《开源软件开发技术》2020年作业（2）

班级 大数据1901 学号 2019011299 姓名 张睿

1. 开源开发有哪些人员角色？

|  |  |
| --- | --- |
| **角色** | **软件架构师Software Architect** |
| **定义** | 主导系统全局分析设计和实施、负责软件构架和关键技术决策的角色 |
| **职责** | l          领导与协调整个项目中的技术活动（分析、设计和实施等）  l          推动主要的技术决策，并最终表达为软件构架  l          确定和文档化系统的相对构架而言意义重大的方面，包括系统的需求、设计、实施和部署等“视图”  l          确定设计元素的分组以及这些主要分组之间的接口  l          为技术决策提供规则，平衡各类涉众的不同关注点，化解技术风险，并保证相关决定被有效的传达和贯彻  l          理解、评价并接收系统需求  l          评价和确认软件架构的实现 |
| **专业技能** | l          技术全面、成熟练达、洞察力强、经验丰富，具备在缺乏完整信息、众多问题交织一团、模糊和矛盾的情况下，迅速抓住问题要害，并做出合理的关键决定的能力  l          具备战略性和前瞻性思维能力，善于把握全局，能够在更高抽象级别上进行思考；  l          对项目开发涉及的所有问题领域都有经验，包括彻底地理解项目需求，开展分析设计之类软件工程活动等  l          具备领导素质，以在各小组之间推进技术工作，并在项目压力下做出牢靠的关键决策  l          拥有优秀的沟通能力，用以进行说服、鼓励和指导等活动，并赢得项目成员的信任；  l          以目标导向和主动的方式来不带任何感情色彩地关注项目结果，构架师应当是项目背后的技术推动力，而非构想者或梦想家（追求完美）  l          精通构架设计的理论、实践和工具，并掌握多种参考构架、主要的可重用构架机制和模式（例如J2EE架构等）；  l          具备系统设计员的所有技能，但涉及面更广、抽象级别更高； |
| **活动** | 确定用例或需求的优先级、进行构架分析、创建构架的概念验证原型、评估构架的概念验证原型的可行性、组织系统实施模型、描述系统分布结构、描述运行时刻构架、确定设计机制、确定设计元素、合并已有设计元素 |
| **工件** | 软件构架文档、参考构架、分析模型、设计模型、实施模型、部署模型、构架概念验证原型、接口、事件、信号与协议 |

|  |  |
| --- | --- |
| **角色** | **设计员Designer** |
| **定义** | 在项目需求、构架和开发流程限制之下负责系统局部的分析设计的角色 |
| **职责** | l          理解、评价并接收系统需求细节  l          理解、评价并接收相关的软件架构  l          依据需求规格分析系统的内部行为，在分析层面识别与定义各系统组成元素的职责、操作  l          识别与定义各设计元素的职责、操作、属性及其相互关系  l          保证其设计符合软件构架，并足够详细到可以进入实施 |
| **专业技能** | l          掌握需求工程概念和技巧，以准确无误地理解相关系统需求；  l          熟悉软件架构模式、概念和技巧，以准确无误地理解相关系统构架；  l          精通软件设计理论、实践和工具，包括面向对象的分析设计技术和统一建模语言等  l          掌握将用于实现系统的相关技术，例如组件开发（J2EE、EJB）、通讯机制、多线程与实时技术等；  l          掌握将用于实现系统的程序设计语言（例如：Java、C++、C、HTML、CSS、XML、 JavaScript、汇编语言）；  l          对目标设计元素的相关问题有深入的了解  l          熟悉项目的设计指南，明了设计与实施的关系，包括在实施之前设计应当达到的详细程度  l          具备实施员的所有技能，但程度更深、抽象级别更高；  l          通常兼任实施员的角色 |
| **活动** | 执行用例分析、设计用例实现、子系统设计、设计类、设计测试包与类库 |
| **工件** | 用例实现、分析类、设计子系统、设计包、设计类、测试类 |

