

05 | 全局视野：洞悉项目开发流程与规范

2022-10-20 郑建勋 来自北京



课程介绍 >

《Go进阶·分布式爬虫实战》



讲述：郑建勋

时长 16:41 大小 15.23M



你好，我是郑建勋。

上节课，我讲解了大型互联网产品开发流程中的需求阶段和设计阶段。这节课，我们继续看看后面五个阶段：研发实现阶段、测试阶段、上线部署阶段、运维阶段和运营阶段。

首先让我们从研发实现阶段说起。

研发实现阶段

确定了设计方案和开发排期之后，我们终于可以进行实际的代码开发了。在大公司，开发过程并没有那么随意，需要遵循多种开发规范，包括**编码规范**、**接口规范**、**日志规范**、**测试规范**、**Commit 规范**、**版本控制规范**、**发布规范**等等。开发规范能够帮助团队更好地协作，同时也能提高代码的质量、提高程序的性能、规避低级的错误、发现隐含的问题。

我们重点看看其中几个规范。

编码规范



编码规范是最受重视的规范，它是团队在程序开发时需要遵守的约定。这种约定在统一代码风格的同时，也为团队定义了什么是好的代码。阅读好的代码就像一本小说，读者脑海中的文字被图像取代，你仿佛看到了角色，听到了声音，体验到了悲怆或幽默。

当我们阅读一段代码时，发现它简单清晰、难以挑出毛病，那我们基本可以断定这段程序是被精心设计过的。相反，如果我们瞥一眼代码就能够发现很多命名问题、无效代码和注释问题，我们也可以预测这种不严谨或者错误已经渗透到了程序的其他角落。

好的代码需要具备如下特性：

- 整洁、一致；
- 高效；
- 健壮；
- 可扩展。

而依照编程规范书写好代码有助于我们促进团队合作、避免错误、提升性能，同时还能够方便我们后续的维护工作。

编码规范是项目开发前的必修课，后面我还会用一节课程专门讲解 Go 语言的代码规范。

接口规范

接口规范指的是定义系统与外界交互方式的协议。在微服务系统中经常涉及到服务之间的调用。一个微服务通常有自己的上下游，一般服务的上游叫做 **caller**，服务的下游叫做 **callee**。

服务之间的通信，最常用的是 HTTP、Thrift 和 gRPC 协议。以使用最多的 HTTP 协议为例，大多数 Web 服务使用了 RESTful 风格的 API。RESTful 规范了资源访问的 URL，规定了使用标准的 HTTP 方法，例如 GET、POST、PUT、DELETE 等，并且明确了这些方法对应的语义。除此之外，接口规范还需要定义状态码如何赋值、如何保证接口向后兼容等一系列问题。

大型公司会单独管理 API 接口，甚至会有一套专门描述软件组件接口的计算机语言，被称为 IDL（接口描述语言，Interface description language）。



IDL 通过独立于编程语言的方式来描述接口，每一种编程语言都会根据 IDL 生成一套自己语言的 SDK。即便是相同的语言，也可能生成不同协议（例如 HTTP 协议、gRPC 协议）的 SDK。使用 IDL 有下面几个好处：

- 作为接口说明文档，IDL 统一规范了接口定义和使用方法，不同使用方不用反复沟通接口的使用方法；
- 不同语言编写的程序可以很方便地相互通信，屏蔽了开发语言上的差异；
- 生成的 SDK 可以提供通用的能力，例如熔断、重试、记录调用耗时等，可以大大节省成本，毕竟如果这些功能要在每一个服务端都实现一遍，是一种成本的浪费。

日志规范

再来看看日志规范。运行中的程序就像一个黑盒，好在日志提供了系统在不同时刻的记录，我们可以基于日志了解系统的运行状态。日志的好处主要有下面四点。

1. 打印调试：打印调试的意思是用日志来记录变量或者某一段逻辑，记录程序运行的流程。虽然用日志来调试通常会被嘲笑为技术手段落后，但它确实能够解决某些难题。例如，一个场景线下无法复现，我们又不希望对线上系统产生破坏性影响，这时打印调试就派上用场了。
2. 问题定位：有时候，系统或者业务出现问题，需要快速排查原因，这时我们就要用到日志的功能了。例如，Go 程序突然 panic，被 recover 捕获之后打印出当前的详细堆栈信息，就需要通过日志来定位。又如，调用的下游系统突然有大量的报错，需要抓取日志查看详细的报错原因。
3. 用户行为分析：日志的大量数据可以作为大数据分析的基础，例如用户的行为偏好等。
4. 监控：日志数据通过流处理生成的连续指标数据，可存储起来并对接监控告警平台，这有助于我们快速发现系统的异常。监控的指标可能包括：核心接口调用量是否突然下降或上升、核心的业务指标变化（例如，GMV 是否同比和环比稳定、是否出现了不合理的订单，是否出现了零元或者天价账单）等。

