GitHub开发指南

极客学院出版

前言

本指南的目的是教会你如何实际运用 Github API。我们将会涉及到你需要知道的所有内容，从身份认证到操作结果，以及怎么将结果与其他服务相结合。

本指南是 GitHub 官方文档 Development Guides (https://developer.github.com/guides/) 的中文翻译版本。

这里的所有指南都有一个项目，并且每个项目都将被存储和记录在我们公共的 platform-samples (https://github.com/github/platform-samples) 库。

可以随意的 Fork，Clone，和改善这些指南。

适用人群

本指南针对需要通过 GitHub API 来构建服务的用户。

学习前提

学习本教程需要开发者了解 Git 基础，熟悉 GitHub 操作和服务器部署相关知识。

你将学会

- GitHub API 基础操作
- 用户认证
- SSH agent 转发
- 数据转换成图表
- 分页
- 自动部署
- ...

图片 .1 image

更新日期	更新内容
2015-06-03	第一版发布

目录

1 准备开始

让我们来通过一些核心的 API 概念为我们处理一些日常用例。

概述

大部分应用都会使用你选择的语言的一个现有 封装好的库 (https://developer.github.com/libraries/) ，但是先熟悉内部的 API HTTP 方法是很重要的。

通过 cURL (http://curl.haxx.se/) 来进行试验是最简单的。

Hello World

首先，我们来测试我们的设置是否正确。打开一个命令提示符，输入下面的命令（不需要 $ 符号）

```
$ curl https://api.github.com/zen

Keep it logically awesome.
```

返回的内容将会是我们的设计理念中随机抽取的一条。

接下来，我们向 Chris Wanstrath's (https://github.com/defunkt) GitHub profile (https://developer.github.com/v3/users/#get-a-single-user) 发送一个 GET 请求:

```
# GET /users/defunkt
$ curl https://api.github.com/users/defunkt

{
  "login": "defunkt",
  "id": 2,
  "url": "https://api.github.com/users/defunkt",
  "html_url": "https://github.com/defunkt",
  ...
}
```

Mmmmm，这看起来像 JSON (http://en.wikipedia.org/wiki/JSON) 。让我们添加 `-i` 参数来包含头部信息。

```
$ curl -i https://api.github.com/users/defunkt

HTTP/1.1 200 OK
Server: GitHub.com
Date: Sun, 11 Nov 2012 18:43:28 GMT
Content-Type: application/json; charset=utf-8
Connection: keep-alive
Status: 200 OK
```

```
ETag: "bfd85cbf23ac0b0c8a29bee02e7117c6"
X-RateLimit-Limit: 60
X-RateLimit-Remaining: 57
X-RateLimit-Reset: 1352660008
X-GitHub-Media-Type: github.v3
Vary: Accept
Cache-Control: public, max-age=60, s-maxage=60
X-Content-Type-Options: nosniff
Content-Length: 692
Last-Modified: Tue, 30 Oct 2012 18:58:42 GMT

{
  "login": "defunkt",
  "id": 2,
  "url": "https://api.github.com/users/defunkt",
  "html_url": "https://github.com/defunkt",
  ...
}
```

在返回的头部信息中有一些有趣的数据。如预期那样， `Content-Type` 属性的值是 `application/json` 。

每一个以 `X-` 开头的头部都是自定义头，它们都不包含于 HTTP 标准。让我们看看其中的几个：

- `x-GitHub-Media-Type` 的值为 `github.v3` 。这告诉我们返回值的媒体类型 (https://developer.github.com/v3/media/) 。媒体类型帮助在 API v3 中确定我们的输出的版本信息。我们将在后面更深入探讨。
- 注意 `X-RateLimit-Limit` 和 `X-RateLimit-Remaining headers` 。这一对头部信息标示出了在一个滚动时间周期内（一般是一个小时）一个客户端能够发起多少请求 (https://developer.github.com/v3/#rate-limiting) 和这些请求多少已经完成。

身份认证

没认证的客户端每小时可以制造60个请求。如果想制造更多，我们需要进行认证。实际上，使用 GitHub API 做任何有趣一点的事情都会要求认证。

基础

GitHub API 最简单的认证方式是使用你的 GitHub 用户名和密码来通过基础认证。

```
$ curl -i -u <your_username> https://api.github.com/users/defunkt

Enter host password for user '<your_username>':
```

这个 `-u` 参数设置用户名，cURL 会提示你填写密码。你可以使用 `-u "username:password"` 来避免这个提醒，但这会使你的密码被记录在 shell 的历史记录中，所以并不推荐这种做法。验证时，你会看到你的速度限制会升到每小时 5000 个请求，这个会在 `X-RateLimit-Limit` 头部信息中标示。

双重认证

如果你启用了双重认证 (https://help.github.com/articles/about-two-factor-authentication)，这个 API 会在以上请求中返回 `401 Unauthorized` 错误码（其他的 API 请求也一样）：

```
$ curl -i -u <your_username> https://api.github.com/users/defunkt

Enter host password for user '<your_username>':

HTTP/1.1 401 Unauthorized
X-GitHub-OTP: required; :2fa-type

{
  "message": "Must specify two-factor authentication OTP code.",
  "documentation_url": "https://developer.github.com/v3/auth#working-with-two-factor-authentication"
}
```

避免这个错误最简单的方式是创建一个 OAuth 令牌和使用 OAuth 认证，而不是使用简单的基础认证。在下面的 OAuth 部分可以看到更详细的信息。

获取自己配置文件

当认证通过时，你可以利用与 GitHub 账户相关联的权限。例如，尝试获取你的用户配置文件：

```
$ curl -i -u <your_username> https://api.github.com/user

{
  ...
  "plan": {
    "space": 2516582,
    "collaborators": 10,
    "private_repos": 20,
    "name": "medium"
  }
  ...
}
```

这一次，除了和早先我们获取 @defunkt (https://github.com/defunkt) 同样的公共信息集合外，你还将看到你自己用户配置的非公共信息。比如，你将在返回值看到一个 `plan` 对象，它给出了账户的 GitHub 计划的细节。

