L7 质量管理(4)
——版本管理及分支策略

大纲

- 版本控制概述
- 悲观锁与乐观锁
- 集中与分布式版本管理
- 分支与合并
- 分支模式
- 在Git中管理分支与合并

大纲

- **版本控制系统概述**
  - 什么是版本控制系统
  - 版本控制系统的历史
  - 版本控制系统的基本概念
- 悲观锁与乐观锁
- 集中与分布式版本管理
- 分支与合并
- 分支模式
- 在Git中管理分支与合并

版本控制概述

- 版本控制系统（也叫源文件控制或修订控制系统）用于维护应用程序每次修改的完整历史。
- 版本控制系统管理的对象包括：
  - 源代码
  - 文档
  - 数据库定义
  - 构建脚本
  - ……
- 版本控制也是让越来越大的团队能够在一起工作的重要机制

版本控制的历史

- 1972年起源于贝尔实验室：SCCS
- 之后产生了许多在此基础上演化而来的工具
  - 开源
    - RCS，CVS，Subversion (SVN)，Git
  - 商业
    - Perforce，StarTeam，ClearCase，Visual SourceSafe (VSS), Team Foundation System (TFS)
- 当前趋势：DVCS（Distributed Version Control System)

版本管理中的基本概念

- 代码仓库(codebase)
  - 一个包含一组文件所有历史修改信息的逻辑单位，通常用于保存有关一个软件产品或某一组件的所有文件信息记录。
- 分支(branch)
  - 对选定的代码基线创建的一个副本。人们可以对这个副本中的文件进行操作，而这些操作与原有代码基线的文件操作是互不影响的。
- 主干(trunk/master)
  - 一个具有特殊意义的分支(branch)，通常在创建代码仓库时即由版本控制系统默认创建，每个代码仓库有且仅有一个这样的分支。其特殊意义在于其与软件的开发活动和发布方式紧密关联。

版本管理的基本概念

- 版本号(revision)
  - 对应在某个分支（branch）上的一次提交操作，是系统产生的一个编号。（有时也被称为一个“版本”）
- 标签(tag)
  - 某个分支上某个具体版本号的一个别名，以方便记忆与查找
- 头(head)
  - 是指某个分支上的最新一次提交对应的版本号。在Git中指的是当前工作分支

版本管理的基本概念

- 合并(merge)
  - 将一个分支上的内容与某个目标分支上的内容进行整合，并在该目标分支上创建一个新版本号。

- 冲突(conflict)
  - 在合入操作时，两个分支上的同一个文件在相同位置上出现不一致的内容。通常需要人工介入，确认如何修改后，方可合入目标分支。

大纲

- 版本控制系统概述
- **悲观锁与乐观锁**
- 集中与分布式版本管理
- 分支与合并
- 分支开发模式
- 在Git中管理分支与合并

悲观锁与乐观锁

- 版本控制系统管理资源并发控制的两种方式
  - 悲观锁
  - 乐观锁

悲观锁和乐观锁

- 悲观锁要求使用资源之前必须锁定。
  - 需要确保任意时刻只有一个人工作在一个对象上
  - 看上去悲观锁是阻止合并冲突的好办法，但事实上，它降低了开发的效率，尤其在大型的团队中。
- 乐观锁不锁定资源，而是假设在大多数时间里，大家不会同时工作在同一个文件上。
  - 乐观锁允许大家同时工作在同一个对象上。
  - 在提交修改时，会使用一些算法来合并这些修改。
  - 合并的最小单位是行。
  - 无法合并的情况下需要手动解决冲突。

悲观锁与乐观锁

- 乐观锁所推荐的实践
  - 频繁地提交代码
  - 降低模块间的耦合度
- 上述实践对持续集成和提高软件质量非常重要
- 对于有些不需要合并的文件，比如二进制文件，可以采用悲观锁。
  - 然而，二进制并不太适合进行在版本管理系统中管理
  - 文本文件比较适合在版本库中管理

- 现代版本管理工具的并发控制方式从以悲观锁为主转向了以乐观锁为主

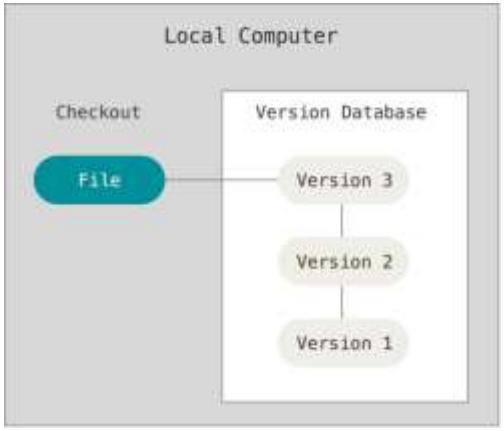
大纲

- 版本控制系统概述
- 悲观锁与乐观锁
- **集中与分布式版本控制**
- 分支与合并
- 分支开发模式
- 在Git中管理分支与合并

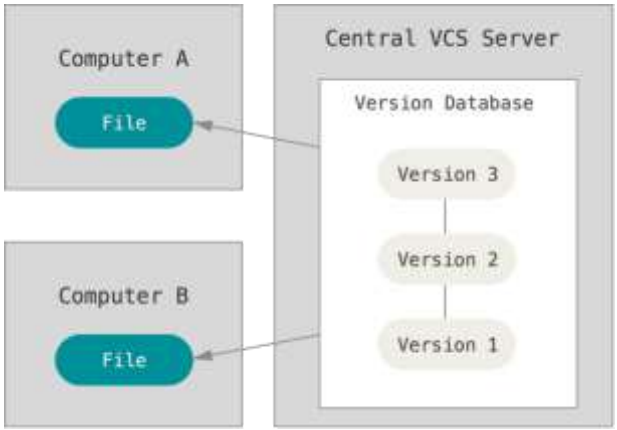
集中式与分布式版本控制

- 比较
  1. 本地版本控制
  2. 集中式版本控制
  3. 分布式版本控制

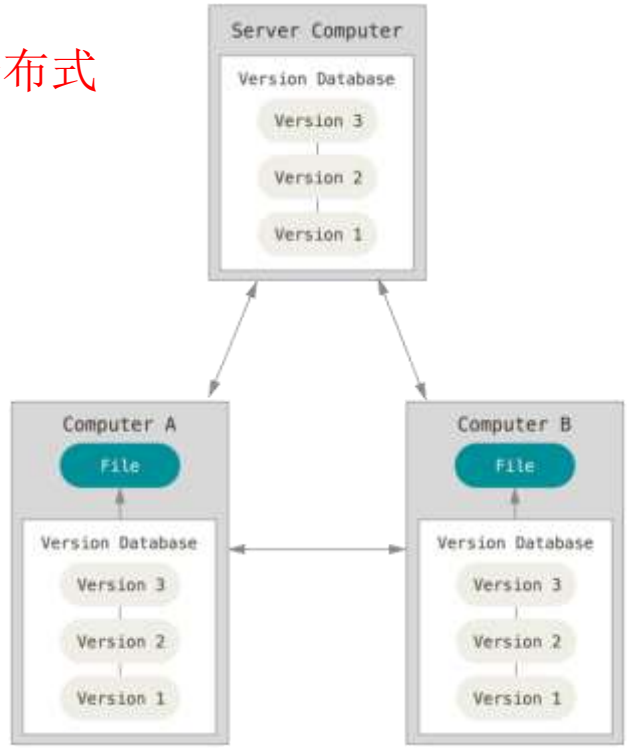
1：本地



2：集中式



3：分布式



集中式版本控制

- 集中式
  - 集中式版本控制解决了多人如何进行协同修改代码的问题。
  - 有一个单一的集中管理的版本控制管理服务器，保存所有文件的历史修订版本记录。
  - 团队成员之间的代码交换必须通过客户端连接到这台服务器，获取自己需要的文件。
  - 每个人如果想获得其他人最新提交的修订记录，就必须从集中式版本控制系统中获得。
  - 典型的集中式版本控制工具是Subversion。

集中式版本控制

- 集中式版本控制的问题
  - 尚未完成的工作难以保存临时版本
    - 要么提交到中央仓库（可能“污染”中央仓库的代码）
    - 要么本地另外找一个地方保存临时版本
  - 同步大量文件可能失败，导致状态不一致
  - 单点故障风险

DVCS：分布式版本控制

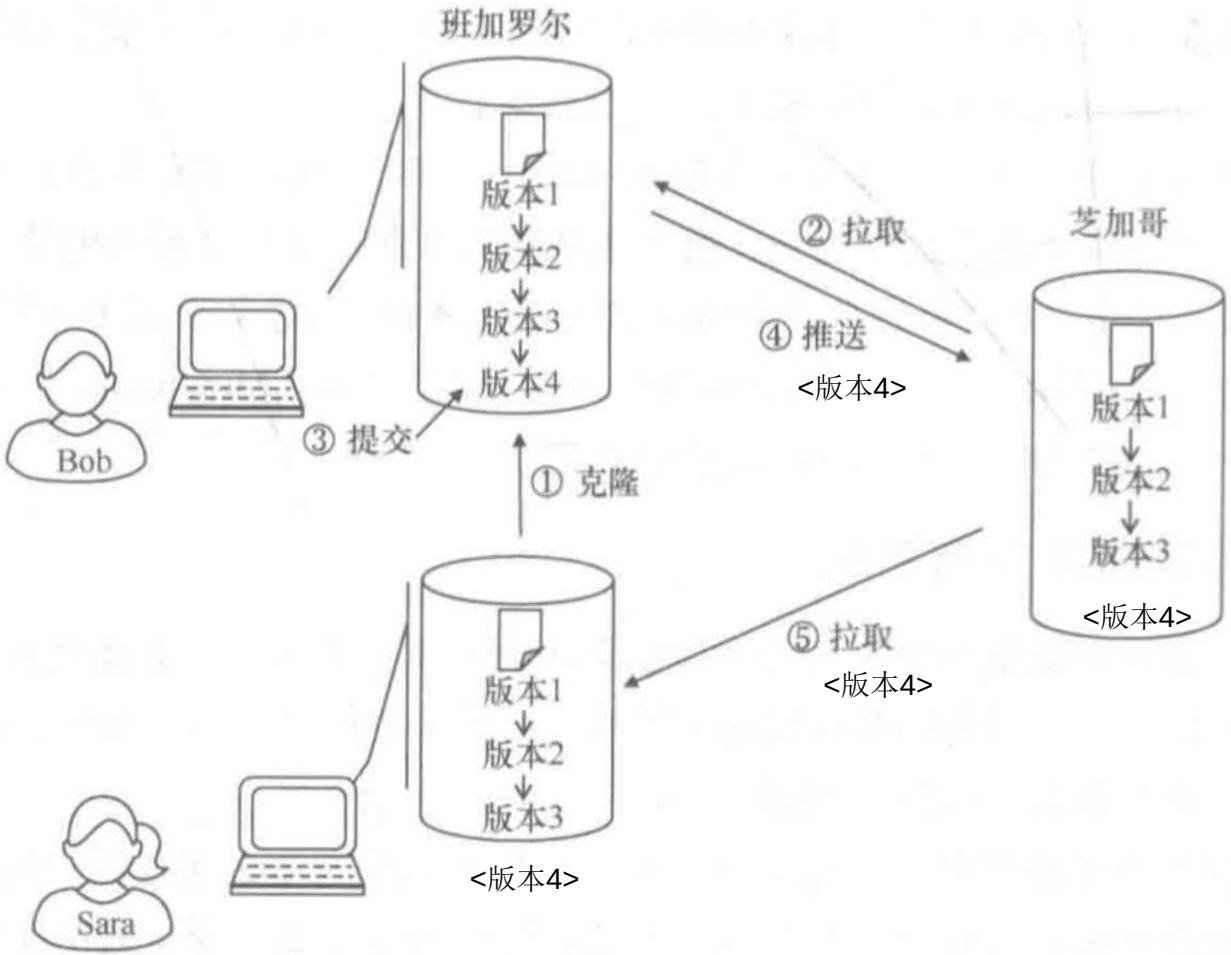
- 分布式版本控制系统（Distributed version control systems /DVCS)允许多个服务器共存
  - 所有的节点都是平等的
  - 团队协作中会指定某个节点作为中央服务器
  - 可以通过传输补丁(patch)完成分布的版本管理服务之间的同步，接受或拒绝补丁也更容易
    - “摘樱桃”(cherry-picking)
  - 在一个DVCS中，并没有一个单一的中心代码仓库，每个用户都有一份代码及代码变更历史的完整记录。
- DVCS的典型代表是Git。

DVCS：分布式版本控制

- 通过patch同步分布的代码
- 断网情况下不影响开发和提交
- 有多个代码库，DVCS具有更好的容错性
- 可以频繁提交至本地代码库作为检查点
- 如果在本地进行了频繁零碎的提交，那么在需要同步至其他代码库时，可以将零碎的提交重新组织，使得提交记录比较清晰（例如利用rebasing）