|  |  |
| --- | --- |
| **角色** | **实施员Implementer** |
| **定义** | 遵照项目标准来开发、测试构件以集成为更大的子系统的角色 |
| **职责** | l          理解、评价并接收相关的软件设计元素  l          在私有开发空间（workspace），遵照项目既定的标准（编程规范等）来开发、测试构件  l          为了测试构件，需要相应的驱动和桩，实施员还负责开发、测试这些测试用构件及相关子系统  l          将测试通过的实施元素（源码等）提交到集成空间，以交付给集成员来集成为更大的子系统  l          重设开发基线（rebase），以便在新的基础上继续实施活动 |
| **专业技能** | l          熟悉软件设计模式、概念和技巧，以准确无误地理解相关设计；  l          掌握软件编程理论、实践和工具，包括面向对象的编程技术和统一建模语言等  l          精通将用于实现系统的程序设计语言和编程技能（例如：Java、C++、C、HTML、CSS、XML、 JavaScript、汇编语言）；  l          精通开发测试（特别是单元测试）和自动化测试工具，熟悉测试驱动编程更佳  l          熟悉被实施构件的所有相关内容，包括其接口、结构与行为  l          熟悉项目的编程指南 |
| **活动** | 实现开发测试（脚本等）、执行开发测试、实现设计元素、实现测试设计元素、分析运行时刻行为、编写构建脚本、开发安装工件 |
| **工件** | 开发测试（脚本等）、实施元素（源码与描述文档等）、构建脚本（makefile）、安装包 |

|  |  |
| --- | --- |
| **角色** | **集成员Integrator** |
| **定义** | 负责计划并执行集成活动从而将实施元素集成为产品构造的角色 |
| **职责** | l          计划在不同的子系统、系统级别下，进行构建和集成的顺序  l          在集成空间执行集成，解决编译与链接冲突，调试构建脚本等  l          建立工作基线  l          将集成测试通过的子系统、系统实施元素提交到更  l          上一级集成空间，以便集成为更大的系统 |
| **专业技能** | l          熟悉将用于实现系统的程序设计语言（例如：Java、C++、C、HTML、CSS、XML、 JavaScript、汇编语言）；  l          精通自动化构建工具（例如make/ant）  l          熟悉被集成子系统、系统的相关内容，包括其各组成部分间的依赖关系、与外部系统间的依赖关系，以及它们的开发和依赖关系如何随时间而演变的预期  l          具备优良的协调能力，以便与多个开发人员协作，来共同保证集成的成功  l          很多时候可以兼任测试员的角色 |
| **活动** | 创建集成空间、计划子系统集成构建、计划系统集成构建、执行子系统集成、执行系统集成、建立基线、提升基线级别 |
| **工件** | 集成构建计划、构建脚本（makefile）、构造（build） |

**开发人员培训方案：**

|  |  |  |
| --- | --- | --- |
| **课　程　名　称** | **课程编号** | **备注** |
| **理论基础** |  |  |
| 面向对象的分析与设计(OOAD) | SZB-CLS-201 |  |
| 统一建模语言(UML)和建模工具 | SZB-CLS-801 |  |
| 统一软件过程RUP | SZB-CLS-803 |  |
| 软件构架设计与构架模式 |  |  |
| 软件设计模式 |  |  |
| 单元测试与测试驱动编程 |  |  |
| 数据模型与O/R Mapping |  | 数据库设计员 |
| 用户界面与可用性设计 |  | 用户界面设计员 |
| **实战演练** |  |  |
| UML/ROSE实践 | SZB-CLS-802 |  |
| UML/RUP项目实战：项目过程概述 |  |  |
| UML/RUP项目实战：构架与详细设计 |  |  |
| UML/RUP项目实战：分析与设计 |  |  |
| UML/RUP项目实战：实施与单元测试 |  |  |
| UML/RUP项目实战：软件模式 |  |  |
| UML/RUP项目实战：自动化构建与持续集成 |  |  |
| **工具操作** |  |  |
| Rational ClearCase基本操作 |  |  |
| Rational RequisitePro与软件需求管理 | SZB-CLS-605 |  |
| Rational ClearQuest与缺陷跟踪 | SZB-CLS-604 |  |
| 白盒测试工具 |  |  |
| **技术应用** |  |  |
| 优秀的开源产品与工具应用 |  |  |
| 代码调试与缺陷快速定位 |  |  |
| J2EE（.Net）架构及程序设计 | SZB-CLS-203 |  |
| BEA WebLogic、BEA Tuxedo | SZB-CLS-204 |  |
| IBM WebSphere、 CICS | SZB-CLS-205 |  |
| Windows NT/2000/XP、Linux/Unix操作系统内幕 |  |  |
| 高级Windows、Linux/Unix GUI应用程序开发 |  |  |
| C++高级开发技巧、 Java高级开发技巧、 C#高级开发技巧、 VB6和VB.NET高级开发技巧 |  |  |
| 组件开发、集成和测试： CORBA、 MS COM+、 Enterprise JavaBeans |  |  |
| 高级的网络应用程序开发、高级的数据库应用程序开发 |  |  |
| IDE：IBM WebSphere Application Developer 、 Borland JBuilder、 MS Visual Studio.Net 、 Sun Forte Developer 、 Borland Delphi |  |  |
| XML及应用程序开发 |  |  |
|  |  |  |

