

## 35 | 快速构建持续交付系统（二）：GitLab 解决代码管理问题

2018-09-22 王潇俊

持续交付36讲

[进入课程 >](#)



讲述：王潇俊

时长 16:25 大小 7.52M



在上一篇文章中，我和你一起理清了我们即将构建的持续交付系统的需求，以及要具备的具体功能。那么，从这一篇文章开始，我们就要正式进入实战阶段了。我会和你详细介绍基于开源工具，从 0 开始搭建一套持续交付平台的详细过程，以及整合各个持续交付工具的一些技术细节。

按照我在前面分享的内容，搭建一套持续交付系统的第一步，就是搭建一套代码管理平台。这里我选择的开源工具是 GitLab，它是一套高仿 GitHub 的开源代码共享管理平台，也是目前最好的开源解决方案之一。

接下来，我们就从使用 GitLab 搭建代码管理平台开始吧，一起来看看搭建 GitLab 平台的过程中可能遇到的问题，以及如何解决这些问题。

## 利用 GitLab 搭建代码管理平台

GitLab 早期的设计目标是，做一个私有化的类似 GitHub 的 Git 代码托管平台。

我第一次接触 GitLab 是 2013 年，当时它的架构很简单，SSH 权限控制还是通过和 Gitolite 交互实现的，而且也只有源码安装（标准 Ruby on Rails 的安装方式）的方式。

这时，GitLab 给我最深的印象是迭代速度快，每个月至少会有 1 个独立的 release 版本，这个传统也一直被保留至今。但是，随着 GitLab 的功能越来越丰富，架构和模块越来越多，也越来越复杂。

所以，现在基于代码进行部署的方式就过于复杂了，初学者基本无从下手。


**因此，我建议使用官方的 Docker 镜像或一键安装包 Omnibus 安装 GitLab。**

接下来，我就以 Centos 7 虚拟机为例，描述一下整个 Omnibus GitLab 的安装过程，以及注意事项。

在安装前，你需要注意的是如果使用虚拟机进行安装测试，建议虚拟机的“最大内存”配置在 4 G 及以上，如果小于 2 G，GitLab 可能会无法正常启动。

## 安装 GitLab

1. 安装 SSH 等依赖，配置防火墙。

 复制代码

```
1 sudo yum install -y curl policycoreutils-python openssh-server
2 sudo systemctl enable sshd
3 sudo systemctl start sshd
4 sudo firewall-cmd --permanent --add-service=http
5 sudo systemctl reload firewalld
```


2. 安装 Postfix 支持电子邮件的发送。

 复制代码

```
1 sudo yum install postfix
```

```
2 sudo systemctl enable postfix
3 sudo systemctl start postfix
```

3. 从 rpm 源安装，并配置 GitLab 的访问域名，测试时可以将其配置为虚拟机的 IP（比如 192.168.0.101）。

 复制代码

```
1 curl https://packages.gitlab.com/install/repositories/gitlab/gitlab-ee/script.rpm.sh | :
2
3 sudo EXTERNAL_URL="http://192.168.0.101" yum install -y gitlab-ee
```

整个安装过程，大概需要 10 分钟左右。如果一切顺利，我们已经可以通过 ["http://192.168.0.101"](http://192.168.0.101) 这个地址访问 GitLab 了。

如果你在安装过程中，遇到了一些问题，相信你可以在[GitLab 的官方文档](#)中找到答案。

## 配置 GitLab

安装完成之后，还要进行一些系统配置。对于 Omnibus GitLab 的配置，我们只需要重点关注两方面的内容：

1. 使用命令行工具 gitlab-ctl，管理 Omnibus GitLab 的一些常用命令。  
比如，你想排查 GitLab 的运行异常，可以执行 gitlab-ctl tail 查看日志。
2. 配置文件 /etc/gitlab/gitlab.rb，包含所有 GitLab 的相关配置。邮件服务器、LDAP 账号验证，以及数据库缓存等配置，统一在这个配置文件中进行修改。  
比如，你想要修改 GitLab 的外部域名时，可以通过一条指令修改 gitlab.rb 文件：