DVCS：分布式版本管理

- 也可以用来做某种程度的集中式的版本控制



DVCS：分布式版本管理

- 在上面的例子中

  ① Bob通过班加罗尔办公网络克隆（clone)一份Sara的代码仓库（Sara也在班加罗尔办公室）。

  ② Bob从芝加哥的中央仓库中拉取（pull)与本地仓库有差异的代码。

  ③ Bob修改代码文件，并提交（commit)到本地仓库，产生一个新的文件版本。

  ④ Bob将这个新的版本推送（push)至中央仓库。

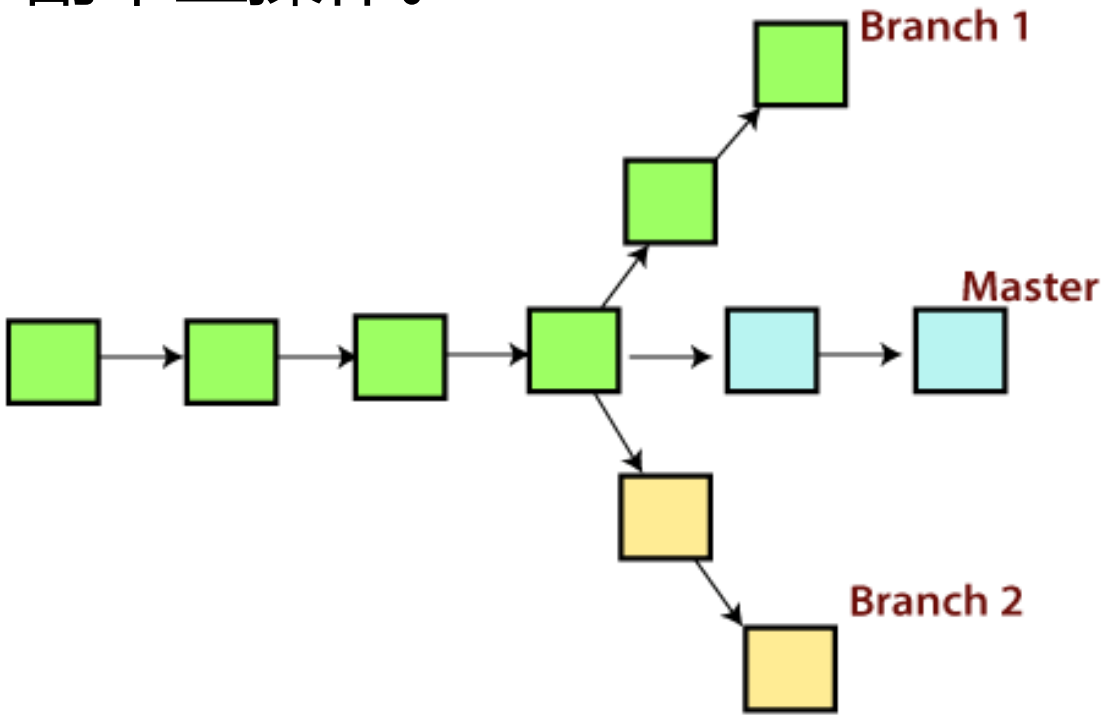
  ⑤ Sara即可从中央仓库拉取（pull)所有的差异代码。

大纲

- 版本控制系统概述
- 悲观锁与乐观锁
- 集中与分布式版本控制
- **分支与合并**
- 分支开发模式
- 在Git中管理分支与合并

分支与合并

- 在一个代码库(code base)上**创建分支**的能力是版本控制系统最重要的特性。
- 分支的作用：选定代码库作为源创建一个副本，之后在这个副本上操作。



注：→ 在这里看作时间先后。但在表示版本时，往往会倒过来，即指向前一个版本。

分支与合并

- 使用分支的原因
  - 物理上：因系统物理配置而分支
  - 功能上：因系统功能配置而分支
  - 环境上：因系统运行环境而分支
  - 组织上：因团队的工作分工而分支
  - 流程上：因团队的工作行为而分支
- 这些分支并不相互排斥

分支与合并

- 分支的优点在于在代码演进的过程中，可以认为各个分支完全独立，可以完全忽略其他分支的存在。
- 分支看上去能迅速避免一些问题，但使用不当会引发另外的问题。
  - 如何管理分支的内容
  - 如何管理分支的合并

分支与合并

- 分支管理不善的案例
  - 一个软件开发团队，为每一个客户构建一个分支，并有一套对分支的命名规则：w.x.y.z，其中，y对应了一个客户，z对应了一个构建
  - 测试团队开发的测试代码在另一个版本控制库中，没有与开发团队的分支保持一致
  - 导致的问题
    - 针对某个客户，每次构建都需要花大量的时间确定需要运行的测试用户
    - 即使测试代码与开发团队有相同的分支名称，甚至在同一个版本控制库中，**分支地狱问题**仍然存在

分支与合并

- 分支的另一个问题是：创建分支通常意味着需要合并，而这是一个容易导致混乱的过程。
  - 在现实中，除非是为了预研，验证而创建的分支，否则总是需要将这个分支的变更合并到另一个分支上。
  - 分支把问题**延迟**到了合并的时刻。

合并：可能的问题

- 合并可能带来如下的问题：
  - 修改对象的直接冲突：
    - 修改了同一行的内容，需要人工判定冲突，确定采用哪个版本的修改。

合并冲突

修改内容位置接近

```
<<<<<<< HEAD
if (carrierImsPackage != null && (SMSDispatcherUtil.isSupportHandleSmsBybinderService(mContext,
        carrierImsPackage))) {
        smsFilterPackages.add(carrierImsPackage);
||||||| merged common ancestors
if (carrierImsPackage != null) {
        smsFilterPackages.add(carrierImsPackage);
=======
if (imsRcsPackage != null) {
        smsFilterPackages.add(imsRcsPackage);
>>>>>>> origin/android-12.0.0_r2
```

git merge生成的冲突块中有大量的情况是由于无关修改位置接近导致的，如左图中，ours修改了if判断语句，theirs修改了add函数中的参数名称，实际上并没有冲突

目前，基本基于AST的工具都可以解决。

合并冲突

同一位置新增不同代码

```
<<<<<<< HEAD
/* DTS2017081607113  xiashaohua/00347572 20170924 begin */
        mPkgName = pic.getPkgName();
/* DTS2017081607113  xiashaohua/00347572 20170924 end */
        mDevice = getDevice();
    }
||||||| merged common ancestors
=======
        mSessionId = pic.mSessionId;
>>>>>>> cebf5c06997b64f4e47a1611edb5f97044509d76
```

同位置新增可以说是在冲突类型中占比最大的一种形式。

同位置新增可以分为两种：

1. 新增的内容之前不存在依赖，不会相互影响

左图中ours新增了两个变量，theirs新增了一个变量，三个变量之间不存在任何依赖，无所谓插入的顺序

除了新增部分体现的依赖，还应该

合并冲突

同一位置新增不同代码

```
<<<<<<< HEAD
    updateNativeAllocation();
    other.updateNativeAllocation();
||||||| merged common ancestors
=======
    mMultiResolutionStreamConfigurationMap = other.mMultiResolutionStreamConfigurationMap;
    updateNativeAllocation();
    other.updateNativeAllocation();
}
```

2. 新增内容中存在相同内容、相互依赖的内容

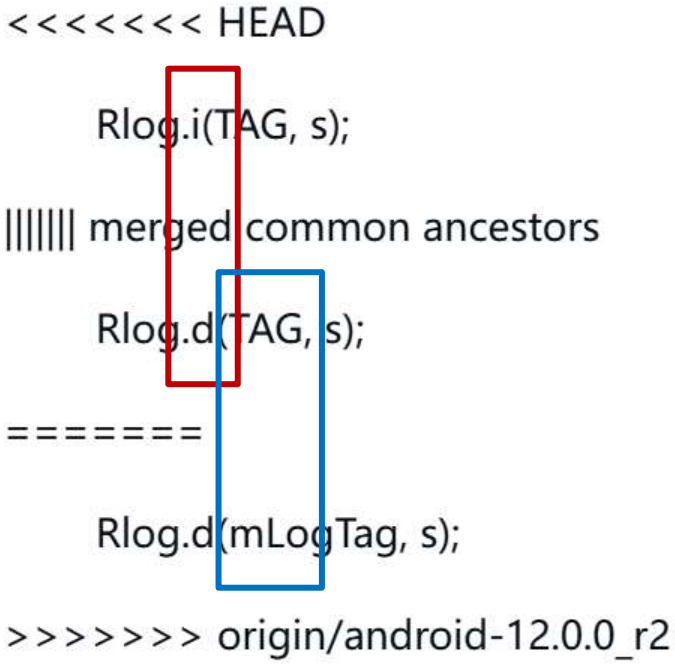
存在的内容中有相似内容，去重？选择某一个版本？

存在相互依赖的内容，顺序会影响顺序的行为逻辑

需要进一步分析内容，给出解决方式

合并冲突

同一位置修改不同，可同时保留，但是是否符合需求?



ours：修改了调用的函数；theirs：修改了参数

一般情况下选择同时保留两种变更，可以通过基于AST的合并算法实现。

虽然同时保留了两侧的修改，但是是否符合实际需求？

合并冲突

同一位置修改不同，但如何确定哪种正确？

```
<<<<<<< HEAD
private final LocalLog mSettingChangeLocalLog = new LocalLog(10);
||||||| merged common ancestors
private final LocalLog mSettingChangeLocalLog = new LocalLog(50);
=======
private final LocalLog mSettingChangeLocalLog = new LocalLog(32);
>>>>>>> origin/android-13.0.0_r1
```

ours与theirs对于同一个参数进行不同的修改，需要结合背景信息与修改者意图进行抉择。

LockLog类的作用，参数代表什么含义，修改成10与32处于什么样的考量

合并：可能的问题

- 合并可能带来如下的问题：
  - 修改对象的直接冲突：
    - 修改了同一行的内容，需要人工判定冲突，确定采用哪个版本的修改。
  - 即便是修改对象没有直接冲突，版本管理工具完成了自动合并，但是依然会出现：
    - 编译错
    - 运行错误
      - 没有自动测试会很容易让这种错误进入最终的产品
      - 隐含的功能错误

分支与合并

- 确定一种分支策略之前，需要确保有一个**合理的合并流程**来支持合并，确保合并是安全的，并且尽可能减少合并的代价。
  - 比如一个流程规定了主干上的代码由开发人员控制，而分支由运维人员创建，并经审核后合后才能合并回主干。
  - 后续会看到更多的分支模式

创建分支的模式

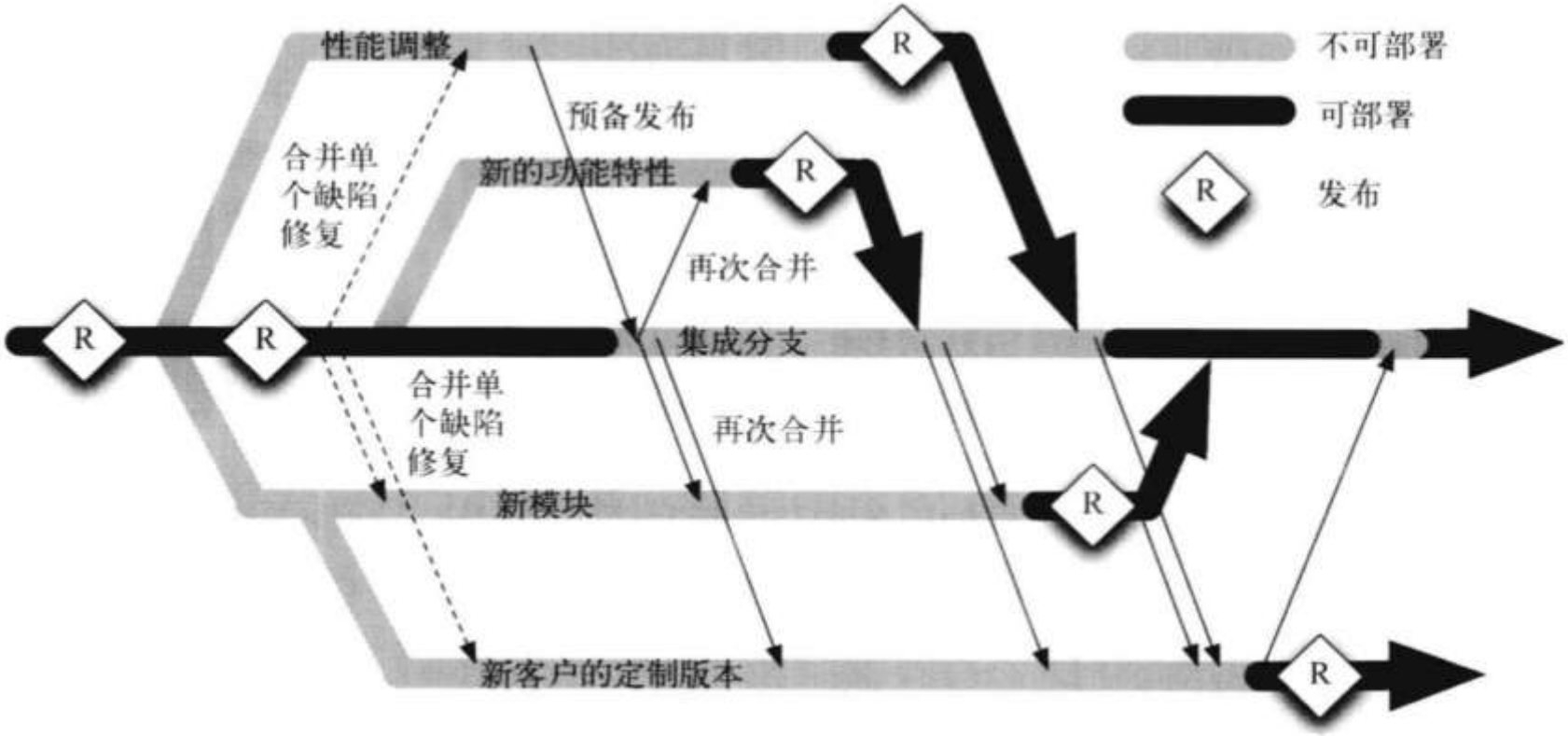
- Early Branching
  - 对每一个新的功能，一开始就创建一个独立的分支。
  - 在新功能完全完成之后，或者是系统发布之前，合并至主线。
  - 问题：需要更多的工作来跟踪所有的分支，合并的痛苦只是被延迟。
- Deferred Branching
  - 尽可能在主线上开发，只有在发布的时候创建日后维护该发布的分支。

分支与持续集成

- **分支**与**持续集成**之间存在着一种矛盾的关系
  - 采用不恰当的分支策略，对导致大量的分支。
  - 如果一个团队的不同成员持续较长时间在不同的分支上工作，那就不是在做持续集成。
  - 除非每天都对分支进行一次合并。