关于日志分级、日志格式和日志库选型在内的日志处理知识，我们在后面的课程中还会详细介绍。

版本控制规范

我们过去常会遇到文件被删除，或者修改之后就无法找回的问题。如果我们好不容易写了大量资料，文档却被人删除，想想就让人抓狂。如果我们想查看之前改动的记录，或因为计划有变想恢复之前的记录，也是很难办到的。

而现在我们使用的在线文档工具（例如谷歌文档、腾讯文档、石墨文档、Notion 等）都能自动保存数据，这满足了团队间协作的要求，允许我们随时查看修改记录，恢复之前的数据。要完成这些功能，背后都离不开版本控制系统。

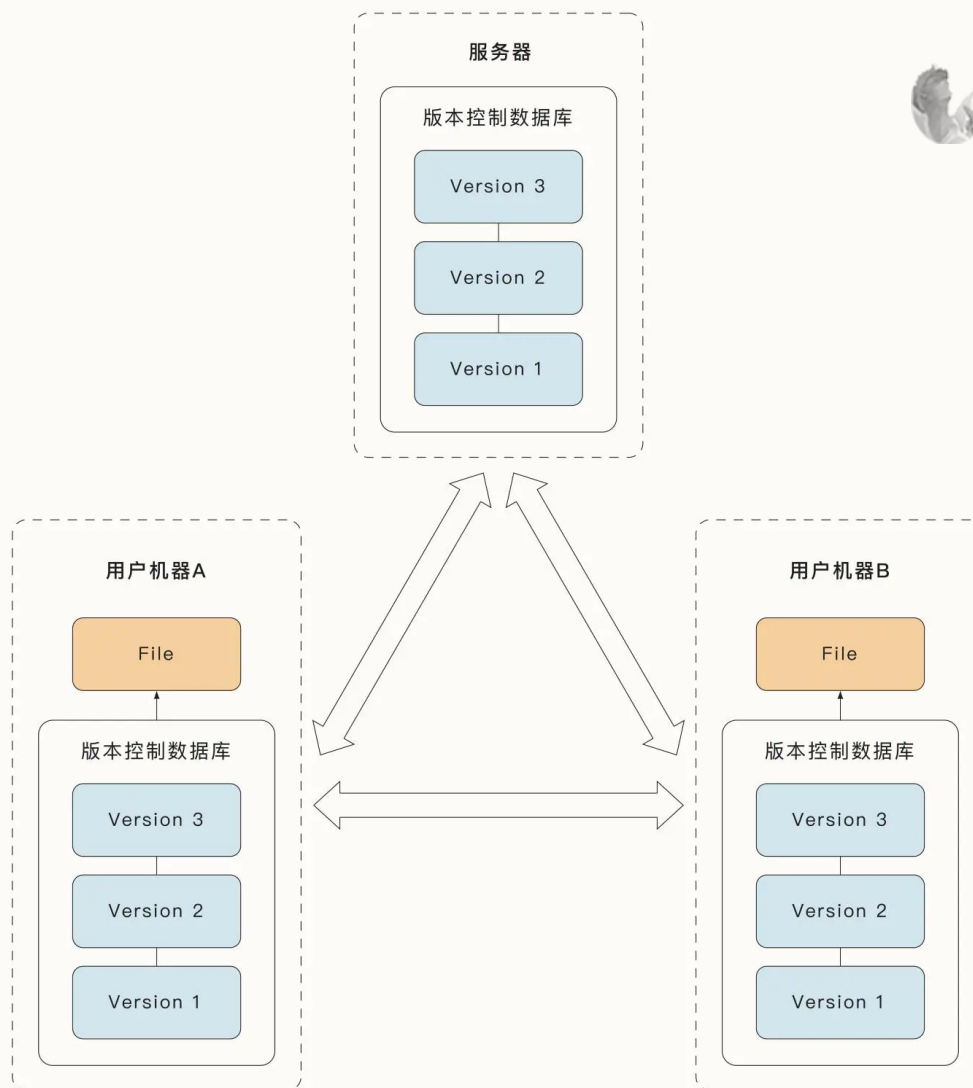
当版本控制应用到程序代码仓库中，我们可以轻松地拥有各个历史阶段代码的快照，快速地退回到过去任意一个版本。我们可以比较任意两个版本代码在细节上的变化，从而助推代码审查、防止修改上的错误。我们也可以方便地查阅何人何时修改了哪些代码。

版本控制系统的这些好处让它成为了代码开发的必备工具。

Git 与 workflow

版本控制系统经历了从本地版本控制系统、到集中化的版本控制系统，再到分布式版本控制系统的演进过程。

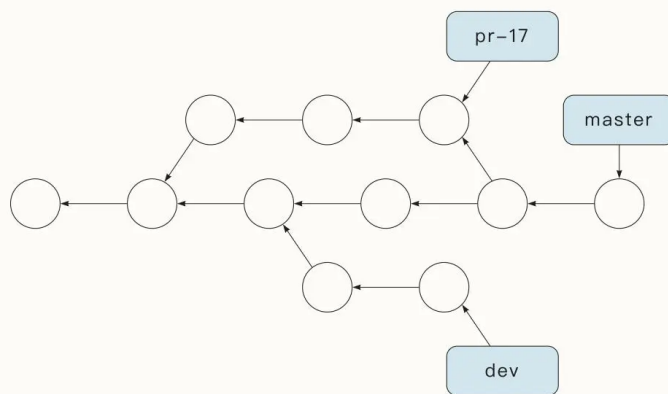
当前使用最多的分布式版本控制系统是 **Git**。我相信任何有编程开发经验的同学对这个工具都不会陌生，我们利用 **GitHub**、**GitLab** 进行代码管理和版本控制时，都离不开 **Git**。这里的分布式，指的是任何客户端从代码仓库中拉取代码时，都包含了所有的代码记录，而不仅仅是最新的代码记录。这意味着即便服务器完全不可用，我们也比较容易从任意一台用户的服务器中恢复代码。