2. 创建和开展开源项目需要注意的问题有哪些？

如果你想发布一个开源库，请确保它有以下特点：

清晰的依赖性和安装说明

至少有一个简要的文档指南

修改日志和仓库中的标签

关于支持的语言、运行时、工具版本的信息和项目的成熟度

一个可以让用户提问和交流的邮件列表

缺少任何一项都会造成一些用户的愤怒和沮丧，当然同时也浪费了时间。

怎样让你的开源项目更棒

每年，越来越多的人发布了自己开发的库并且它们开源。这里我们分享一些我们经验，以便你的用户对你的库满意。

这里有一个经验法则：

不要让你的用户生气！

也可以理解为：

不要让你的用户有想要砸电脑的冲动

现在让我们通过一些努力来实现这个目标。

创建一个实用的README

即使你的项目有一个很棒的网站，潜在的用户第一次接触这个项目很可能就是通过阅读README文件。我们需要确保它很棒并且包含了有用的信息。

提供依赖信息

那么说你会发布你的开源项目。这说明你很聪明，真有你的！不幸的是，不是所有人都像你那样，而且有一些人对这门语言或者你在做的系统完全不了解。这意味着对你来说很显然的事情对他们来说就一点也不显然了。

其中一件就是缺少依赖说明或者安装说明

我到底怎么安装这个东西，难道不能说得清楚一些吗?

这很快就能让用户生气。在源代码里找你的包或者构件的名字是很烦人的，有些项目对构件使用一些特别有才的名字，完全不符合仓库的名字。

让你的用户从这些糟糕的事情中脱离出来吧。问题是怎样添加依赖性，理想状况下可以通过复制粘贴一小段代码。

如果需要例子的话可以点击 Welle。

清楚的说明项目的成熟度

你在生产中使用这个项目有几个月了吗？你是否觉得它还是不完整的？你是否希望API在下一个版本会彻底地修改？你的项目是否在要求最多并且很老的项目中也能稳定安全的使用？

要把这些说得清楚。下次你就不会因为做了一个错误的介绍，但是没有的提供任何项目成熟度的信息而项目浪费一周的时间了。你会意识到几句短短的话就能产生很大的影响。

运行时、语言、工具版本的文档支持

当考虑到向后兼容时，Clojure有一个很好的跟踪记录。它好的几乎让人难以置信。包括1.2到1.3的升级，之后的升级对绝大多数的项目来说就是一个简单替换。同样地，那些高于1.2的项目大多使用了最新的稳定版本。

然而，不会一直都是这样。在某些情况下，未来版本的Clojure会打破兼容性。我们怎么让我们的用户不愤怒？通过在README中清楚的说明哪些版本是支持的。

这只需要写一行文字。这样，在你发布的那一周就少了抱怨，同时也减少了初学者的很多麻烦。

如果你需要一个例子，有一个来自 Welle的例子。

说明你使用了什么许可证

你可能并不太关心许可证，但是那些在大公司中想用你的库的人很关心。他们必须知道！当他们想用Clojure/Node.js/Scala/Go等等的时候，可能不能使用。

因此清楚的说明你的许可证。也请你使用一些对商业友好的协议，除非你有自己的理由。（ Apache Public License 2.0和Eclipse Public License）是不错的选择。注意到一些许可证(比如MIT)的确很友好、流行，但是不提供任何专利保护，在当前的法律环境下也不应该忽视。

最后，记得你可以使用双许可证，如果你真的是许可证中立的话可以使用，比如APL2/GPLv2。那个你的用户就可以选择最适合他们的许可证了。

疑惑的时候，可以参考摘要：合法、开源许可证用白话概括（但是别把它当作合法的建议）

怎么把它搞糟

如果你想坑你的用户，可以试试：

在你项目的根目录放一个空的README

在末尾写上“欢迎加补丁”