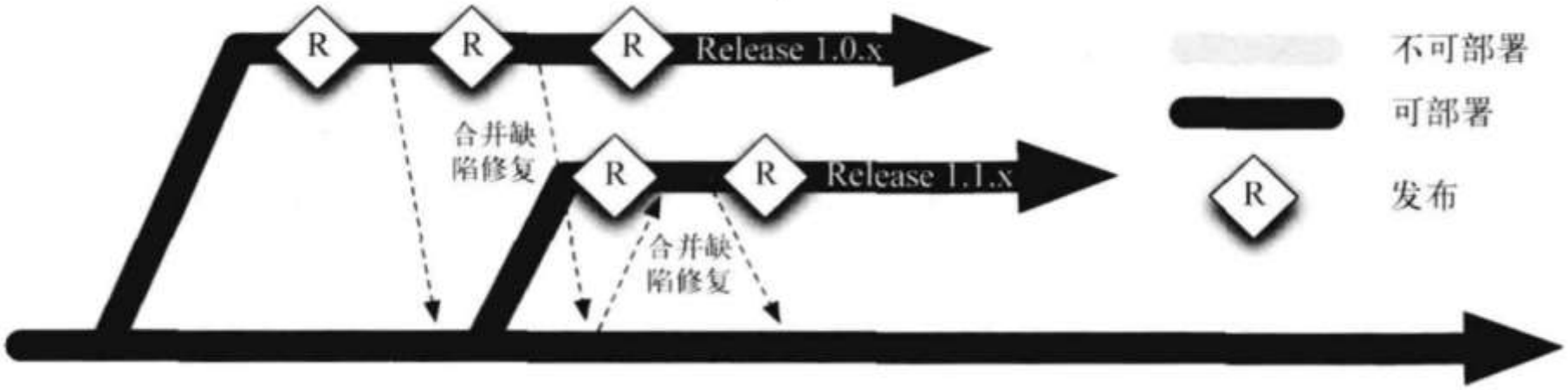
分支与持续集成

- 下面是一个分支管理失败的案例：
  - 大量的、长时间游离的分支，导致不可部署的情况较多



分支与持续集成

- 下面就是一个可控的分支策略
  - 因为及时进行合并，几乎没有不可部署的情况



大纲

- 版本控制概述
- 悲观锁与乐观锁
- 集中与分布式版本管理
- 分支与合并
- **分支模式**
  - 主干开发，主干发布
  - 主干开发，分支发布
  - 分支开发，主干发布
  - 三驾马车分支模式
  - Gitflow分支模式
  - GitHubFlow分支模式
- 在Git中管理分支与合并

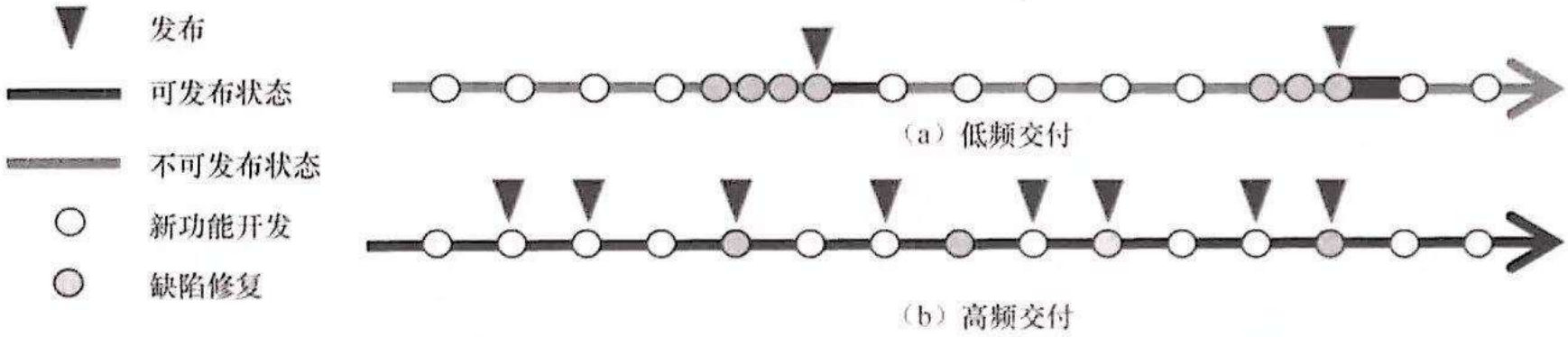
常见的分支开发模式

- 主干开发，主干发布
  - Trunk-based Development & Release
- 主干开发，分支发布
  - Trunk-based Development & Branch-based Release
- 分支开发，主干发布
  - Branch-based Development & Trunk-based Release

主干开发，主干发布

“主干开发，主干发布”是指工程师向主干上提交代码（或者每个分支的生存周期很短，如数小时，或少于1天），并用主干代码进行软件交付。

- 所有新特性的开发，代码均提交到主干（trunk)上
- 当需要发布新功能时，直接将主干上的代码部署到生产环境上。



主干开发，主干发布

- 有两种场景
  - 低频交付
    - 常见于一些周期比较长的大型软件开发项目。
    - 在低频工作模式下，其主干代码总是长时间处于不可用的状态
    - 版本控制系统的作用仅仅是确保代码不丢失，用作代码备份仓库。
  - 高频交付
    - 高频交付通常每天都会发布一次，甚至更多
    - 高频交付需要比较完备的交付基础设置，包括：
      - 自动化配置构建，自动化测试，自动化运维，自动化监控与报警

主干开发，主干发布

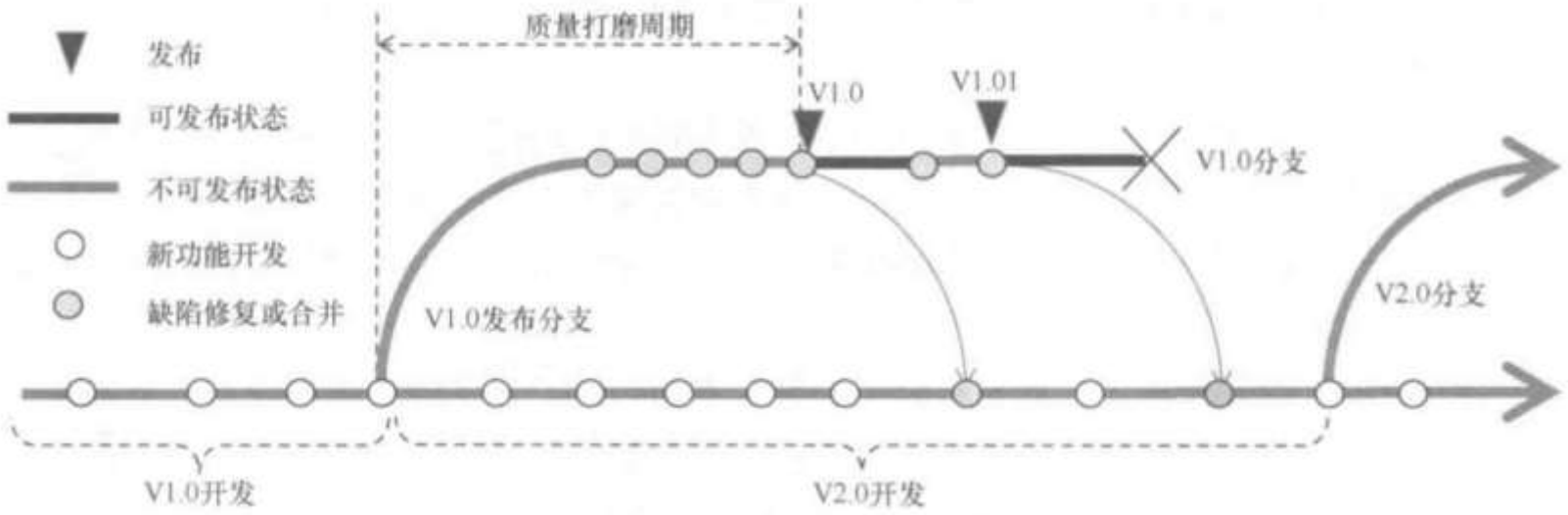
- 优点
  - 分支方式简单，分支管理的工作较少。
- 缺点
  - 对于低频模式：系统经常性处于不可发布状态；项目后期缺陷修复阶段，并不是所有团队成员都需要做缺陷修复，会有一定的资源浪费
  - 对于高频模式，主干代码变动非常快，导致开发人员每天从主干向本地合并代码的工作量较大。但如果不做每日更新，那么可能造成本地与主干上的代码差异过大而无法再合并回去。
    - 高频模式下，未完成的功能也应被允许提交到主干中，前提是不影响正常使用和发布
      - 需要特定的技术管理手段（比如开关技术或抽象分支）

主干开发，主干发布

- 缺点案例：无法完成的“合并任务”
  - 2011年，百姓网（一个生活分类服务网站）的研发团队只有12名工程师。他们使用高频交付模式，每天早上7点做一次生产环境发布。为了对某个重要模块进行重大重构，其技术负责人曾经创建了一个专有分支。然而，一周以后，他不得不宣布放弃该分支的所有代码，因为其他工程师在主干上已经做了太多的改动，专有分支已经无法合并回主干。

主干开发，分支发布

- 开发人员将写好的代码提交到主干
- 当新版本的功能全部开发完成或者已经接近版本发布时间点时，从主干上拉出一个新的分支
- 在这个新的分支上进行集成测试，并修复缺陷，进行版本质量打磨。当质量达标后，再对外发布该版本。



主干开发，分支发布

- 这种模式的特点如下：
  - 主干代码提交活动频繁，对保障主干代码质量有较大的挑战
  - 分支只修复缺陷，不增加新功能
  - 新版本发布后，如果发现严重缺陷，而且必须立即修复的话，只要在该版本所属的分支上修复后，再次发布补丁版本，然后将分支上的修改合并回主干
  - 也可以在主干上修复缺陷，然后将针对该缺陷的修复代码挑出来（cherry-pick）合并到该缺陷所在的分支上

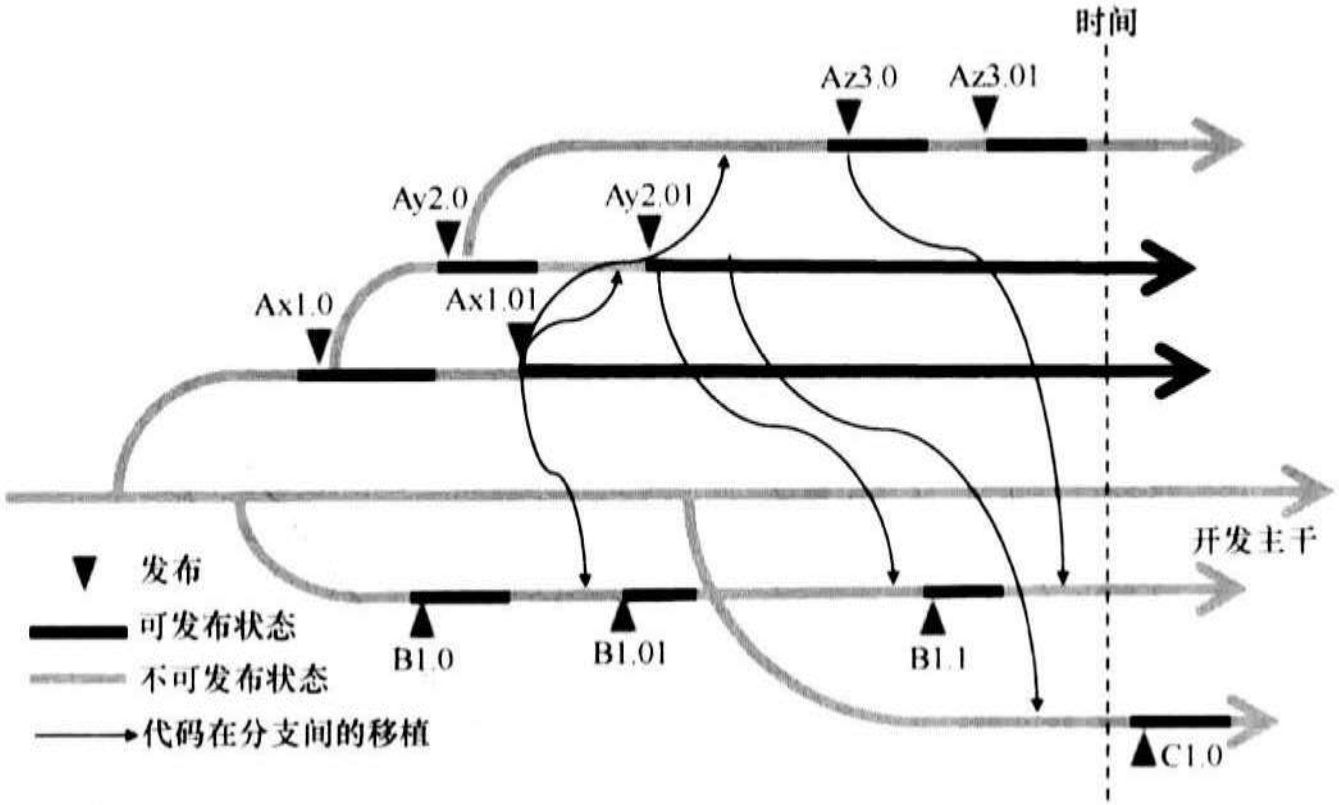
主干开发，分支发布

- 优点
  - 与将要发布的新功能无关的人员可以**持续工作在开发主干**上，不受版本发布的影响；
  - 新发布的版本出现缺陷后，可以直接在其自己的版本发布分支上进行修复，简单便捷。即使当前开发主干上的代码已经发生了较大的变化，该分支也不会受到影响。

主干开发，分支发布

- 缺点

  使用这种开发模式，对发布分支的数量不加约束，并且分支周期较长，很**容易出现“分支地狱”**倾向，这种倾向常见于“系列化产品簇 + 个性化定制”的项目，例如某硬件设备的软件产品研发的分支模式



主干开发，分支发布

HP LaserJet Firmware team（2008年）	
时间占比	工作任务
10%	代码集成
20%	做详细计划
25%	在分支间移植代码
25%	已发布产品的技术支持
15%	手工测试
约5%	新功能开发