Git 让你能够基于项目的稳定代码库开辟新的分支，和团队成员并行工作。还可以确保新的特性或实验性代码实现。创建“分支”是一种非常常见的做法，它可以确保主开发线的完整性，避免任何意外的更改破坏主分支。

在 **Git** 中，分支的概念被认为是轻量级且廉价的，你可以在本地轻松切换分支，因为 **Git** 中的分支名其实是一个指针，指向了某一个 **commit**，分支的切换就是单纯指针的切换。

对于你创建的每个分支，**Git** 都会跟踪该分支的一系列提交。如下图所示，当我们新提交一个 **commit** 时候，代表分支名的指针会指向新的 **commit**，同时，新的 **commit** 会指向它的父 **commit**，这样我们就能够追踪到整个分支的所有的 **commit** 了。如果你想更进一步地了解 **Git** 内部的一些原理，我推荐你阅读《**Version Control with Git, 2nd Edition**》，这本书在 2022 年底将会推出第三版。



Git 的特性催生了基于 Git 的多种工作流模式，包括：

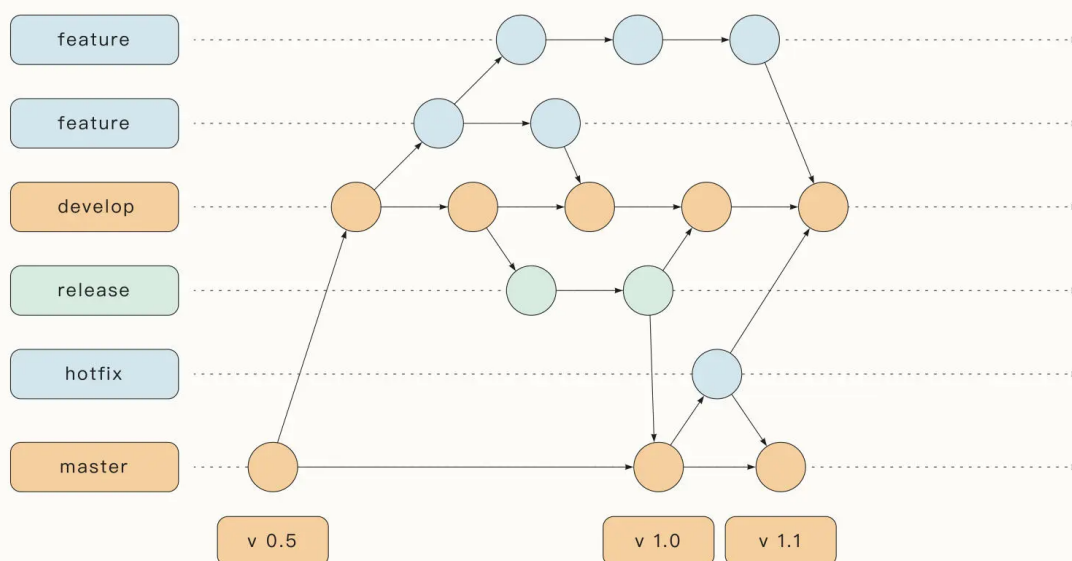
- 集中式工作流
- Git Flow 工作流
- GitHub Flow 工作流
- GitLab Flow 工作流

我主要介绍一下目前大型项目用得比较多的 **Gitflow 工作流** 和 **GitHub Flow 工作流**。

Gitflow 工作流（Gitflow Workflow）是 2010 年 Vincent Driessen 在他的一篇 [博客](#) 里提出来的。它定义了一套完善的基于 Git 分支模型的框架。Gitflow 工作流结合了版本发布的研发流程，适合管理具有固定发布周期的大型项目。它对于分支的定义有下面几点。

- **Master 分支：**作为唯一一个正式对外发布的分支，是所有分支里最稳定的。
- **Develop 分支：**是根据 Master 分支创建出来的。Develop 分支作为一种集成分支 (Integration Branch)，专门用来集成已经开发完的各种特性。
- **Feature 分支：**根据 Develop 分支创建出来。Gitflow 工作流里的每个新特性都有自己的 Feature 分支。当特性开发结束以后，这些分支上的工作会被合并到 Develop 分支。
- **Release 分支：**当积累了足够多的已完成特性，或者预定的系统发布周期临近的时候，我们就会从 Develop 分支创建出一个 Release 分支，专门做和当前版本发布有关的工作。Release 分支一旦创建，就不允许再有新的特性被加入到这个分支了，只有修复 Bug 或者编辑文档之类的工作才能够进入该分支。Release 分支上的内容最终会被合并到 Master 分支。

