


# 第5章 作品的审阅与维护

## 本章提要


我写这本书的目的之一，就是想阐述一种“像写程序一样写作”的方法论。在我个人看来，写作和写程序代码在本质上并没有多少差异，两者的首要工作都应该都是用语言工具将问题或观点表达清楚，然后在表达清晰的基础上，我们再来追求一些表达的效率。该简单明了的地方，不要拐弯抹角、废话连篇，该突出重点的地方不要语焉不详、故弄玄虚。最后才是追求美感，锦上添花。如今有些文学流派，完全本末倒置，一个简单的故事，他偏偏要堆砌一堆优美的废话，无病呻吟，还号称自己“三年得二句，捻断数根须”，而且他表达的这种意境还要有缘人得之，这听上去，就像是那件只有聪明人才能看见的衣服，大概只有不谙世事的小孩说点实话。人间本就很复杂，我们大可不必如此画蛇添足，自欺欺人。有时候，作品的成功在于知道如何在设计上做减法，而不是盲目地往里添加各种其实并没有什么用的东西。

另一方面，和程序中始终会存在bug一样，我们在写作时表达中也始终会存在各种各样的问题，有些是表达逻辑上的缺陷，有些是表达视角上的问题，甚至有些只是一些语病、错别字问题。我们可能写不出没有bug的程序和没有问题的文章，但可以一直持续地完善自己的作品，优秀的文章和优秀的程序一样，来自于写完之后的精心打磨和维护，在这一章中，我们将继续以之前的论文为导引，讨论一下 `Markdown` 文档的审阅、修改与维护。

## 5.1 选择合适的审阅工具

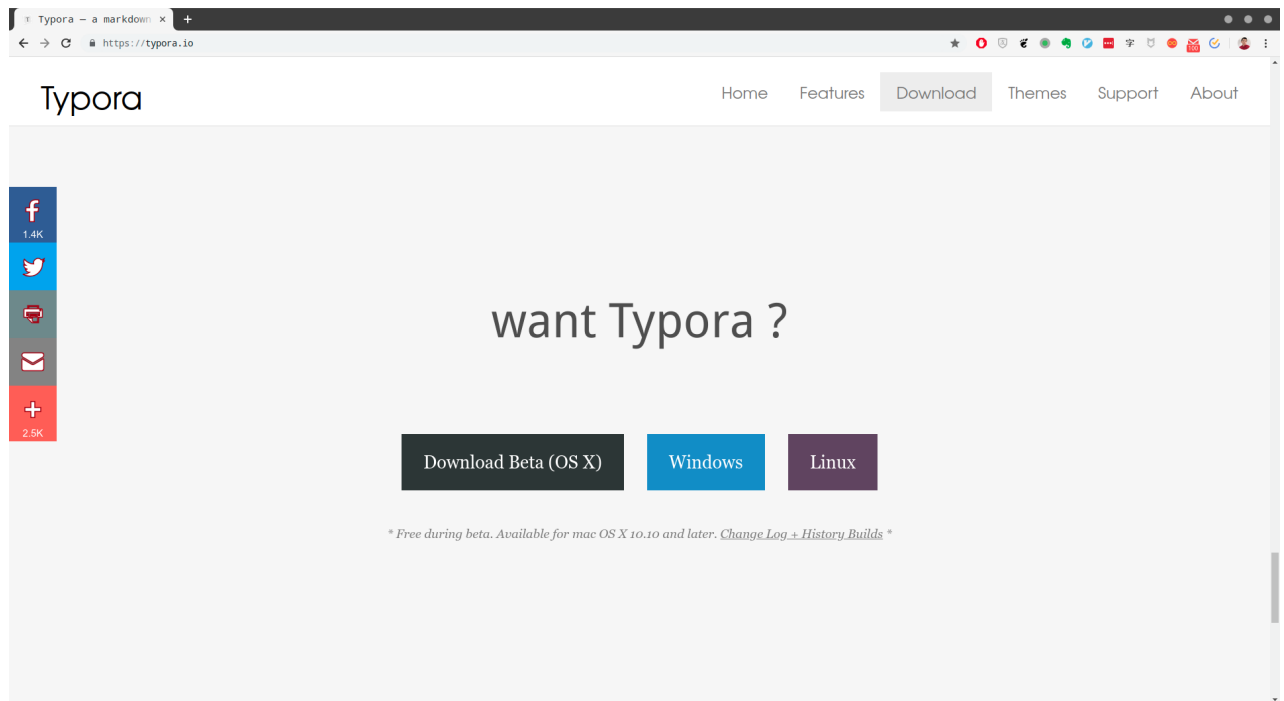
在一件作品完成之后，其质量的第一个把关者通常都应该是作者自己。每一个作者在写作过程中，多多少少都会存在一些输入性错误（相当于纸和笔时代的笔误），他们应该尽可能地找出作品中错误的文字、标点、代码以及图表。除此之外，我们的作品中通常还少不了会存在一些词不达意、表达不够充分的，甚至是表达错误，存在语病的文字。最后，作品本身的结构，譬如章节的先后顺序可能也需要做一些调整。所有的这一切，都是我们在自我检阅过程中要完成的任务。

要想较好地完成自我检阅工作，我们首先要做的就是通读自己的作品。在通读过程中，我们应该尽可能地专注于作品本身的内容，这时候任何与作品内容无关的事物都应被视为干扰，即使是最简单的 `Markdown` 标记也不例外。为了在通读作品时排除这些干扰，最简单直接的做法就是使用 `Markdown` 的预览器来完成通读工作。但这种做法有一个明显的缺陷：这些预览器只能显示 `Markdown` 标记的渲染效果，它们并不支持文章的直接修改。也就是说，每当我们发现作品中存在问题时，就要去编辑器中找到该问题所在的相应位置才能修改，这终归让人感觉没有 `Microsoft Word` 那样做到“所见即所得”的使用体验。所以在这里，我会建议大家使用同样实现了“所见即所得”的 `Typora` 编辑器来完成作品检阅阶段的工作。下面，我们就来介绍一下这款编辑器的具体使用。让我们先从利用这款编辑器的优势开始：

首先，当然是这款编辑器的跨平台特性。`Typora` 支持 `MacOS`、`Windows` 以及各种 `Linux` 桌面发行版，我们可以在各种工作环境中使用这款编辑器，不受具体操作系统的限制。譬如，我的台式机上运行的是 `Ubuntu`，而我的笔记本电脑是 `Macbook Air`，它运行的是 `MacOS`，我通常会在不同的时间段里分别在这两台电脑上对相同的文章进行编辑  `Typora` 在两边的用户体验是完全一致的，完全感觉不到使用环境的不同。下面，我们就来简单介绍一下如何在 `MacOS`、`Windows` 和 `Ubuntu` 这三个系统中如何下载安装 `Typora` 吧。

`Typora` 在 `MacOS` 和 `Windows` 下的安装过程极为简单，其步骤如下：

- 第一步. 在浏览器中打开下载地址：`https://typora.io/`
- 第二步. 在打开的页面中找到如下图所示的下载链接，下载相应系统的安装包，并根据安装向导完成安装即可。



请注意：MacOS的版本需在10.10以上。

而该编辑器在Ubuntu上的安装则更复杂一点，我们需要使用apt包管理器来安装（当然你也可以直接下载二进制安装包），这样做的好处是，今后我们可以直接使用apt命令来进行软件更新，其步骤如下：

- 第一步. 在浏览器中打开相关下载页面：`https://typora.io/#linux`
- 第二步. 复制页面中显示的 `shell` 命令，并在终端中执行。

```
1  # or run:
2  # sudo apt-key adv --keyserver keyserver.ubuntu.com --recv-keys BA300B7755AFCFAE
3  wget -qO - https://typora.io/linux/public-key.asc | sudo apt-key add -
4
5  # add Typora's repository
6  sudo add-apt-repository 'deb https://typora.io/linux .'
7  sudo apt-get update
8
9  # install typora
10 sudo apt-get install typora
```

请注意：上述下载页面也提供了二进制安装包的下载链接，其位于上面这段 `shell` 代码的正下方，显示为 `>> or,download binary file x63 x86` 字样。

然后是最重要的**即时预览**功能。Markdown编辑器常见的界面布局通常都是并列着两个窗口，左边为编码区，用于编写Markdown标记源码，右边则是预览区，用于显示相应的HTML渲染效果。在写作过程中，为了及时掌握标记的作用，在编写源码时同步查看一下渲染效果是可以接受的，因为有些标记的复杂应用（譬如制作图表，编写数学公式）还是需要我们在专注于标记本身的编码，但如果到了自我检阅阶段，需要专注于作品内容时这种安排就确实让人觉得有些不便了，这时候Typora的即时渲染功能，所见即所得的优势就凸显了出来。下面是该编辑器打开之前那篇论文第2章时的效果：



