版本控制系统 Git

版本控制系统

版本控制系统 (Version Control Systems, VCSs) 是一类用于追踪源代码（或其他文件、文件夹）改动的工具。顾名思义，这些工具可以帮助我们管理代码的修改历史；不仅如此，它还可以方便团队协作编码。

例如，在实际项目开发中的一些场景：

- 需要查看当前版本和上一版本之间的**差异**
  - 更细节的：这个文件的这一行是什么时候被编辑的？是谁作出的修改？修改原因是什么？
- 当前版本上线出现严重 bug，需要**回退**到上一版本
  - 还需要知道：在哪一个版本快照导致了单元测试失败？

版本控制是项目管理的一大关键。实践中，我们会使用最常用的版本控制工具 Git 来管理代码：它是一个**分布式版本控制系统**，客户端并不只提取最新版本的文件快照，而是把代码仓库完整地**克隆**下来，包括完整的历史记录。

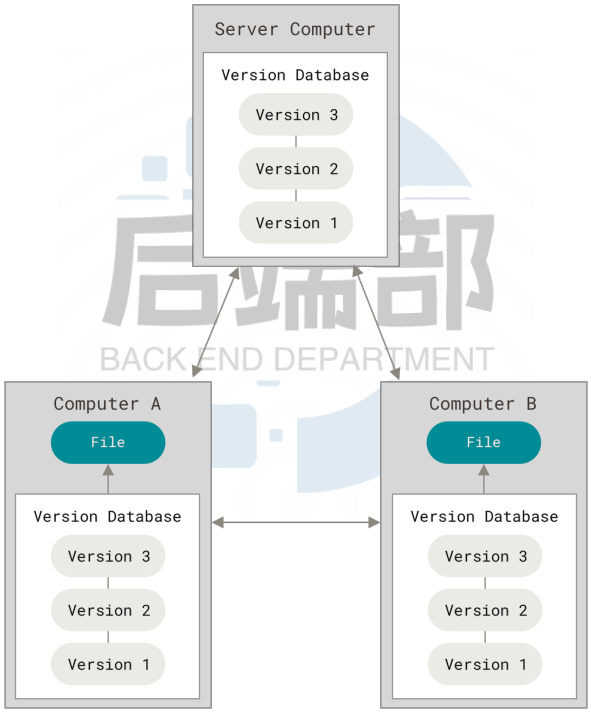
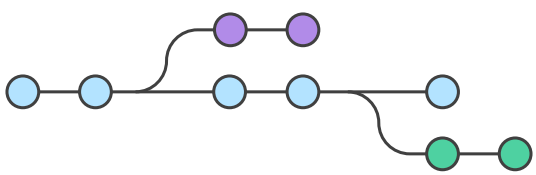


Figure 1: 分布式 VCS 的一个优势是，就算中心服务器发生故障，事后都可以用任何一个镜像出来的本地仓库恢复数据库历史。

你可以指定和不同的远端仓库交互，在同一个项目中，分别和不同的人相互协作。Git 的**分支**功能非常强大，它支持非线性的开发流程，并且能够有效地管理像 Linux 内核这样庞大的项目。



总而言之，假若你将来的工作和计算机沾边，Git 绝对是你离不开的开发工具。

Git 的实现

Git 的后端实现相当优雅，虽然在理解上确实存在一定的复杂度。我们会简单讲解它的原理，以便大家能够真正了解 Git 的相关概念。如果你对这部分不感兴趣，可以转至下一小节。

怎样记录版本差异: 记录快照

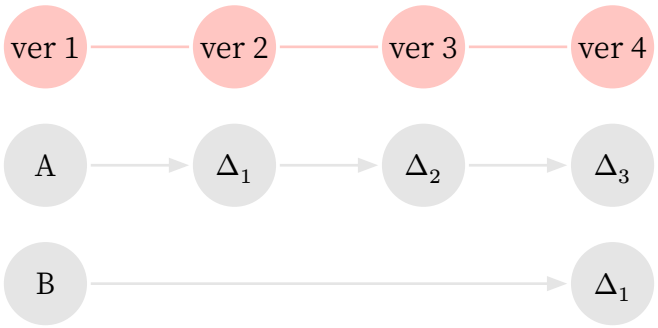


Figure 3: 基于**差异** (delta-based) 的版本控制

Git 不存储每个文件与初始版本的差异，假设你的改动是添加了一行文本，它并不会记录这行新增的文本——它只存储文件的**快照**，每当你提交更新或保存项目状态时，它基本上就会对当时的**全部文件**创建一个快照（副本）并保存这个快照的**索引**。