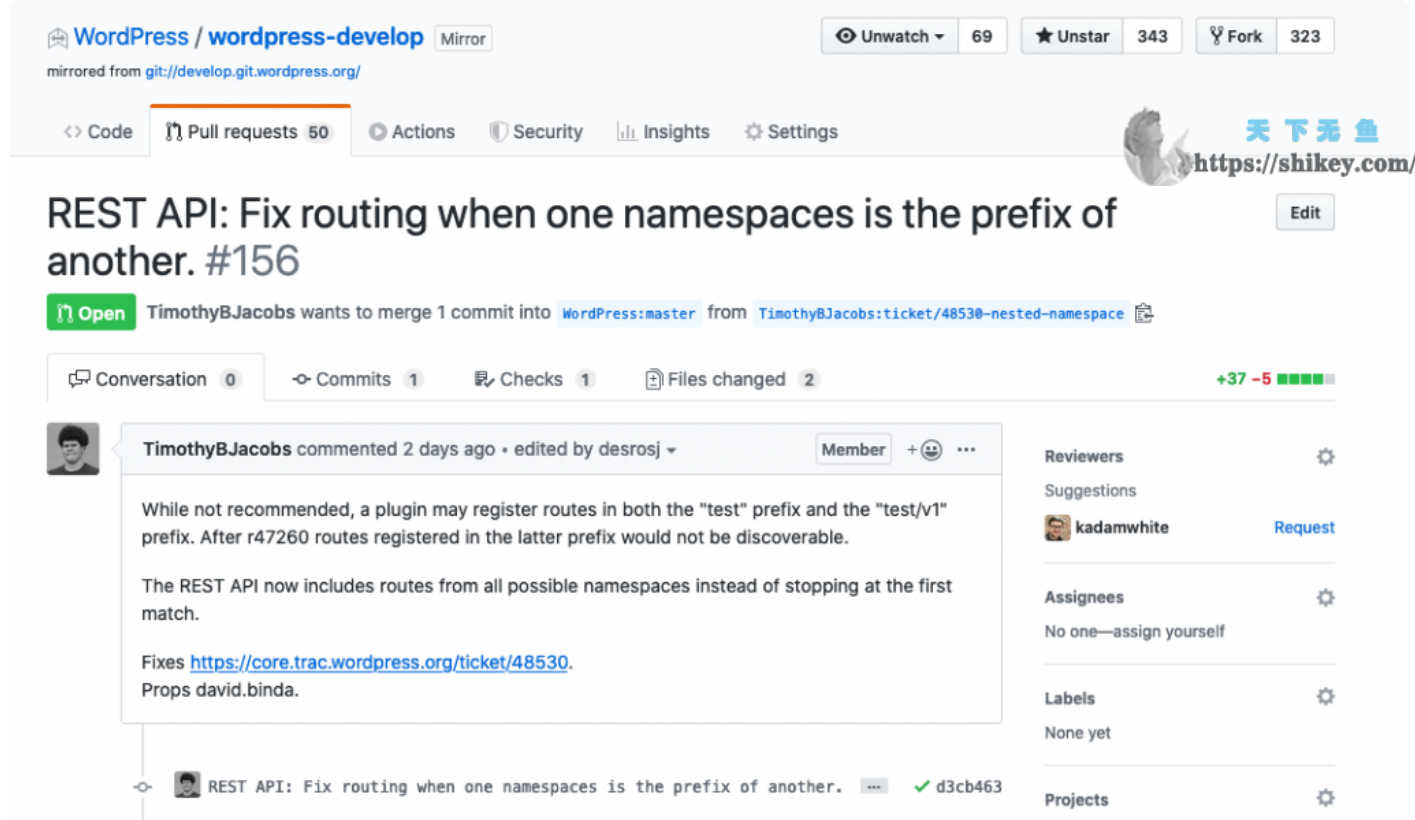
- **Hotfix 分支**：直接根据 Master 分支创建，目的是给运行在生产环境中的系统快速提供补丁。当 Hotfix 分支上的工作完成以后，可以合并到 Master 分支、Develop 分支以及当前的 Release 分支。如果有版本的更新，也可以为 Master 分支打上相应的 Tag。



Gitflow 工作流诞生于 2010 年，在这 10 年间软件开发的模式发生了很大变化，敏捷开发通过持续迭代的方式，让产品发布不再具有固定的发布周期。因此，Gitflow 工作流虽然很有名，但它不是灵丹妙药，也不适用于所有场景。而更适合敏捷开发且流程更简单的 **GitHub Flow 工作流逐渐变为了主流**。

在 GitHub Flow 工作流中，通常有一个管理者维护的主仓库。一般开发者无法直接提交代码到主仓库，但是可以为主仓库代码提交变更。在通过了自动化 CI 校验和代码评审（Code Review）之后，维护者会将代码合并到主分支中。GitHub Flow 工作流的详细过程如下。