接下来是其支持的**扩展功能**：Typora主要支持的是 `GitHub Flavored Markdown (GFM)` 风格的语法，并提供了任务列表、表情符号、数学公式、代码高亮、图表生成等扩展支持，当然，部分扩展在默认设定下是没有被打开的，我们需要通过依次点击「文件-偏好设置」菜单，打开设置界面，在如下界面中勾选相关选项：



在设置完成之后，我们就可以来看看该编辑器对于这些扩展的支持情况，譬如下面是 $LaTeX$ 数学公式和 `Mermaid` 图库生成的渲染效果：

文件(F) 编辑(E) 段落(P) 格式(O) 视图(V) 主题(T) 帮助(H)

文件 大纲

Markdown扩展测试

- 乘方与开方
- 巨算符公式
- 条件公式
- 流程图
- 序列图

$$z = \sqrt{x^2 + y^2 - 1}$$

## 巨算符公式

$$\Pr\left(X > \frac{1}{3} \middle| Y = 0\right) = \int_0^1 p(t) \, dt \bigg/ (N^2 + 1)$$

## 条件公式

$$f(n) = \begin{cases} n/2, & \text{若} n \text{为偶数} \\ 3n + 1, & \text{若} n \text{为奇数} \end{cases}$$

## 流程图

## 序列图

< </>

239 词

当我们将光标移动到相关元素上时，编辑器就会自动显示该公式和图形元素背后的 Markdown 标记，我们可以随时就发现的问题对它们进行修改。譬如下面是修改序列图时的状态：

文件(F) 编辑(E) 段落(P) 格式(O) 视图(V) 主题(T) 帮助(H)

文件 大纲

Markdown扩展测试

- 乘方与开方
- 巨算符公式
- 条件公式
- 流程图
- 序列图

## 序列图

```
1 sequenceDiagram
2   participant 张三
3   participant 李四
4   participant 王五
5   张三 ->> 王五: 老王，你论文写好了吗？
6   loop 检查论文
7     王五 ->> 王五: 确认完成度
8   end
9   Note right of 王五: 再三确认 <br/>再回答.....
10  王五 -->> 张三: 写好了，但还需修改。
11  王五 ->> 李四: 你怎么样？
12  李四 -->> 王五: 我已经发给老师了！
```

< </>

238 词

除此之外，Typora还对几乎所有的 Markdown 语法都提供了快捷操作，我们可通过「段落」和「格式」这两个菜单中的选项来快速设置 Markdown 文档中的元素标记。当然，在我们足够熟练之后，还可以通过这些菜单项旁边标注的快捷键来进行更为高效的操作。而在阅读体验方面，Typora不仅界面设计得简约美观，还默认自带了6种 Markdown 的预览主题，我们可通过「主题」菜单进行随意切换，以进一步改善我们的阅读体验。譬如，下面是以 Newsprint 主题来显示上述论文第2章时的效果；



## 5.2 转换成更通用的格式

当然，一部作品仅仅靠自我审阅是远远不够的，因为每个作者都有其自身的思考盲区，总有一些问题靠他自己是发现不了的。所以下一步就是要找若干个可靠的人来对作品进行更细致而广泛地审阅或试读，并根据他们的反馈来进行进一步的修改。这时候，我们就必须要解决一个问题，那就是，*作品要以什么格式分发给我们的审阅者或试读者？*

### 5.2.1 目标格式与转换工具

要想解决这个问题，我们就得先来想一想参与作品审阅或试读的会是哪一些人。譬如，我们在这里用来举例的是一篇本科生的毕业论文，它的审阅者或试读者可能就是同专业的同学，以及直接指导这篇论文的老师，当然也有可能有一些愿意帮助你完成论文的热心网友。虽然 Markdown 文档本身纯文本的特性很适合作为分发和传阅，但我们必须考虑 Markdown 的实际普及情况，对于很多人来说，Markdown 可能还完全是一个陌生事物，我们不能在邀请别人审阅或试读自己作品的时候跟他说：“嗨，你应该先了解一下 Markdown 语法，还有，最好先安装一个叫做 Typora 的编辑器”。何况这些人中可能还有我们的老师和前辈。所以，更合适的做法是将 Markdown 文档视为作品的源文件，然后像使用程序源代码一样用编译器或解释器输出用户可以直接使用的文档格式。

对于程序来说，这种文档格式应该是一些二进制的可执行文件（譬如 Windows 下的 .exe 或 .dll 文件）。但对于文字作品来说，这种文档格式应该是怎么样的呢？首先，它应该是一种主流的跨平台文档格式，也就是说，这种格式的文档应该在所有主流操作系统上都只需要通过简单安装一款软件就能直接打开，并且这款软件应该是大多计算机用户都会安装，且是日常都在使用的，不需要进行复杂的系统配置。其次，这种格式的文档应该带给用户良好且一致的阅读体验。它应该像印刷品一样有美观的排版，将平台因素对文档呈现效果的影响降到最低。最后，这种文档格式应该要支持一定的批阅功能，以便读者在对作品提出审阅意见的同时不必修改文档本身的内容。综合以上要求，我们会发现，目前最能满足需求的应该就只有 PDF 和 docx 这两种文档格式了。

在确定了目标文档格式之后，接下来的问题是，我们要用什么工具来做格式转换呢？换言之，将 Markdown 文档输出成目标格式的“编译器”或“解释器”是什么呢？事实上，这个问题我们可以在 Typora 编辑器上找到答案。在该编辑器的「文件-导出」菜单中，我们可以看到它支持将 Markdown 文档导出哪些格式，但在点击你要导出的格式之后（譬如 docx），它会提示你这些功能需要先安装 Pandoc。这就是我们要使用的转换工具。

## 5.2.2 Pandoc的安装与使用

Pandoc是一款用 `Haskell` 语言实现的著名的标记语言转换工具。该工具支持多种操作系统平台，采用命令行交互界面，主要用于执行不同标记语言之间的格式转换，堪称文档处理领域中的“瑞士军刀”。在本节，我们就来详细介绍一下这款工具的安装与使用。

首先，我们需要根据自己所在的操作系统平台来安装Pandoc（如果需要用它将相关格式转换成 `PDF` 文件的话，还必须同时安装 `LATEX` 组件）。正如上面所说，Pandoc支持几乎所有的操作系统，我们在这里照例以Windows、MacOS和Ubuntu三个系统为例来介绍Pandoc的安装步骤。

首先是在Windows下，其安装步骤如下：

- 步骤1. 在浏览器中打开下载页面：`https://github.com/jgm/pandoc/releases/latest`
- 步骤2. 下载最新的安装包：`pandoc-<版本号>-windows.msi`
- 步骤3. 双击安装包后按照向导提示一步步完成安装
- 步骤4. 设置好Pandoc的环境变量

接着是在MacOS下，Pandoc有两种安装方式：

1. 二进制安装包：

- 步骤1. 在浏览器中打开下载页面：`https://github.com/jgm/pandoc/releases/latest`
- 步骤2. 下载最新的二进制安装包：`pandoc-<版本号>-osx.pkg`
- 步骤3. 双击安装包后按照向导提示一步步完成安装

2. homebrew包管理器，这种方式只需要在终端中执行以下命令即可：

```
1 | sudo brew install pandoc
```

最后，在Ubuntu下，Pandoc的安装方式也有两种：

1. 二进制安装包：

- 步骤1. 在浏览器中打开下载页面：`https://github.com/jgm/pandoc/releases/latest`
- 步骤2. 下载最新的二进制安装包：`pandoc-<版本号>-amd64.deb`
- 步骤3. 双击安装包后按照向导提示一步步完成安装

2. apt包管理器，这种方式只需要在终端中执行以下命令即可：

```
1 | sudo apt-get install pandoc
```

除此之外，我们在所有的操作系统中都可以选择先安装 `Haskell` 平台，然后使用其中的 `cabal` 工具来安装Pandoc，其命令如下：

```
1 | cabal update
2 | cabal install pandoc
```

在完成安装之后，可以在终端中输入 `pandoc -v` 来查看我们所安装的版本，只要看到类似于下面这样的输出，就说明Pandoc安装成功了：

```
1 / 1 + [ ] [ ] Tilix: 默认
1: _zsh_tmux_plugin_run
owlman ~ 1 pandoc -v
pandoc 1.19.2.4
Compiled with pandoc-types 1.17.0.5, texmath 0.9.4.4, skylighting 0.3.3.1
Default user data directory: /home/owlman/.pandoc
Copyright (C) 2006-2016 John MacFarlane
Web: http://pandoc.org
This is free software; see the source for copying conditions.
There is no warranty, not even for merchantability or fitness
for a particular purpose.
owlman ~ 1
0 0* > ~ 0.2 0.3 0.3 2019-05-11 < 19:29 owlman-Lenovo
```