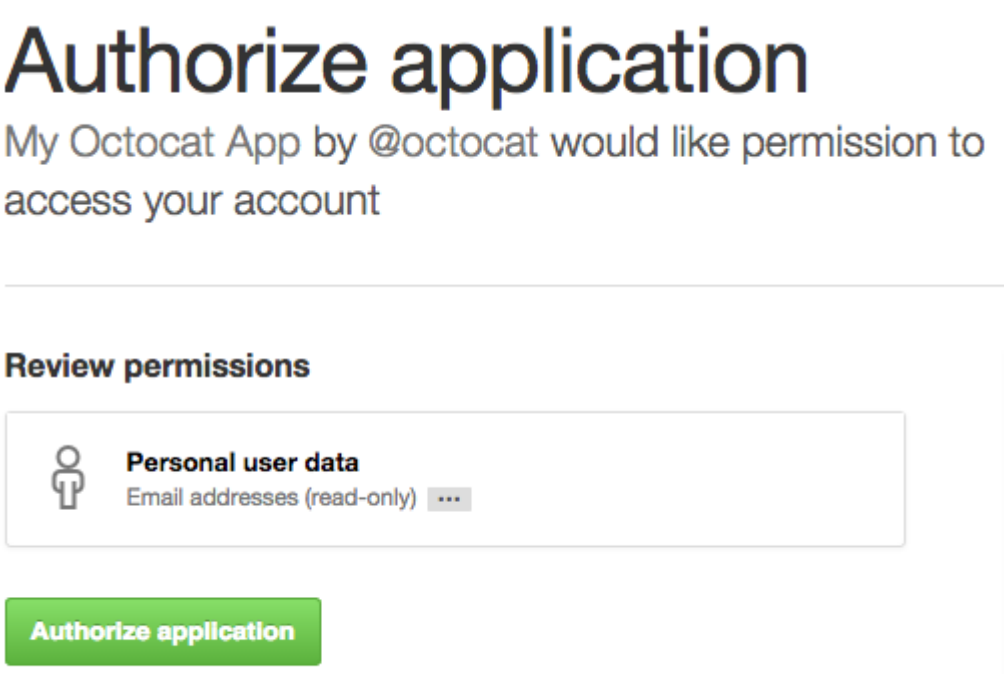
OAuth

基本认证虽然方便，但是并不理想，因为你不应该将你的 GitHub 用户名和密码共享给任何人。需要通过 API 读写另一个用户的私有信息时必须使用 OAuth。

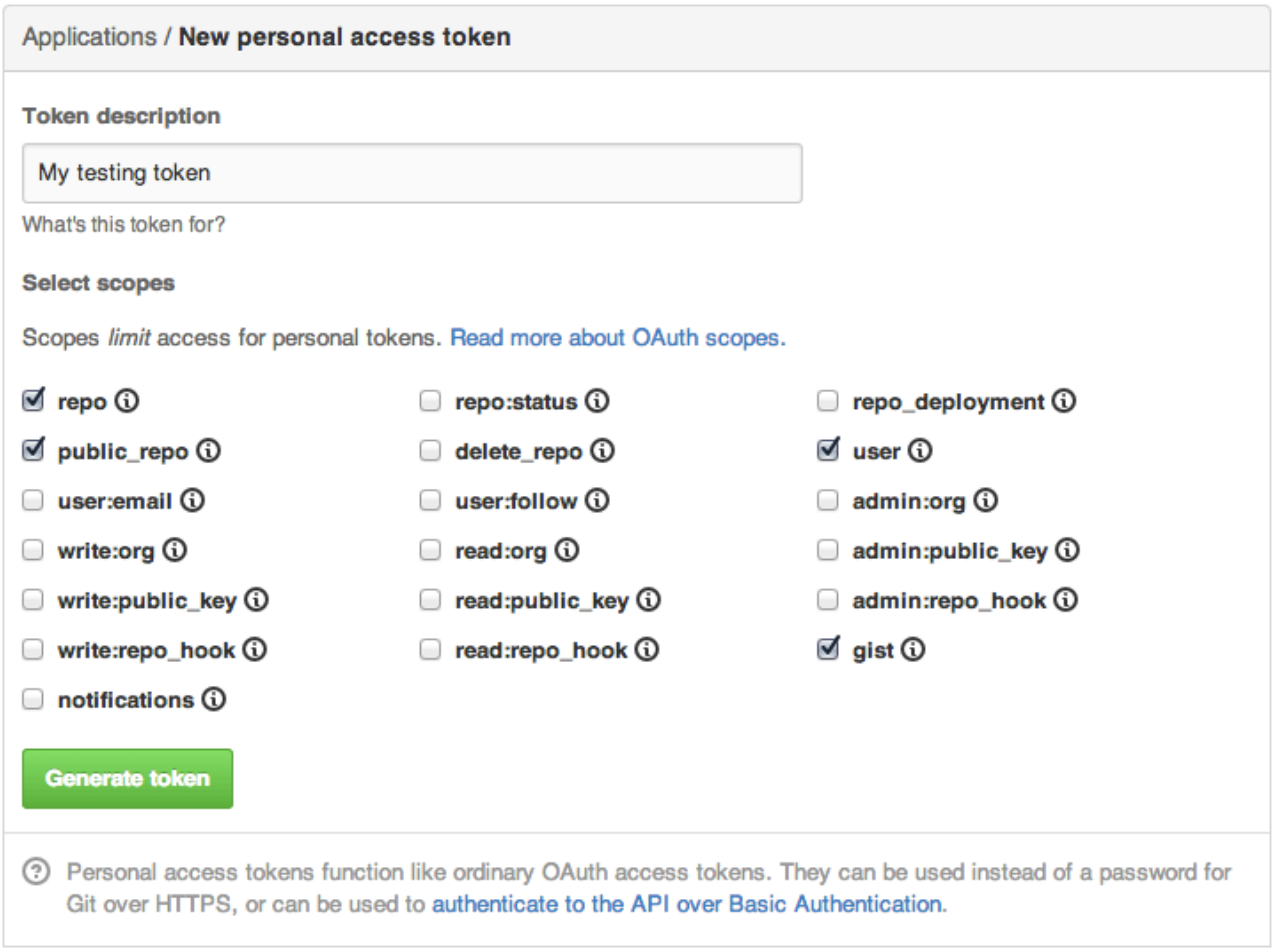
OAuth 用令牌（token）替代了用户名和密码。令牌有两大特色：

- 可撤销访问：用户能够在任何时候撤销对第三方 app 的认证。
- 有限访问：用户能够在授权一个第三方 app 之前检验特定的准入权限。

一般来说，令牌能够通过一个 web 流 (https://developer.github.com/v3/oauth/#web-application-flow) 来创建。一个应用将用户跳转到 Github 登陆。然后 GitHub 会显示对话框来标明应用的名字，同时也显示应用曾经被授予的用户权限。当用户授权之后，GitHub 会将用户重定向跳转回应用：



然而，开始使用 OAuth 令牌并不需要你配置整个 web 流。获取一个令牌的简单方法是通过个人准入令牌设置页 (https://github.com/settings/tokens) 创建一个个人准入令牌 (https://help.github.com/articles/creating-an-access-token-for-command-line-use)：



同时，授权 API (https://developer.github.com/v3/oauth_authorizations/#create-a-new-authorization) 使得通过基本授权创建一个 OAuth 令牌变得简单。试试粘贴并运行以下命令：

```
$ curl -i -u <your_username> -d '{"scopes": ["repo", "user"], "note": "getting-started"}' \
https://api.github.com/authorizations

HTTP/1.1 201 Created
Location: https://api.github.com/authorizations/2
Content-Length: 384

{
  "scopes": [
    "repo",
    "user"
  ],
  "token": "5199831f4dd3b79e7c5b7e0ebe75d67aa66e79d4",
  "updated_at": "2012-11-14T14:04:24Z",
  "url": "https://api.github.com/authorizations/2",
  "app": {
    "url": "https://developer.github.com/v3/oauth/#oauth-authorizations-api",
```

```
    "name": "GitHub API"
  },
  "created_at": "2012-11-14T14:04:24Z",
  "note_url": null,
  "id": 2,
  "note": "getting-started"
}
```

这个小调用里面包含了很多过程，让我们来分解一下。首先，`-d` 标志表明我们正在做一个 `POST` 调用，使用的是 `application/x-www-form-urlencoded` 内容类型(和 `GET` 相对)。所有对 GitHub API 发起的 `POST` 请求都必须用 JSON 编写。

接下来，让我们看看我们在这个请求播送的 `scopes` 字段。当创建一个新的令牌时，我们包含了一个可选的数组 *scopes*，或用来标示这个令牌能够获取的信息的准入等级。在这个例子中，我们设置该令牌拥有 `repo` 权限，这个权限将授予用户读写公共和私有仓库的权限；该令牌还有 `user` 域权限，这将授予用户读写公共和私有用户简介数据。查看 区域文档 (https://developer.github.com/v3/oauth/#scopes) 可以获得所有区域的列表。为了避免因可能的侵入行为吓到用户，你应该只请求你的应用所实际需要的权限。 `201` 的状态码告诉我们调用是成功的，并且返回的 JSON 包含了新 OAuth 令牌的详细信息。

如果你已经打开双重授权 (https://help.github.com/articles/about-two-factor-authentication) ,API 将对上述请求返回前面所述的 `401 Unauthorize` 错误码。你能够通过在 X-GitHub-OTP 请求头 (https://developer.github.com/v3/auth/#working-with-two-factor-authentication) 包含一个 2FA OTP 码来回避这个错误:

```
$ curl -i -u <your_username> -H "X-GitHub-OTP: <your_2fa_OTP_code>" \
    -d '{"scopes": ["repo", "user"], "note": "getting-started"}' \
    https://api.github.com/authorizations
```

如果你在一个移动应用打开了 2FA ，可以通过你的手机的一次性密码获取一个 OTP 码。如果你通过短信打开了 2FA，发起此请求后，你将受到一条包含你的 OTP 码的短信。

现在，我们能够在剩下的例子中使用这个40字节的令牌来替代用户名和密码。让我们再次获取我们的个人信息，这一次我们使用 OAuth:

```
$ curl -i -H 'Authorization: token 5199831f4dd3b79e7c5b7e0ebe75d67aa66e79d4' \
    https://api.github.com/user
```

请向对待密码一样对待 OAuth 令牌! 不要和其他用户分享令牌或者将令牌存储在不安全的地方。这些例子中的令牌是伪造的，名字也已经修改过，以免影响无关用户。

现在我们知道了如何进行授权请求，接下来我们来看Repositories API (https://developer.github.com/v3/repos/) 。

Repositories

几乎所有有意义的 GitHub API 使用会包含了某种程度的 repositories 信息。我们能够像我们前面获取用户详细信息一样获取 repository 详情：

```
$ curl -i https://api.github.com/repos/twbs/bootstrap
```

用同样的方式，我们能够为授权用户查看 repositories (https://developer.github.com/v3/repos/#list-your-repositories)：

```
$ curl -i -H 'Authorization: token 5199831f4dd3b79e7c5b7e0ebe75d67aa66e79d4' \
  https://api.github.com/user/repos
```

或者我们能够查看另一个用户的 repositories (https://developer.github.com/v3/repos/#list-user-repositories)：

```
$ curl -i https://api.github.com/users/technoweenie/repos
```

或者，我们能够查看一个组织的 repositories (https://developer.github.com/v3/repos/#list-organization-repositories)：

```
$ curl -i https://api.github.com/orgs/mozilla/repos
```