1. 项目维护者将代码推送到主仓库。
2. 开发者克隆（Fork）此仓库，做出修改。
3. 开发者将修改后的临时代码分支推送到自己的公开仓库。
4. 开发者创建一个合并请求（Pull Request），包含进行本次更改有关的信息（例如目标仓库、目标分支、关联要修复的 issue 问题），以便维护者进行代码评审。



WordPress / wordpress-develop Mirror

mirrored from [git://develop.git.wordpress.org/](https://github.com/WordPress/wordpress-develop)

<> Code Pull requests 50 Actions Security Insights Settings

天下无鱼 <https://shikey.com/>

REST API: Fix routing when one namespaces is the prefix of another. #156

Open TimothyBJacobs wants to merge 1 commit into WordPress:master from TimothyBJacobs:ticket/48530-nested-namespace

Conversation 0 Commits 1 Checks 1 Files changed 2 +37 -5

TimothyBJacobs commented 2 days ago • edited by desrosj

While not recommended, a plugin may register routes in both the "test" prefix and the "test/v1" prefix. After r47260 routes registered in the latter prefix would not be discoverable.

The REST API now includes routes from all possible namespaces instead of stopping at the first match.

Fixes <https://core.trac.wordpress.org/ticket/48530>.
Props david.binda.

REST API: Fix routing when one namespaces is the prefix of another. ✓ d3cb463

Reviewers: kadamwhite (Request)

Assignees: No one—assign yourself

Labels: None yet

Projects

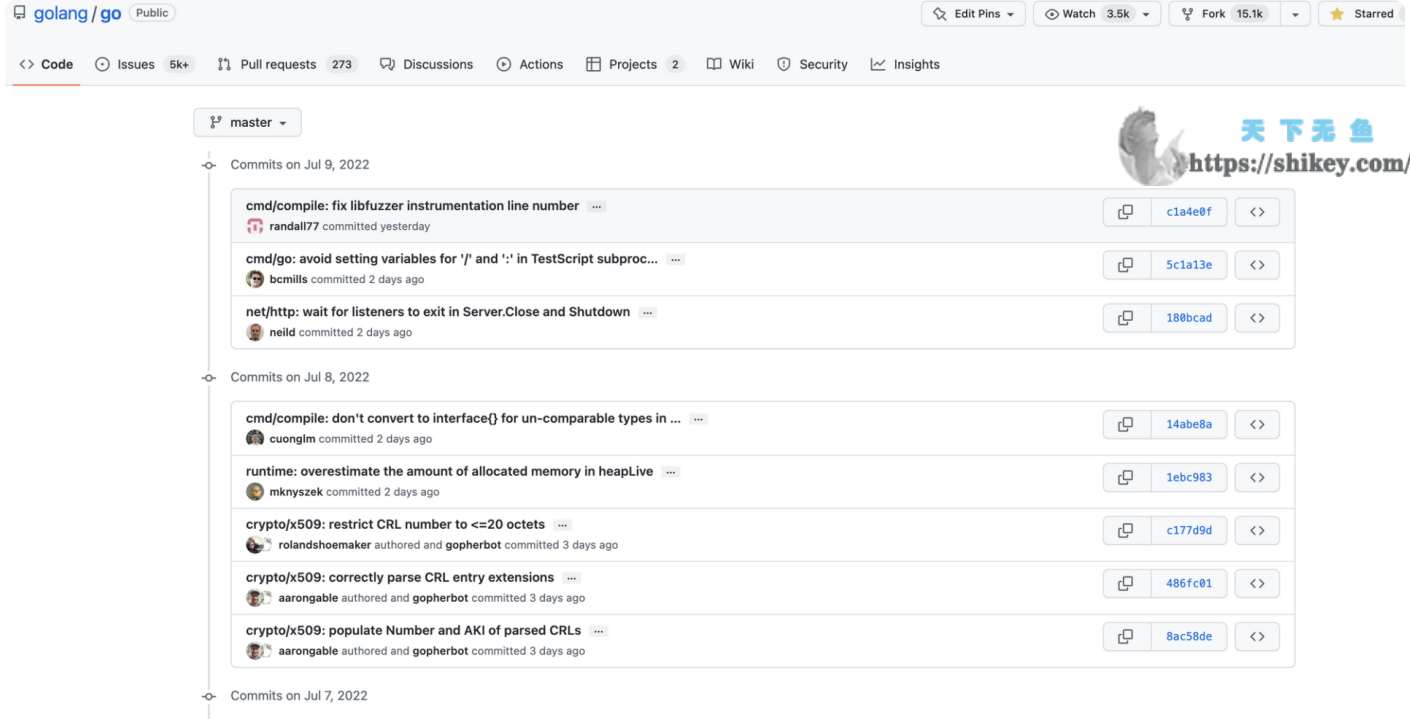
5. RP 通过后，临时代码分支将被合并到指定的分支，合并前可能有一些需要解决的代码冲突。合并完成后，GitHub 在合并分支的 commit 记录中可以连接到之前 PR 的页面，帮助我们了解更改的历史、背景和评论。
6. 合并拉取请求后，维护者可以删除临时代码分支。这表明该分支上的工作已经完成，同时也可以防止其他人意外使用旧分支。