在确认安装了Pandoc之后，接下来就可以开始使用Pandoc了。我们先来看看Pandoc到底支持哪一些标记语言：

Pandoc可读取的源格式	Pandoc可生成的目标格式
Markdown	HTML格式：包括XHTML，HTML5及HTML slide。
reStructuredText	文字处理软件格式：包括docx、odt、OpenDocument XML。
textile	电子书格式：包括EPUB（第2版及第3版）、FictionBook2
HTML	技术文档格式：包括DocBook、GNU TexInfo、Groff manpages、Haddock。
DocBook	页面布局格式：InDesign ICML。
LaTeX	大纲处理标记语言格式：OPML。
MediaWiki标记语言	TeX格式：包括LaTeX、ConTeXt、LaTeX Beamer。
OPML	PDF格式：需要LaTeX支持。
Org-Mode	轻量级标记语言格式：包括Markdown、reStructuredText、textile、Org-Mode、MediaWiki标记语言、AsciiDoc。
Haddock	自定义格式：可使用lua自定义转换规则。

或许，上述表格中列出的很多格式可能都让人觉得很陌生，但这没有关系，我们在这里的主要目标就是将 Markdown 文档转换成 PDF 和 docx 格式，并不需要你掌握Pandoc支持的所有文档格式。下面，我们就先来看看如何用Pandoc将 Markdown 文档输出为Microsoft Word文档（即 docx 格式）。

通过在终端中输入 `pandoc -h`，我们可以了解到Pandoc命令的基本格式如下：

```
pandoc [options] [input-file] ...
```

并且有以下常用选项：



选项	含义
<code>-f FORMAT,</code> <code>-r FORMAT,</code> <code>--from=FORMAT,</code> <code>--read=FORMAT</code>	指定输入文件的格式，若不指定，pandoc可以从明显的文件后缀名中推测，若无明显提示，默认的输入文件格式是 Markdown，默认的输出文件格式是 HTML。
<code>-t FORMAT,</code> <code>-w FORMAT,</code> <code>--to=FORMAT,</code> <code>--write=FORMAT</code>	指定输出文件的格式。
<code>-o FILE,</code> <code>--output=FILE</code>	写输出到FILE文件而不是到标准输出。
<code>--list-input-formats</code>	列出支持的输入文件格式。
<code>--list-output-formats</code>	列出支持的输出文件格式。
<code>--list-extensions</code>	列出支持的 Markdown 扩展，+代表默认支持，-代表默认不支持。
<code>-s,</code> <code>--standalone</code>	产生输出文件时附带适当的头注和脚注（比如 HTML）。
<code>-c URL,</code> <code>-css=URL</code>	链接到 CSS 样式表。该选项能够使用多次来引入多个文件，所指定的文件能够以指定的顺序依次引入。

所以根据帮助信息中给出的说明，如果要想将 Markdown 文档转换为 Microsoft Word 文档，我们需执行如下命令：

```
1 | pandoc input.md -o output.docx -c Github.css
```

在这里，input.md是输入文件，-o 选项后面跟的参数代表的输出文件，即output.docx，-c 选项指定的是 Markdown 文档的渲染样式，这里选择的是Github的样式。另外，如果我们在 Markdown 文档中使用了 $LaTeX$ 标记，那就还需要在上述命令的后面再加上一个--latex-engine=xelatex 参数，以用来指定 $LaTeX$ 引擎。

在通常情况下，Pandoc会根据文件的后缀名自动判断格式。当然，我们也可以显式地指定输入文件和输出文件格式，譬如：

```
1 | pandoc -f markdown -t docx input.md -o output.docx -c Github.css
```

在上述命令中，-f 选项代表的是输入文件的格式，这里指定了 markdown。而 -t 选项代表的则是输出文件的格式，这里指定的是 docx。读到这里，有过编译C/C++或Java程序经验的读者肯定已经发现了，Pandoc的使用方式与我们曾经使用过的gcc、javac等编译器是非常类似的。这意味着，我们也可以使用 makefile 这样的项目配置文件来管理 Markdown 项目的输出，真正实现像管理程序项目一样管理我们的写作项目，譬如，下面是我为自己论文项目编写的一个 makefile 文件，它会根据我修改的 Markdown 文档来更新输出的 Microsoft Word 文档：

```
1 | # 时间：2019年04月22日
2 | # 作用：将markdown文件转换成docx格式
```



```

3
4 src = $(shell find ./ -name "*.md")
5 docx = $(src:%.md=../docx/%.docx)
6
7 all: $(docx)
8     ls -l ../docx
9
10 ../docx/%.docx:%.md
11     pandoc -f markdown+tex_math_dollars -t docx $< -o $@ --latex-engine=xelatex
12
13 clean:
14     rm -rf $(docx)

```

当然，我们需要先像管理程序带一样将上面这个 `makefile` 文件和所有的 `Markdown` 文件一起放入项目的 `src` 子目录中，然后再创建一个 `docx` 目录以作为输出目录，最后才能进到 `src` 目录中执行 `make` 命令。如果有读者对 `makefile` 的写法并不熟悉，可以参考我在[附录A: Makefile 简易教程](#)中所做的相关介绍。

对于将 `Markdown` 文档转成 `PDF` 格式，我们当然也可以采用类似的方法，只需要将 `Pandoc` 命令修改如下即可：

```

1 pandoc -f markdown -t pdf input.md -o output.pdf -c Github.css --latex-engine=xelatex
  -V mainfont=heiti

```

但我们在这里其实可以用一种更简单的方式，即我们在一开始就提到的，`Typora` 编辑器的「文件-导出」菜单。在安装了 `Pandoc` 之后，`Typora` 编辑器支持将 `Markdown` 文档导出为 `HTML`、`PDF`、`odt`、`rtf`、`epub`、`LaTeX`、`Media Wiki` 等格式。譬如，下面是之前论文第2章导出的 `PDF` 文档：



当然，如果你对输出的 `PDF` 文档在排版时有更细致的要求（譬如想调整图片的大小、实现图文绕排），我们也可以选择先将 `Markdown` 文档导出为 `TEX` 格式的源文件，在使用 `LATEX` 进行排版处理之后再输出 `PDF` 格式。

### 5.3 使用版本控制系统

在完成毕业论文的初稿之后，指导老师往往会针对性地提出各种不同的问题，让我们进行多轮修改，有时候甚至要改上七八稿以上。在这种情况下，我们应该如何管理这篇论文的各个修改版本呢？众所周知，我们通常都是在空间这个维度上对项目进行管理的。毕竟，项目中的源文件（譬如C源码文件、`Markdown`文件）、输出文件（譬如可执行文件、`PDF`文件）以及配置文件（譬如`makefile`文件）都是以文件和目录的形式存储在磁盘空间上的，但现在我们要管理的是自己不同时间段里所做的修改，这种时间维度上的项目管理应该怎么做呢？答案就是：使用版本控制系统（Version Control System）。

版本控制系统是一种记录文件修改演化的系统，它的作用就是方便我们找回项目在某个特定时间点（即版本）上的文件、查看这些版本前后都做了哪些修改，并对这些修改进行比对，以修正一些致命性的错误。从版本控制系统的发展历史来看，它主要经历了本地版本控制、集中式版本控制和分布式版本控制三个阶段：

- **本地版本控制(Local Version Control System)**：这顾名思义就是本地化的版本控制系统，它没有网络协作等较为先进的版本控制的概念。
- **集中式版本控制(Centralized Version Control System)**：这种版本控制系统通常设置有一台用于执行版本控制服务的中央服务器，这台服务器会一直处于运行状态，由用户上传或下载其项目的各个版本。这类系统在很长一段时间里都是版本控制的主流方式，著名的`CVS`与`SVN`都是这一类版本控制系统的代表。
- **分布式版本控制(Distributed Version Control System)**：这种版本控制系统解决了集中式版本控制可能因为其中央服务器故障而带来的麻烦，它可以让每一台客户端都完全镜像整个被纳入版本控制的项目。也就是说，在分布式版本控制系统中，任何一台机器都可以视为版本控制服务器。即使有一台服务器失去服务能力，其它服务器还可以继续协作维持版本控制系统的正常运转。

对于毕业论文这种个人属性比较强，而又需要与人分享协作的项目来说，显然分布式版本控制系统是比较合适的选择。所以，接下来我们就以时下最流行的分布式版本控制系统——`git`——为例来介绍一下`Markdown`项目的管理。

### 5.3.1 git简介

`git`是林纳斯·托瓦兹（Linus Torvalds）为更好地管理Linux内核开发而设计的一个分布式版本控制软件。这款软件与其它主流版本控制系统最大的区别是：在项目版本更新的过程中，`git`并没有直接去记录基于初始文件的变化数据，而是通过一系列快照（Snapshot，就像是个小型的文件系统）来保存记录每个文件。在此过程中，对于那些没有发生变化的文件，它们在新版本中就会是一个指向其最近一次更新的链接。此外，几乎所有`git`的操作都是在本地进行的，所以并不存在服务器“延迟”的问题，几乎所有的操作都是瞬间完成的。例如，如果我们想要查看项目的版本历史，就不必特地到服务器上抓取历史记录，直接在本地浏览即可。而且，这种几乎完全本地化的操作方式也为许多特殊的应用场景提供了支持，譬如：我们在开发过程中都可能会遇到要在无网络环境中需要对自己的项目进行修改，并同时要使用版本控制系统将这些修改记录下来的尴尬场景，这时，`git`无疑就是最好的选择了。