 复制代码

```
1 external_url 'http://newhost.com'
```

然后，执行 gitlab-ctl reconfigure 重启配置 GitLab 即可。

关于 GitLab 更详细的配置，你可以参考[官方文档](#)。

## GitLab 的二次开发

在上一篇文章中，我们一起分析出需要为 Jar 包提供一个特殊的发布方式，因此我们决定利用 GitLab 的二次开发功能来满足这个需求。

对 GitLab 进行二次开发时，我们可以使用其官方开发环境 gdk (<https://gitlab.com/gitlab-org/gitlab-development-kit>)。但，如果你是第一次进行 GitLab 二次开发的话，我还是建议你按照<https://docs.gitlab.com/ee/install/installation.html> 进行一次基于源码的安装，这将有助于你更好地理解 GitLab 的整个架构。

为了后面更高效地解决二次开发的问题，我先和你介绍一下 GitLab 的几个主要模块：

Unicorn，是一个 Web Server，用于支持 GitLab 的主体 Web 应用；

Sidekiq，队列服务，需要 Redis 支持，用以支持 GitLab 的异步任务；

GitLab Shell，Git SSH 的权限管理模块；

Gitaly，Git RPC 服务，用于处理 GitLab 发出的 git 操作；

GitLab Workhorse，基于 Go 语言，用于接替 Unicorn 处理比较大的 http 请求。

# GitLab Application Architecture

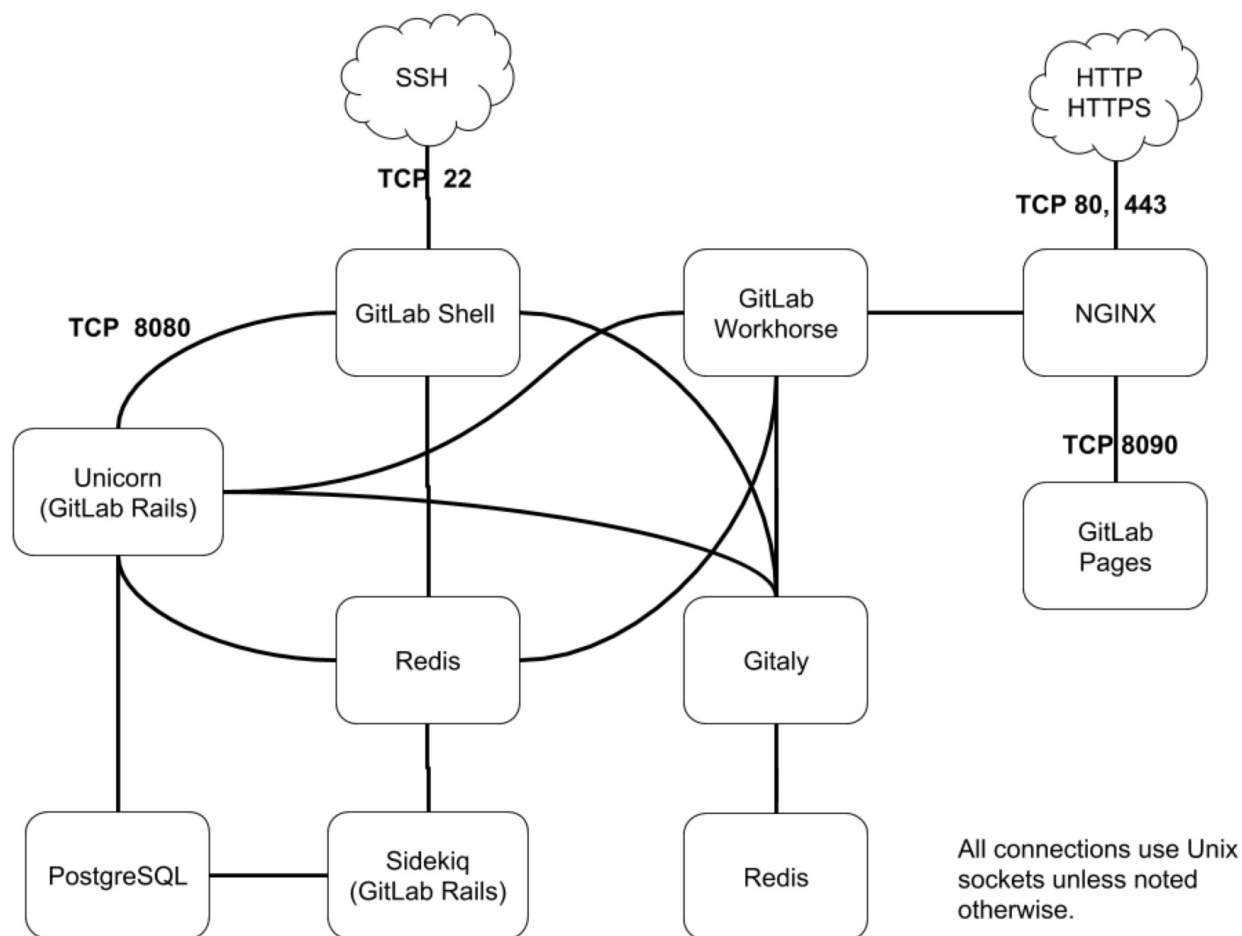


图 1 GitLab 架构图 (引自 GitLab 官网)

对 GitLab 应用层的修改，我们主要关注的是 GitLab Rails 和 GitLab Shell 这两个子系统。

接下来，我们一起看一个二次开发的具体实例吧。

## 二次开发的例子

二次开发，最常见的是对 GitLab 添加一个外部服务调用，这部分需要在 `app/models/project_services` 下面添加相关的代码。

我们可以参考 GitLab 对 Microsoft Teams 的支持方式：

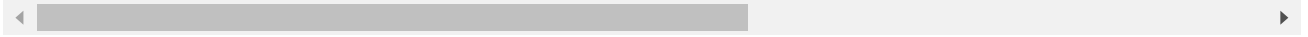
1. 在 `app/models/project_services/microsoft_teams_service.rb` 下，添加一些可配置内容及其属性，这样我们就可以在 GitLab 的 service 模块页面下看到相应的配置项了。

```
1 # frozen_string_literal: true
2
3 class MicrosoftTeamsService < ChatNotificationService
4   def title
5     'Microsoft Teams Notification'