- HP激光打印机部件团队2008年研发资源分布
  - ¼的时间在分支间移植代码

分支开发，主干发布

- 分支开发，主干发布是最为广泛的工作方式

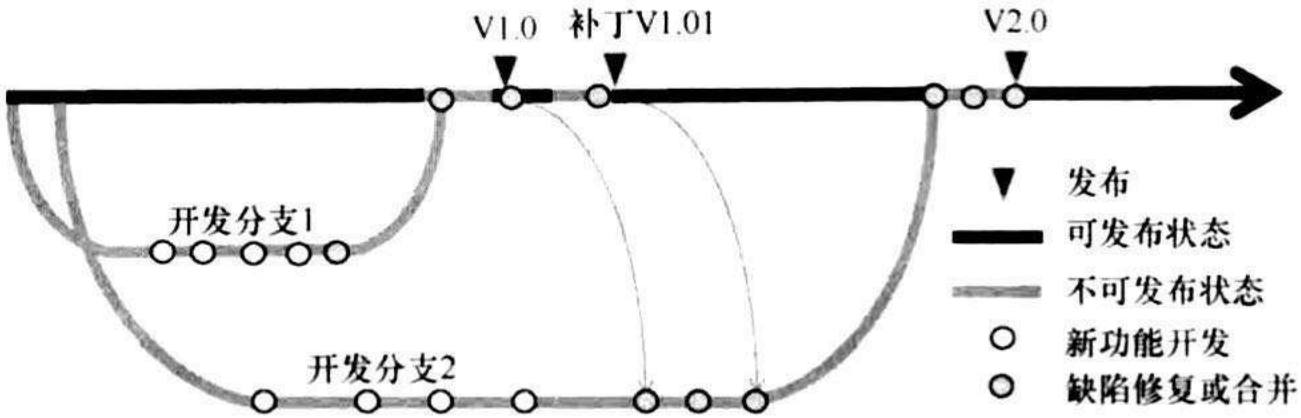


图8-8 “分支开发，主干发布”模式

分支开发，主干发布

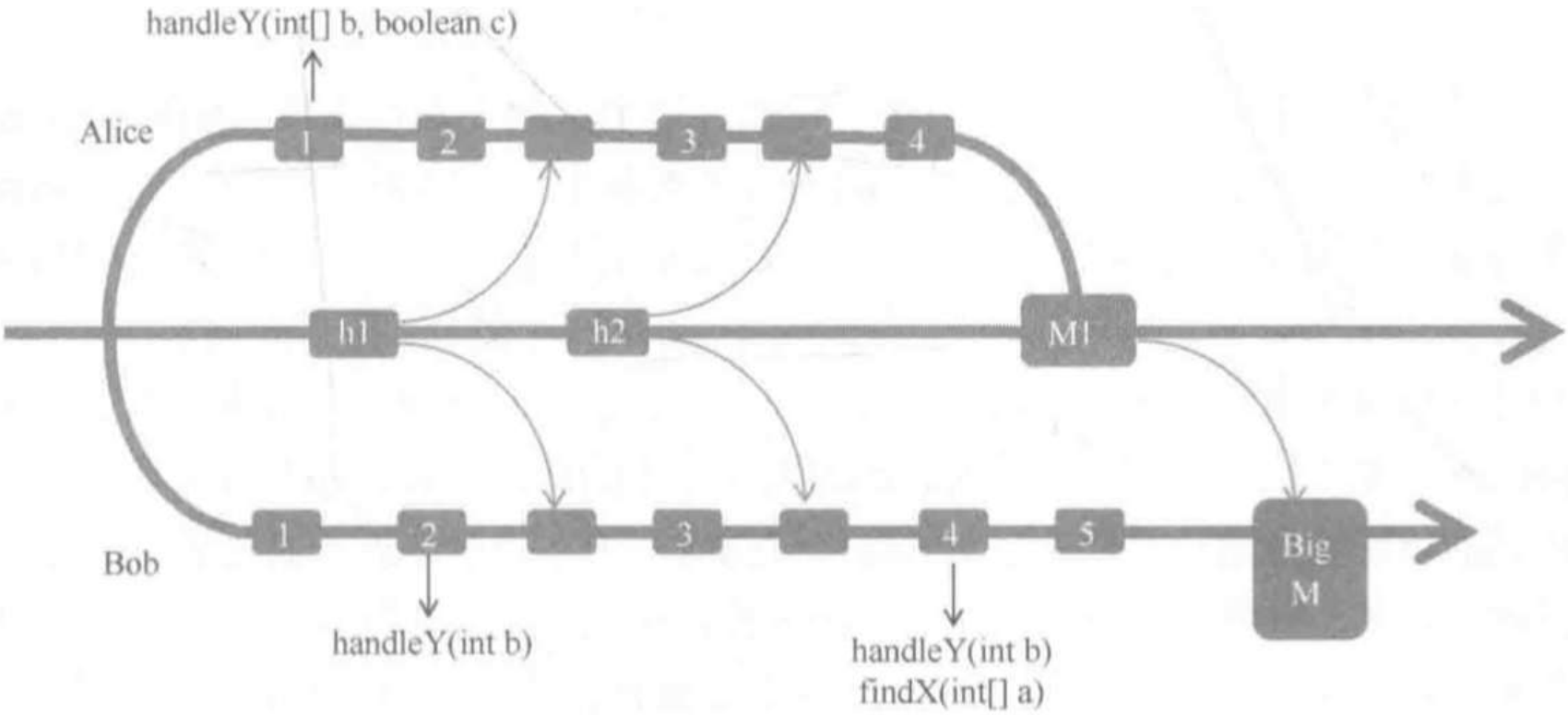
- 这种模式是指:
  - 团队从主干上拉出分支，并在分支上开发软件新功能或修复缺陷
  - 当某个分支(或多个分支)上的功能开发完成后要对外发布版本时，才合入主干
  - 通常在主干上进行缺陷修复，质量达标后，再将主干上的代码打包发布。

分支开发，主干发布

- 优势
  - 在分支合并之前，每个分支之间的开发活动互相不受影响
  - 团队可以自由选择发布哪个分支上的特性
  - 如果新版本出现缺陷，可以直接在主干上进行修复或者使用hotfix分支修复，简单便捷，无须考虑其他分支
- 劣势
  - 为了分支之间尽量少受影响，开发人员通常会减少向主干合并代码的频率（**危险！**）
  - 分支过多，分支的生命周期过长，代码合并成本会快速增加

分支开发，主干发布

- 不频繁提交导致巨型代码合并



分支开发，主干发布

- 成功使用这种模式的关键点在于：
  - 让**主干**尽可能一直保持在**可发布**的状态
  - 让每个**分支**的生命周期**尽可能短**
  - 主干代码**尽早**与分支**同步**
  - 一切**以主干代码为准**，不要在分支之间合并代码

分支开发，主干发布

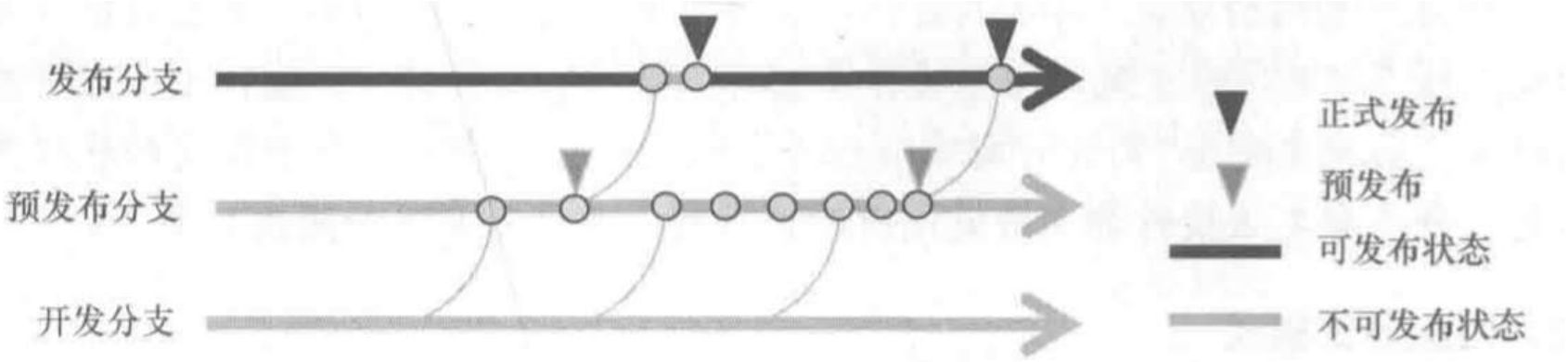
- 特性分支与团队分支
  - 大多数情况下，“分支开发、主干发布”模式中的分支指的是**特性分支**。但是如果特性分支过多，会带来合并成本，多个特性同时完成的情况下，合并更加具有挑战性。
  - 对于大型的开发团队，可以考虑**团队分支**，一个团队负责了一系列特性的多个组件的开发。这样可以减小与主干合并的压力。
- 在实践中，“主干开发”是初级团队的首选
  - 特性分支开发需要严格遵守前一页提到的规则
  - “按特性拉分支”与持续集成之间存在难以调和的关系（Martin Fowler）

分支模式的演化

- 三驾马车分支模式
- Gitflow分支模式
- GitHubFlow分支模式

三驾马车分支模式

- 软件开发团队仅维护3个分支
  - 开发分支
    - 开发人员的目标分支
  - 预发布分支
    - 从开发分支上选取的一组特性准备发布。只做缺陷修复。（Alpha版本、Beta版本）
  - 发布分支
    - 预发布分支上功能稳定后，合入发布分支（RC版本，正式版）



Gitflow分支模式

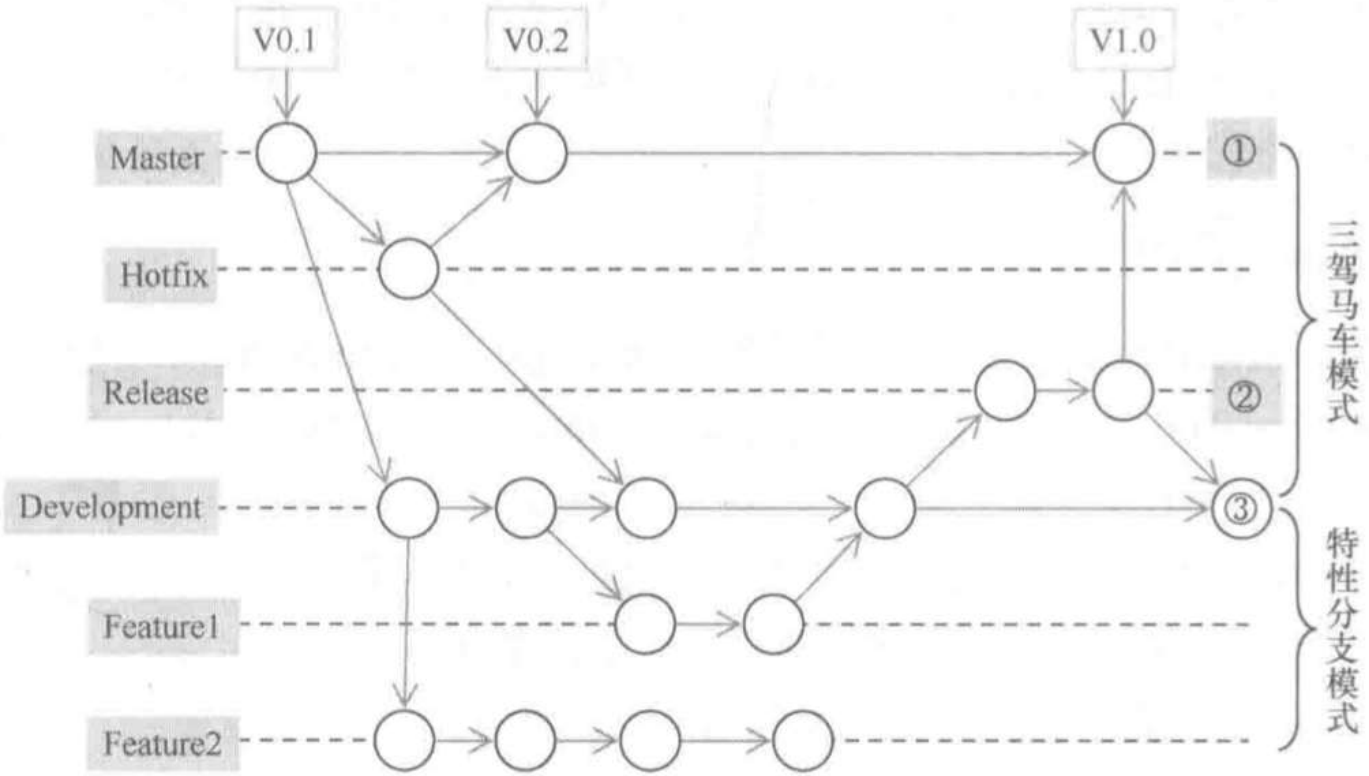
- Gitflow是目前很多企业所应用的分支模式，包括下面类型的分支：
  - Master
    - 正式版本的发布分支。
  - Release
    - 预发布分支。如果Release分支的质量达标，就可以将其合入Master分支，同时也需要将代码合入Development分支。
  - Development
    - 对新功能进行集成的分支。

Gitflow分支模式

- Feature
  - 为了开发某一功能特性，开发人员从Development分支上拉出的分支。当特性开发完成后，合入Development分支。
- Hotfix
  - 如果已经发布的版本(如V0.1)出现了严重的缺陷，从Master分支上V0.1版本标签处拉出Hotfix分支，与此同时，由于Development分支上也有同样的缺陷存在，因此开发人员还要将Hotfix分支的代码移植到Development分支上，以修复Development分支上的缺陷。

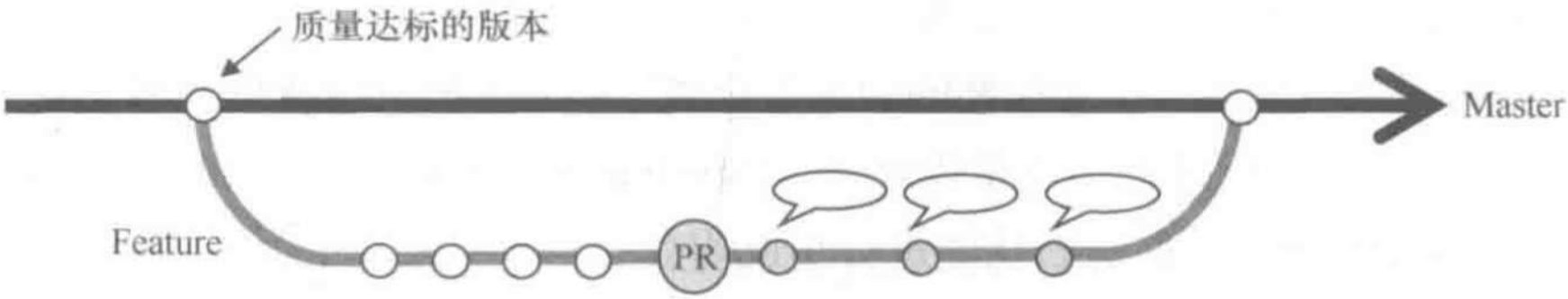
Gitflow分支模式

- 特性分支与三驾马车的组合
  - 分支职责清晰，但数量多，需要注意避免特性分支的不足



GitHubFlow分支模式

- GitHubFlow这种开发分支模式的名称来源于GitHub团队的工作实践。工作步骤：
  - 从Master上创建一个新的分支，以这个特性或缺陷的编号命名该分支
  - 在这个新创建的分支上提交代码
  - 功能开发完成，并自测通过，创建Pull Request (简称为PR)
  - 其他开发人员对这个PR进行审查，确认质量合格后，合入Master