Commit 规范

与 Git 有关的另一个概念是 Commit。Commit 是一次开发者触发的代码提交，可以理解为往代码仓库中存储了当前所有文件的快照，每一次 Commit 提交都有必要的提交信息，而规范这些信息有如下好处：

1. 格式统一，内容更加清晰和易读；
2. 可以通过提交记录来了解本次提交的目的，更好地 CR 和重构；
3. 更容易了解变更、定位和发现问题；
4. 由于每个提交描述都是经过思考的，这就可以改善提交的质量。

我们可以看看 Go 语言官方仓库遵守的 [提交规范](#)。



Go 源码仓库中的 Commit 示例如下所示：

复制代码

```
1 math: improve Sin, Cos and Tan precision for very large arguments
2
3 The existing implementation has poor numerical properties for
4 large arguments, so use the McGillicutty algorithm to improve
5 accuracy above 1e10.
6
7 The algorithm is described at https://wikipedia.org/wiki/McGillicutty_Algorithr
8
9 Fixes #159
```


第一行通常是对变更的简短描述，并且在最开始指明了本次修改受影响的 package。

正文部分详细地论述了本次修改的背景和目的。要注意的是，不要使用 HTML、Markdown 或任何其他标记语言。

特殊符号“Fixes #159”将当前 commit 与 github issue159 关联起来。当前变更被合并后，github issue 工具会自动将该 issue 标记为已修复。

Go 代码仓库太大，Packages 很多，而上述的 Commit 规范有助于开发者快速区分要修改哪一部分代码。当前也存在其他一些有名的 Commit 规范，例如 **Vue**、**React**、**Angular** 都在用的 **Commit 规范**。

测试阶段

遵循项目开发规范完成开发之后，我们还需要完成必要的测试。这种测试可分为六类。 <https://shikey.com/>


- 代码规范测试：包括对代码风格、命名等的检测，主要工具有 `gofmt`、`goimport`、`golanci-lint` 等。
- 代码质量测试：包括代码的覆盖率。主要工具有 `go tool cover` 等。
- 代码逻辑测试：包括并发错误测试、新增功能测试。主要工具有 `race`、单元测试等。
- 性能测试：包括 `Benchmark` 测试、性能对比。主要工具有 `Benchmark`、`ab`、`pprof`、`trace` 等。
- 服务测试：测试服务接口的功能与准确性，以及服务的可用性测试。
- 系统测试：包括端到端测试，确保上下游接口与参数传递的准确性、确保产品功能符合预期。


在后面的课程中，我们还会实践爬虫项目的测试。

上线部署阶段

在代码被开发者提交到指定分支，发布 `PR` 进行代码评审之前，一般就会对代码进行自动的检查。这种检查包括：代码能否成功通过编译、静态扫描代码是否满足代码规范、自动化测试和单元测试能否通过。通过这些自动化的测试降低了变更出错的风险。

当代码通过检查之后，就进入到了代码评审阶段了。一般重要项目至少需要两个团队成员对代码进行评审。成功通过评审之后，代码会合并到主干分支作为稳定版本。

代码合并完成之后，还会再次进行代码的编译、镜像打包、自动化测试等动作。这些将开发与运维结合起来，使用自动、连续和迭代的过程来构建、测试和部署的过程也被称为  **CI/CD**（持续集成 / 持续交付，`continuous integration/continuous delivery`）。

CI/CD 是  **DevOps** 的一种最佳实践，通过 **CI/CD** 能够提高软件开发和交付的效率、速度 and 安全性。有些项目可以不用人工干预，将打包后的镜像自动部署到生产集群中。不过在实践中，一些重要的项目还是需要完成项目的上线审批。上线单需要 **QA** 与项目负责人审批，**审批人需要二次检查上线步骤、检查项和回滚方案**。