下面我们来简单介绍一下`git`的存储机制。在`git`中，我们的项目文件主要有三种状态，它们分别是：

- **提交**：在这种状态下，文件或数据已经安全的存放在了`git`本地数据库中。
- **修改**：在这种状态下，文件或数据已经修改但是尚未提交到数据库。
- **暂存**：在这种状态下，文件或数据已被标记要放入到下一次要提交的版本中。

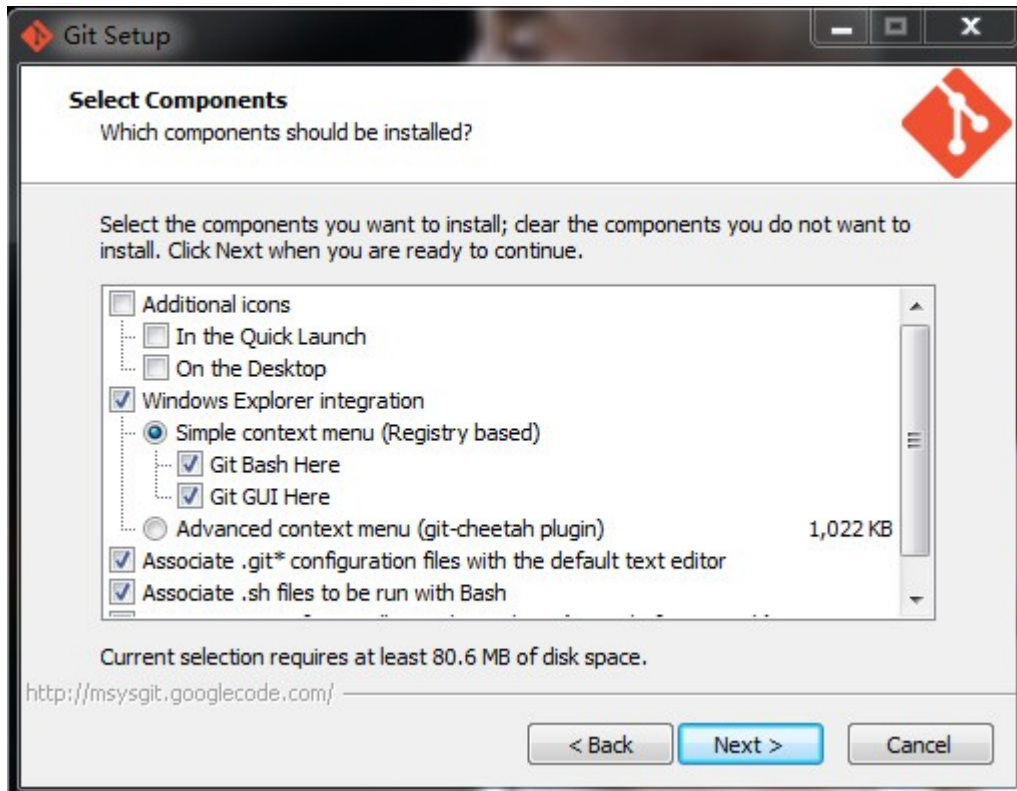
为了实现这种不同状态的存储，一个`git`项目应该要被分成三个组成部分（这里假设我们的项目目录叫`git-repo`）：

- **git目录**：该目录是存放项目中所有元数据以及对象的地方（即`git-repo/.git/`）
- **工作目录**：该目录用于存储从`git`项目数据库中`checkout`出的一个单独的（默认情况下是最新的）项目版本，用于对指定项目版本中的文件进行修改和编辑（即`git-repo/`）
- **暂存区**：这个区域实际上是存放在`git`目录（`git-repo/.git/`）中的一个简单的文件，里面存放着下一次需要`commit`的文件的信息。

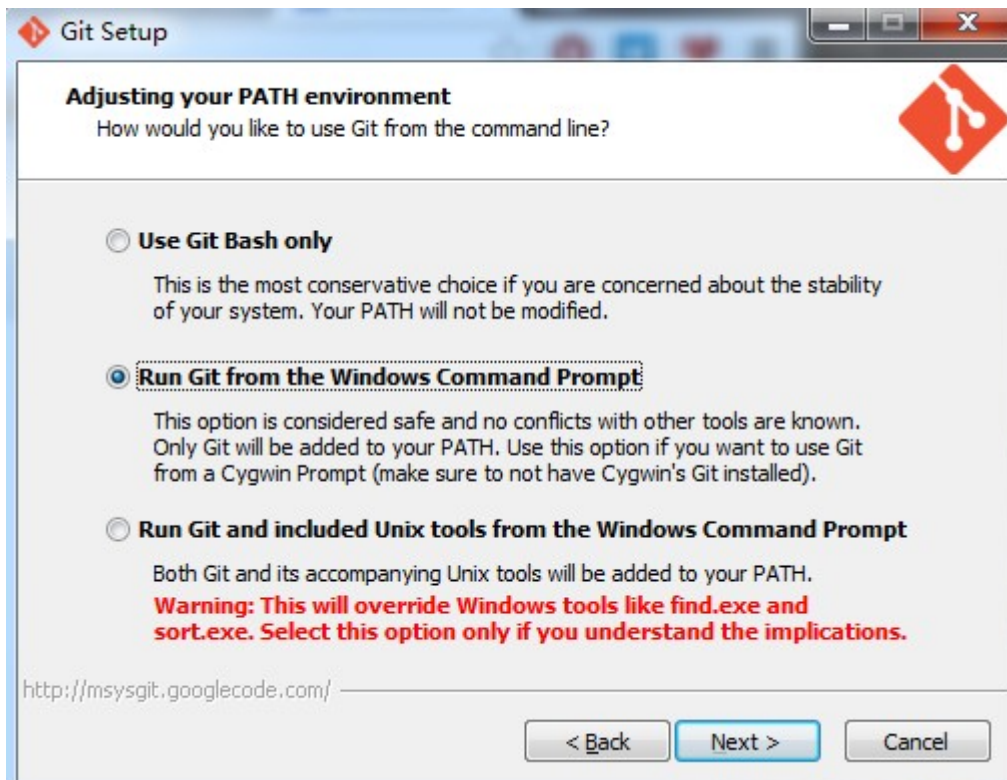
### 5.3.2 git的安装与配置

git支持多种操作系统，采用命令行交互界面，在这里，我们以Windows为例来介绍一下如何安装和配置git（MacOS和Ubuntu下的安装则更为简单，使用相应的包管理器即可安装，请读者参考相关帮助信息）：

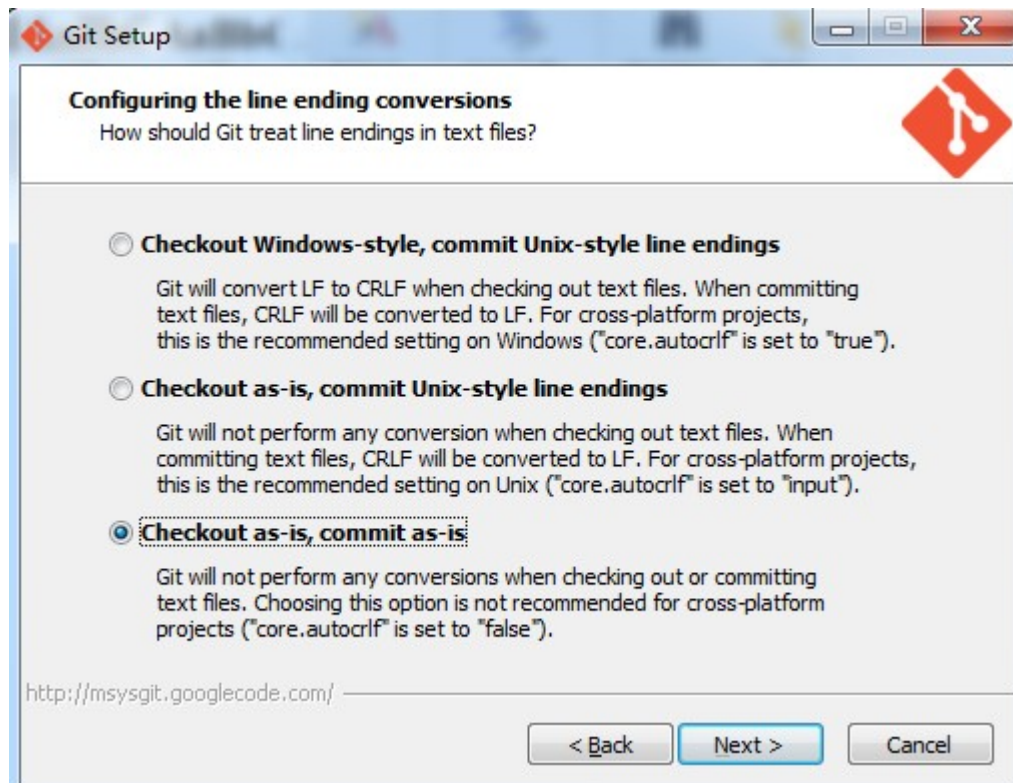
- 步骤1. 在浏览器中打开git for Windows的下载页面：<https://gitforwindows.org/>
- 步骤2. 下载安装包，并打开安装包启动安装向导，在这一步请选择好安装路径和要安装的组件：