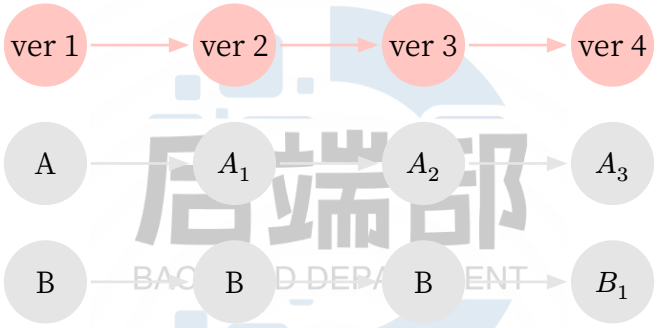


Figure 4: 存储随时间改变的快照

给每个版本做快照显然很耗空间！对此，Git 的优化是：如果文件没有修改，就不再重新存储该文件，而是只保留一个**指针（链接）**，指向之前存储的文件。

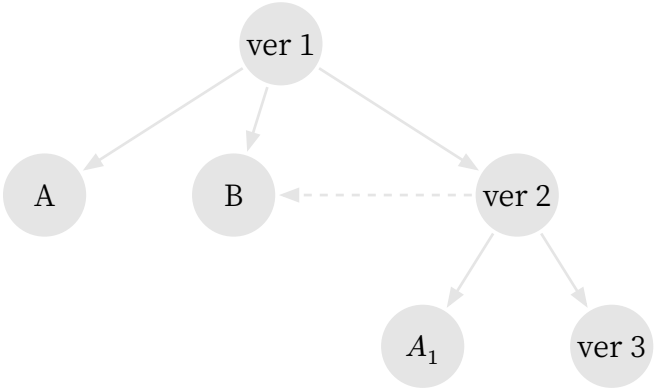


Figure 5: 由快照构成的历史记录，虽然版本 2 指向 B，图中出现了环的形状，但 B 没有其它出边，不会出现循环，也因此不构成环

从数据结构的角度来看这种设计，历史记录就成了一个由快照组成的有向无环图。一次快照就是一棵**树**，包含了全部的文件，每一次快照更新的**提交**(commit)包含了上面的树，附带了提交信息、作者、时间戳等元数据。

怎样存储文件: 一切都是对象

操作系统为我们抽象出**文件**的概念：我们通过文件路径来定位文件元数据所在的磁盘位置，再根据文件的元信息定位文件在硬盘上的具体位置，最终访问到文件的内容。

而 Git 有着独属于它的文件**内容寻址**系统。它并不基于文件路径寻址，路径不再作为获取文件内容的键，而文件**内容本身就是这个键**。

Git 将项目所有受控文件内容、文件的状态通过 SHA1 算法进行**哈希化**、压缩后作为 Blob 对象存储在 `.git\objects` 内。Git 数据库中保存的信息都是以文件内容的哈希值来索引，所有的数据在存储前都计算校验和，然后以校验和来引用。

我们用一个小例子来体现哈希化的过程：

```
>>> import hashlib
>>> data = 'hello git'
>>> content = f'blob {len(data)}\x00{data}'
'blob 9\x00hello git'
>>> hashlib.sha1(content.encode()).hexdigest()
'f09e9c379f5fe8f4ce718641c356df87906d87a6'
$ echo -n "hello git" | git hash-object --stdin
f09e9c379f5fe8f4ce718641c356df87906d87a6
```

对文件名、目录等信息的保留依赖于先前提到的**树**。假如你的最新提交中更改了一个文件的文件名，而没有改变它的内容，Git 不会添加新的 Blob 对象，而只会更改树对象中指向这个 Blob 对象的名称。

树、提交（以及标记）这些元数据也是 Git 对象，同样会经过哈希化同文件一起存入 `objects` 目录下。正因为它们共同继承“对象”的统一接口，关联这些概念只需要提供哈希索引。