完成上面一系列检测和审批之后，就可以进行最终的上线部署了。上线部署过程中需要遵循如下流程。



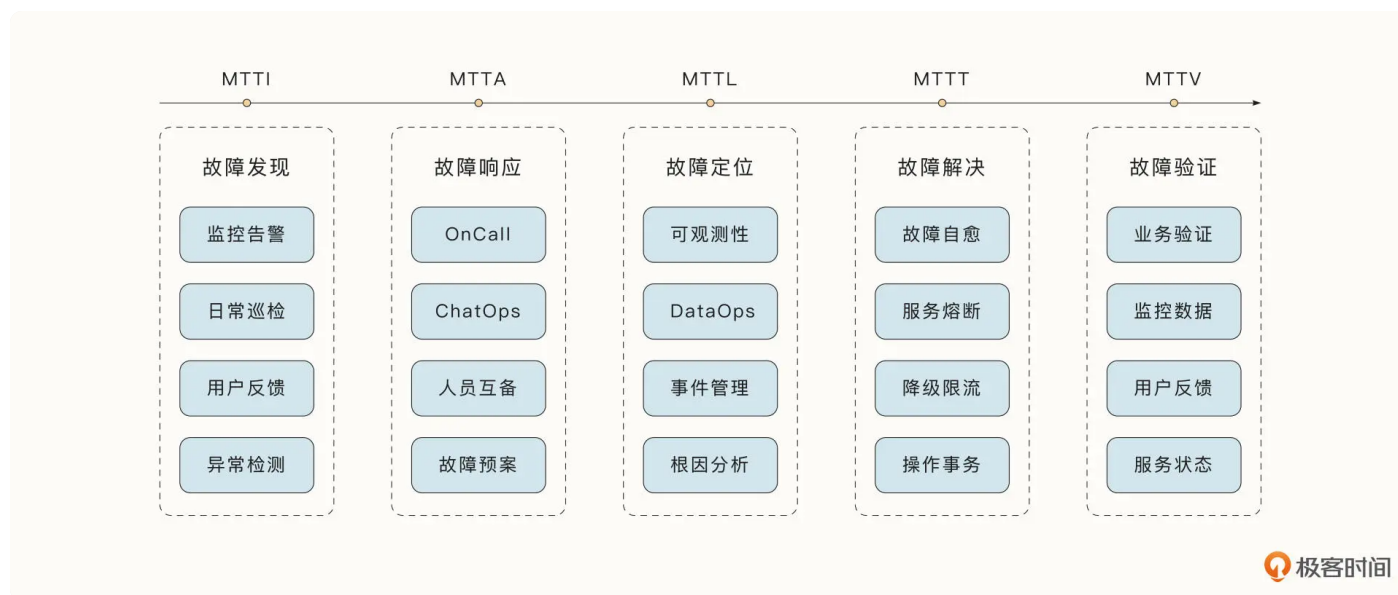
- 上线避开高峰期上线，尽量不要在节假日之前上线变更较大的版本，不在业务量大的时期做任何变更操作。
- **提前通报利益相关者，例如项目组成员、组内成员、有影响的上下游。**
- 严格规范上线执行步骤、确定检查清单与回滚方案。
- 灰度发布，并且对于要变更的功能，采取逐步放量的方式。例如只放量 A 城市 M 品类 30% 的流量，这样 A 城市 M 品类 30% 的用户才会体验到新功能，最大限度地减少变更时候的风险。借助灰度发布也能进行 A/B 实验，这样可以观察不同策略下用户的行为，从而做出科学的决策。
- **分级发布，遵循先少量，再部分，最后全量的原则；严格控制每一次部署的时间间隔。**通常大公司还有接近线上环境的预发环境，首先在预发环境中部署服务并验证服务的检查清单。对于核心服务来说，上千个容器是很常见的事情，分级发布有助于减少问题出现时的影响面，因为大部分问题都是服务的变更引起的。
- 另外，在上线时进行检查，观察当前服务指标是否异常，如发现异常应尽快回滚，**止损后再查明问题原因**。部署过程中，需要观察程序的核心指标，包括系统指标（上下游调用错误率、接口的平均响应时间和 P99 响应时间、CPU、内存、磁盘的利用率等）和业务指标（服务错误率、核心接口请求量等）。以打车业务为例，监控的指标包括：是否出现天价账单、特定费用项（如时长费、动态调节费）是否出现大的波动，平台是否抽成过高等问题。
- 最后，在上线过程中还要及时关注报警群，关注上下游的反馈，收集错误日志，并抓取 case 查看。

服务部署完成后，一些重要变更还需要 QA 工程师进行回归测试，验证服务在线上是否符合预期。

运维阶段

产品新功能上线后，其实更多的时间还需要花在对产品的运维上。当前运维的趋势是减少人工介入，通过平台化建设与自动化手段自动发现和修复问题，并且衍生出了 SRE 工程师的新岗位。

SRE 工程师日常会涉及到开发工作，他们用系统来维护系统，通过自动化、工具化等手段提升服务管理效率，确保集群的可观测性、稳定性、可用性。下图是腾讯的 SRE 稳定性建设的全景图。



SRE 工程师通常是围绕着缩减下面的几个时间来提高整个系统的稳定性水平：

- MTTI (Mean Time To Identify) 平均故障发现时间
- MTTA (Mean Time To Acknowledge) 平均故障确认时间
- MTTL (Mean Time To Location) 平均故障定位时间
- MTTT (Mean Time To Troubleshooting) 平均故障解决时间
- MTTV (Mean Time To Verify) 平均故障验证时间

而要确定服务是否稳定运行，需要对目标进行量化。所有运维的工作都围绕着 SLO（服务水平目标）的定制、执行、跟踪和反馈来展开。Google 提出了下面这套 VALET 法帮助我们定制自己的 SLO 指标，包括了如下几个因素：

- **Volume (容量)**：服务承诺的最大容量。比如常见的 QPS、TPS、会话数、吞吐量以及活动连接数等等。
- **Availability (可用性)**：服务是否正常 / 稳定。比如请求调用 HTTP 200 状态的成功率，任务执行成功率等。
- **Latency (时延)**：服务响应速度。有时我们需要判断时延是否符合正态分布，或者指定不同的区间，比如常见的 P90、P95、P99 等。