- 如果特性分支存在时间很短可以认为是高频的“主干开发、主干发布”

大纲

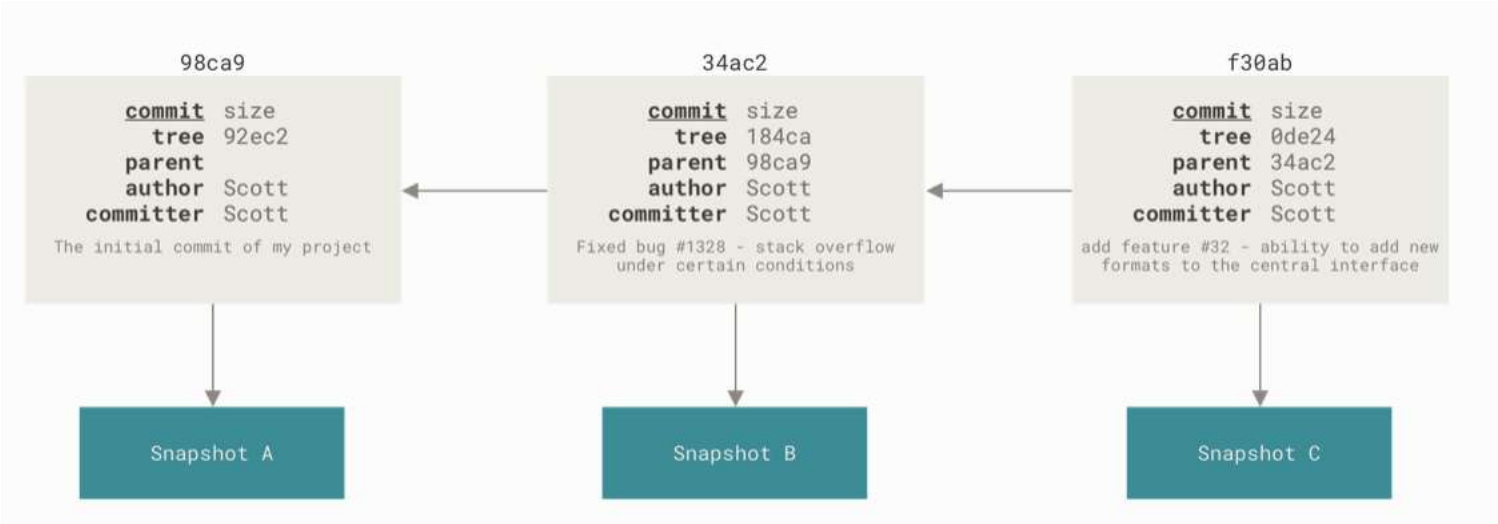
- 版本控制概述
- 悲观锁与乐观锁
- 集中与分布式版本管理
- 分支与合并
- 分支模式
- **在Git中管理分支与合并**
  - 分支管理
    - 创建分支
    - 使用cherry-pick管理分支
  - 合并管理
    - 管理合并：Merge
    - 管理合并：Rebase

在Git中管理分支与合并

- **分支管理**
  - 创建分支
  - 使用cherry-pick管理分支
- 合并管理
  - 管理合并：Merge
  - 管理合并：Rebase

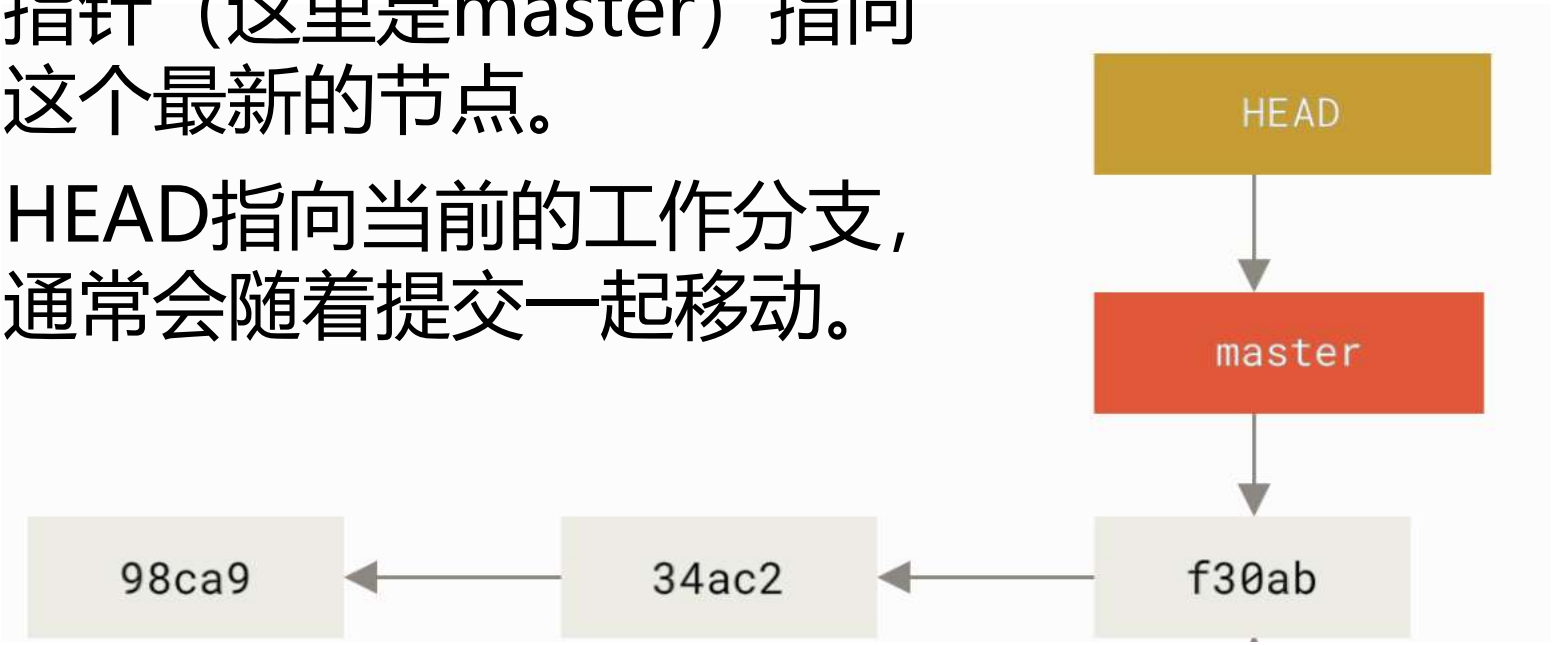
在Git中管理分支与合并

- Git中管理着一系列的提交(commit)，每个提交指向了一个资源集的快照(Snapshot)。在没有分支的情况下（或者说，只有一个默认分支），这些提交构成了一个前后衔接的链：



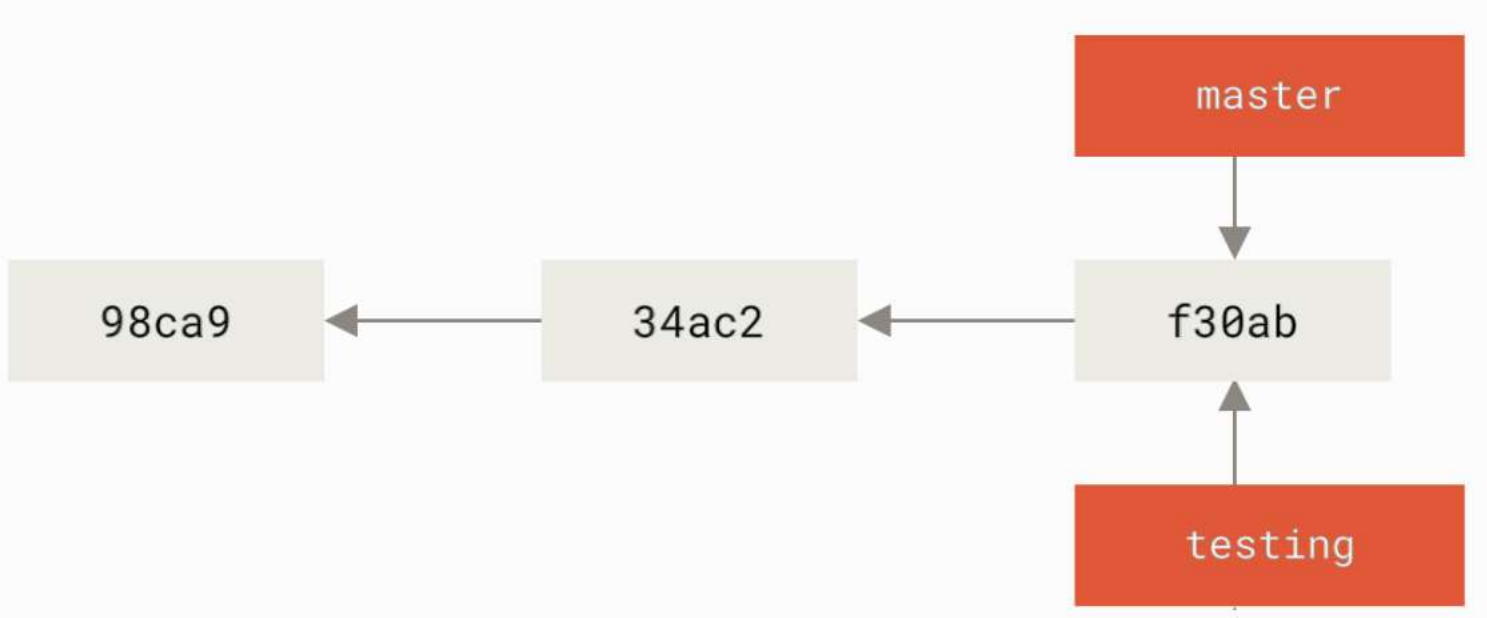
在Git中创建分支

- Git在初始化仓库的时候，给这条默认的链一个名字：master。一个分支其实就是一个指针，指向了这条链最新的commit。
- 每次提交会生成一个新的commit，并将该分支的指针（这里是master）指向这个最新的节点。
- HEAD指向当前的工作分支，通常会随着提交一起移动。



在Git中创建分支

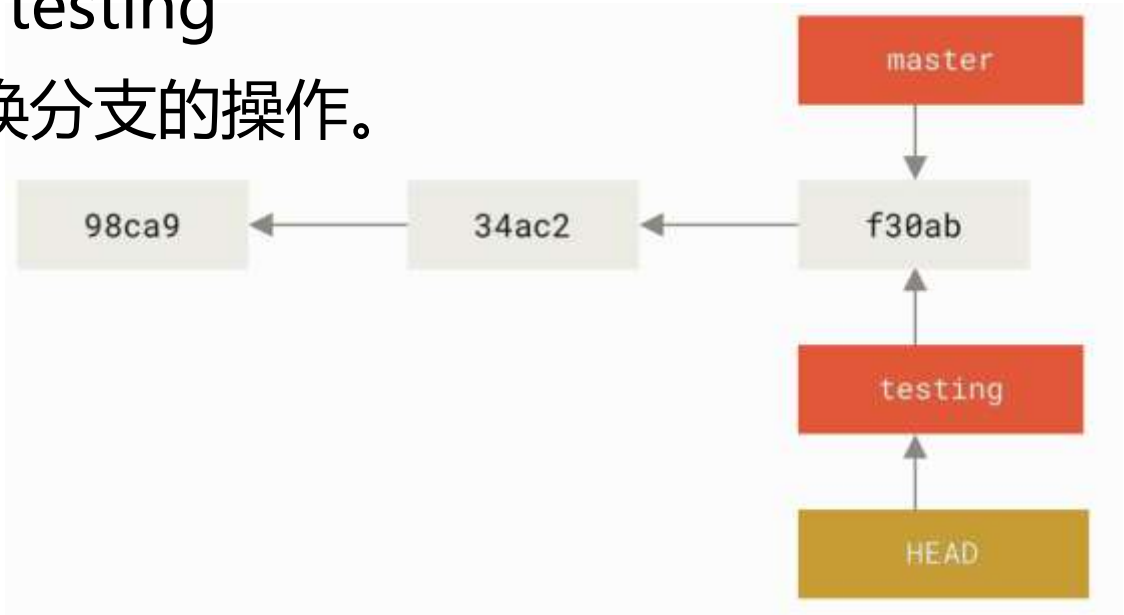
- 如果要创建一个新的分支，比如说 testing，可以使用Git命令：
  - git branch testing
- 这时候会产生一个新的分支，这实际上是一个指向某个提交的指针。



