压合并。在“完成拉取请求”对话框上选择“当合并时积压更改”去积压合并主题分支。



# 取消更改-随重置和恢复在Git中取消更改

当在Git中取消更改，首先决定你正在尝试取消更改的类型。这些更改分为三个类：

* 在你自己的系统上本地未提交更改没有被添加到Git。
* 在你自己的系统上本地已提交的更改添加到Git而没有推送到其他人。
* 已提交更改到Git推送到一个远程分支。

注意当随Git取消更改，Git通过提交仅仅记住你添加的库是重要的。当你有未提交更改的那些命令正在执行时练习提醒你想要保留。当覆盖文件作为它的部分取消命令时Git不提示确认。

如果你仅仅是需要让小的升级例如去修复一个错误或者小问题引导进你的最后的提交，考虑修订你的上一次提交而不是取消全部提交。

## 取消本地文件更改

最基本的取消场景是你不想更改一个文件并且想恢复这些一个已知好版本的内容。你能够使用签出命令并且给它文件名去更改。Git接受通配符而没有对取消更改到多文件的一个问题。

> git checkout approuter.js

这不同于更早使用的用于在活动分支之间移动Git的“checkout”命令。在一个“checkout”里Git不明确关于发生什么-Git将告诉你如果它是在输出里的分支之间更改一个文件或者交换，并且抱怨如果你在相同的时间正在尝试去做两者。

### 自从上次提交取消所有本地更改

如果你需要在你的本地库里带一个分支回一个以前提交的层级内容，通过重置你能够做那个。重置命令的最通常用法是简单地放弃自上一次提交的所有更改的文件并且返回这些文件到它们在最近提交里的状态。这是“reset”命令去做：

> git reset --hard HEAD

命令的—hard部分告诉Git重置这些文件到上一次提交的状态并且放弃一些已暂存更改。HEAD参数告诉Git重置本地库到最新的提交。

注意这个命令：

* 在这个库上的当前分支里重置所有文件，在你的当前目录里不仅仅文件。
* 从另一个分支不重置这个库到你合并更改以前的状态或者从远程拉取更新到这个分支。它仅仅放弃没有被提交的更改。

## 取消多本地提交

Git reset也能够被用来取消提交并且在这个历史里返回这个分支到一个指定的提交。当这个重置命令运行时由于它不是简单地恢复更改所以这应该被格外小心。用户与分支不能被合并到它们的更改一起工作而无须执行一个复杂的rebase操作。

> git reset --hard commit\_id

Git的黄金规则适用：从来不重写被共享到一个远程库的提交的历史。

这个命令将返回这个库到在commit\_id里的状态。所有更改生成自commit\_id将被丢失。在分支历史使用里你能够发现一个提交的列表：

* 从命令行git log
* Visual Studio内部分支历史。
* 从团队服务web门户的分支历史。

由于运行此命令危险，随推送更改你不应该在分支上运行它。当看到回滚更改到一个指定点以下使用git恢复是更安全的，作为保留这个分支的历史。

## 在推送提交里取消更改

由于它能够重写一个分支的提交历史Reset命令是危险的。当与在那个分支里由于其他人工作并执行提交的共享分支一起工作时也必须重写这个分支的历史去和你的更改同步是有问题的。

当用“push”提交revert工作时Git提供一个安全的命令。Revert命令创建一个新的取消在一个上一次提交之上的更改的提交，以使这个分支的历史保留。

> git revert --no-commit commit\_id..HEAD

> git commit

这个命令将取消在commit\_id做到的最近提交生成的所有更改，返回分支到在commit\_id里的状态。然后你提交随一个新提交的更改。--no-commit防止Git对于在commit\_id和HEAD之间的有序的每一个提交进行一个新还原提交。

这个revert命令是灵活的而且它需要一个分支历史以及提交标识符去使用。通过在命令行上git日志你能看到分支的一个历史，在Visual Studio内部，或者在团队服务使用分支历史。

# 随变基应用更改-随Git变基管理历史以及合并更改

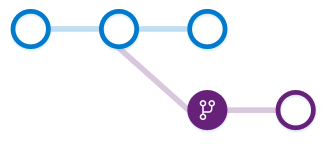
对于日常开发Git工作流通常是卓越的。这个工作流的一个方面是在整个项目的上下文里它不积极地管理更改的历史。通过“提交”和“拉取请求”Git记录每一个提交生成的master的主分支。这个选择是有意图的因为它保持工作流简单并且捕捉历史的一个全面记录，而它导致两个挑战：