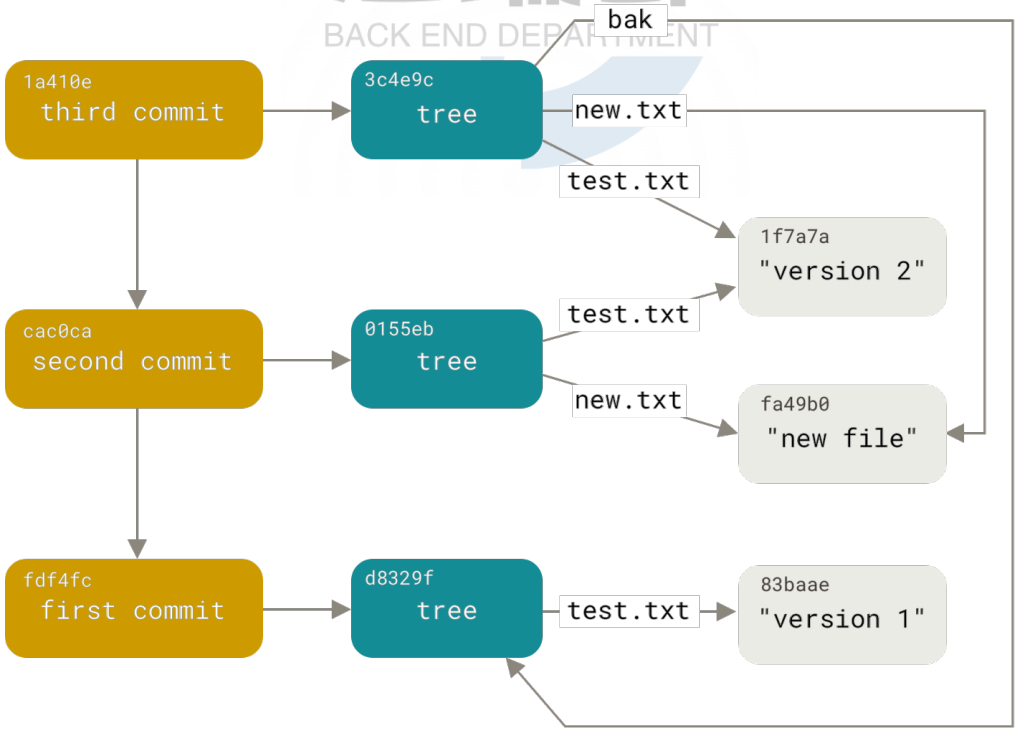


Figure 6: Git 对象模型示例，第三次提交的快照添加了 `bak/test.txt`，内容和文件名和初次提交一样。Git 不会添加新的 Blob 对象，只需令其引用第一次提交的树对象即可。

简洁而又高效的设计。

怎样设置分支: 引用

现在，所有的快照都可以通过它们的 SHA-1 哈希值来引用，不过我们并不需要记住这一串串 40 位的十六进制字符。Git 支持短哈希，也支持对哈希值做**别名**(alias)，**标记**(tag)快照号：

```
$ git tag -a v0.1.0 4892c7dc -m 'version 0.1.0 released'
```

之后我们就可以用 `v0.1.0` 更便于人类记忆的编号来引用这个快照号了。

Git 内部维护**引用**(reference)使得某一项别名可以指向最新的提交，当前所在位置会用一个特殊的 `HEAD` 索引标记。

Figure 7: 例如，`master` 引用通常会指向主分支的最新一次提交，此时 `HEAD` 引用指向 `master`。

创建**分支**其实就是新建一个引用，分支切换只需要改变 `HEAD` 文件内分支的指向，而分支合并（merge）则是创建一个新的提交对象，其父节点指向两个分支的最新提交，然后合并二者的文件。