6   end
7
8   def description
9     'Receive event notifications in Microsoft Teams'
10  end
11
12  def self.to_param
13    'microsoft_teams'
14  end
15
16  def help
17    'This service sends notifications about projects events to Microsoft Teams channels
18    To set up this service:
19    <ol>
20      <li><a href="https://msdn.microsoft.com/en-us/microsoft-teams/connectors">Getting
21      <li>Paste the <strong>Webhook URL</strong> into the field below.</li>
22      <li>Select events below to enable notifications.</li>
23    </ol>'
24  end
25
26  def webhook_placeholder
27    'https://outlook.office.com/webhook/...'
28  end
29
30  def event_field(event)
31  end
32
33  def default_channel_placeholder
34  end
35
36  def default_fields
37    [
38      { type: 'text', name: 'webhook', placeholder: "e.g. #{webhook_placeholder}" },
39      { type: 'checkbox', name: 'notify_only_broken_pipelines' },
40      { type: 'checkbox', name: 'notify_only_default_branch' }
41    ]
42  end
43
44  private
45
46  def notify(message, opts)
47    MicrosoftTeams::Notifier.new(webhook).ping(
48      title: message.project_name,
49      summary: message.summary,
50      activity: message.activity,
```


```

51     attachments: message.attachments
52   )
53 end
54
55 def custom_data(data)
56   super(data).merge(markdown: true)
57 end
58 end