- **Error（错误率）**：服务错误率，比如 5XX、4XX，以及自定义的状态码。
- **Ticket（人工干预）**：服务是否需要人工干预，面对一些复杂的故障场景，就需要人工介入来恢复服务。



除此之外，我们还需要根据自己业务场景的不同，制定相对应的业务指标体系。对于单台容器请求耗时过大，或者服务器请求量上涨等异常，可以用自动触发漂移和自动扩容来解决。而另外一些需要人工介入的异常，就需要根据报警的级别将报警信息通过群、短信和电话的形式发送给相关责任人，以便相关责任人以最快的速度完成发现、止损、修复过程。

在运维阶段，**SRE** 工程师还需要与开发工程师保持良好的沟通合作，提升 **CI/CD** 水平，定时进行故障演练、压测演练、降级演练来保证服务运行状况符合预期，各项降级预案（限流、熔断、开关降级、切流）正常。

运营阶段

对于新上线的功能，运营需要评估它的收益是否符合预期目标，例如是否显著提高了日均活跃用户数量 (**DAU, Daily Active User**) 和商品交易总额 (**GMV, Gross Merchandise Volume**)。通过实验的结论和数据的分析，洞悉用户需求变化，为进一步策略调整、产品研发提供决策依据。

总结

这两节课，我们讲解了大型互联网产品的开发流程。可以看到，从需求分析、设计再到运维、运营的每一个阶段，都有比较完备的操作规范与相对应的工种。这种严谨的开发流程在保证产品质量的同时，也高效推进了互联网产品的敏捷开发。这种开发流程也推动了社会的分工，不同的开发工程师、UI 设计师、测试工程师、运维工程师在中间发挥着自己的专业能力。

互联网产品的这种开发流程不一定适用于小规模的企业，有一些流程（比如灰度发布和分级发布）可能并不是必要的。但是它确实为我们提供了一些值得借鉴的好思路，可以帮助管理者设计好当下和未来的研发流程。

课后题

最后，我也给你留两道思考题。

1. Git 作为非常强大的版本控制系统，每一次 commit 都保存了所有文件的快照，那么它是如何做到轻量、分支切换快速的呢？
2. 在实践中，程序经过时间的积累通常会逐步变得难以维护，这就像动物陷入了油坑中。你觉得造成这种现象的原因是什么？



天下无鱼

http://wiley.com/

欢迎你在留言区与我交流讨论，我们下节课再见！

分享给需要的人，Ta购买本课程，你将得 20 元

生成海报并分享

👍 赞 3

🔗 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 04 | 敏捷之道：大型Go项目的开发流程是怎样的？

下一篇 06 | 免费的宝库：什么是网络爬虫？

精选留言 (3)

💬 写留言



抱紧我的小鲤鱼

2022-10-20 来自北京

这个快照并不是这次 commit 时项目中所有文件的复制，而是一种索引，通过这个索引可以找到这次 commit 时的所有文件，而 git 在每次 commit 时对项目中所有文件进行扫描，如果某文件发生变化则会对应生成一个新的文件 blob，记录当前 commit 时该文件的文件内容，而文件名是对文件内容的一次 SHA-1 运算得到的 40 位字符串，如果该文件内容没有发生变化，就不会产生新的 blob 对象（因为相同内容的 SHA-1 hash 值唯一），而快照就是记录所有这次提交 commit 时 blob 文件名 SHA-1 hash 字符串集合，所以就可以通过某次 commit 的快照找到当时所有文件的 blob 对象文件 hash 字符串集合从而找到所有的文件，加载所有文件后就还原到了当时 commit 的项目状态。时间久了 blob 过多还会进行压缩 pack，只保存 diff 信息。

作者回复： 有点细呀，更详细的推荐《Version Control with Git, 2nd Edition》



G55

2022-10-20 来自北京



天下无鱼

<https://shikey.com/>

回答下第二个问题吧: 就我工作的情况看

1. 业务迭代太快，大家都是完成任务为第一要素，对于质量和可维护性往往不那么看重。
2. `code review`机制不严格 对于代码风格可维护性检查不够 只检查能不能实现功能。
3. 大量实验性的代码堆砌在项目中，即使某些模块已经无效了但是代码没有被删掉， 导致出现一个文件几千行的情况。维护困难，新人接手成本高。

作者回复: 理想和现实的差距呀

共 2 条评论 >

👍 3



徐曙辉

2022-10-21 来自北京

思维导图有错别字，函数与线（栈），接口动调（态）调用，思维导图是不是少了网络部分？期待实战更有深度，前面几讲偏理论

作者回复: 你说的是03讲的学习路线哈，看得非常细致，我会修复一下。学习路线里面更多的是与Go自身有关的知识进阶，包括网络、文件系统等标准库很多涉及到了对系统调用API的封装，也涉及到原生协程的使用，所以我暂时没放上去。实际上这个项目07、16、17会对Go网络部分进行深入的介绍，涉及到标准库的实现和操作系统的实现，可以关注一下。



👍 1