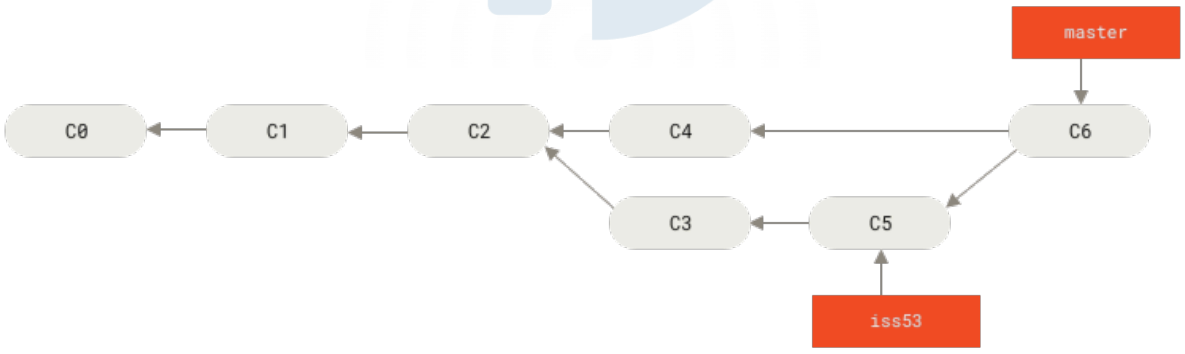


Figure 8: 分支开发的一个场景。为了解决出现的 bug，你在主分支之上新建了一条 iss53 分支，在修正了问题 #53 后，你在主分支将其合并。

总结

TL;DR: Git 以**快照**的形式记录版本差异，将要存储于仓库的数据抽象为**对象**，具体来说：Blob 保存了文件内容，**树**保存了目录结构和文件名，**提交**保存了一次快照的提交信息、作者、时间戳等元数据。对象根据其内容的**哈希值**进行索引、关联。同时，Git 内部维护**引用**，允许用一个更用户友好的别名来指向最新的提交记录，使得我们可以进行强大的**分支**操作。在这种简洁的设计之下，Git 的操作就是对一些“对象”和“引用”的**图操作**。

使用 Git

安装与配置

当前 Git 的最新稳定版本为 2.47.0。用 `git --version` 查看你的系统上是否安装了 Git。

- 对于 Windows 用户，下载地址。如果无法下载，尝试科学上网或清华镜像、阿里镜像：
- 对于 MacOS 用户，Git 通常是系统自带的工具；如果没有可以尝试下面这个图形化的 Git 安装工具，或者尝试 `homebrew` 安装（学习下装一个包管理工具）。
- 对于 Linux 用户，根据你系统的发行版安装包管理器安装 Git。
  - 例如 Ubuntu/Debian：`sudo apt-get install git`

我们对 Windows 安装包额外做一些说明：通常来说按照默认设置安装 Git 即可，但需要稍微注意一些个性化的设置（如换行符、编辑器）。我们建议你参考这篇文章来辅助你决策。

安装了 Git 之后，在 shell 里使用如下命令配置你的用户名和邮箱。参考这篇文章：

```
git config --global user.name <name>
git config --global user.email <email-address>
```

Git 接口

在学习 Git 接口之前，我们先来了解一下 Git 文件的状态：

- **未跟踪**(Untracked)：Git 尚未跟踪的文件，即还没有纳入版本管理的文件。
- **已修改**(Modified)：修改了文件，但还没保存到数据库中。
- **已暂存**(Staged/Added)：对一个已修改文件的当前版本做了标记，使之包含在下次提交的快照中。
- **已提交**(Committed)：数据已经安全地保存在本地数据库中。

这会让我们的 Git 项目拥有三个阶段：工作区、暂存区以及 Git 目录。

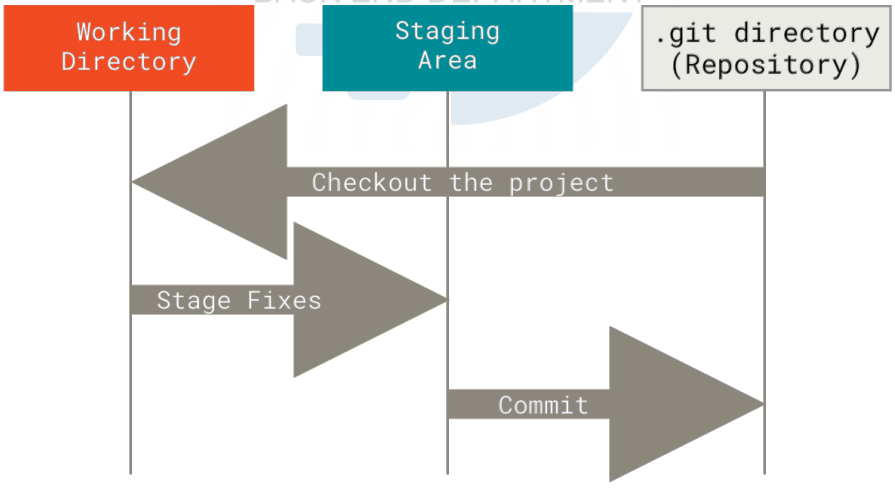


Figure 9: 在一次提交中，选中提交的文件会从工作区记录到暂存区、最后经哈希对象化存放到 `.git` 目录下

创建一次 `commit` 的流程如下：

- 在工作区中修改文件。
- 将你想要下次提交的更改选择性地暂存，这样只会将更改的部分添加到暂存区。(`git add`)
- 提交更新，找到暂存区的文件，将快照永久性存储到 Git 目录。(`git commit`)