这些调用返回的信息将依赖于我们怎样被授权的：

- 使用基本授权，返回将包含所有用户在 github.com 能够查看的 repositories。
- 使用 OAuth 的话，如果 OAuth 令牌包含了 `repo` 域，则将只返回私有 repositories。

就像文档 (https://developer.github.com/v3/repos/) 中指出的一样，这些方法包含了一个 `type` 参数以便根据用户对 repository 拥有的访问权限类型来过滤返回的 repositories。通过这种方式，我们能够单独获取直接拥有的 repositories，组织的 repositories，或者用户通过一个小组合作的 repositories。

```
$ curl -i "https://api.github.com/users/technoweenie/repos?type=owner"
```

在这个例子中，我们获取了 technoweenie 用户拥有的 repositories，而不是那些他正在和其他人合作的 repositories。注意上面被引号包括的 URL。依赖于你的 shell 设置， cURL 有时候会要求 URL 被引号包括，否则将忽略查询字符串。

创建一个 repository

获取一个已存在的 repository 的信息是非常常见的应用，但是 GitHub API 也支持创建新的 repositories。为了创建一个 repository (https://developer.github.com/v3/repos/#create)，我们需要 `POST` 一些包含细节和配置选项的 JSON：

```
$ curl -i -H 'Authorization: token 5199831f4dd3b79e7c5b7e0ebe75d67aa66e79d4' \
  -d '{ \
    "name": "blog", \
    "auto_init": true, \
    "private": true, \
    "gitignore_template": "nanoc" \
  }' \
  https://api.github.com/user/repos
```

在这里最小例子里面，我们为我们的博客（可能是架设在 GitHub Pages (http://pages.github.com/)）创建了一个新的 repository。 虽然博客将会是公共的，但是我们将 repository 设置为私有的。同样，在这个请求里，我们用一个 README 文件和一个 nanoc-flavored .gitignore 模板初始化这个 repository。

创建的 repository 将可以在 `https://github.com/<your_username>/blog` 找到。要创建一个你拥有的组织下的 repository，仅需要修改一个 API 方法，将 `/user/repos` 变更为 `/orgs/<org_name>/repos`。

接下来，让我们获取我们新创建的 repository：

```
$ curl -i https://api.github.com/repos/pengwynn/blog

HTTP/1.1 404 Not Found

{
  "message": "Not Found"
}
```

这是怎么一回事？返回了一个 `404` 错误。因为我们将 repository 设置为私有的，我们需要授权才能查看它。如果你是一个 HTTP 的老用户，你可能会期望一个 `403` 而不是 `404`。但是因为我们不想泄露私有 repositories 的任何信息，所以 GitHub API 在这种情况下返回一个 `404`，表示我们不能确定或者否认这个 repository 的存在。

Issues

GitHub Issues UI 目标在于当你远离你的方向的时候給你提供适合的工作流。通过 GitHub Issues API (https://developer.github.com/v3/issues/)，你可以使用其他工具提取数据或者创建 Issues 来給你的团队创建一个工作流。

就像 github.com, 这个 API 給认证用户提供了一些查看 issues 的方法。查看你的全部 issues (https://developer.github.com/v3/issues/#list-issues)，使用 `GET/issues`：

```
$ curl -i -H 'Authorization: token 5199831f4dd3b79e7c5b7e0ebe75d67aa66e79d4' \
https://api.github.com/issues
```

想要获取你其中一个组织下的 issues (https://developer.github.com/v3/issues/#list-issues)，使用 `GET /orgs/<org>/issues`：

```
$ curl -i -H 'Authorization: token 5199831f4dd3b79e7c5b7e0ebe75d67aa66e79d4' \
https://api.github.com/orgs/rails/issues
```

我们也可以获取单一 repository 中的全部 issues (https://developer.github.com/v3/issues/#list-issues-for-a-repository)：

```
$ curl -i https://api.github.com/repos/rails/rails/issues
```

分页

一个项目的 Rails 大小拥有上千个 issues。我们将需要进行分页 (https://developer.github.com/v3/#pagination) 处理，使用多种 API 来获取数据。重复最后一条命令，这时候需要注意回复的 headers：

```
$ curl -i https://api.github.com/repos/rails/rails/issues

HTTP/1.1 200 OK

Link: <https://api.github.com/repos/rails/rails/issues?page=2>; rel="next",
<https://api.github.com/repos/rails/rails/issues?page=14>; rel="last"
```

`Link` 的头部信息 (http://www.w3.org/wiki/LinkHeader) 将提供一个方法来响应链接外部资源，用这种方式来获取额外页面的数据。当我们请求获取超过30条（默认的页面大小）issues 的时候，这个 API 将会告诉我们哪里可以获取下一页和最后一页的结果。

创建一个 issue

我们已经知道怎么将 issues 分页，现在让我们使用 API 来创建一个 issue (https://developer.github.com/v3/issues/#create-an-issue) 。

要创建一个 issue，我们需要先通过认证，所以在请求的头部需要传递 OAuth 令牌。同时，我们需要传递 JSON body 中的 title，body 和 labels 到我们想要创建 issue 的 repository 下面的 `/issues` 路径：

```
$ curl -i -H 'Authorization: token 5199831f4dd3b79e7c5b7e0ebe75d67aa66e79d4' \
-d '{ \
    "title": "New logo", \
    "body": "We should have one", \
    "labels": ["design"] \
  }' \
https://api.github.com/repos/pengwynn/api-sandbox/issues

HTTP/1.1 201 Created
Location: https://api.github.com/repos/pengwynn/api-sandbox/issues/17
X-RateLimit-Limit: 5000

{
  "pull_request": {
    "patch_url": null,
    "html_url": null,
    "diff_url": null
  },
  "created_at": "2012-11-14T15:25:33Z",
  "comments": 0,
  "milestone": null,
  "title": "New logo",
  "body": "We should have one",
  "user": {
    "login": "pengwynn",
    "gravatar_id": "7e19cd5486b5d6dc1ef90e671ba52ae0",
    "avatar_url": "https://secure.gravatar.com/avatar/7e19cd5486b5d6dc1ef90e671ba52ae0?d=https://a248.e.akamai.ne
    "id": 865,
    "url": "https://api.github.com/users/pengwynn"
  },
  "closed_at": null,
  "updated_at": "2012-11-14T15:25:33Z",
  "number": 17,
  "closed_by": null,
  "html_url": "https://github.com/pengwynn/api-sandbox/issues/17",
```

```
  "labels": [
    {
      "color": "ededed",
      "name": "design",
      "url": "https://api.github.com/repos/pengwynn/api-sandbox/labels/design"
    }
  ],
  "id": 8356941,
  "assignee": null,
  "state": "open",
  "url": "https://api.github.com/repos/pengwynn/api-sandbox/issues/17"
}
```

这个 response 給我们回复了新创建 issue 的一系列指针（pointers），有响应头的 `位置` 还有 JSON response 的 `url` 域。

条件请求

作为一个好的api用户很重要的一点就是缓存信息（从来不改变的）以配合api的速度限制。这个 API 提供条件请求和帮你做正确的事。试想一下我们用来获取 defunkt’s 配置的第一条命令：

```
$ curl -i https://api.github.com/users/defunkt

HTTP/1.1 200 OK
ETag: "bfd85cbf23ac0b0c8a29bee02e7117c6"
```

除了 JSON body，我们注意 HTTP 的状态码200和 ETag 的头部信息。ETag (http://en.wikipedia.org/wiki/HTTP_ETag) 是 response 的“指纹”。如果我们在随后的访问中传递它，我们就可以告诉 API 让它再一次回复之前的 resource，前提是它已经发生改变。

```
$ curl -i -H 'If-None-Match: "bfd85cbf23ac0b0c8a29bee02e7117c6"' \
https://api.github.com/users/defunkt

HTTP/1.1 304 Not Modified
```

状态码304表明这个资源在我们之前访问到现在时间内都没有发生改变，并且这个 response 将不包含 body。作为奖励，这个304响应将不会影响你的速率限制。

喔！现在我们知道了 GitHub API 的基础。

- 基础 & OAuth 认证
- Fetching 和 创建 repositories 和 issues

- 条件请求

继续学习下一个 API 指南 身份认证基础 () !