- 步骤3. 设置环境变量。如果我们希望在Windows自带的终端中使用git，在这里就必须选择第二项或第三项：



- 步骤4. 配置文本文件的换行符形式：



接下来，我们还要来做一些基本配置。在这里，我用 Python 专门编写了一个 git 的配置脚本，以便日后重复使用这些配置（具体配置请看代码中的注释说明）：

```
1  #!/usr/bin/env python
2  '''
3      @author: lingjie
4      @name:   git_configuration
5  '''
6
7  import os
8  import sys
9  import platform
10
11  title = "= Starting " + sys.argv[0] + "..... ="
12  n = len(title)
13  print(n*'=')
14  print(title)
15  print(n*'=')
16
17  cmds = [
18      # 配置用户名
19      "git config --global user.name 'lingjie'",
20      # 配置邮箱，请注意，该邮箱纯属虚构，如有雷同必是巧合
21      "git config --global user.email 'lingjie@mail.com'",
22      # 配置push命令的默认行为
23      "git config --global push.default simple",
24      # 配置git以多种颜色输出信息
25      "git config --global color.ui true",
26      # 解决中文文件名乱码问题，前提是终端环境要支持utf8编码
27      "git config --global core.quotepath false",
28      # 配置日志信息输出编码为utf8
```



```

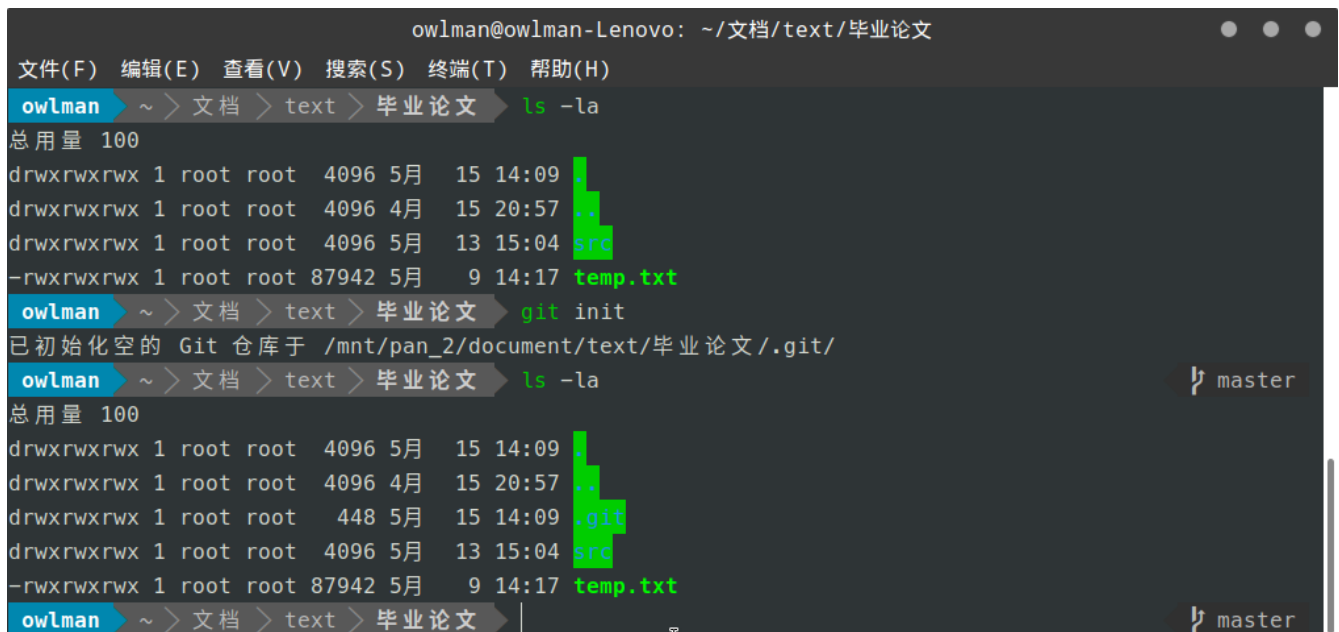
29     "git config --global i18n.logOutputEncoding utf-8",
30     # 配置提交信息输出编码为utf8
31     "git config --global i18n.commitEncoding utf-8"
32 ]
33
34 # 根据所在操作系统设置换行符
35 if platform.system() == "Windows":
36     cmds.append("git config --global core.autocrlf true")
37 else:
38     cmds.append("git config --global core.autocrlf input")
39
40 for cmd in cmds:
41     print(cmd)
42     os.system(cmd)
43
44 print(n*'=')
45 print("=      Done!" + (n-len("=      Done!")-1)*' ' + "=")
46 print(n*'=')

```

当然，如果没有 Python 执行环境，我们也可以手动在终端中执行上述代码存储在 `cmds` 数组中的配置命令，除了不可重用外，其他效果都是一样的。

### 5.3.3 git的基本操作

在完成了git的安装和配置之后，我们就可以用git来管理和维护我们的论文项目了。首先，我们需要将论文项目纳入版本控制系统，用git的术语来说，就是将其初始化成是一个git仓库。好了，现在让我们进入到论文项目所在的目录，然后执行 `git init` 命令：



```

owlman@owlman-Lenovo: ~/文档/text/毕业论文
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
owlman ~ > 文档 > text > 毕业论文 ls -la
总用量 100
drwxrwxrwx 1 root root 4096 5月 15 14:09 .
drwxrwxrwx 1 root root 4096 4月 15 20:57 ..
drwxrwxrwx 1 root root 4096 5月 13 15:04 src
-rwxrwxrwx 1 root root 87942 5月 9 14:17 temp.txt
owlman ~ > 文档 > text > 毕业论文 git init
已初始化空的 Git 仓库于 /mnt/pan_2/document/text/毕业论文/.git/
owlman ~ > 文档 > text > 毕业论文 ls -la
总用量 100
drwxrwxrwx 1 root root 4096 5月 15 14:09 .
drwxrwxrwx 1 root root 4096 4月 15 20:57 ..
drwxrwxrwx 1 root root 448 5月 15 14:09 .git
drwxrwxrwx 1 root root 4096 5月 13 15:04 src
-rwxrwxrwx 1 root root 87942 5月 9 14:17 temp.txt
owlman ~ > 文档 > text > 毕业论文

```

在执行完 `git init` 命令之后，项目目录下就多出了一个叫做 `.git` 的隐藏子目录。这就是我们之前所说的git目录，该项目后续提交的所有版本都会被存放在其中。接下来，让我们执行 `git status` 命令来查看一下这个git仓库的状态。在使用git的过程中，我们将会反复使用到这个命令：

```
owlman@owlman-Lenovo: ~/文档/text/毕业论文
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
owlman ~ > 文档 > text > 毕业论文 git status master
位于分支 master

尚无提交

未跟踪的文件：
（使用 "git add <文件>..." 以包含要提交的内容）

src/
temp.txt

提交为空，但是存在尚未跟踪的文件（使用 "git add" 建立跟踪）
owlman ~ > 文档 > text > 毕业论文 master
owlman ~ > 文档 > text > 毕业论文 master
```

我们可以看到，`git status` 命令输出了相当详细的信息。其中，第一行表明的是当前所在的版本分支（默认情况下自然是 `master` 分支）。接下来，它告诉我们目前没有任何需要提交的内容。然后当前项目中有哪些文件可以被纳入到仓库中。正如之前所说，我们将所有的 `Markdown` 文件都存放在 `src` 目录中，下面让我们根据提示，用 `git add` 命令将这些文件纳入到版本系统中来，然后再次查看仓库状态：

```
owlman@owlman-Lenovo: ~/文档/text/毕业论文
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
owlman ~ > 文档 > text > 毕业论文 git add src/ master
owlman ~ > 文档 > text > 毕业论文 git status master
位于分支 master

尚无提交

要提交的变更：
（使用 "git rm --cached <文件>..." 以取消暂存）

新文件：    src/01_系统概述.md
新文件：    src/02_系统数据库的设计.md
新文件：    src/03_功能模块的划分.md
新文件：    src/04_开发环境与工具的选择.md
新文件：    src/05_各功能模块的实现.md
新文件：    src/06_系统程序的发布.md
新文件：    src/07_设计总结.md
新文件：    src/img/3-1.png
新文件：    src/makefile

未跟踪的文件：
（使用 "git add <文件>..." 以包含要提交的内容）

temp.txt

owlman ~ > 文档 > text > 毕业论文 master
```