为什么要引入暂存区这个看似多余的步骤？想象一下，当你开发了两个独立的特性，希望创建两个独立的提交，这时候就可以有选择地进行提交。

另外一些常用的命令行操作：

- `git clone <repository>`: 克隆远程仓库到本地。有些老师也喜欢用 Git 远程仓库来发布代码作业。
- `git status`: 查看当前仓库的状态。
- `git push`: 将本地仓库的更新推送到远程仓库（前提是你已经关联好了远程仓库）。当你的 IDE 因出现网络原因无法推送时可以尝试在命令行中推送。
- `git commit --amend`: 编辑历史提交的 commit 信息。
- `git stash`: 暂时移除工作目录下的修改内容。这在切换分支的时候很有用。

我们推荐大家阅读奇点先前编写的在线文档来学习更多的接口知识。

IDE 里使用图形界面

学习 Git 操作的最好是通过命令行，这也是学习命令行的好机会。

但我们同时推荐你去了解 IDE 里集成的 Git 使用：在日常工作中，CLI 并不常用。（除了某些特定 GUI 不管用的场景）

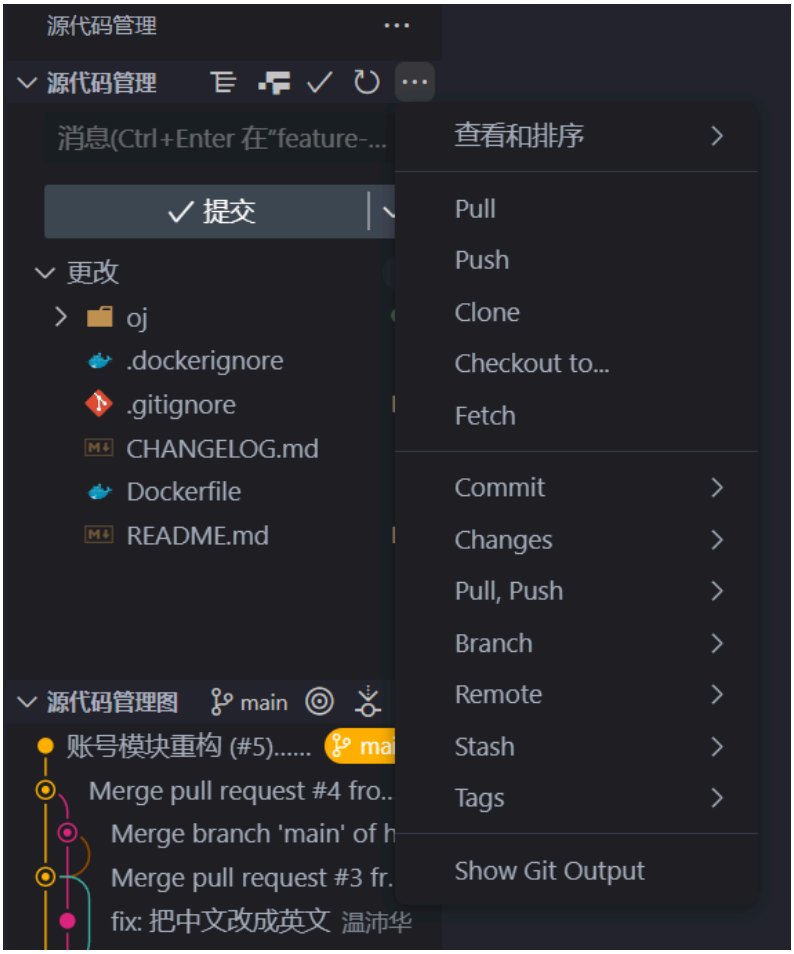


Figure 10: VSCode 里的源代码管理界面，在提供 Git 的图形化界面操作以外，也提供了对分支的良好可视化

Git 的工作流

由于 Git 的使用非常灵活，在实践当中衍生了很多种不同的工作流程，不同的项目、不同的团队会有不同的协作方式。下面将介绍一种在各种大小项目中经过总结出来的工作流。