2

身份认证基础

在这一节，我们将重点讲身份认证的基础知识。明确地说，我们将使用Sinatra (http://www.sinatrarb.com/) 创建一个 ruby 服务，该服务将用几种不同的方式来实现一个应用的 web 流程。

> 你能够从平台范例仓库 (https://github.com/github/platform-samples/tree/master/api/) 下载这个工程的完整源代码。

注册你的应用

首先，你需要 注册你的应用 (https://github.com/settings/applications/new) 。每一个已注册的 OAuth 应用将被指定一个唯一的 Client ID 和 Client Secret。注意不要共享你的 Client Secret！包括将该字符串提交到你的 repo 中。

你能够根据你的喜好任意填写每一个信息，除了**授权回调 URL**。它无疑是配置你的应用最重要的部分。它是 Github 在成功认证用户之后返回的回调 URL。

因为我们是运行一个普通的 Sinatra 服务，本地实例的地址被设置为 `http://localhost:4567` 。所以让我们将回调 URL 填写为 `http://localhost:4567/callback` 。

接受用户授权

现在，让我们开始编写我们简单的服务。创建名为 *server.rb* 的文件并且将以下内容粘贴到文件中：

```
require 'sinatra'
require 'rest-client'
require 'json'

CLIENT_ID = ENV['GH_BASIC_CLIENT_ID']
CLIENT_SECRET = ENV['GH_BASIC_SECRET_ID']

get '/' do
  erb :index, :locals => {:client_id => CLIENT_ID}
end
```

你的 client ID 和 client secret 密匙来自你的 应用配置页 (https://github.com/settings/applications) 。你绝不应该将这些值存储于 Github 或其他公共区域。我们建议将它们保存为 环境变量 (http://en.wikipedia.org/wiki/Environment_variable#Getting_and_setting_environment_variables) ，这也是我们这里所做的。

接下来，在 *views/index.erb* 中，粘贴以下内容：

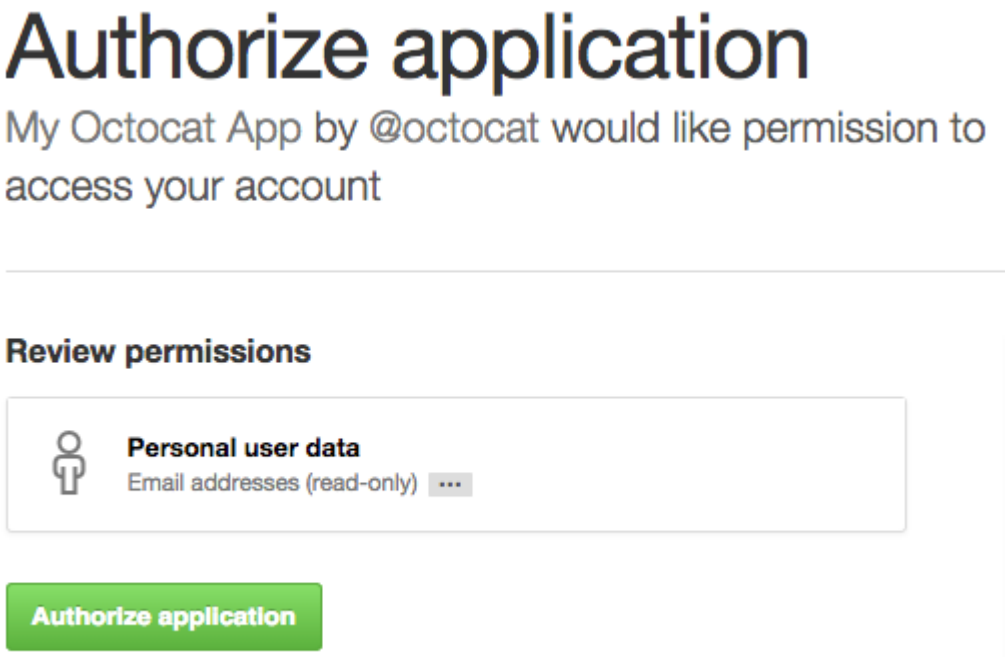
```
<html>
  <head>
```

```
  </head>
  <body>
    <p>
      Well, hello there!
    </p>
    <p>
      We're going to now talk to the GitHub API. Ready?
      <a href="https://github.com/login/oauth/authorize?scope=user:email&client_id=<%= client_id %>">Click here</a> to
    </p>
    <p>
      If that link doesn't work, remember to provide your own <a href="/v3/oauth/#web-application-flow">Client ID</a>!
    </p>
  </body>
</html>
```

（如果你不熟悉 Sinatra 是如何工作的，我们建议阅读 Sinatra 指南 (https://github.com/sinatra/sinatra-book/blob/master/book/Introduction.markdown#hello-world-application) ）

同样的，注意代码中的 URL 使用 `scope` 查询参数来定义应用程序所要求的权限区域 (https://developer.github.com/v3/oauth/#scopes) (scopes)。对于我们的应用，我们请求 `user:email` 权限区域来读取私有 email 地址。

在你的浏览器中打开 `http://localhost:4567` 。点击该链接后，你将跳转至 GitHub，并且显示类似以下对话框：



如果你信任你自己，点击 Authorize App。哇哦， Sinatra 跳出来一个 `404` 错误。这是怎么回事?

好吧，记得我们指定了一个回调 URL 为 `callback` 吗？我们并没有为它提供路由，所以 GitHub 在验证 app 之后，不知道把用户往哪里丢。现在我们来解决这个问题！

提供一个 callback

在 *server.rb* 中，加入一个 route 来指明 callback 应该做什么：

```
get '/callback' do
  # get temporary GitHub code...
  session_code = request.env['rack.request.query_hash']['code']

  # ... and POST it back to GitHub
  result = RestClient.post('https://github.com/login/oauth/access_token',
                           {:client_id => CLIENT_ID,
                            :client_secret => CLIENT_SECRET,
                            :code => session_code},
                            :accept => :json)

  # extract the token and granted scopes
  access_token = JSON.parse(result)['access_token']
end
```

在一次成功的 app 授权认证之后， GitHub 提供了一个临时的 `code` 值。你将需要将这个值 `POST` 回 GitHub 以交换一个 `access_token` 。我们使用rest-client (https://github.com/archiloque/rest-client) 来简化我们的 GET 和 POST HTTP 请求。注意，你可能永远不会通过 REST 来访问这些 API 。对于一个更加正式的应用，你很可能使用一个你选择的语言所写的库 (https://developer.github.com/libraries/) 。

确认被授予的权限区域

在此之后，用户将能够编辑你请求的权限区域 (https://developer.github.com/changes/2013-10-04-oauth-changes-coming/) ，你的应用也可能被授予少于你默认请求的数量的权限区域。所以，在你使用该 token 进行任何请求前，你应该确定用户授予了该 token 哪些权限区域。

被授予的权限区域被作为交换 token 时返回值的一部分被返回。

```
# check if we were granted user:email scope
scopes = JSON.parse(result)['scope'].split(',')
has_user_email_scope = scopes.include? 'user:email'
```

在我们的应用中，我们使用 `scopes.include?` 来检查我们是否被授予了 `user:email` 区域的权限，我们需要使用该权限来获取授权用户的私人 email 地址。如果应用程序要求更多其他区域的权限，我们也可以以同样的方式检查。

还有，因为权限区域之间有着继承的关系，你必须检查你被授予了所请求的最低级别的权限。比如说，如果应用请求了 `user` 区域权限，但是它可能只被授予了 `user:email` 区域权限。在这种情况下，应用将不会被授予它所请求的权限，但是已经被授予的区域权限仍然是有效的。

仅在进行请求之前检查区域授权情况是不够的，因为用户可能在你检查授权情况和进行实际请求之间改变了区域权限。如果这种情况发生了，原本你预计成功的请求，可能会失败并返回一个 `404` 或 `401` 状态，或者返回一个不同的信息子集。

为了让你能够更优雅地处理这些情况，所有有效 token 发起的 API 请求的返回值都包含一个 `X-OAuth-Scopes` 头部。这个头部包含了该 token 用来发起请求的区域列表。除此之外，授权 API 还提供了一个终端来检查一个 token 的有效性 (https://developer.github.com/v3/oauth_authorizations/#check-an-authorization)。使用这个信息来检测 token 授权区域的改变，并且告知你的用户可用应用功能的改变。

发起已认证请求

最后，使用这个 access token，你将能够作为一个已登录用户发起已认证的请求。

```
# fetch user information
auth_result = JSON.parse(RestClient.get('https://api.github.com/user',
                                        {:params => {:access_token => access_token}}))

# if the user authorized it, fetch private emails
if has_user_email_scope
  auth_result['private_emails'] =
    JSON.parse(RestClient.get('https://api.github.com/user/emails',
                              {:params => {:access_token => access_token}}))

erb :basic, :locals => auth_result
```

我们能够使用我们的结果做任何我们想要的事。在这里，我们仅仅是将它们直接输出到 *basic.erb* 中：

```
<p>Hello, <%= login %>!</p>
<p>
  <% if !email.nil? && !email.empty? %> It looks like your public email address is <%= email %>.
  <% else %> It looks like you don't have a public email. That's cool.
  <% end %>
</p>
<p>
```

```
  <% if defined? private_emails %>
  With your permission, we were also able to dig up your private email addresses:
  <%= private_emails.map{ |private_email_address| private_email_address["email"] }.join(', ') %>
  <% else %>
  Also, you're a bit secretive about your private email addresses.
  <% end %>
</p>
```