在Git中创建分支

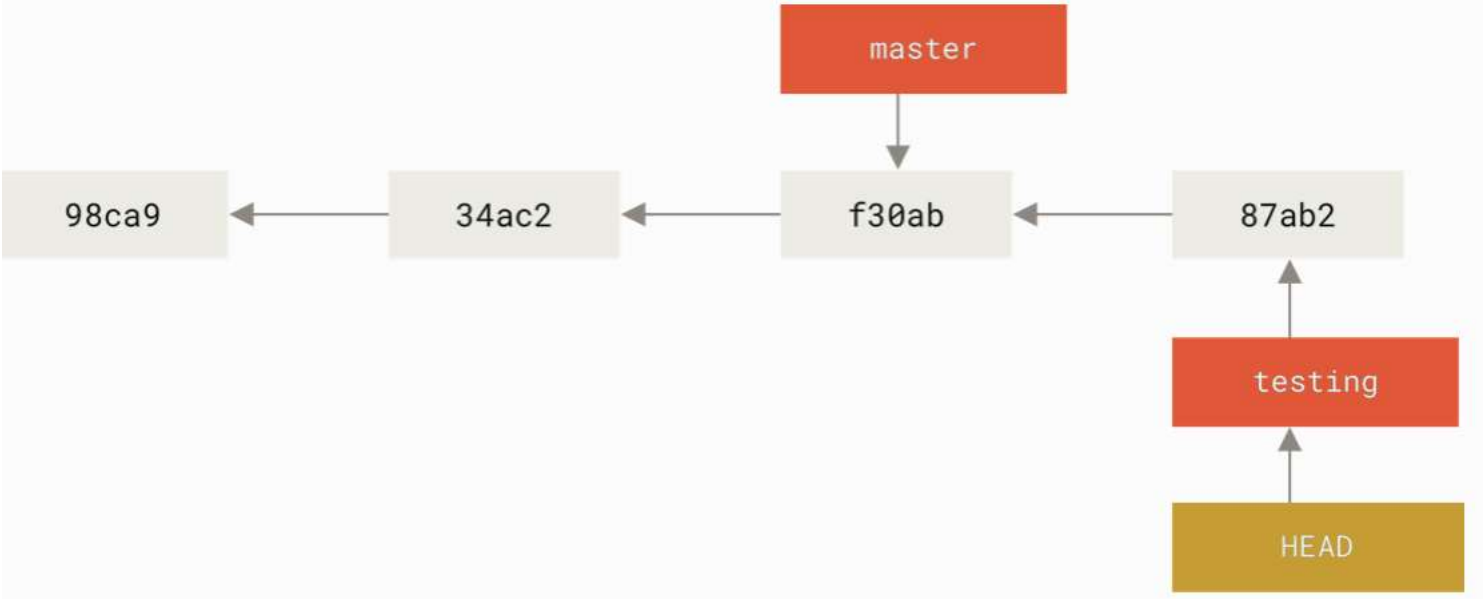
- 可以使用
  - git checkout testing
- 此时工作分支就切换为 testing，接下来所有的操作，包括提交，都工作在testing分支上。
- 可以使用
  - git checkout –b testing

  直接完成创建和切换分支的操作。



在Git中创建分支

- 在testing分支上提交一次后，仓库的结构变为如下形式：



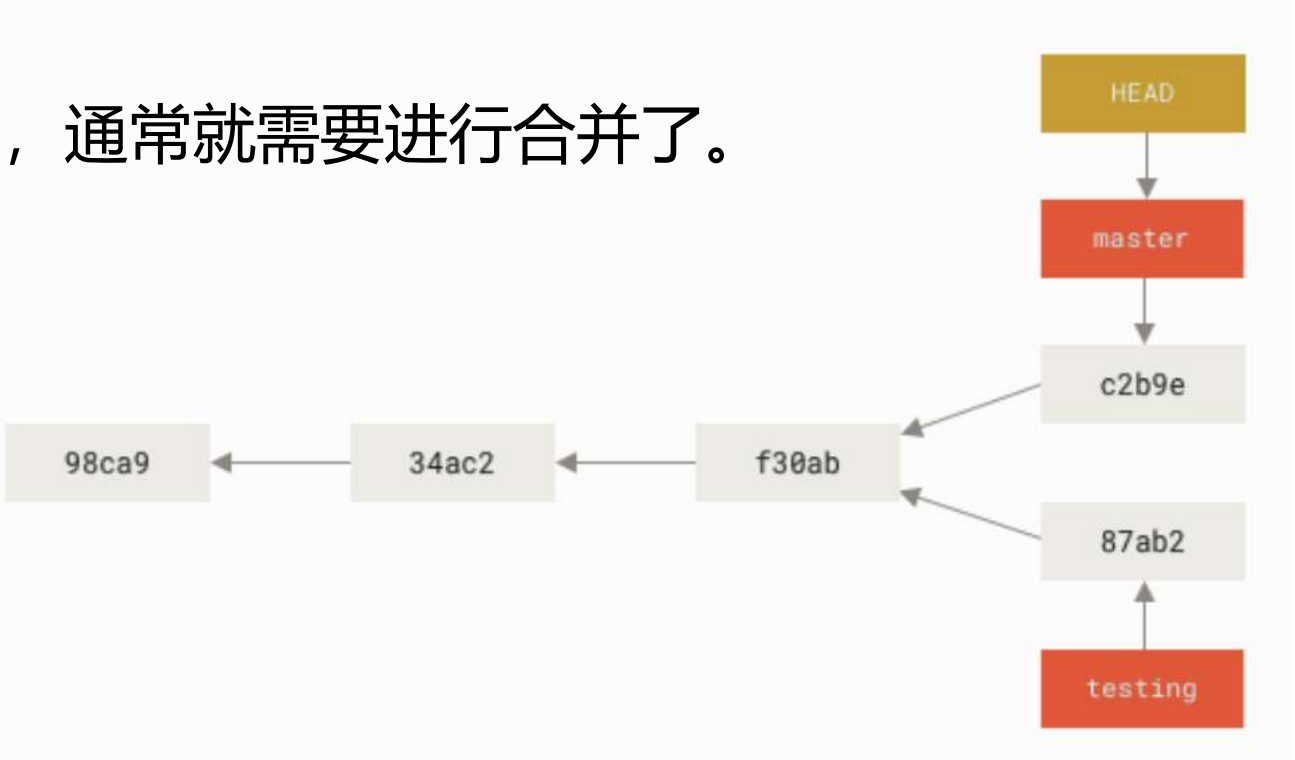
- 这时两个分支就产生了分离

在Git中创建分支

- 可以使用
  - git checkout master

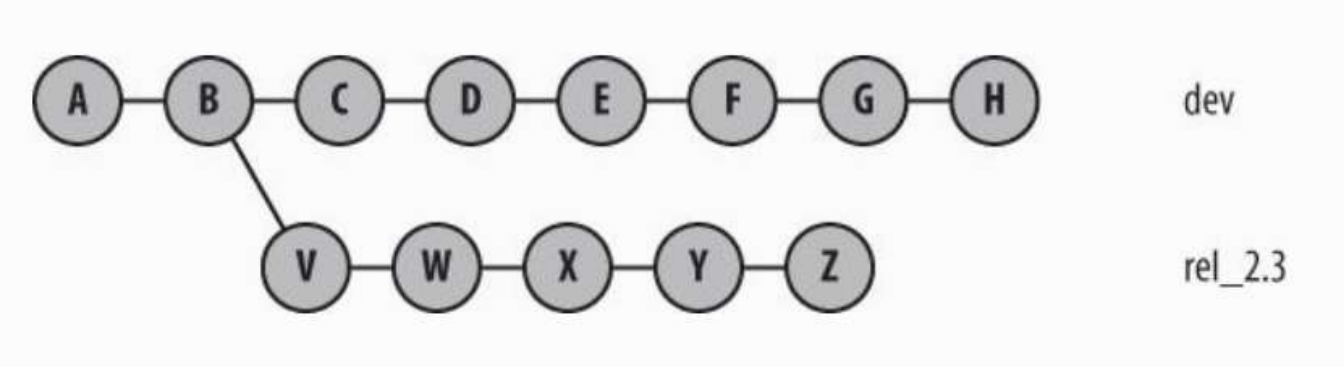
  重新切换到master分支，然后在master上继续提交，此时分支的结构变为如下的形式：

  这种情况下，通常就需要进行合并了。



在Git中使用Cherry Pick管理分支

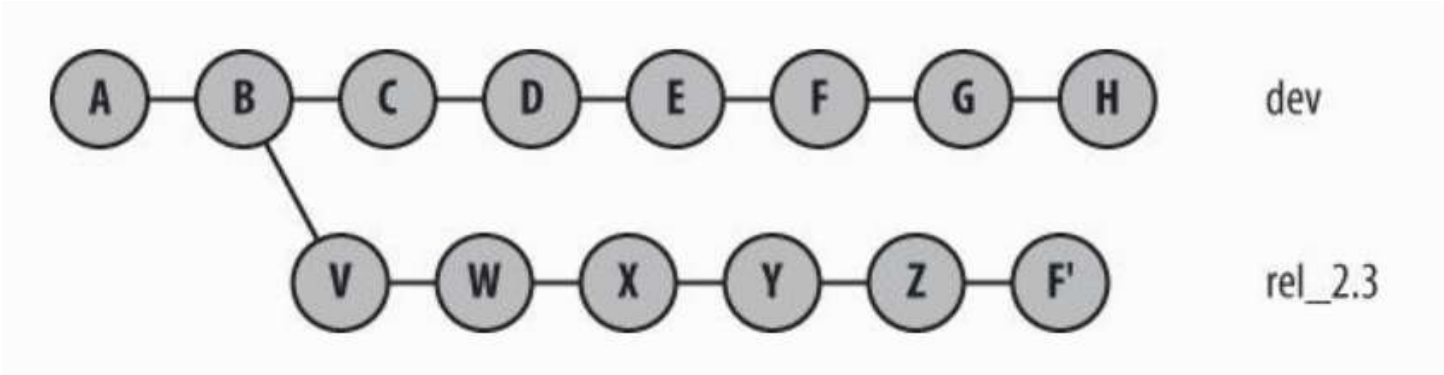
- 假设有下面两个分支，一个是开发分支，一个是发布分支



- dev分支中的提交F是一个公共的bugfix，需要反映到rel_2.3中，而其他的提交不需要反映到rel_2.3中。

在Git中使用Cherry Pick管理分支

- 此时我们可以使用cherry-pick命令：
  - git checkout rel_2.3
  - git cherry-pick dev~2 //dev~0为H，dev~2为F
- 此时结构如下：



  - F’ 为将dev中的F应用到rel_2.3的结果。当然可能需要手工解决冲突。

在Git中管理分支与合并

- 分支管理
  - 创建分支
  - 使用cherry-pick管理分支
- **合并管理**
  - 管理合并：Merge
  - 管理合并：Rebase

在Git中管理合并

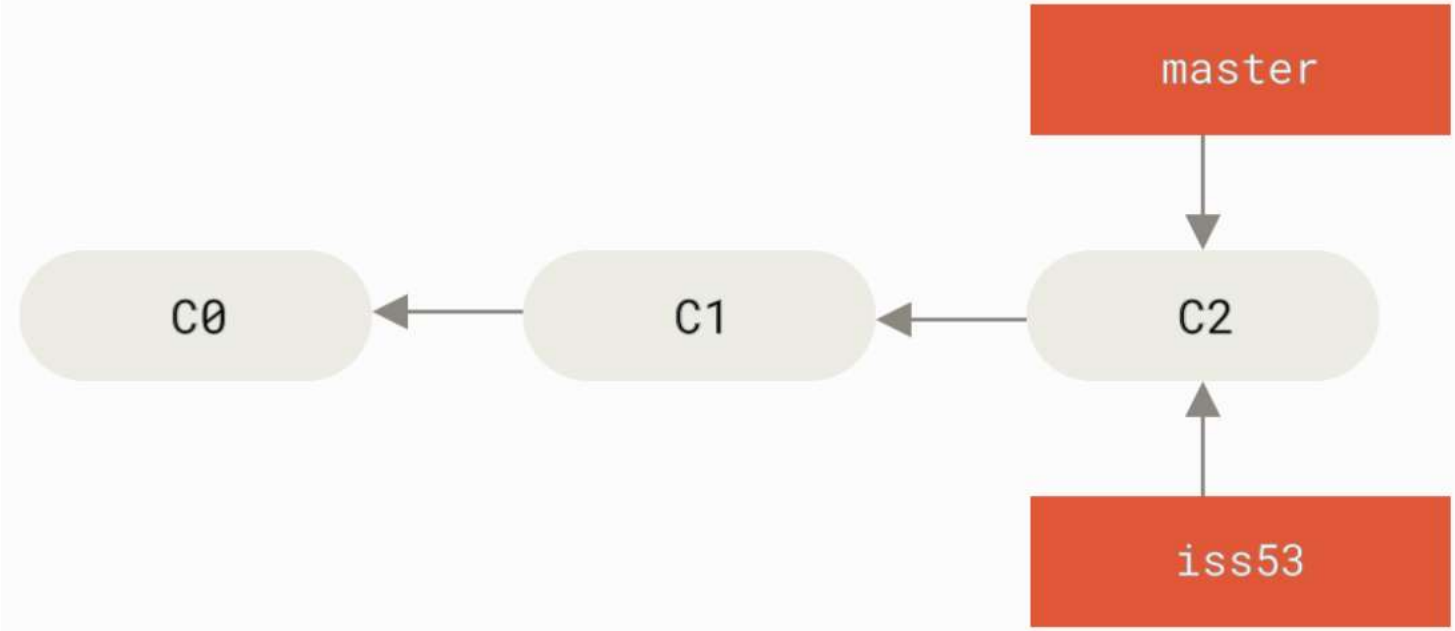
- 假设当前的仓库结构如图：

- 考虑下面的场景：
  - master为主线。需要针对issue 53开发一个名为iss53的PR，为此创建了一个名为iss53的分支，完成后需要合并到master
  - 在开发iss53的过程中，客户发现了一个bug需要紧急修复，为此创建了一个名为hotfix的分支，完成后也需要合并至master