请注意，我们在这里所添加的文件只是被 `git` 标记成了“要提交的变更”，也就是说，它们都处于“暂存”状态。接下来，我们可以根据自己的需要来选择是使用 `git commit` 命令提交这些文件，还是使用 `git rm -cached` 命令撤销一些文件的“暂存”状态。在这里，我们来将之前完成的论文初稿提交为项目的第一个版本：

```
owlman@owlman-Lenovo: ~/文档/text/毕业论文
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)

owlman ~ > 文档 > text > 毕业论文 git commit -m "初稿" master
[master (根提交) a6215b7] "初稿"
9 files changed, 393 insertions(+)
create mode 100644 src/01_系统概述.md
create mode 100644 src/02_系统数据库的设计.md
create mode 100644 src/03_功能模块的划分.md
create mode 100644 src/04_开发环境与工具的选择.md
create mode 100644 src/05_各功能模块的实现.md
create mode 100644 src/06_系统程序的发布.md
create mode 100644 src/07_设计总结.md
create mode 100644 src/img/3-1.png
create mode 100644 src/makefile

owlman ~ > 文档 > text > 毕业论文 git status master
位于分支 master
未跟踪的文件：
  （使用 "git add <文件>..." 以包含要提交的内容）

    temp.txt

提交为空，但是存在尚未跟踪的文件（使用 "git add" 建立跟踪）
owlman ~ > 文档 > text > 毕业论文 master
```

由于git生成的版本文件名都是一些用Hash算法生成的字符串，所以我们在使用 `git commit` 进行版本提交时经常会用 `-m` 参数来注释一下这次提交的内容，比如我们这次提交的是论文的第一稿，我就将其注释为“初稿”。这样做的目的是为日后的查找和维护提供方便。

在完成提交之后，当我们再次执行 `git status` 时就会看到，当前项目下没有被纳入版本控制的就只有 `temp.txt` 这个临时文件了，我们也没有打算将其纳入项目。接下来，我们来对论文的第1章做一些模拟修改，改变一下前两段的某些措辞，然后再次执行 `git status` 命令。然后我们从输出信息中可以看到，`src/01_系统概述.md` 已经被修改过了：

```
owlman@owlman-Lenovo: ~/文档/text/毕业论文
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)

owlman ~ > 文档 > text > 毕业论文 git status master
位于分支 master
尚未暂存以备提交的变更：
  （使用 "git add <文件>..." 更新要提交的内容）
  （使用 "git checkout -- <文件>..." 丢弃工作区的改动）

    修改：    src/01_系统概述.md    I

未跟踪的文件：
  （使用 "git add <文件>..." 以包含要提交的内容）

    temp.txt

修改尚未加入提交（使用 "git add" 和/或 "git commit -a"）
owlman ~ > 文档 > text > 毕业论文 master
```

并且，我们可以通过 `git diff` 命令来查看自己所做的修改：



```
git diff
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
diff --git a/src/01_系统概述.md b/src/01_系统概述.md
index 6e72b98..7d49bfc 100644
--- a/src/01_系统概述.md
+++ b/src/01_系统概述.md
@@ -1,6 +1,6 @@
# 第1章：系统概述

-正如最近几年新闻媒体大量报道的，Web 应用进入了一个新的时代。那么，我们是否对这个被称为 Web 2.0 的新
概念有着在技术层面的理性的认知呢？要回答这个问题，我们就需要回到早期的 Web 开发中，了解一下以 ASP 为
代表的服务器脚本，并找到其中存在的问题。而了解这个问题的最好方法就是到具体开发中去亲身体会。
+正如近几年新闻媒体上大量报道的，Web 应用进入了一个新的时代。那么，我们是否对这个被称为 Web 2.0 的新
概念有着在技术层面的理性的认知呢？要回答这个问题，我们就需要回到早期的 Web 开发中，了解一下以 ASP 为
代表的服务器脚本，并找到其中存在的问题。而了解这个问题的最好方法就是到具体开发实践中去探索。

## 1.1 系统的设计目的和意义

:
```

这个时候，如果我们后悔了，可以使用 `git checkout -- src/01_系统概述.md` 命令来撤销自己在当前工作区中对第 1 章所做的修改。但如果你已经使用 `git add` 命令将修改加入到了暂存区中，那就得使用 `git reset HEAD src/01_系统概述.md` 命令来撤销修改了：

```
owlman@owlman-Lenovo: ~/文档/text/毕业论文
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
owlman ~ > 文档 > text > 毕业论文 git status master
位于分支 master
要提交的变更：
（使用 "git reset HEAD <文件>..." 以取消暂存）

    修改：      src/01_系统概述.md

未跟踪的文件：
（使用 "git add <文件>..." 以包含要提交的内容）

    temp.txt

owlman ~ > 文档 > text > 毕业论文 git reset HEAD src/01_系统概述.md master
重置后取消暂存的变更：
M       src/01_系统概述.md
owlman ~ > 文档 > text > 毕业论文 git status master
位于分支 master
尚未暂存以备提交的变更：
（使用 "git add <文件>..." 更新要提交的内容）
（使用 "git checkout -- <文件>..." 丢弃工作区的改动）

    修改：      src/01_系统概述.md

未跟踪的文件：
（使用 "git add <文件>..." 以包含要提交的内容）

    temp.txt

修改尚未加入提交（使用 "git add" 和/或 "git commit -a"）
owlman ~ > 文档 > text > 毕业论文 | master
```

但这里，我们会选择将修改内容提交为第二个版本，并注释为“第二稿：第1章的部分措辞变更”。这样一来，这个论文项目就有了两个版本。我们可以使用 `git log` 命令来查看版本历史：

```
git log

文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
commit b7e8ee37fc2bb63918c4b8a3af2507f251f1f94d (HEAD -> master)
Author: owlman <jie.owl2008@gmail.com>
Date: Thu May 16 14:29:58 2019 +0800

    “第二稿：第1章的部分措辞变更”

commit a6215b7e2522f05bff60e7fada7a65d905231a5b
Author: owlman <jie.owl2008@gmail.com>
Date: Wed May 15 15:56:00 2019 +0800

    “初稿”

~
(END)
```

如你所见，git详细记录了我们提交的每一个版本的具体信息（包括checksum值、提交者信息、提交时间），我们可以使用 `git checkout` 命令将文件恢复到之前的任意一个版本上，操作方式和之前撤销工作目录中的修改是一样的，那次操作实际上就是拿最近版本中的同名文件覆盖掉当前被修改的文件。

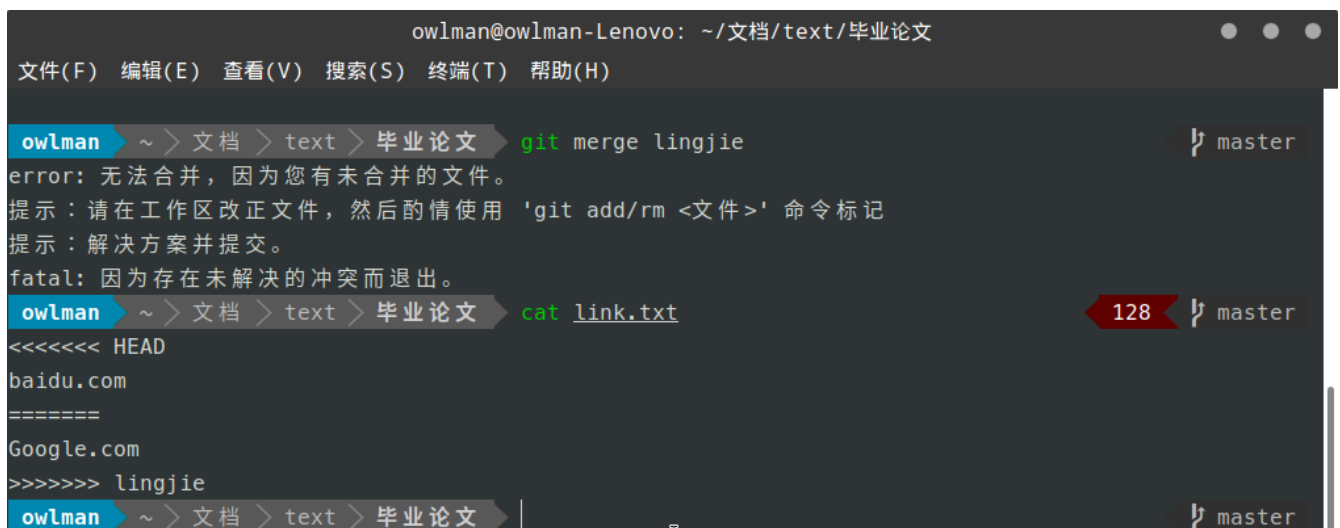
下面，我们再来简单介绍一下版本控制中的另一种管理操作：分支管理。git的分支管理是异常的简单和方便，可以使用 `git branch` 命令进行非常直观的操作。首先可以在工作目录下查看当前的项目存在多少分支：

```
1 $ git branch
2 * master
```