发明你自己的许可证或者使用一个完全不熟悉的，比如WTFPL

那么你的项目就永远不会有用户了。我保证。

为你的项目写文档

写文档不容易同时也是需要花费一些时间的。然而，文档是你能为你的用户做的最好的事了。不仅能够节省他们大量的时间，也可以让他们确信你的库不是被遗弃的软件。

文档能够让你的用户完成他们起初使用你的库的任务。像Rob Pike说的，它“让这些任务成为可能”。这让你的用户知道你重视这一点，让他们知道你是个有血有肉的人，不是一个产生代码的机器。

在ClojureWerkz上工作将近两年后，我可以自信地说，我们的用户最感谢我们的就是我们写的项目文档：

Elastisch documentation

Welle documentation

Neocons documentation

Monger documentation

Langohr documentation

写出优秀的文档需要花些时间。幸运的是，现代工具可以帮到你并且大大减少你必须解决的一些烦人的事。

我们为ClojureWerkz项目开源了我们的基于Jekyll的文档模板。我们在CSS和设计中视觉效果方面不是很擅长，所以我们使用了Twitter的BootStrap库。我们的文档站点可以更好看，但是相比大多数开源项目来说已经很不错了。你可以使用我们的模板或者为你的项目开发类似的工具。

更好的是，如果你开源了你的文档站点(这似乎没有理由不那么做)，你会看到人们会比贡献代码的修改更早的贡献出小的改进。

如果你仍然不确定是否值得为你的项目写文档，看一下 Jacob Kaplan-Moss的这个报告：

怎么把它搞糟

如果你想坑你的用户，可以试试：

不要写一个文档说明，甚至连例子也不写

确保你的API说明已经有三个月没有更新了

声明那些不愿意读代码去理解即使是最基本的东西的用户是愚蠢的，并且应该去卖汉堡！

更容易升级

某些时候，你想要发行项目的另一个版本。这可能是让你的用户很开心，因为他们已经使用了你的库，或者很生气，浪费了他们时间。

不关心向后兼容

关于软件开发的一件很令人生气的事就是当你升级一个库但是数百个测试失败了。更让我生气的就是我还要重写我一半的基础代码，因为有人在没有任何警告的前提下决定打破公共的API。

因此，致力于维护向后兼容性。当然你没有必要像OpenJDK那样支持15年以前的项目。但是在移除之前建议不使用一些东西能够更容易发现哪些地方改动了。

你怎么做到这点呢？维护一个修改日志。

拥有一个修改日志

有时，你的用户会升级（关于这一点在下文会更多的介绍）。他们会问自己一个问题：

这次发布改动了什么地方呢？

然后

我的代码会不能用吗？我是不是一定要重写？

最后

Joe，那个运维的家伙会因为我升级讨厌我吗？

所有这些问题都能通过一个修改日志得到解答。它像推特一样只不过它真的很实用，它是这样用的：

每次你解决一个bug，在日志里加一个简单的记录

每次你加入一个新特性，在日志里简单地提一下，并且用几个代码例子解释它。

每次你做了重大的API改动，在日志中用粗体清楚的说明

就是这些了。没有第三步！

修改日志一般把最新的记录放在最前面。改动是按版本分类的。如果你有多个分支(比如master和1.0.x)，每一个都应该有一个独立的修改日志。

就是这些了。可以看看， Welle的修改日志

给版本加上标签

又是那个时候了，你已经升级版本并且马上就要发布构件了。停一停，先做一件事：给这次提交加上标签。没有标签的话，找两个版本之间的不同会很痛苦的。

一些依赖性(比如Bundler, Rebar)和配置管理工具可以使用标签，开发者系统这些标签是可用的。

使用统一的版本信息，比如v1.0.0-alpha1, v1.0.0, v1.1.2等。标签不一致绝对会导致运维的人整天讨厌你的项目。

宣布版本发行

在你发布一个版本字之后就是要写一个博客日志，或者在你们项目的邮件列表或更大的相关的邮件列表中发个更新(比如Clojure邮件列表或者RabbitMQ)