实现持久授权

如果我们要求用户每次进入网页的时候都需要登录 app ，那是非常糟糕的。例如，尝试直接打 `http://localhost:4567/basic` 。你将看到一个报错。

假如我们能够跳过整个“点击这里”的过程，而是仅仅记住它，只要用户登录了 GitHub，它们就能够使用这个应用，那会怎样？请保持淡定，因为这就是接下来我们要做的。

我们上面缩写的小服务器是非常简单的。为了能够嵌入一些智能的认证机制，我们将转而使用回话来保存 token。这将使得认证对用户来说是透明的。

另外，因为我们要在会话中保持授权区域，我们需要处理用户在我们检查之后更新了区域，或者撤消了标识的情况。为了做到这一点，我们将使用一个 `rescue` 区块并检查第一个成功的 API 调用，这确认了 token 还是有效的。然后我们会检查 `X-OAuth-Scopes` 应答头来确认用户还没有撤消 `user:email` 区域。

创建一个名为 *advanced_server.rb* 的文件，并且将下面的代码粘贴到其中：

```
require 'sinatra'
require 'rest_client'
require 'json'

# !!! DO NOT EVER USE HARD-CODED VALUES IN A REAL APP !!!
# Instead, set and test environment variables, like below
# if ENV['GITHUB_CLIENT_ID'] && ENV['GITHUB_CLIENT_SECRET']
#  CLIENT_ID        = ENV['GITHUB_CLIENT_ID']
#  CLIENT_SECRET    = ENV['GITHUB_CLIENT_SECRET']
# end

CLIENT_ID = ENV['GH_BASIC_CLIENT_ID']
CLIENT_SECRET = ENV['GH_BASIC_SECRET_ID']

use Rack::Session::Pool, :cookie_only => false

def authenticated?
  session[:access_token]
```

```
end

def authenticate!
  erb :index, :locals => {:client_id => CLIENT_ID}
end

get '/' do
  if !authenticated?
    authenticate!
  else
    access_token = session[:access_token]
    scopes = []

    begin
      auth_result = RestClient.get('https://api.github.com/user',
                                   {:params => {:access_token => access_token},
                                    :accept => :json})
    rescue => e
      # request didn't succeed because the token was revoked so we
      # invalidate the token stored in the session and render the
      # index page so that the user can start the OAuth flow again

      session[:access_token] = nil
      return authenticate!
    end

    # the request succeeded, so we check the list of current scopes
    if auth_result.headers.include? :x_oauth_scopes
      scopes = auth_result.headers[:x_oauth_scopes].split(', ')
    end

    auth_result = JSON.parse(auth_result)

    if scopes.include? 'user:email'
      auth_result['private_emails'] =
        JSON.parse(RestClient.get('https://api.github.com/user/emails',
                    {:params => {:access_token => access_token},
                     :accept => :json}))
    end

    erb :advanced, :locals => auth_result
  end
end

get '/callback' do
```

```
  session_code = request.env['rack.request.query_hash']['code']

  result = RestClient.post('https://github.com/login/oauth/access_token',
                           {:client_id => CLIENT_ID,
                            :client_secret => CLIENT_SECRET,
                            :code => session_code},
                            :accept => :json)

  session[:access_token] = JSON.parse(result)['access_token']

  redirect '/'
end
```

大部分的代码看起来都很熟悉。比如说，我们仍然使用 `RestClient.get` 来调用 GitHub API ，并且我们仍然将我们的结果传给一个 ERB 模板来渲染。（ 这回，文件名是 `advanced.erb` ）

而且，我们现在使用 `authenticated?` 方法来检查用户是否已经认证过了。如果没有， `authenticate!` 方法将被调用，这个方法将执行 OAuth 流程并且使用被授 予的标识和区域来更新回话。

接下来，在 *views* 中创建一个文件 *advanced.erb*，并将以下内容粘贴进去：

```
<html>
  <head>
  </head>
  <body>
    <p>Well, well, well, <%= login %>!</p>
    <p>
      <% if !email.empty? %> It looks like your public email address is <%= email %>.
      <% else %> It looks like you don't have a public email. That's cool.
      <% end %>
    </p>
    <p>
      <% if defined? private_emails %>
      With your permission, we were also able to dig up your private email addresses:
      <%= private_emails.map{ |private_email_address| private_email_address["email"] }.join(', ') %>
      <% else %>
      Also, you're a bit secretive about your private email addresses.
      <% end %>
    </p>
  </body>
</html>
```

从命令行调用 `ruby advanced_server.rb` ，将在 4567 端口启动你的服务端，和 我们使用简单的 Sinatra app 时同样的端口。当你浏览 `http://localhost:4567` 时，app 调用 `authenticate!` 将你重定向到 `/callback` 。然后 `/callback` 将我们又送回了 `/` ，由于现在我们已经认证了，页面将渲染 *advanced.erb* 。

我们能够通过在 GitHub 将我们的回调 URL 指定为 `/` 来完全简化这个往返的过程。但是，因为 `server.rb` 和 `advanced.rb` 都依赖于同一个回调 URL，我们必须多绕点弯来让它正确工作。

而且，如果我们从来没有授权这个应用去获取我们的 GitHub 数据，我们将从更早的弹出窗口看到相同的确认对话框和警告。

如果你有兴趣，你可以查看 yet another Sinatra-GitHub auth example (https://github.com/atmos/sinatra-auth-github-test) 作为另一个工程进行实验。

探索用户资源

当发出经过认证的请求给 GitHub API 时，应用程序通常会需要获取当前用户的存储库和组织。这个指南会向您解释如何可靠地发现这类资源。

要和 GitHub API 互动，我们将使用 Octokit.rb (https://github.com/octokit/octokit.rb) 。您可以在platform-samples (https://github.com/github/platform-samples/tree/master/api/ruby/discovering-resources-for-a-user) 存储库中找到这个示例项目的完整源代码。

准备开始

您应该先阅读认证基础指南(如果您还没有的话)，再来尝试本指南所提供的示例。下面的示例默认您已经注册了一个 OAuth 应用程序并且该程序有一个为用户提供的 OAuth 令牌。

探索应用程序可让用户访问的存储库

除了拥有自己的个人存储库之外，一个用户还可能是其他用户或组织的存储库的一个合作者。总而言之，刚才提到的都是用户拥有访问特权的存储库，要不就是用户拥有读写权限的私人存储库，要不就是用户拥有写权限的公共存储库。

OAuth 域 (https://developer.github.com/v3/oauth/#scopes) 和组织应用策略 (https://developer.github.com/changes/2015-01-19-an-integrators-guide-to-organization-application-policies/) 会决定哪些存储库可以通过您的应用让用户访问。使用下面的工作流来发现这些存储库。

和往常一样，我们会 require GitHub 的 Octokit.rb (https://github.com/octokit/octokit.rb) Ruby 库。然后将其配置为自动为我们处理分页 (https://github.com/v3/#pagination) 。

```
    require 'octokit'

    Octokit.auto_paginate = true
```

接下来，我们选择加入列出存储库的 API 的未来改进存储库。并设置媒体类别让我们得以访问那个功能。

```
language-ruby
Octokit.default_media_type = "application/vnd.github.moondragon+json"
```

现在，将我们应用程序的指定用户的 OAuth 令牌传过去：

```
# 在真正的应用内永远不要用硬编码把值写死！
# 而是设置环境变量并测试，和下例所示
client = Octokit::Client.new :access_token => ENV["OAUTH_ACCESS_TOKEN"]
```

接下来，我们就可以获取应用程序可以为用户获取的存储库了：

```
client.repositories.each do |repository|
 full_name = repository[:full_name]
 has_push_access = repository[:permissions][:push]

 access_type = if has_push_access
               "write"
              else
               "read-only"
              end

 puts "User has #{access_type} access to #{full_name}."
end
```

探索应用程序可让用户访问的组织

应用程序可以为用户开展一系列组织相关的任务。要开展这些任务，程序需要一个OAuth 认证和足够的权限。举例来说， `read:org` 域允许您列出队伍，而 `user` 域则可以让你公布用户的组织会员身份。 当一个用户授予一个或多个这样的域给你的应用程序时，你就可以获取该用户的组织信息了。

就和我们发现存储库一样，我们还是从 require GitHub’s Octokit.rb (https://github.com/octokit/octokit.rb) Ruby库开始，并将其设置为为我们自动操作分页：

```
require 'octokit'

Octokit.auto_paginate = true
```

接下来，我们会加入列出组织信息的 API 的未来改进存储库。并设置媒体类别让我们得以访问那个功能。

```
Octokit.default_media_type = "application/vnd.github.moondragon+json"
```

现在，将我们应用程序的指定用户的 OAuth 令牌传过去，来初始化我们的 API 客户端：

```
# 在真正的应用内永远不要用硬编码把值写死！
# 而是设置环境变量并测试，和下例所示
client = Octokit::Client.new :access_token => ENV["OAUTH_ACCESS_TOKEN"]
```