如你所见，目前项目中只有一个叫做 `master` 的主分支，星号（\*）代表的是我们当前所在的分支。下面，让我们使用 `git branch lingjie` 命令来为项目添加一个名为 `lingjie` 的分支，并使用 `git checkout lingjie` 命令切换到这个新的分支上：

```
1 $ git branch lingjie
2 $ git checkout lingjie
3 $ git branch
4 * lingjie
5 master
```

现在，我们可以看到自己已经位于新建的 `lingjie` 分支上了。在版本控制系统中，分支的作用是隔离暂时无法确定的修改，使这些修改在不影响主分支的情况下进行，以确定这些修改的可行性。也就是说，从我们建立 `lingjie` 分支的那一刻起，我们在该分支上所做的所有修改都将独立于 `master` 分支而存在。如果我们确认了这些修改的可行性，就只需在 `master` 分支上执行 `git merge lingjie` 命令将这些修改合并到主分支上即可。当然，如果我们在建立 `lingjie` 分支之后，在两个分支上都对同一个文件做了修改，并提交它们，那在执行合并命令时，git会要求你就该文件上的修改做一些版本冲突的处理。例如，我们在两个分支上都创建一个 `link.txt`，在 `master` 分支上该文件内容为“baidu.com”，在 `lingjie` 分支上其内容则为“Google.com”。然后，我们在 `master` 分支上执行 `git merge lingjie` 命令就会看到：



```
owlman@owlman-Lenovo: ~/文档/text/毕业论文
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)

owlman ~ > 文档 > text > 毕业论文 git merge lingjie master
error: 无法合并，因为您有未合并的文件。
提示：请在工作区改正文件，然后酌情使用 'git add/rm <文件>' 命令标记
提示：解决方案并提交。
fatal: 因为存在未解决的冲突而退出。
owlman ~ > 文档 > text > 毕业论文 cat link.txt 128 master
<<<<<< HEAD
baidu.com
=====
Google.com
>>>>>> lingjie
owlman ~ > 文档 > text > 毕业论文 master
```

这时候，我们就要对 `link.txt` 文件中的内容进行修改，去除掉里面标识冲突的字符，合并才能成功：

```
owlman@owlman-Lenovo: ~/文档/text/毕业论文
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)

owlman ~ > 文档 > text > 毕业论文 cat link.txt master
baidu.com
Google.com
owlman ~ > 文档 > text > 毕业论文 git add link.txt master
owlman ~ > 文档 > text > 毕业论文 git commit -m "master合并 lingjie" master
[master d3d7102] "master合并 lingjie"
owlman ~ > 文档 > text > 毕业论文 git merge lingjie master
已经是最新的。
owlman ~ > 文档 > text > 毕业论文 | master
```

在合并完成之后，如果我们要删除该分支，可以使用 `git branch -d lingjie` 命令：

```
1 $ git branch -d lingjie
2 已删除分支 lingjie (曾为 63c0da1).
3 $ git branch
4 * master
```

这时候，我们就会看到 `lingjie` 分支已经不见了。

### 5.3.4 远程仓库操作

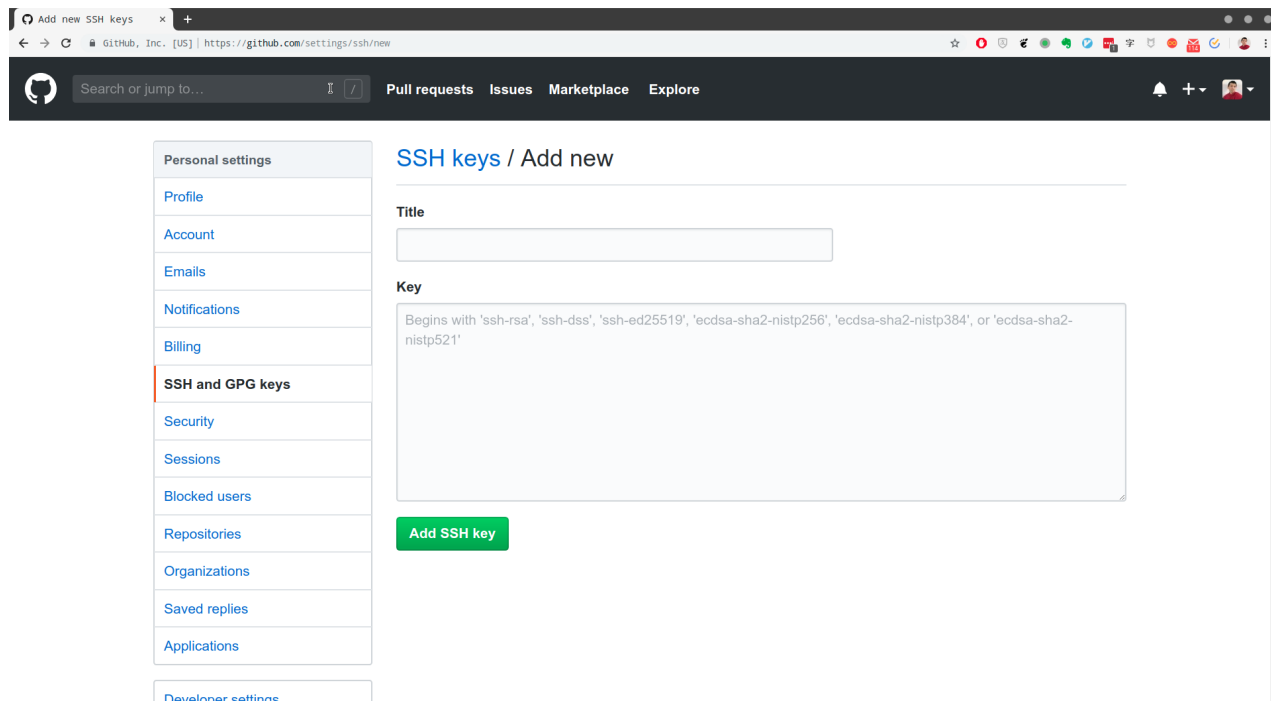
要参与任何一个git项目的网络协作，必须要了解该如何管理远程仓库。远程仓库是指托管在网络上的项目仓库。同他人协作开发某个项目时，需要管理这些远程仓库，以便推送或拉取数据，分享各自的工作进展。管理远程仓库的工作，包括添加远程库、移除废弃的远程库、管理各式远程库分支、定义是否跟踪这些分支，等等。在本节，我们就以目前世界上最大的git源码托管服务github为例，来简单介绍一下这些操作。

在使用Github远程仓库之前，我们需要先就其ssh链接方式来做一些设置：

- **步骤1. 首先来配置SSH的密钥：**我们可以先查看系统的用户目录（在windows下通常是 `C:\Documents and Settings\`，其他系统通常都是 `/home/`），如果当前系统中已经完成了对SSH的配置，用户目录下就会存在一个名为 `.ssh` 的隐藏目录，该目录下会存有 `id_rsa`、`id_rsa.pub` 这两个文件。现在请将其备份一下，然后生成新的，在用户目录中打开终端输入以下命令（请注意，请将用尖括号标注的信息替换成你要配置的具体参数）：

```
1 $ ssh-keygen -t rsa -C "lingjie@mail.com" <此处应输入你自己的电子邮件地址>
2 Enter file in which to save the key (~/ssh/id_rsa):
3 Enter passphrase (empty for no passphrase): <在此处输入你为其设定的密码>
4 Enter same passphrase again: <重复一遍你设定的密码>
5 Your identification has been saved in ~/ssh/id_rsa.
6 Your public key has been saved in ~/ssh/id_rsa.pub.
7 The key fingerprint is:
8 e8:ae:60:8f:38:c2:98:1d:6d:84:60:8c:9e:dd:47:81 lingjie@mail.com
```

- **步骤2. 将生成的公钥知会给我们的github账户：**请打开你的Github账户设置页面，然后找到并点开"SSH and GPG Key"选项页面，点击"New SSH key"按钮，将你的公钥（`id_rsa.pub`）字符串复制到其中即可。