```



## 2. 在 lib/microsoft\_teams/notifier.rb 内实现服务的具体调用逻辑。

 复制代码

```

1 module MicrosoftTeams
2   class Notifier
3     def initialize(webhook)
4       @webhook = webhook
5       @header = { 'Content-type' => 'application/json' }
6     end
7
8     def ping(options = {})
9       result = false
10
11       begin
12         response = Gitlab::HTTP.post(
13           @webhook.to_str,
14           headers: @header,
15           allow_local_requests: true,
16           body: body(options)
17         )
18
19         result = true if response
20       rescue Gitlab::HTTP::Error, StandardError => error
21         Rails.logger.info("#{self.class.name}: Error while connecting to #{@webhook}: #·
22       end
23
24       result
25     end
26
27     private
28
29     def body(options = {})
30       result = { 'sections' => [] }
31       result['title'] = options[:title]
32       result['summary'] = options[:summary]
33       result['sections'] << MicrosoftTeams::Activity.new(options[:activity]).prepare
34
35       attachments = options[:attachments]

```



```
36     unless attachments.blank?
37       result['sections'] << {
38         'title' => 'Details',
39         'facts' => [{ 'name' => 'Attachments', 'value' => attachments }]
40       }
41     end
42
43     result.to_json
44   end
45 end
46 end
```



以上就是一个最简单的 Service 二次开发的例子。熟悉了 Rails 和 GitLab 源码后，你完全可以以此类推写出更复杂的 Service。

## GitLab 的 HA 方案

对于研发人员数量小于 1000 的团队，我不建议你考虑 GitLab 服务多机水平扩展的方案。GitLab 官方给出了一个内存对应用户数量的参照，如下：

- 16 GB RAM supports up to 2000 users
- 128 GB RAM supports up to 16000 users

从这个配置参照数据中，我们可以看到一台高配的虚拟机或者容器可以支持 2000 名研发人员的操作，而单台物理机（128 GB 配置）足以供上万研发人员使用。

在携程，除了要支持开发人数外，还要考虑到高可用的需求，所以我们经过二次开发后做了 GitLab 的水平扩展。但是，即使在每天的 GitLab 使用高峰期，整机负载也非常低。因此，对于大部分的研发团队而言，做多机水平扩展方案的意义并不太大。