奇点的开发团队参考的也是这一套流程。

远程 Git 仓库

我们提到，Git 是一个分布式版本控制系统，这意味着每个开发者都有一份完整的仓库，称为**本地仓库**。实现同步的方式是拥有一个可靠的中心**远程仓库**，一个 Git 服务器。

一些常用的 Git 服务器有：

- GitHub：一个基于 Git 的代码托管服务平台，同时也是重要的开源社区。
- GitLab：类似 Github，但主要面向企业、组织等内部使用。也可以用官方提供的镜像部署。
- Gitee：具有中国特色的代码托管平台

奇点的两个开发部门目前使用 Github 作为 Git 远程仓库。

分布式开发

源仓库

在项目的开始，项目的发起者构建起一个项目的最原始的仓库，称为**源仓库**。它的目的是：

- 汇总参与该项目的各个开发者的代码
- 存放趋于稳定和可发布的代码

开发者仓库

源仓库建立以后，每个开发者需要做的事情就是把源仓库的“复制”，在其基础之上 `fork`(分叉)开发，作为自己日常开发的仓库。

每个开发者所 `fork` 的仓库是完全独立的，互不干扰。而在开发工作完成以后，开发者可以向源仓库发送 `Pull Request`(推送请求)，请求管理员把自己的代码合并到源仓库中。

Figure 11: 一个开源项目 Qexo 的 PR，记得在 PR 里描述清楚自己的贡献

集成代码

当然，多条 PR 可能存在代码冲突！项目管理者首先需要对代码进行**代码审阅** (code review) 解决冲突以外，我们还需要检验集成后代码的正确性。**自动化构建工具**会根据流程自动编译构建安装应用，并执行**单元测试框架**的自动化测试来校验提交的修改。

一个成熟的项目通常包含完备的**持续集成和持续交付** (CI/CD)。它通过自动化流程和工具自动帮助项目管理者构建应用、测试应用、部署应用。

例如，Github Action 通过 YAML 文件的配置定义工作流程以构建执行 CI/CD 流水线，并可以触发不同事件时（如 `push`、`Pull Request`）自动执行这些工作流程。

当你的 PR 通过了 CI/CD 的测试，以及相关的 code review，项目管理者就可以采用如 squash-merge 的方式将你的代码合并到主分支。

分支管理

称分支管理是 Git 的灵魂也不为过。上面我们提到了 Git 分布式开发的基本流程，为了更好地利用 CI/CD 工作流，项目团队通常对分支的命名和权限的管理有着一定的规定，下面就介绍一种比较普遍适用的分支模型，在这种模型中，分支有两类，五种：

永久性分支

- `main`: 主分支，它用于存放经过测试，已经完全稳定的代码；在项目开发以后的任何时刻当中，主分支存放的代码应该是可作为产品供用户使用的代码。
- `develop`: 开发分支，它用于开发者存放基本稳定代码。

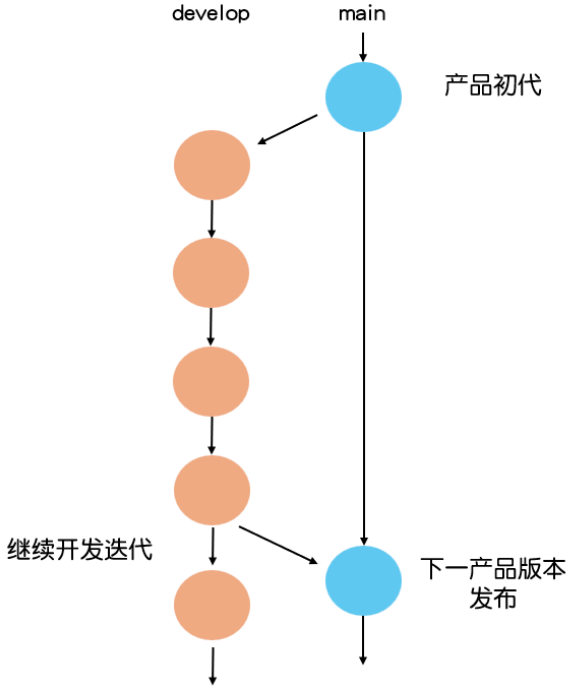


Figure 12: 所有开发者开发好的功能会在源仓库的 `develop` 分支中进行汇总，经测试稳定后最后合并到 `main` 分支。这就是一个产品不断完善和发布过程。