- 步骤3. 测试配置是否成功：请在终端中输入：

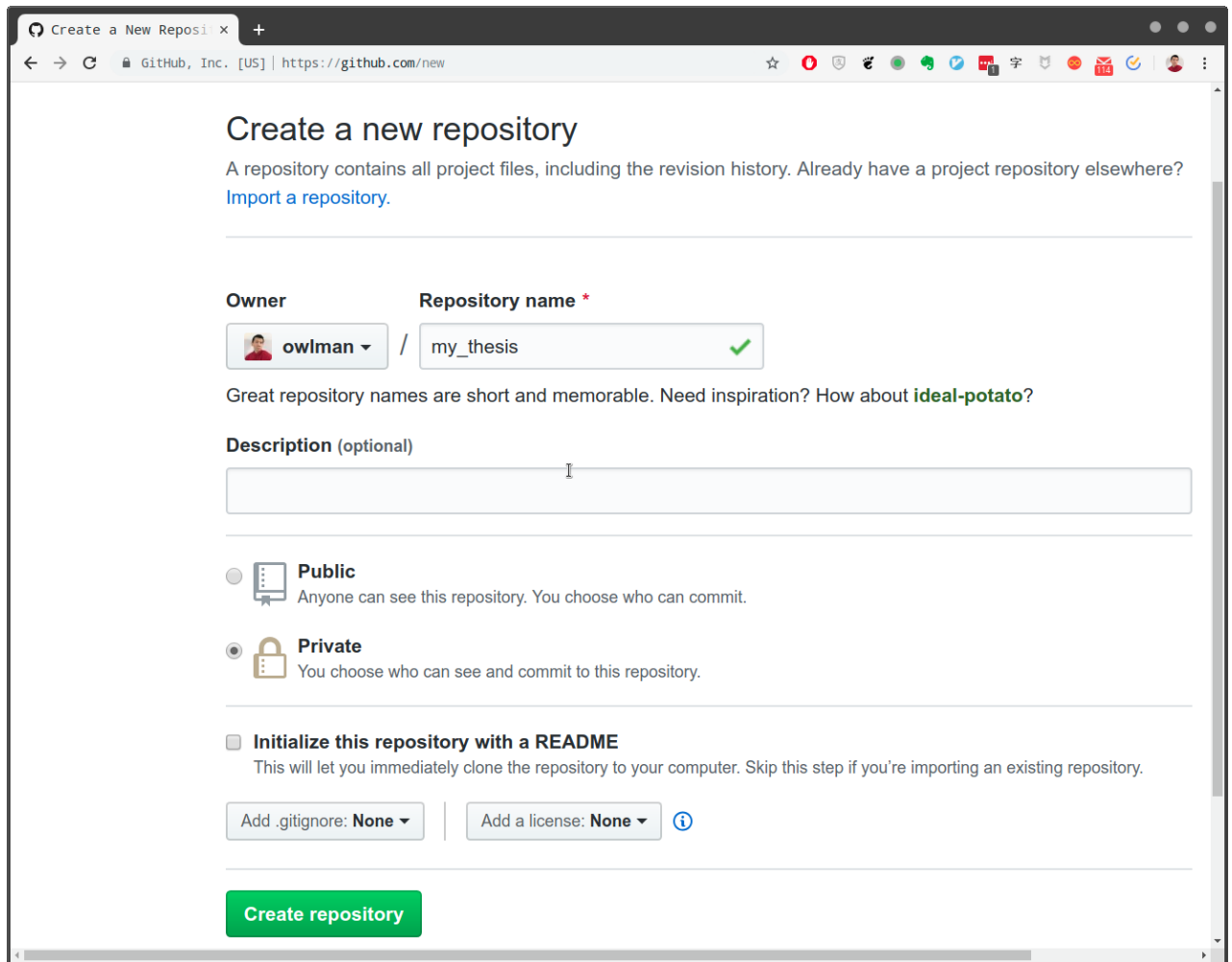
```
1 | ssh -T git@github.com
```

如果终端输出类似下面这样的信息，即表示我们的设置成功，可以执行后续操作了。

```
1 | Hi lingjie You've successfully authenticated, but GitHub does not provide shell access.
```

接下来，我们就来为当前的论文项目创建一个Github远程仓库，其具体步骤如下：

- 步骤1. 在登录到github.com之后，我们单击首页中的 **New** 按钮，打开远程仓库的创建页面。
- 步骤2. 填入相关信息之后，点击“Create Repository”按钮。



- 步骤3. 打开终端，进入项目目录，使用 `git remote add` 命令将一个名为 `origin` 的远程仓库添加到项目中，然后执行 `git remote show` 命令查看当前项目的远程仓库，并进一步使用 `git remote show origin` 命令查看新增仓库的具体信息：

```
owlman@owlman-Lenovo: ~/文档/text/毕业论文
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
owlman ~ > 文档 > text > 毕业论文 git remote add origin git@github.com:owlman/my_thesis.git
owlman ~ > 文档 > text > 毕业论文 git remote show                                ? master
origin
owlman ~ > 文档 > text > 毕业论文 git remote show origin                        ? master
* 远程 origin
  获取地址：git@github.com:owlman/my_thesis.git
  推送地址：git@github.com:owlman/my_thesis.git
  HEAD 分支：(未知)
owlman ~ > 文档 > text > 毕业论文 |                                           ? master
```

在这里，`origin` 为 git 默认远程仓库约定俗成的名称，在执行某些远程仓库操作（譬如 `git pull`）时，这个名称是可以省略不写的。除此之外，同一个项目也可以拥有多个远程仓库，我们也可以使用 `git remote -v` 命令列出这些远程仓库更详细的信息（注：`-v` 选项为 `--verbose` 的简写，取首字母），譬如下面是本书自身项目拥有的两个远程仓库。除此之外，我们还可以通过 `git remote show <远程仓库名>` 这个命令格式来查看某个远程仓库的详细信息：

```
owlman@owlman-Lenovo: ~/文档/text/markdown_guide
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
owlman ~ > 文档 > text > markdown_guide git remote -v master
gitee git@gitee.com:owlman/markdown_guide.git (fetch)
gitee git@gitee.com:owlman/markdown_guide.git (push)
origin git@github.com:owlman/markdown_guide.git (fetch)
origin git@github.com:owlman/markdown_guide.git (push)
owlman ~ > 文档 > text > markdown_guide git remote show origin master
* 远程 origin
  获取地址：git@github.com:owlman/markdown_guide.git
  推送地址：git@github.com:owlman/markdown_guide.git
  HEAD 分支：master
  远程分支：
    master 已跟踪
  为 'git pull' 配置的本地分支：
    master 与远程 master 合并
  为 'git push' 配置的本地引用：
    master 推送至 master (最新)
owlman ~ > 文档 > text > markdown_guide master
```

接下来，我们需要将论文项目的数据推送（push）到远程仓库上了。项目进行到一个阶段，我们既需要将项目备份在可靠的地方，也有让别人加入项目，进行审阅的网络协作需求。实现这个任务的命令很简单，基本格式为 `git push <远程仓库名> <分支名>`。譬如，如果我们想要将当前项目的 `master` 分支推送到 `origin` 远程仓库中，可以运行下面的命令：

```
owlman@owlman-Lenovo: ~/文档/text/毕业论文
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
owlman ~ > 文档 > text > 毕业论文 git push origin master master
对象计数中：24，完成。
Delta compression using up to 4 threads.
压缩对象中：100% (17/17)，完成。
写入对象中：100% (24/24)，244.90 KiB | 824.00 KiB/s，完成。
Total 24 (delta 3)，reused 0 (delta 0)
remote: Resolving deltas: 100% (3/3)，done。
To github.com:owlman/my_thesis.git
 * [new branch]      master -> master
owlman ~ > 文档 > text > 毕业论文 master
```

同理，如果我们想要将远程仓库中的数据取回，并将其合并到当前项目的 `master` 分支中，就只需要执行 `git pull origin master` 命令即可，这里就不再演示了。无论如何，我们在这里只是针对管理 Markdown 项目的需要，对 git 做了一点最基本的介绍。但 git 是一个非常强大的工具，如果你需要更好地使用这项工具，应该找一本 git 的专著来学习一番。<sup>1</sup>

## 本章小结

在本章，我们围绕着如何“像维护程序项目一样维护 Markdown 项目”的议题展开了一系列的讨论。首先，我们介绍了一款可以让人们更专注于文字内容审阅和修改的 Markdown 编辑器：Typora。这款编辑器所见即所得的特性，以及对 Markdown 扩展语法的强大支持会给我们的自我审阅，以及可直接使用 Markdown 文档来进行审阅的人们带来极大的便利。接下来，考虑到 Markdown 的应用不够普及的现实问题，为了让更多的人参与我们作品的审阅，我们为大家介绍了一款专用于转换标记语言格式的工具：Pandoc。通过这个格式转换器，我们可以非常轻松地将 Markdown 文档转换成人们普遍习惯的 PDF 和 Microsoft Word 文档。最后，为了从时间维度上对项目的修改进行管理，我们也对如何用 git 版本控制系统对 Markdown 项目进行管理和维护，做了一个基本介绍。



在下一章中，我们将会介绍几个 Markdown 在特定领域中的应用，让读者进一步体验 Markdown 的开放、便捷与强大。我真心地期待有一天，Markdown 能成为像儿时的铅笔和橡皮一样文案工具，简单即强大。

---

1. 注释：趁机打个广告，笔者曾经参与翻译过一本《git学习指南》，已由人民邮电出版社出版，如有需要可找来一读。[👉](#)