* Master的提交历史不是一条直线。Master上的你的开发历史将包含大量的分支发散并且合并回master分支里。当你在多分支里即将有大量并发更改时，它变得难以遵循一个项目的整体历史。
* 每一个提交生成的库是master分支的历史的一部分。因此如果当做它们的工作时另一个开发人员频繁地生成提交，他们能够有一个非常大规模的提交历史添加到这个项目中。这使得分支甚至难以遵从因为你将不得不经历这些提交的所有去重读开发的story。

Git提供一个工具叫rebase去解决这些问题。Rebase从一个分支备份更改到另一个分支而无须执行在两个分支之间的一个合并，并且提供一个替代方式去当保持一个线性时从一个分支添加更改到另一个分支，为了这个分支简单遵从提交历史。

## 变基怎样工作？

Rebase是重播在另一个的历史之上的你的当前分支的提交以使提交看上去像它们被写作在目标分支之上的历史的一部分。这个重播将在新分支之上创建新包含相同文件更改的提交作为在你的当前分支之上的提交，而有不同的提交ID。





在第一个图里我们看到在紫色特性分支里的更改，创建原始分支。第二个显示一个rebase后从特性分支备份到主分支里的那些提交。由于没有合并被执行，没有合并提交并且在原始分支上的提交历史还是一条直线。

## 当rebase时小心使用

变基是一个对于移动提交并且创建你想要的确切的Git历史的强有力的工具，而你需要小心它的使用。尤其，总是避免变基已经“推送”到”源”或者一些其它的库的提交。当在现存工作分支上的一个提交被随已经重播的提交之一里的一个更改重播冲突时变基也能够发生冲突。这些冲突被以相同的合并冲突的方式被解决。

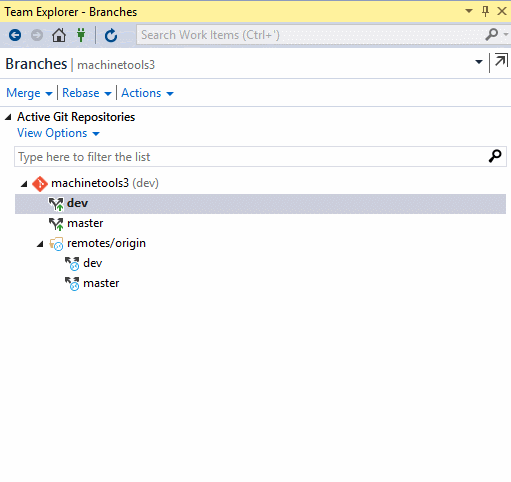
### 这不更改开发的历史吗？

使用rebase替代合并分支导致一个更容易去遵从但不确切提交的历史。Git提供rebase命令作为卖出开发历时的准确性的一种方式换取审查历史的易用性。它面对你的团队去决定当去rebase一个分支与当随一个拉取请求merge一个分支。一个被建议的方法是随时变基本地你已经生成而没有与其他人被共享的更改，而一旦你正在和其他人共享更改推迟拉取请求并且合并工作流。这允许你管理已经完成当确保其他人审查尽可能多的信息去工作的历史知识的提交历史。

### 在Git里变基：

Visual Studio

在团队资源管理器里，到分支视图。点击“变基”链接。你将看到一个提示从你的当前分支变基这些更改，并且然后一个下拉列表去指定在当前分支里更改应该被重播到哪一个分支上。如果有一个冲突，它能够被以在Visual Studio里“合并冲突“被解决的相同方式解决。



## 通过变基积压提交

一个特殊rebase的使用是能够浓缩这些生成一套提交到一个单个提交的更改。它们被推送到一个远程分支以前这是一个非常有用的技术去清理提交历史。像所有rebase操作，它导致一个真实开发历史的重写，而它通常是在足够小规模里你还能描述生成多提交的更改到一个单个提交。提高可读性的好处是明显-积压的四个提交到一个单个提交里添加了一段时间。

积压提交，第一决定关于你喜欢积压多少提交。当你运行这个命令你没有确切地因为将得到挑出确切地积压的你一个提交。如果你有五个命令去积压，然后你不确保没有未暂存更改并且运行：

> git rebase -i HEAD~5

这将打开一个编辑器你将看到像这样的事情：

pick 7b43f3f Updated router.js

pick 00859d9 Updated README.md

pick 9d247f7 initial commits

pick 7068b09 Updated README.md with build instructions

你将保留这些更改的第一个（丢弃它pick），并且然后对于这些更改的剩余部分更改pick到squash。保存并且关闭这个编辑器。Git现在将尝试凝结这些提交到一个单个提交中去。这些提交被积压之后，你将有一个新编辑器打开Git要求你为随合并的更改的提交给一个提交信息。你将想浓缩这到文本的一行（就像你将为了一个正常提交）并且保存和退出这个编辑器。