同时，实现 GitLab 的完整水平扩展方案，也并不是一件易事。



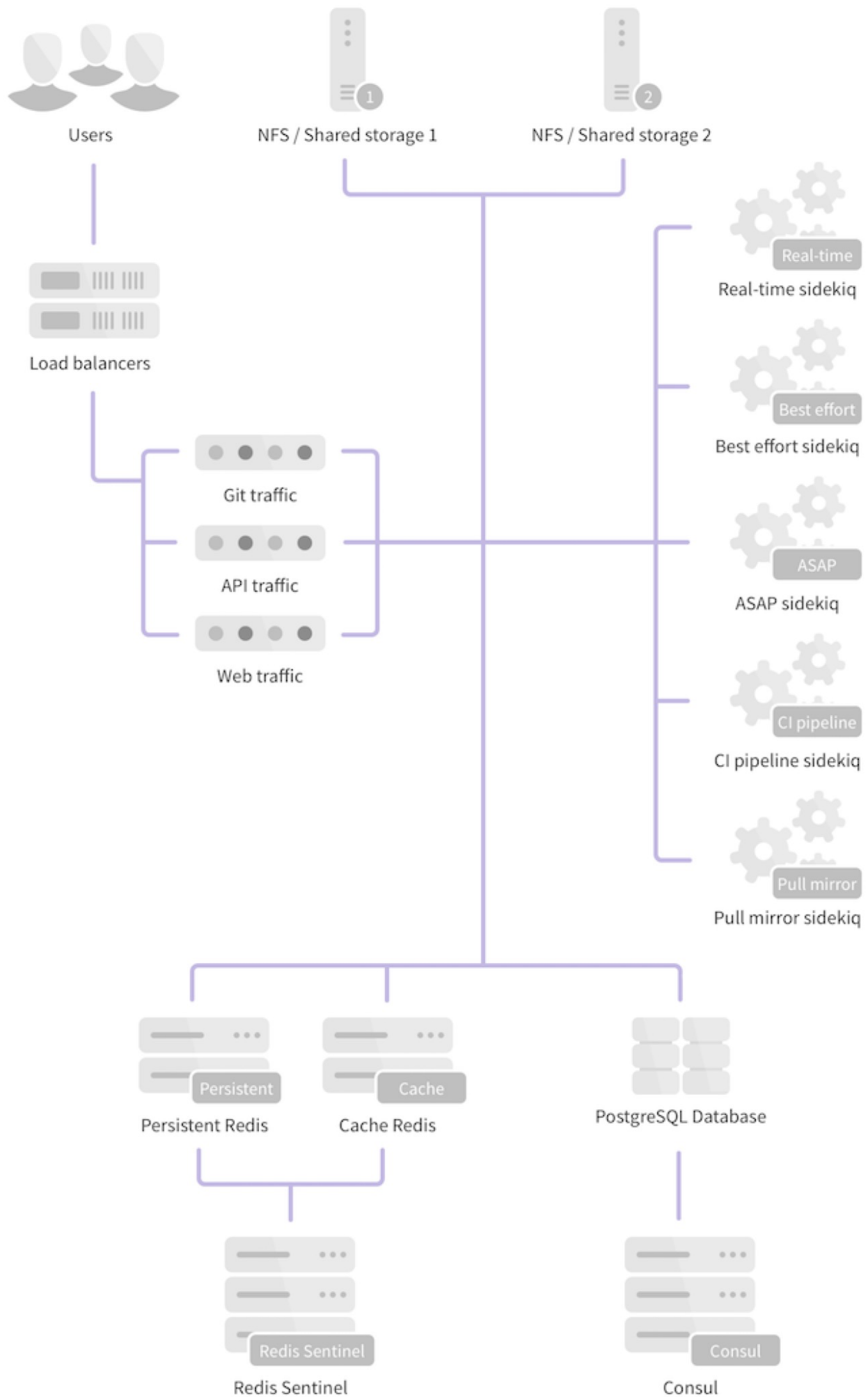


图 2 GitLab 官方 HA 方案 (引自 GitLab 官网)

我们先看一下社区版的 GitLab，官方提供的 HA 方案的整体架构图可参考图 2。从整体架构上看，PostgreSQL、Redis 这两个模块的高可用，都有通用的解决方案。而 GitLab 在架构上最大的问题是，需要通过文件系统在本机访问仓库文件。于是，**水平扩展时，如何把本地的仓库文件当做数据资源在服务器之间进行读写就变成了一个难题。**

官方推荐的方案是通过 NFS 进行多机 Git 仓库共享。但这个方案在实际使用中并不可行，git 本身是 IO 密集型应用，对于真正在性能上有水平扩展诉求的用户来说，NFS 的性能很快就会成为整个系统的瓶颈。我早期在美团点评搭建持续交付体系时，曾尝试过这个方案，当达到几百个仓库的规模时，NFS 就撑不住了。

对于水平扩展这部分内容，有一个非常不错的分享：阿里的[《我们如何为三万人的公司横向伸缩 GitLab》](#)。但是，实施这个方案，你需要吃透 Git 的底层，所以并不容易实施。

而携程的解决方案就比较简单了：

我们在应用层处理这个问题，根据 Git 仓库的 group 名字做了一个简单切分，并使用 ssh2 对于 Git 访问做一次代理，保证对于不同项目的 http 访问，能够分配到确定的机器上。

