

千葉工業大学 社会システム科学部

プロジェクトマネジメント学科

平成25年度 卒業論文

OSSプロジェクトにおけるプロジェクト

マネジメントの実態調査

The project in the open source software project

Survey of Management

プロジェクトマネジメントコース

矢吹研究室

1042067 関口元基／Genki SEKIGUCHI

|  |  |
| --- | --- |
| 指導教員印 | 学科受付印 |
|  |  |

目次

[第1章 緒論 2](#_Toc369272159)

[1.1研究背景 2](#_Toc369272160)

[1.2研究目的 3](#_Toc369272161)

[1.3研究方法 3](#_Toc369272162)

[1.4プロジェクトマネジメントとの関連 3](#_Toc369272163)

[第2章 オープンソーススフトウェアの概要 6](#_Toc369272164)

[2.1 オープンソースとは何か 6](#_Toc369272165)

[2.2 オープンソーススフトウェアを定義する 7](#_Toc369272166)

[2.3 オープンソーススフトウェア誕生の背景 10](#_Toc369272167)

[2.4 ソフトウェア産業におけるオープンソース 11](#_Toc369272168)

[第3章 オープンソーススフトウェアプロジェクトの現状 14](#_Toc369272169)

[3.1オープンソースソフトウェアの種類 14](#_Toc369272170)

[3.2商用ソフトウェアとの比較 15](#_Toc369272171)

[3.3 オープンソースを活用したプロジェクトのメリット 17](#_Toc369272172)

[3.3.1開発者側のメリット，デメリット 17](#_Toc369272173)

[3.3.2 ユーザ側のメリット，デメリット 19](#_Toc369272174)

[3.4 ノマドワーキングとは 20](#_Toc369272175)

[3.4.1 ノマドワーカーの実態 21](#_Toc369272176)

[3.4.2ノマドワーキングのメリット，デメリット 21](#_Toc369272177)

[第4章 ホスティングサイトの台頭 26](#_Toc369272178)

[4.1 ホスティングサイトとは 26](#_Toc369272179)

[4.2 匿名性とプロジェクト参加 26](#_Toc369272180)

[4.3 ツールが一通り揃ったホスティングサイト 26](#_Toc369272181)

[4.4 ホスティングサイトを選ぶ 28](#_Toc369272182)

[4.5 Githubとは何か 30](#_Toc369272183)

[4.5.1バージョン管理システム「Git（ギット）」 32](#_Toc369272184)

[4.5.2 履歴を管理する「リポジトリ」 32](#_Toc369272185)

[4.5.3 変更を記録する「コミット」 33](#_Toc369272186)

[4.5.4 リポジトリに変更を記録する「ワークツリー」と「インデックス」 33](#_Toc369272187)

[4.5.5 リポジトリの共有「プッシュ」と「クローン」と「プル」 34](#_Toc369272188)

[4.5.6 変更履歴の統合「変更履歴のマージ」 35](#_Toc369272189)

[4.5.7 変更履歴の統合「競合の解決」 35](#_Toc369272190)

[第5章 オープンソーススフトウェアプロジェクトのマネジメント実態調査 37](#_Toc369272191)

[5.1 調査方法 37](#_Toc369272192)

[5.2「Ruby on Rails3アプリ開発プロジェクト」 37](#_Toc369272193)

[5.2.1Rubyとは 37](#_Toc369272194)

[5.2.2 Ruby on Railsとは 38](#_Toc369272195)

[5.3「Ruby on Rails3アプリ開発プロジェクト」の概要 38](#_Toc369272196)

[5.3.1「Gitlab」 38](#_Toc369272197)

[5.3.2「WhiteHall」 39](#_Toc369272198)

[5.3.3「Spree」 39](#_Toc369272199)

[5.3.4「Shapado」 39](#_Toc369272200)

[5.4「cms開発プロジェクト」 40](#_Toc369272201)

[5.4.1 CMSとは 40](#_Toc369272202)

[5.5「cms開発プロジェクト」の概要 40](#_Toc369272203)

[5.5.1「RefineryCMS」 40](#_Toc369272204)

[第6章 結論 42](#_Toc369272205)

[6.1実証分析結果 42](#_Toc369272206)

[6.1.1 総合統計結果 42](#_Toc369272207)