确保主题是以ANN或者[ANN]开头的，这意味着这是一个通告。比如

ANN Welle 1.5.0 发布了

在你的通告中，清楚的说明你的项目是做什么的，它是否向后兼容，并且有到修改日志的链接，可以让用户找到更多的细节。就是这样了。

开发时使用预览或者快照版本

你有没有曾经看到一个项目用同一个版本，比如0.2.1将近半年？你怎么知道哪一个版本才是0.2.1呢？这是一个还在开发中的版本吗？是不是有人升级后忘了修改版本号？到底怎么回事？

这会让所有的开发者疯掉的！千万别做那样的人！在项目中用预览或者快照版本，当你快要发布一个版本的时候才揭开那个版本。然后立即升级那个版本。

举几个开发版本的例子：

1.1.0.pre1

1.1.0-alpha1

1.1.0-SNAPSHOT

任何其他开发版本的命名格式是不清楚的，并且会你的用户很不愉快。

怎么把它搞糟

如果你想完全坑你的用户，试试下面：

* 随意打破公用的API，最好巧妙地，连你的测试也不会发现API的修改
* 忘了升级版本信息
* 从不给版本加标签
* 从不宣布版本发行

使用GitHub

我和gitHub没有友好关系，也不要假设Git是“最好”的版本控制系统。但是它真的不错。最近几乎所有人都在使用GitHub。

GitHub让下面几件事变得更简单：

发现你的项目

浏览和搜索代码

通过填问题或者@使你能够关注问题

为小的改动做出贡献

可能最重要的是，GitHub对不是技术大牛的很友好。是的，它的确是，同时他们正努力让它变得更好。

使用GitHub意味着你能尤其简单地使用CI的服务(Travis CI)。

如果你不让你的用户去处理补丁、为了提交问题在网上到处找你的email、通过糟糕的3G网络复制你300M的仓库只是为了编辑一个排版错误，你将得到更多的赞赏。

@old\_sound @g3rtm bitbucket毫无疑问是很好的服务。但对于使用公开代码的人开始显得有点难了。– Michael Klishin (@michaelklishin) 21 de enero de 2013

不要把事情弄得困难。

提供一个让用户可以得到帮助的地方

如果你的项目达到了一定程度的流行度，你必须回答关于它的一些问题。为了这一点，设置一个邮件列表(一个Google群)或者如果你经常上IRC的话，开启一个通道吧。

认为你没有足够的时间？使用邮件列表最好的部分就是如果你给了一个途径，用户会互相帮助。所以在你项目的README中清楚地说明可以获得帮助的途径。

在Twitter上经常搜你项目的名字，你就会发现各种各样的问题，批评和表扬的。如果你频繁地使用Twitter，为你的项目创建一个独立的帐号，就像我们的@ClojureWerkz。

这可以让你创建一个社区，让你知道人们是怎么使用你的项目的、还有什么地方可以提高。最后，它会帮助你找到可以帮助你维护你项目的人。这不仅能节省你的时间，也会鼓励人们到处宣传你的项目。

如果你需要一个例子，Welle README有一节关于社区和支持的。

怎么把它搞糟

如果你想完全坑你的用户，试试下面：

关闭你Github上问题的功能

用开发协议，那么用户必须写纸质信到坦桑尼亚

即使在README中修改了一行也要使用补丁

把你的项目放到Darcs，即使它是Ruby、JavaScirpt或者Clojure的项目

让人们很难找到项目在哪儿

这可以防止人们为你的项目做出贡献或者从你地方偷一点想法。

传给别人

到了一定时候，你可能对维护你的项目变得不感行却了。可能你已经换了一个新工作，或者不再使用你自己的项目了。在邮件列表中宣布这件事，让其他人来 接管这个项目。不久以后，有人会来帮忙的。在Github上对这种事是有好处的，特别是他们已经公布了一个让你转你仓库管理权的新特性。

不管你做什么，不要让你的项目变成没人负责的项目。这是最确信的方式可以让你现在或者未来的用户可以继续小猫大屠杀。

把项目移交给别人总是比之后找借口更好的。

怎么把它搞糟

如果你想完全坑你的用户，试试下面：

没有解释地直接停止贡献代码，回答邮件列表的问题

忽略提交请求，说他们的提交没有用，应该提交其它

说你是一个一旦问题解决就没有任何兴趣的人

这样就可以确保你的项目最后会被复制至少300次，最后一个代替的项目会被创建，因为搞清楚那个复制项目解决了那个问题是很烦人的。

最后的思考

正如你看到的，让你的项目可以被接受不是那么难吧。除了文档说明，让你的用户不生气，让运维的人不讨厌你也不需要花太多的时间。

维护一个开源项目是需要时间和精力的。但是它也是有回报的。我从GitHub还在测试的时候就已经在使用它了，而且几乎可以说没有其它什么事情让我有更多的专业机会。我很高兴我能有今天而不是活跃在开源社区。