这个方案的优点是，实施起来相对简单，缺点是无法向上兼容，升级 GitLab 会比较麻烦。

当然，你还可以参考[GitLab 的官方建议](#)，并结合我分享的经验完成自己的 HA 方案。

## 如何应对代码管理的需求？

我们先一起回忆一下，上一篇文章中，我们对代码管理平台的需求，即要求能够支持 3 个团队的开发工作，且具备 code review 和静态代码检查的功能。

要实现这些需求，我需要先和你介绍一下 GitLab 提供的几个比较重要的功能。

## 了解 GitLab 提供的功能

Gitlab 作为开源的代码管理平台，其原生也提供了不少优秀的功能，可以直接帮助我们解决上一篇文章中的一些需求。这些功能主要包括：

## 1. Merge Requests

分支代码审核合并功能，关于 Merge Request 和分支策略。你可以回顾一下第四篇文章 [《一切的源头，代码分支策略的选择》](#) 和 第七篇文章 [《“两个披萨”团队的代码管理实际案例》](#) 的内容。

之后就是，我们根据不同的团队性质，选择不同的分支管理策略了。

比如，在我们的这个系统中：中间件团队只有 6 个开发人员，且都是资深的开发人员，他们在项目的向下兼容方面也做得很好，所以整个团队选择了主干开发的分支策略，以保证最高的开发效率。

同时，后台团队和 iOS 团队各有 20 个开发人员，其中 iOS 团队一般是每周三下午进行发布，所以这两个团队都选择了 GitLab Flow 的分支策略。

## 2. issues

可以通过列表和看板两种视图管理开发任务和 Bug。在携程，我们也有一些团队是通过列表视图管理 Bug，通过看板视图维护需求和开发任务。

## 3. CI/CD

GitLab 和 GitLab-ci 集成的一些功能，支持 pipeline 和一些 CI 结果的展示。携程在打造持续交付系统时，GitLab-ci 的功能还并不完善，所以也没有对此相关的功能进行调研，直接自研了 CI/CD 的驱动。

不过，由于 GitLab-ci 和 GitLab 天生的集成特性，目前也有不少公司使用它作为持续集成工作流。你也可尝试使用这种方法，它的配置很简单，可以直接参考官方文档。而在专栏中我会以最流行的 Jenkins Pipeline 来讲解这部分功能。

## 4. Integrations

Integrations 包括两部分：

GitLab service，是在 GitLab 内部实现的，与一些缺陷管理、团队协作等工具的集成服务。

Webhook，支持在 GitLab 触发代码 push、Merge Request 等事件时进行 http 消息推送。

我在下一篇文章中介绍的代码管理与 Jenkins 集成，就是通过 Webhook 以及 Jenkins 的 GitLab plugin 实现的。

理解了 GitLab 的几个重要功能后，便可以初步应对上一篇文章中的几个需求了。之后，搭建好的 GitLab 平台，满足代码管理的需求，我们可以通过三步实现：

1. 创建对应的代码仓库;
2. 配置 Sonar 静态检查;
3. 解决其他设置。

接下来，我和你分享一下，每一步中的关键点，以及具体如何满足相应的代码需求。

## 第一步，创建对应的代码仓库

了解了 GitLab 的功能之后，我们就可以开始建立与需求相对应的 Projects 了。

因为整个项目包括了中间件服务、业务后台服务，以及业务客户端服务这三个职责，所以相应的我们就需要在 GitLab 上创建 3 个 group，并分别提交 3 个团队的项目。

对于中间件团队，我们创建了一个名为 framework/config 的项目。这个项目最终会提供一个配置中心的服务，并且生成一个 config-client.jar 的客户端，供后台团队使用。