然后，我们可以列出应用程序可以为用户访问的组织了：

```
client.organizations.each do |organization|
puts "User belongs to the #{organization[:login]} organization."
end
```

不要依赖公共组织 API

如果你完整阅读了文档，也许会已经发现一个列出用户公共组织会员身份的 API 方法。大多数应用程序应该避开这个 API 方法。因为这个方法只返回用户的公共组织会员身份，不包括私人组织会员身份。

作为一个应用程序，你通常会希望用户加入的所有组织（包括公共的和私人的）都能授权给你的应用程序访问。而本文所描述的工作流正是让你实现初衷的。

4

管理部署密钥

在你自动化部署脚本时，一共有四种方式来管理你的服务器的 SSH 密钥：

- SSH agent 转发
- HTTPS + OAuth 令牌
- 部署密钥
- 机器用户

这个指南会帮你确定哪一种策略是最适合你。

SSH agent 转发

在许多情况下，特别是在一个项目的初始阶段，SSH agent 转发是最简单、最快捷的办法，该功能使用的密钥和你本地开发电脑所使用的密钥相同。

优点

- 你不需要生成或者监视记录任何新的密钥。
- 没有密钥管理；用户在服务器上拥有和本地一样的权限。
- 没有任何密钥存放在服务器上，所以即使服务器被攻陷，你也不需要寻找并删除已经沦陷的密钥。

缺点

- 用户**必须**通过 SSH 连接来部署；这意味着不能使用自动化部署技术。
- SSH agent 转发对于 Windows 用户来说运行起来稍显繁琐。

设置

1. 要在本地打开 agent 转发，请查看我们的 SSH agent 转发指南来获取更多信息。
2. 将你的部署脚本设定为使用 agent 转发。举例来说，在 bash 脚本里，启用 agent 转发应该和这个代码类似：`ssh -A serverA 'bash -s' < deploy.sh`

通过 OAuth 令牌进行 HTTPS clone

如果你不想使用 SSH 密钥，你可以使用带 OAuth 令牌的 HTTPS (https://help.github.com/articles/git-automation-with-oauth-tokens) 。

优点

- 任何具有访问服务器权限的人都能部署存储库。
- 用户不必变更他们的本地 SSH 设置。

- 多个用户不需要准备多个令牌；一个服务器只需要一个令牌。
- 令牌可以随时被废除，几乎可以当作一个一次性密码。
- 可以很简单地用脚本通过OAuth API (https://developer.github.com/v3/oauth_authorizations/#create-a-new-authorization) 生成新令牌。

缺点

- 您必须确认你的令牌的访问域已经被正确配置。
- 令牌的本质就是密码，所以需要和密码一样被保护起来。

设置

请参照通过令牌的 Git 自动化指南 (https://help.github.com/articles/git-automation-with-oauth-tokens)。

部署密钥

部署密钥是一个存放在你的服务器上并且可以授权访问一个 GitHub 存储库的 SSH 密钥。这个密钥是直接附在存储库上的，而不是个人用户账户。

优点

- 任何具有访问存储库权限的人都可以部署工程。
- 用户不需要改变他们本地的 SSH 设定。

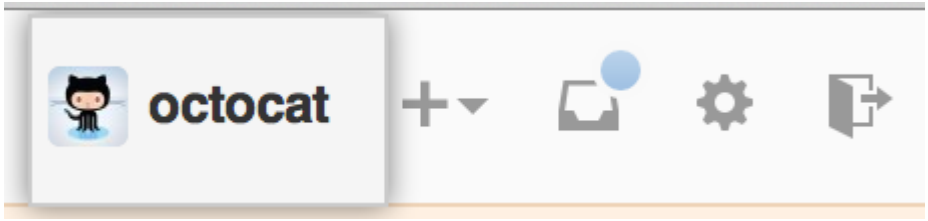
缺点

- 一个部署密钥只能授权一个存储库。而更复杂的工程可能会在同一个服务器上对许多存储库发出 pull 操作。
- 部署密钥总是提供对一个存储库的完整读写访问权限。
- 部署密钥通常没有经过密码保护，如果服务器被攻陷这些密钥将会很容易被获取。

设置

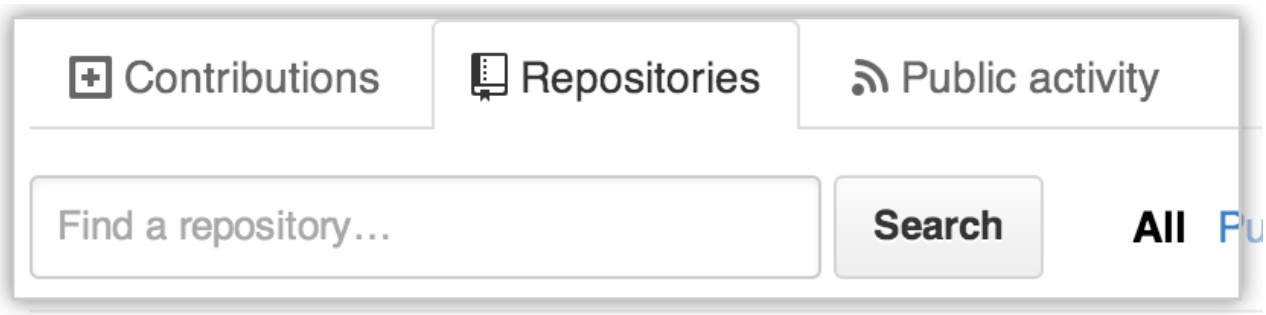
1. 在你的服务器上执行 `ssh-keygen` 程序 (https://help.github.com/articles/generating-ssh-keys)。