在Git中管理合并

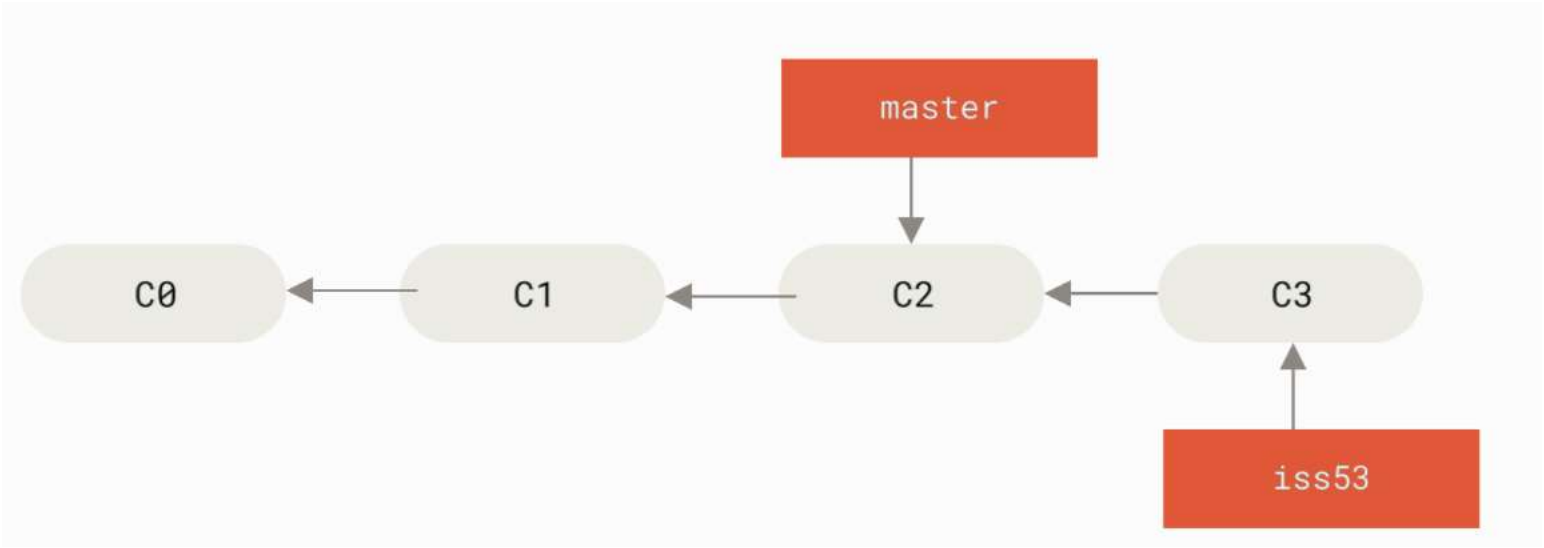
- 创建iss53的分支

```
$ git checkout -b iss53
Switched to a new branch "iss53"
```



在Git中管理合并

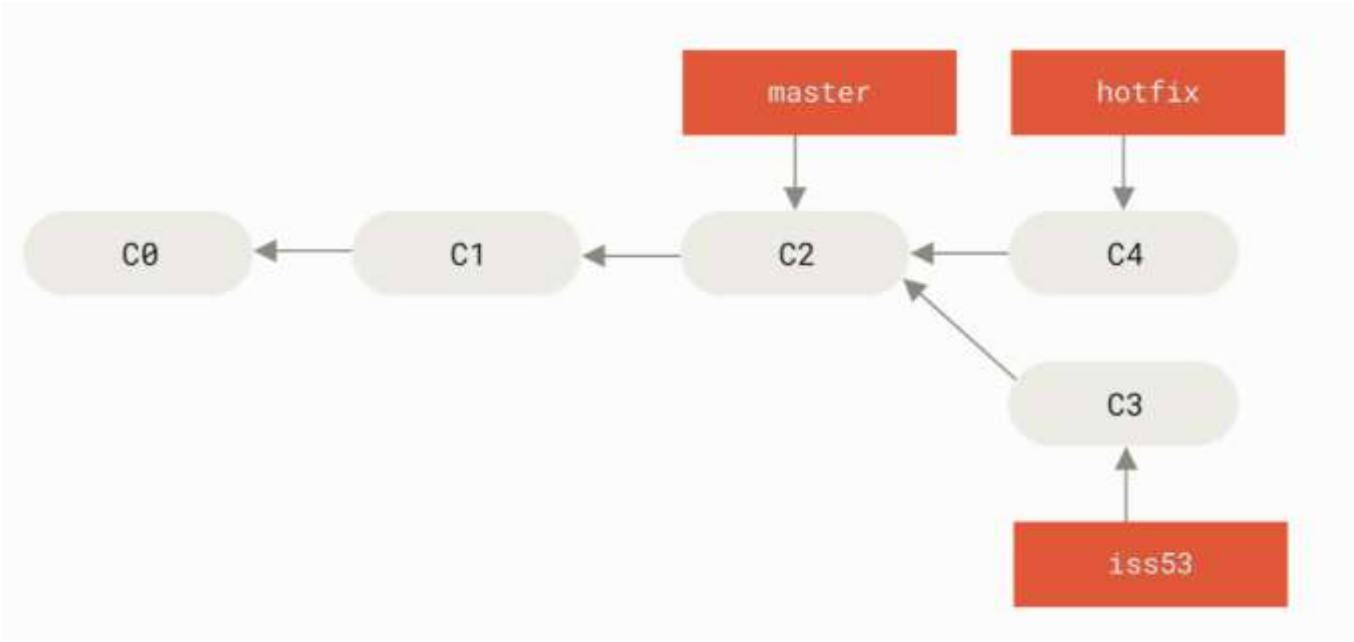
- 为修复issue 53修改代码，并提交iss53
  - 但修复工作还未完成



在Git中管理合并

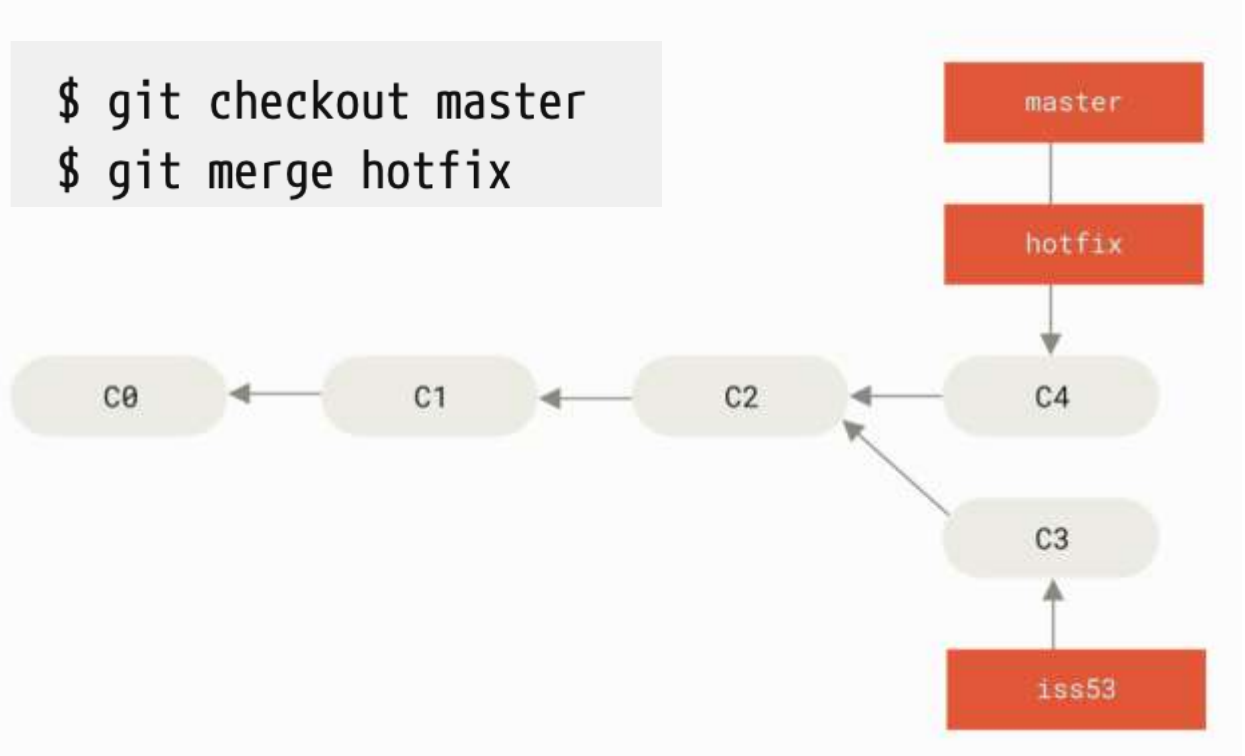
- 需要在主线上完成一个紧急修复。此时iss53还未完成。

```
$ git checkout master
$ git checkout -b hotfix
$ git commit -a -m 'Fix broken email address'
```



在Git中管理合并

- 下面有了三个分支。
  - 首先需要将hotfix与master合并，完成紧急修复。
    - 此时master实际上是只要改一个指针就好（快进）



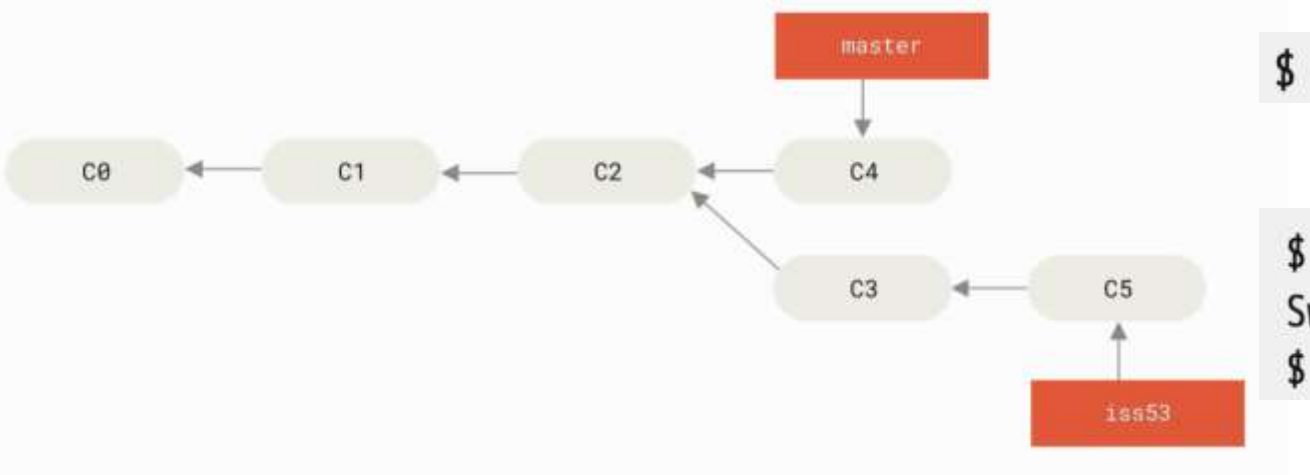
在Git中管理合并

- 如果不想留下太多的支，可以把hotfix分支删除

```
$ git branch -d hotfix
```

在Git中管理合并:Merge

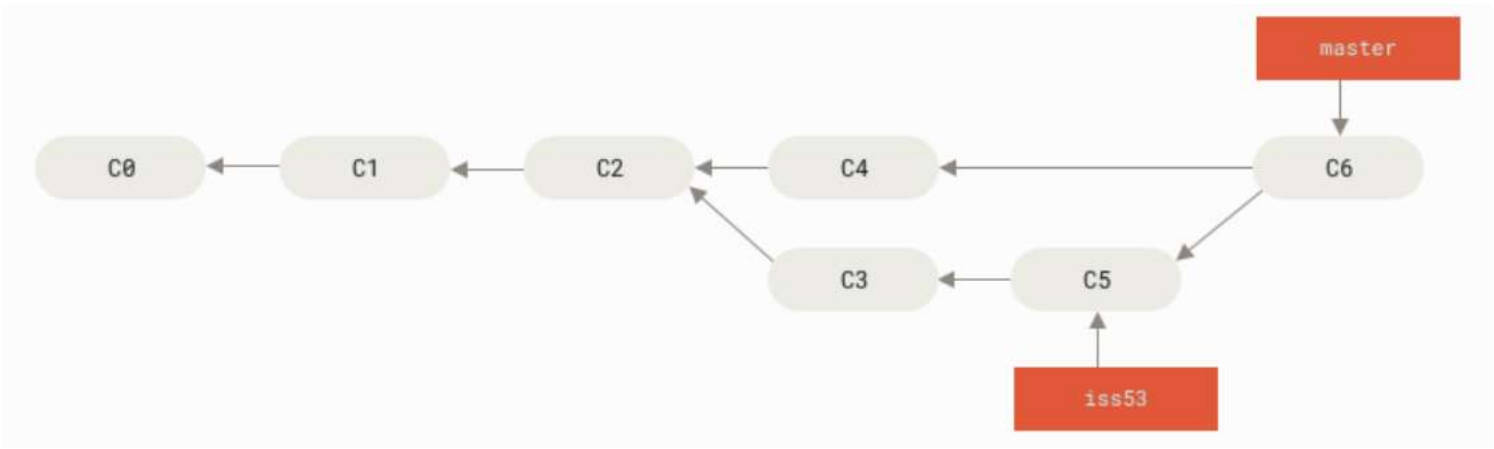
- 回到iss53分支继续开发，在完成了iss53后需要将这个分支也合并到master上。




```
$ git checkout iss53
```

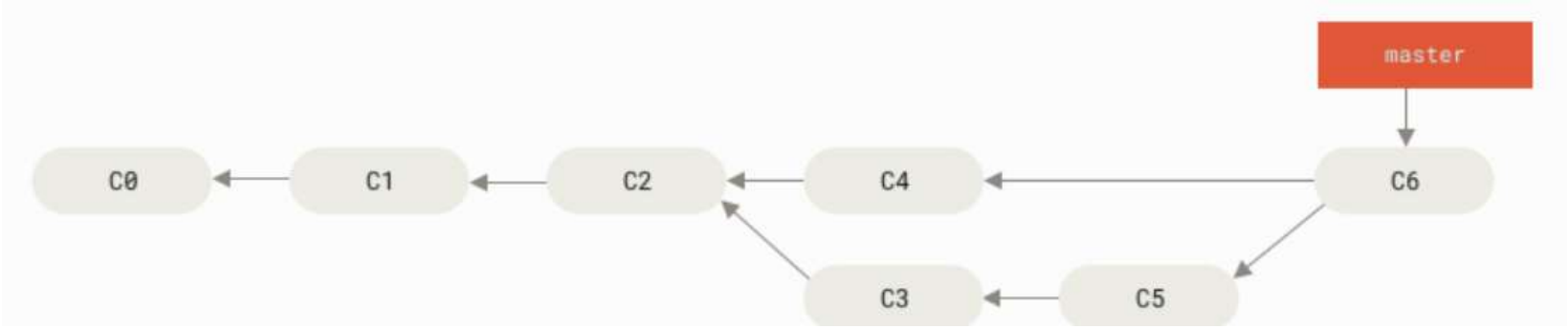
修改并提交...

```
$ git checkout master
Switched to branch 'master'
$ git merge iss53
```