后台服务团队的项目名为：waimai/waimai-service，产物是一个 war 包。

移动团队创建一个 React Native 项目 mobile/waimai-app。

## 第二步，配置 Sonar 静态检查

创建了三个代码仓库之后，为了后续在构建时进行代码静态检查，所以现在我们需要做的就是配置代码静态扫描工具。而在这里，我依旧以 Sonar 为例进行下面详解。

我们在使用 SonarQube 服务进行静态检查时，需要注意的问题包括：

Sonar 的搭建比较简单，从 <https://www.sonarqube.org/downloads/> 下载 Sonar 的压缩包以后，在 conf/sonar.properties 中配置好数据库的连接串，然后执行 bin/linux-x86-64/sonar.sh start 命令。之后，我们可以再查看一下日志 logs/sonar.log，当日志提示 “SonarQube is up” 时就可以通过 http://localhost:9000 访问 sonar 了。（如果你有不明白的问题，可以参考

<https://docs.sonarqube.org/display/SONAR/Installing+the+Server>)

和 GitLab 的扩展一般只能通过二次开发不同，Sonar 通过 plugin 的方式就可以完成扩展。在 extensions/plugins 目录下面已经预置了包含 Java、Python、PHP 等语言支持，以及 LDAP 认证等插件。你可以通过直接安装插件的方式进行扩展。

插件安装完成后，我们就可以尝试在本地使用 Maven 命令，对中间件和后台团队的 Java 项目进行静态检查了，React Native 项目则是通过 sonar-scanner 配合 ESLint 完成静态检查的。

GitLab 的 Merge Request 需要通过触发 Jenkins 构建 Sonar 来驱动代码的持续静态检查，至于如何集成我会在下一篇文章中和你详细介绍。

关于静态检查的更多知识点，你可以再回顾一下第二十五篇文章 [《代码静态检查实践》](#)。

### 第三步，解决其他设置

经过创建对应的代码仓库、配置 Sonar 静态检查这两步，再配合使用 GitLab 提供的 Merge Request、Issues、CI/CD 和 Integration 功能，代码管理平台基本上就算顺利搭建完毕了。

之后剩余的事情包括：

1. 为项目添加开发者及对应的角色；
2. 根据分支策略，设定保护分支，仅允许 Merge Request 提交；
3. 创建功能分支。

至此，我们需要的代码管理平台就真的搭建好了，开发人员可以安心写代码了。

### 总结及实践

在上一篇文章中，我们已经清楚了整个持续交付体系中，代码管理平台要具备的功能，所以今天我就在此基础上，和你一起使用 GitLab 完成了这个代码管理平台的搭建。

首先，我介绍了 GitLab 的安装及配置过程，并通过 Microsoft Teams 这个具体案例，介绍了如何完成 GitLab 的二次开发，以应对实际业务的需求。同时，我还介绍了 GitLab 的高可用方案。

然后，我针对代码管理平台要支持 3 个团队的 code review 和代码静态扫描的需求，和你分享了如何使用三步实现这些需求：

第一步，创建对应的代码仓库；

第二步，配置 Sonar 静态检查；

第三步，解决其他设置。

完成以上工作后，我们的代码管理平台就可以正式运作了，也为我们下一篇文章要搭建的编译构建平台做好了准备。

最后，希望你可以按照这篇文章的内容，自己动手实际搭建一套 GitLab，以及配套的 Sonar 服务。

千里之行始于足下，如果搭建过程中，遇到了什么问题，欢迎给我留言一起讨论。

 极客时间

# 持续交付36讲

量身定制你的持续交付体系

王潇俊 携程系统研发部总监



新版升级：点击「 请朋友读」，10位好友免费读，邀请订阅更有**现金**奖励。

© 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 34 | 快速构建持续交付系统（一）：需求分析

下一篇 36 | 快速构建持续交付系统（三）：Jenkins 解决集成打包问题

精选留言 (3)

 写留言



熙

2018-09-28



非常赞的分享

展开 ▾



铭熙

2018-09-24



GitLab的高可用，能否详细说下携程的实现？

展开 ▾

作者回复: 携程的方案就是做sharding+互备，利用ssh2作为代理达到对ssh请求的分流，这样每台服务器上只服务特定仓库，再配置仓库互备，在部分服务器出现问题时修改sharding策略引流到备机就可以了



江湖小虾

2018-09-22



谢谢，老师的分享

展开 ▾