2. 在任意 GitHub 页面的右上角，点击你的用户相片。



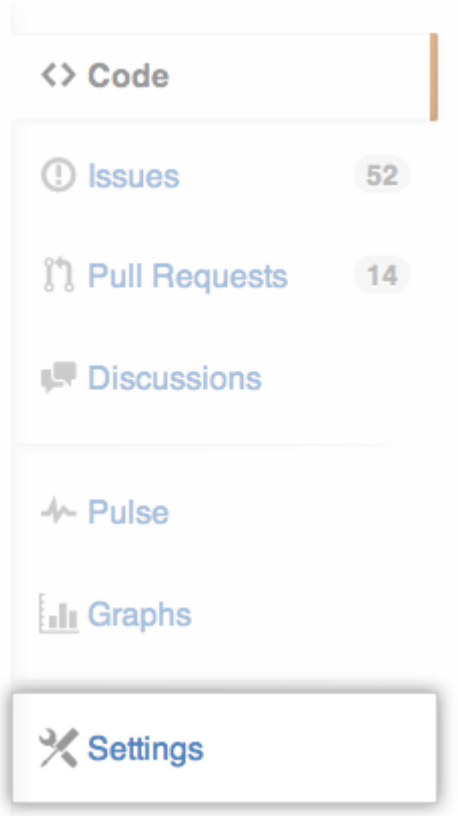
图片 4.1 Sample of an avatar

3. 在你的用户页面内，点击 Repositories（存储库）标签页, 然后点击你的存储库的名字。



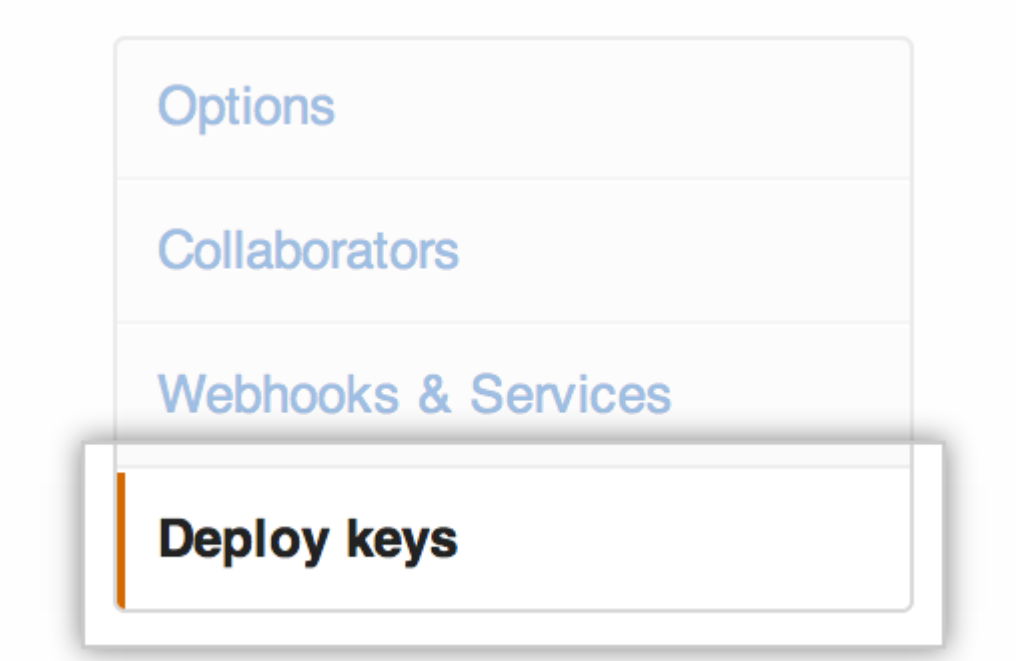
图片 4.2 Repository tab

4. 在你的存储库的右边栏中，点击 Settings（设定）。



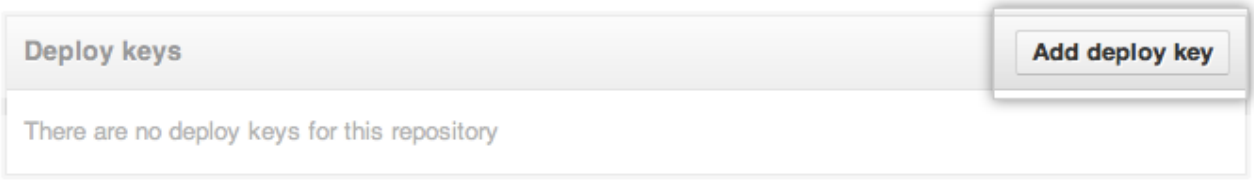
图片 4.3 Settings tab

5. 在侧边栏中，点击 Deploy Keys（部署密钥）。



图片 4.4 Deploy Keys section

6. 点击 Add deploy key（添加部署密钥）。将你的公钥粘贴进去，并且提交。



图片 4.5 Add Deploy Key button

机器用户

如果你的服务器需要访问多个存储库，你可以选择创建一个新的 GitHub 账户然后附上一个仅用于自动化操作的 SSH 密钥。因为这个 GitHub 账户不会被任何人类使用，所以被称作机器用户。接下来，你可以添加机器用户为合作者 (https://help.github.com/articles/how-do-i-add-a-collaborator)，或者将机器用户添加到队伍里 (https://help.github.com/articles/adding-organization-members-to-a-team) 并给与自动化所需的存储库访问权限。

注意：添加一个机器用户作为合作者将会赋予其完整读写访问权限。而添加一个机器用户进队伍则会赋予它队伍的权限。

提示：我们的服务条款 (https://help.github.com/articles/github-terms-of-service) 特地提到“ *‘bot’（机器人）或者其他自动化手段所创建的用户是不被允许的*”，还有“*一个自然人或者法人不得持有多于一个免费账户*”。不过不要害怕，我们不会派出极端的律师来和你纠缠到底，只要你所创建的机器用户只是用于服务器部署脚本。本文提到的机器用户是完全合法的。

优点

- 任何有权限访问存储库和服务器的人都能部署工程。
- 没有任何（人类）用户需要更改他们的本地 SSH 设定。
- 不需要多个密钥；一个服务器一个密钥已然足够。

缺点

- 只有组织有权限创建队伍；所以只有组织可以用其来限制机器用户的权限为只读。个人存储库只能对机器用户授予合作者权限，也就是读写。
- 机器用户密钥，和部署密钥一样，通常没有密码保护。

设置

1. 在你的服务器上执行 `ssh-keygen` 程序 (https://help.github.com/articles/generating-ssh-keys)，然后为机器用户账户附上公钥。
2. 给那个账户所需要访问的存储库的访问权限。你可以通过添加账户为合作者 (https://help.github.com/articles/how-do-i-add-a-collaborator) 或者在组织内添加账户到队伍 (https://help.github.com/articles/adding-organization-members-to-a-team) 来做到这点。

5

SSH agent 转发

SSH Agent 转发功能可以让您在部署至服务器的过程更简便。比起让您的密钥（无密码保护的）存放在服务器上，SSH 代理转发功能可以让您使用本地的 SSH 密钥。

如果您已经设置好 SSH 密钥来和 Github 交换数据，可能已经对 ssh-agent 感到熟悉。 ssh-agent 是一个在后台运行的应用程序，它会缓存您已经加载到内存中的密钥，这样便不必每次使用这个密钥都输入密码了。更棒的是，您可以选择让服务器访问您的本地 ssh-agent ，效果如同 ssh-agent 已经在服务器上运行。这有点像当您借用朋友的电脑时，让朋友帮忙输入密码以便让您使用。

查看 Steve Friedl 的技术提示指南 (http://www.unixwiz.net/techtips/ssh-agent-forwarding.html) 来获得关于 SSH Agent 转发功能更详细的解释。

设置 SSH agent 转发功能

请确保您自己的 SSH 密钥已经设置并且可用。如果还没有完成这步，可以查看我们的生成 SSH 密钥指南 (https://help.github.com/articles/generating-ssh-keys) 。

您可以在终端中使用命令 `ssh -T git@github.com` 来测试您的本地密钥是否可用：

```
$ ssh -T git@github.com
# 尝试通过SSH连接Github
Hi username! You've successfully authenticated, but GitHub does not provide
shell access.
```

我们现在有了一个良好的开端。让我们设置 SSH 来允许 agent 转发到您的服务器。

1. 使用您最喜欢的文本编辑器, 打开文件 `~/.ssh/config` 。 如果这个文件不存在，可以在终端中使用命令 `touch ~/.ssh/config` 来创建一个。
2. 将以下文本输入到文件中，用您的服务器的域名或者 IP 地址来替换 `example.com` ：

```
Host example.com
ForwardAgent yes
```

> 警告: 您也许会想投机取巧地使用类似Host *这样的带通配符语句来设置所有的 SSH 连接。这并不是一个好办法，因为这样做会分享

测试 SSH agent 转发功能

您可以用 SSH 连接到您的服务器并再次执行命令 ssh -T git@github.com 来测试 agent 转发是否有效。如果情况顺利，您会得到和

如果您不能确定目前是否正在使用本地密钥，可以通过检视服务器上的 SSH_AUTH_SOCK 变量：

$ echo "$SSH_AUTH_SOCK" # 打印输出 SSH_AUTH_SOCK 变量 /tmp/ssh-4hNGMk8AZX/agent.79453

如果这个变量尚未设定，说明 agent 转发功能没有运作：

$ echo "$SSH_AUTH_SOCK" # 打印输出 SSH_AUTH_SOCK 变量 [No output] $ ssh -T git@github.com # 尝试通过 SSH 连接 Github Permission denied (publickey).

SSH agent 转发功能的疑难解答

如果您在 SSH agent 转发功能中遇到问题，以下几点可以助您排除故障。

您必须使用 SSH URL 来检查代码

SSH 转发功能只能在 SSH URL 下使用，不包括 HTTP(s) URL。检查服务器上的 *.git/config* 文件并且确保 URL 是 SSH 格式的，

[remote "origin"] url = git@github.com:yourAccount/yourProject.git fetch = +refs/heads/*:refs/remotes/origin/*

您的 SSH 密钥必须本地可用

在您的密钥能通过 agent 转发功能使用之前，这些密钥必须在本地上也能正常使用。我们的生成SSH密钥指南 (https://help.github.co

您的系统必须允许 SSH agent 转发功能

某些时候，系统配置会不允许 SSH agent 转发功能。您可以通过终端输入以下命令来检查目前是否有系统配置文件正在生效：

$ ssh -v example.com # 连接到 example.com，附带详细除错输出 OpenSSH_5.6p1, OpenSSL 0.9.8r 8 Feb 2011 debug1: Reading configuration data /Users/you/.ssh/config debug1: Applying options for example.com debug1: Reading configuration data /etc/ssh_config debug1: Applying options for * $ exit # 返回到您的本地命令提示符

在上面的示例中，*~/.ssh/config* 文件最先被载入，然后是 */etc/ssh_config* 文件。我们可以通过下面的命令来确认后来被载入的文件是

$ cat /etc/ssh_config # 打印输出 /etc/ssh_config 文件 Host * SendEnv LANG LC_* ForwardAgent no

在这个示例中，我们的 */etc/ssh_config* 文件显然写着 ForwardAgent no，这是阻止 agent 转发功能工作的方法之一。将这行从文件

您的服务器必须允许 SSH agent 转发功能可以用于入站连接上

agent 转发功能也可能是被您的服务器阻止了。您可以通过 SSH 连接服务器并且执行 sshd_config 来检查 agent 转发功能是否被许可

您的本地 ssh-agent 必须处于运行中状态

在大多数电脑上，操作系统会自动为您启动 ssh-agent。然而在 Windows 上，您需要手动设置。我们有一个指南教您如何随打开 Git

如果想确认 ssh-agent 正在您的电脑上运行，请在终端中输入以下命令：

$ echo "$SSH_AUTH_SOCK" # 打印输出 SSH_AUTH_SOCK 变量 /tmp/launch-kNSlgU/Listeners

您的密钥必须对于 ssh-agent 可见

您可以通过以下命令检查您的密钥是否对 ssh-agent 可见：

ssh-add -L

如果该命令指出没有身份可用，那么您需要通过以下命令添加您的密钥：

ssh-add 您的密钥 ```

> 在 Mac OS X 上，当系统重新启动后，ssh-agent 再次启动时会 “忘记” 这个密钥。不过您可以通过以下命令将您的 SSH 密钥导入到密钥链中：
>
> /usr/bin/ssh-add -K 您的密钥

6

数据渲染成图表

在这个指南中，我们将要用 API 来获取我们拥有的储存库(repository)的信息以及编写它们所用的编程语言。然后，用 D3.js (http://d3js.org/) 库把那些信息用几种不同的方式进行可视化。在此我们用一个极好的 Ruby 库 Octokit (https://github.com/octokit/octokit.rb) 来和 GitHub API 交流。

如果您还没准备好，应该先去阅读“认证基础” (https://developer.github.com/guides/basics-of-authentication/) 指南再尝试这个示例。 您能在 platform-samples (https://github.com/github/platform-samples/tree/master/api/ruby/rendering-data-as-graphs) 存储库中找到这个示例项目的完整源代码。

让我们开始吧!