在Git中管理合并Merge

- 可以将iss53这个分支也删除，删除后成为如下的结构：

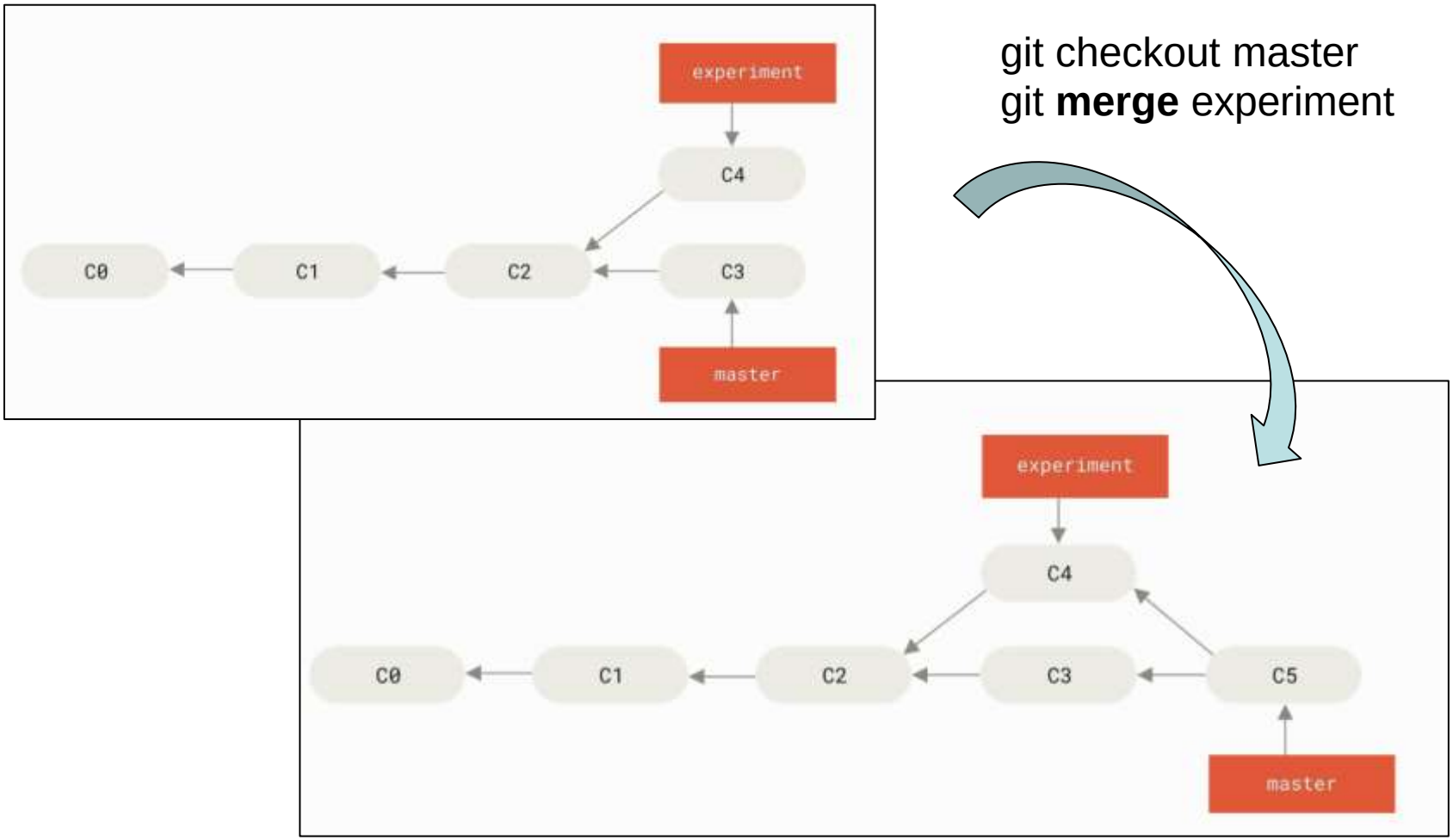


在Git中管理合并：Rebase

- 在开发过程中频繁使用分支与合并，会导致：
  - 一个分支的各个提交的关系构成复杂的依赖图，导致依赖关系不清晰。
  - 某开发人员在本地分支上可能进行一次或多次提交，合并到远程分支上后，导致远程分支提交链过长，提交之间的依赖关系变得不清晰。
- 为了让提交关系变得简单，Git提供了rebase作为合并的另一个方式

在Git中管理合并：Rebase

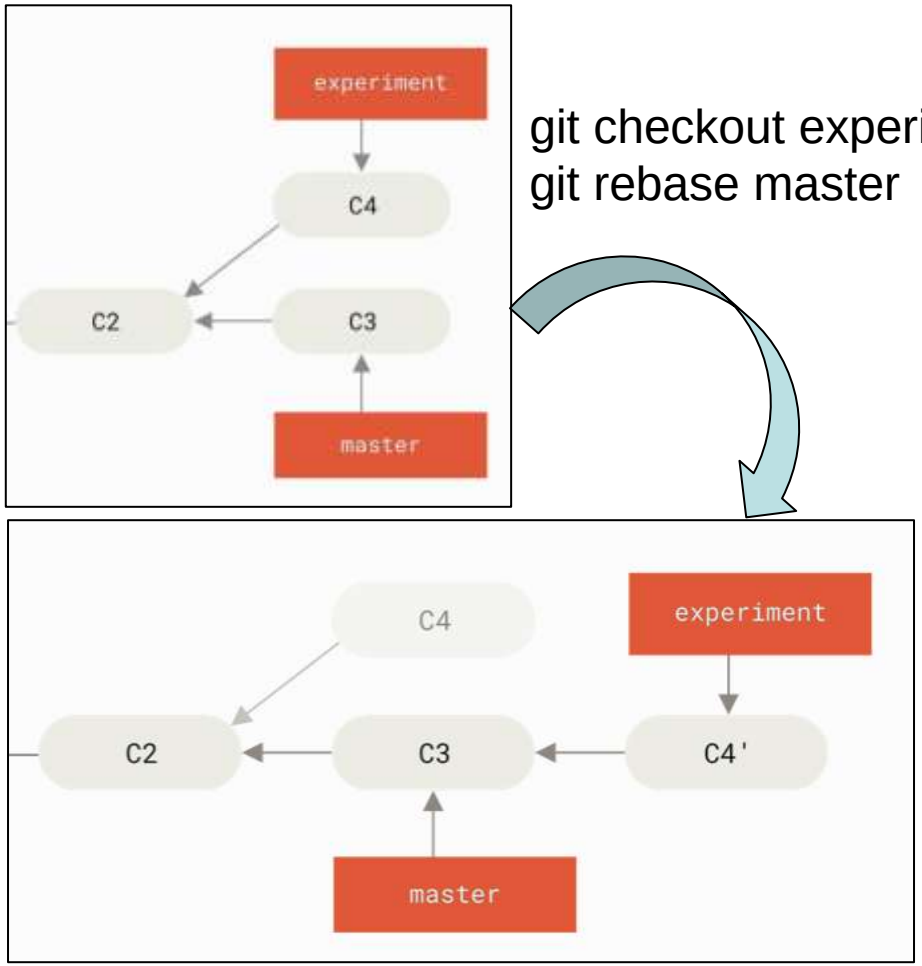
- 使用merge，对下面experiment和master两个分支合并：



在Git中管理合并：Rebase

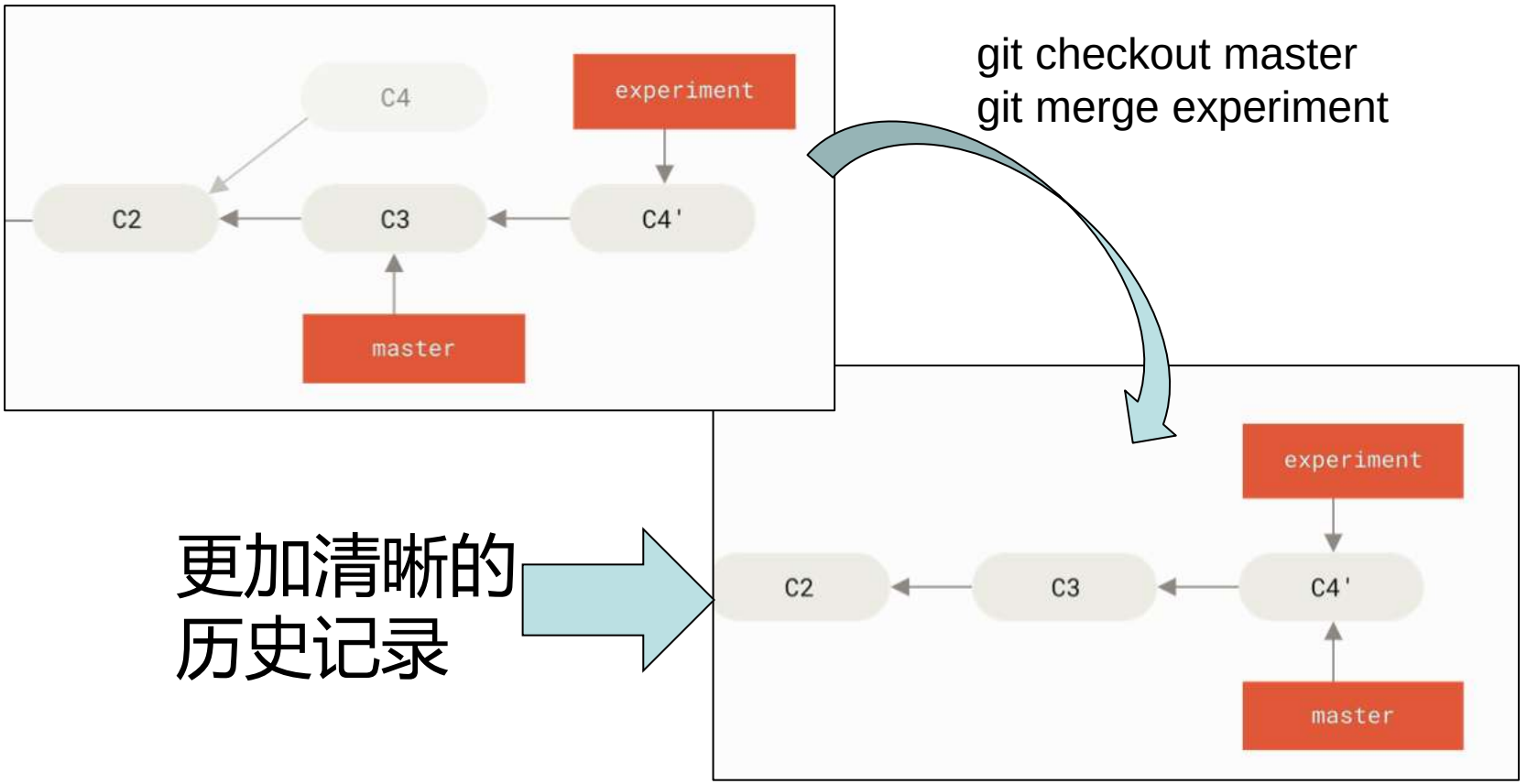
- Rebase的效果是，将两个分支以线性的结构组织起来。

- 在这种情况下，可以认为experiment是在master的最新commit之后才开始的。



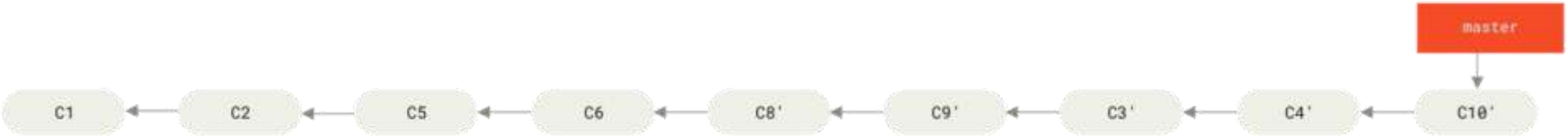
在Git中管理合并:Rebase

- 在待合并分支上进行rebase之后，再回到目标分支上进行merge



在Git中管理合并：Rebase

- 如果在本地修改的时候，管理本地的提交记录，做了多次提交



- 但是在最后需要与远程分支合并的时候，希望某些提交合并为一个提交，此时可以用rebase –i命令

  git rebase -i <after-this-commit>

  将指定的commit后的所有提交合并为一个。

后续课程安排

- 开启第二轮分组实践讨论
  - 2周分批现场分组讨论（5月9日，5月16日）
  - 目标：完成第一轮迭代（实现开发任务细分，确认并通过第一轮验收测试用例），启动第二轮迭代（初步确定第二轮验收测试用例）
    - 更新DevCloud或所采用CI环境上的任务进度
    - 代码持续提交更新
      - 最终需要提交整个代码仓
- 后续初步安排
  - 最后一次理论课（5月23日）
  - 两周期末报告，每组12分钟（5月30日，6月6日）

课程项目实践

- 代码托管
  - 规范提交，分支策略等
- CI服务器编译构建
  - 自动运行测试，测试失败则构建失败
- 部署和发布
  - 根据应用类型建立部署任务
- 创建流水线
  - 添加任务，手工或者自动执行
- 持续迭代
  - 管理需求完成情况

课程项目实践

- 用户故事的验收测试用例
  - 场景：正常预约（预约时段晚于当前时间，且座位可用）
    - **给定**用户A，座位S，且座位S的占用时段为空，且当前时间为2023年4月6日8:00
    - **当**A预定S，预定时间段为2023年4月6日10:00-11:00
    - **那么**预约成功
  - 场景：预约时间冲突
    - 测一下所选座位的整段预约时间已经被他人约过
    - 测一下所选座位的部分预约时间已经被他人约过
    - 测一下所选座位的预约时间与自己已经约过的时间有交集
    - ……
- 单元测试+集成测试