提示：分支保护很重要，不要为了图一时省事不建立开发分支或随意合并至主分支！

临时性分支

临时性分支类似于开发分支的一个缓冲区。在开发完毕、合并后，这些临时性分支会得到删除。通常来说，各个开发者会在 `develop` 分支上继续分叉出特性分支 `feature-*`，例如 `feature-OAuth` 表示实现 OAuth 登录功能的特性分支。

另外两个临时分支在更大规模的团队开发中会运用到：

- `release`: 预发布分支，当产品即将发布的时候，要进行最后的调整和测试，就可以开出预发布分支
- `hotfix`: 修复 bug 分支，在产品上线之后发布热补丁进行紧急的 bug 修复工作，通常来说经过测试后会合并回 `main` 分支。

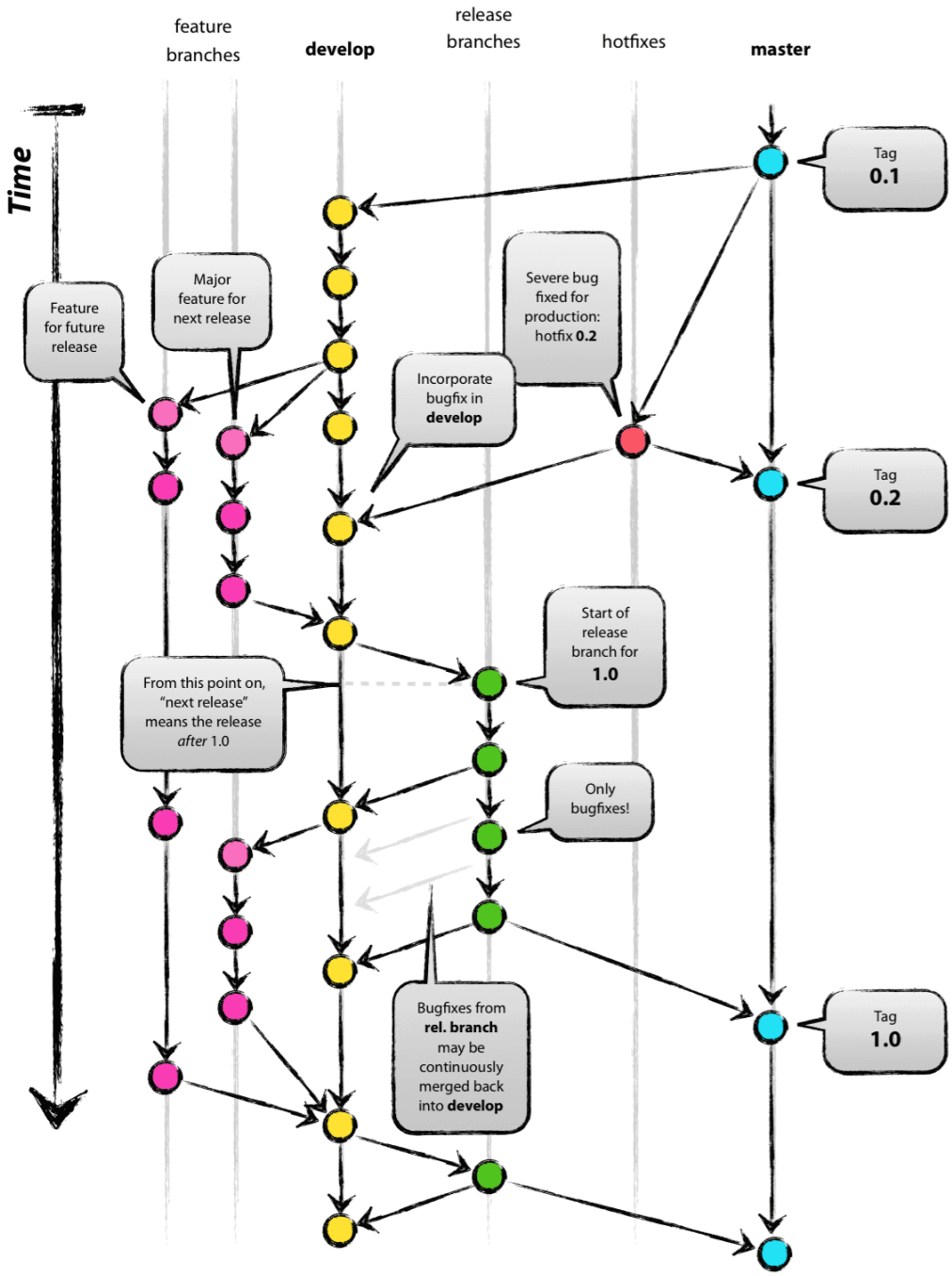


Figure 13: 比较标准的大型项目的 Git 分支模型。在这种规范下，历史记录看起来很复杂但是相当清楚。

提交消息规范

对分支规范后，我们还会对提交消息进行规范，主要的原因是：

- 利用工具来自动化生成 CHANGELOG，例如 `cz-conventional-changelog` 工具
- 统一提交消息格式，不必纠结到底该怎么写提交消息
- 方便项目更好设计 CI/CD 流程

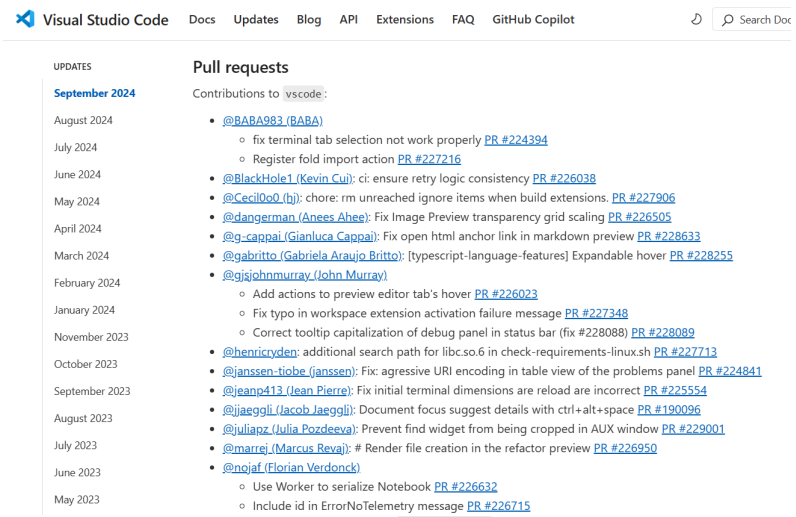


Figure 14: 一个好的开源软件都会在每个版本号对提交或者 PR 记录改动日记，例如 VSCode

我们的参考是 约定式提交规范。具体来说，如果采用 eslint 格式的格式规范：

- 每个提交都**必须**使用类型字段前缀，如 `fix`，同时加上以及必要的冒号（英文半角）和空格.
- 当一个提交为应用或类库实现了新功能时，使用 `feat` 类型；为应用修复了 bug 时，使用 `fix` 类型。