设置一个 OAuth 应用程序

首先, 在 GitHub 上注册一个新应用程序 (https://github.com/settings/applications/new) 。 设置主 URL 和回调 URL 为 `http://localhost:4567/` 。和之前 (https://developer.github.com/guides/basics-of-authentication/) 一样，我们通过 sinatra-auth-github (https://github.com/atmos/sinatra_auth_github) 来应用 Rack 中间件来处理 API 的认证:

```
require 'sinatra/auth/github'

module Example
  class MyGraphApp < Sinatra::Base
    # !!! 在真正的应用内永远不要用硬编码把值写死 !!!
    # 而是设置环境变量并测试，和下例所示
    # if ENV['GITHUB_CLIENT_ID'] && ENV['GITHUB_CLIENT_SECRET']
    #  CLIENT_ID       = ENV['GITHUB_CLIENT_ID']
    #  CLIENT_SECRET    = ENV['GITHUB_CLIENT_SECRET']
    # end

    CLIENT_ID = ENV['GH_GRAPH_CLIENT_ID']
    CLIENT_SECRET = ENV['GH_GRAPH_SECRET_ID']

    enable :sessions

    set :github_options, {
      :scopes    => "repo",
      :secret    => CLIENT_SECRET,
      :client_id => CLIENT_ID,
      :callback_url => "/"
    }

    register Sinatra::Auth::Github
```

```
    get '/' do
      if !authenticated?
        authenticate!
      else
        access_token = github_user["token"]
      end
    end
  end
end
```

和之前的例子一样，设置一个类似的 *config.ru* 文件：

```
language-ruby ENV['RACK_ENV'] ||= 'development' require "rubygems" require "bundler/setup"

require File.expand_path(File.join(File.dirname(__FILE__), 'server'))

run Example::MyGraphApp
```

获取存储库信息

这次，为了能和 GitHub API 交流，我们将用 Octokit Ruby 库 (https://github.com/octokit/octokit.rb) 。这要比直接进行一大堆 REST 调用简单许多，更别说 Octokit 就是由 GitHub 用

通过 API 和 Octokit 库来进行认证是很简单的，只需要将您的登陆信息和令牌(token)作为参数穿给 Octokit::Client 构造器即可：

```
if !authenticated? authenticate! else octokit_client = Octokit::Client.new(:login => github_user.login, :o
auth_token => github_user.token) end
```

我们来对我们的存储库信息做点有趣的事情吧，例如查看他们使用的各种编程语言，并且数出哪一种是最常用的。要达成这个目的，首

```
repos = client.repositories
```

接下来，我们会迭代每一个存储库，然后对 GitHub 所关联的编程语言进行计数：

language_obj = {} repos.each do |repo| # 某些情况下，language 可能为0 if repo.language if !languag e_obj[repo.language] language_obj[repo.language] = 1 else language_obj[repo.language] += 1 end e nd end

```
languages.to_s
```

当您重启您的服务器时，网页应该会显示类似下面这样的内容：

{"JavaScript"=>13, "PHP"=>1, "Perl"=>1, "CoffeeScript"=>2, "Python"=>1, "Java"=>3, "Ruby"=>3, "G o"=>1, "C++"=>1}

到目前为止都十分顺利，不过这样的表现方法对人类不是十分友好。一个可视化图可以很好地帮助我们了解这些语言计数的分布情况。

可视化编程语言计数

D3.js，或者直接叫 D3，是一个功能全面的库，专门用于创建许多不同种类的图表和互动可视化图。D3 的详细使用方法已经超出本文

D3 是一个 JavaScript 库，并且喜欢把数据以数组形式处理。所以，我们先把 Ruby Hash 转换成一个 JSON 数组，以便在浏览器内

languages = [] language_obj.each do |lang, count| languages.push :language => lang, :count => coun t end

```
erb :lang_freq, :locals => { :languages => languages.to_json}
```

我们简单地迭代对象中的每个*键–值对*并将他们写入一个新的数组。之所以在早些时候没做这步，是为了避免在建立 language_obj 对

这时，lang_freq.erb 会需要一些 JavaScript 语句来渲染条形图，在本例中您可以直接使用下方提供的代码，如果您想了解 D3 如何工

Check this sweet data out:

```
</body>
<script>
  var data = <%= languages %>;

  var barWidth = 40;
  var width = (barWidth + 10) * data.length;
```

```
  var height = 300;

  var x = d3.scale.linear().domain([0, data.length]).range([0, width]);
  var y = d3.scale.linear().domain([0, d3.max(data, function(datum) { return datum.count; })]).
    rangeRound([0, height]);

  // 为 DOM 添加 canvas
  var languageBars = d3.select("#lang_freq").
    append("svg:svg").
    attr("width", width).
    attr("height", height);

  languageBars.selectAll("rect").
    data(data).
    enter().
    append("svg:rect").
    attr("x", function(datum, index) { return x(index); }).
    attr("y", function(datum) { return height - y(datum.count); }).
    attr("height", function(datum) { return y(datum.count); }).
    attr("width", barWidth);

  languageBars.selectAll("text").
    data(data).
    enter().
    append("svg:text").
    attr("x", function(datum, index) { return x(index) + barWidth; }).
    attr("y", function(datum) { return height - y(datum.count); }).
    attr("dx", -barWidth/2).
    attr("dy", "1.2em").
    attr("text-anchor", "middle").
    text(function(datum) { return datum.count;});

  languageBars.selectAll("text.yAxis").
    data(data).
    enter().append("svg:text").
    attr("x", function(datum, index) { return x(index) + barWidth; }).
    attr("y", height).
    attr("dx", -barWidth/2).
    attr("text-anchor", "middle").
    text(function(datum) { return datum.language;}).
    attr("transform", "translate(0, 18)").
    attr("class", "yAxis");
 </script>
</html>
```

呼！再次地，您不用太担心这堆代码在干什么。重点是位于相对上方的 var data = <%= languages %>; 语句，这语句将我们先

正如《 凡人用 D3 》指南所说的，这或许不是 D3 的最佳用例，但至少展示了如何结合 Octokit 来使用这个库, 如何来做一些真正炫目的

结合不同的 API 调用

坦白的时候到了：存储库内的 language 属性只能识别“主要”的编程语言，这意味着如果您有一个存储库使用了几种不同的编程语言

我们来结合几种 API 调用来获得一个能显示哪种编程语言拥有最多字节的代码的*真实表示*。treemap (http://bl.ocks.org/mbostock/4

[{ "name": "language1", "size": 100}, { "name": "language2", "size": 23} ...]

因为我们之前已经有了一个存储库列表，所以直接检视每一个存储库，并调用列出编程语言的 API 方法 (https://developer.github.co

repos.each do |repo| repo_name = repo.name repo_langs = octokit_client.languages("#{github_use r.login}/#{repo_name}") end

接着, 在“主表”中累加每个找到的编程语言：

repo_langs.each do |lang, count| if !language_obj[lang] language_obj[lang] = count else language_ob j[lang] += count end end

然后, 我们将内容的格式转换成 D3 能理解的结构：

language_obj.each do |lang, count| language_byte_count.push :name => "#{lang} (#{count})", :count => count end

```
# 一些必须的格式化操作
language_bytes = [ :name => "language_bytes", :elements => language_byte_count]
```

(若想获取更多关于 D3 tree map 原理的信息，参见这个简单的教程 (https://developer.github.com/v3/repos/#list-languages))

最后, 将这个 JSON 信息传给一样的 ERB 模板：

erb :lang_freq, :locals => { :languages => languages.to_json, :language_byte_count => language_bytes.to_json}

和之前一样, 这是一段 JavaScript 代码，您可以将其直接复制到您的模板中：