如果你不想做一些很酷的事情，可能不要在第一时间就发布它。

3. 开源开发为什么要进行版本控制，如何进行版本控制？

上图是一个CAD设计师的施工图文件夹，他需要修改原来的施工图，但是他又不想把修改前的文件状态和内容给破坏掉，因为如果直接修改，万一修改的有问题，或者后悔修改了，如果以前的文件状态还在的话，大不了把修改糟糕了的删除了就是了，所以他其实就用到了最简单的版本控制方法，那就是复制重命名；  
  具体写程序的过程也是一样的道理，其实就是一个反复编辑，不厌其烦的编辑的过程。你修改的时候不希望破坏掉修改前的状态，最好就是每修改一段事件，能够保存一个状态，类似系统的自动快照一样，当后面出现问题的时候，可以自由选择复原到之前的某个快照状态；  
  还有一个原因，那就是，最后发布软件的时候很多时候会有多个版本，而生成软件的源代码却往往只有一份，只是在最后编译生成的时候用到不同的部分，大部分代码还是共用的，所以往往需要版本控制，几个版本复制几个文件夹出来；  
  版本控制的过程也是过程追踪记录，成就达成的过程。  
  还有就是很多时候做一个项目，往往不是单枪匹马单打独斗的，是需要很多个人之间合作的，多人合作自然也需要进行多份文档可以编辑，最后再来合并；  
  当然，上面主要是说明版本控制的方法并不是现在软件行业的版本控制方法，因为缺点很多：

* 首先，需要手工操作复制，慢，繁琐，而且大部分时候我们改动的只是很小的一部分，但每次都要把整个文件夹复制一份出来，这个自然是不合适的。
* 其次，多份快照之间到底有什么不同，这个怎么知道？靠记忆显然扯淡，每个快照附一份说明区别的文件显然也太扯淡了。
* 最后，多个备份的硬盘占用问题；

**版本控制的方式**

  鉴于上面的原因，于是就出现了很多版本控制的工具，这些工具能够做到随时提交快照，并附带上一个小的log说明这次更改的内容；而且控制方式是采用增量的方式，所以项目体积不会占用太多；如果你想知道前后有什么区别，这些工具还能给你提供基于文本对比，给你展示哪些地方进行过修改；如果编辑出问题，可以随时会到之前提交过的任一快照。多人合作，权限控制，多个分支等等，Good and Strong.

**集中式版本控制**

  集中式顾名思义就是代码集中到服务器，用的时候每个人把自己需要的被授权的那部分代码下载到自己的计算机上，提交也是最后提交到服务器上，服务器可以对代码做很好的控制，但是需要有网络，网络断了，就没法工作了。  
  代表就是SVN,Team Foundation等

**分布式版本控制**

  分布式自然就是每个人的地方多有一份完全的代码，提交和管理都是在本地进行，虽然有远端仓库，不过那是最终提交用的，没联网本地也是完整代码的，只是每次需要最新的代码的时候才必须联网；分布式的代表就是git，对应的网上仓库比较有名的就是github.

**版本控制的问题**

  主要是多个人合作的时候，可能会发生两个人同时提交一个文件的不同更改，这时候版本控制上就会出现冲突，就必须人工介入来决定保留谁的，否决谁的，当然这个事情是由仓库的建立者决定的。  
  目前的版本控制对文本内容有很好的支持，但是对二进制文件几乎没有支持，所以尽量不要在二进制文件上用这些版本控制工具，这不是他们的长处，二进制文件还是老老实实复制重命名吧，所以说其他设计行业的人苦逼啊。

4. 注册Github网站，填写个人资料（个人介绍，位置，个人网站等）。在Github创建一个仓库（repository），把本作业的文档上传到这个仓库中。将你的Github地址写到下面作为答案供检查。

https://github.com/Z139-tech/fantastic-guide.git