- 类型字段**可以**指定范围，例如 `fix(api)` 指明是对接口代码的修复。当然你也可以不带.
- 每个提交**必须**要有对提交的描述，对代码变更的简短总结。可以使用中文。
- 在类型后添加感叹号，表明这是一个**破坏性变更**(Breaking Change)。即，对现有功能或接口进行的修改。如果你改了接口的某一个字段，导致无法兼容旧版本，那么就需要加上感叹号。

更多规则见上方给出的文档。以上几条作为我们的开发规范就够用了。

示例

- `feat!: send an email to the customer when a product is shipped`
- `docs: correct spelling of CHANGELOG`
- `feat(lang): add polish language`
- `feat`: 登录相关接口

推荐大家去阅读正式一些的开源项目来了解提交规范。例如，有的项目会要求在 PR 里指出相关的 issue 编号，等等。

相关工具

你可以在终端里使用 commitizen，commitlint 这些工具省去手敲前缀的麻烦。

参考资料

这篇文档主要参考的是 Pro Git。许多配图以及内容都来自这本书，强烈推荐自己亲自读一遍前五章！

奇点工作室暑期训练营后端文档：这是奇点在 23 年暑假开展的暑期训练营，这篇文档也是在线文档的一次重置翻新——主要是对原理部分进行了重写（因为初版确实有些不那么容易理解）。如果你对其它部分感兴趣，欢迎阅读文档！

MIT CS 教育缺失的一课: 版本控制(Git): 整个 Missing-semester 课程都推荐大家学习, 非常好的一门课, 学习到很多好用的工具, 除了拓宽视野更能帮助你在未来的职业生涯中节省时间.

Learn Git Branching 通过基于浏览器的游戏来学习 Git.

A successful Git branching model：有关分支工作流的文章，文章的图 13 来自于此。