[6.2プロジェクトマネジメントありかた． 42](#_Toc369272208)

[6.3オープンソーススフトウェアプロジェクトの抱える課題と今後のありかた 42](#_Toc369272209)

[おわりに 42](#_Toc369272210)

**第1章**

**緒論**

# 緒論

## 1.1研究背景

オープンソースソフトウェア（以下OSS）を利用したプロジェクトが増えてきている.[1]OSSとはソフトウェアの設計図にあたるソースコードを，インターネットなどを通じて無償で公開し，誰でもそのソフトウェアの改良，再配布が行えるようにすることでありそのようなソフトウェアの名称である．オープンソースのソフトウェア開発のためには，バージョン管理システムやWiki，バグ追跡システム，メーリングシステムがよく利用される．これらのツールをまとめて提供するのがホスティングサイトであり，その代表的なものにGitHubがある．GitHubとはプログラマのためのソーシャルネットワーキングサイトともいい，今まで主流であったリポジトリホスティングとは大きく異なり，プロダクトの世界だけに情報が閉じていたが，開発者が公開しているソースコードはすべて閲覧できるようになっており，どのリポジトリに誰が興味を持っているのを把握できるようになった．これにより，人に目を向けられるようになったことが大きな特徴である．現在，OSSプロジェクトの開発者は毎日コードを書いたりレビュ－をしたり，バグレポートを書きバグをについて議論したり，ドキュメントやwikiを更新するなど，何百ものプロジェクトに貢献している．しかし，こうした活動を通じて生まれたデータをアーカイブ化して分析することは難しい．私たちの身近な問題としてプロジェクトを行う上で共有サーバ上に保存したデータがほかのメンバによって上書きされてしまいデータが消えてしまうことがある．また，データファイルの更新時，ファイル名が同じであるファイルをメンバが操作してしまうと，更新日がばらつきどのファイルが最新なのかを把握できなくなってしまうリスクが発生してしまう．そのようなときに便利なものがGitHubである．GitHubの登場により，ソフトウェア開発の内情を調べることが容易になっている．たとえば，ソースコードの変更履歴を調べれば，その開発状況を調べればその開発過程を知ることが可能であるし，バグ追跡システムの履歴を調べれば，開発者同士のコミュニケーションの様子もしることができる．これによりソフトウェア開発が盛んになり，OSSプロジェクトでの利用頻度は増えてきている.[2][3]そのような中で，私が体験してきたプロジェクトの経験を生かしOSSプロジェクトではどのようなプロジェクトマネジメントの手法が使われているのかを調査・研究し，管理体制の向上やマネジメント手法を提案することで更なる効果を期待できるのではないかと考えた．

1.2研究目的

GitHubなどのネット上で公開されているOSSを調査し，ソフトウェア開発の実態を明らかにする．GitHubからは，成果物(ソースコード)以外にも，ソフトウェア開発にかかわるさまざまな情報を引き出すことができるため，それらを収集して解析することによって，OSS開発における，プロジェクトマネジメントの実態や課題を明らかになることが期待できる．

1.3研究方法

研究方法は以下の通りである．

1. OSSのプロジェクトマネジメントの実態把握のための調査項目の決定をする
2. データの収集方法を検討
3. データを収集し，解析する．

具体的には， GitHubに重点を置き，GitHub内で実際に行われているプロジェクトを分析する．

たとえば，何人規模のメンバで機能しているのか，リリース（バージョン１への到達）までにかかった時間，ソフトウェアの規模などのデータを収集し，それを統計あるいはデータマイニング的な手法で解析することによって，OSSプロジェクトではどのようなマネジメントの手法が行われているかを研究する．

## 1.4プロジェクトマネジメントとの関連

OSSの開発が，どのようにマネジメントされているかを調査する研究である．

参考文献

[1] 経済産業省. OSSの現状と今後の課題について. 2011. http://www.meti.go.jp/kohosys/press/0004397/1/030815opensoft.pdf.

[2] GitHub Social Coding. https://github.com/,2013-09-11.

[3] 大塚 記. WEB+DBPRESS. 初版,2012, vol.69, p.18-23.

**第2章**

**オープンソーススフトウェアの概要**

# 第2章 オープンソーススフトウェアの概要

## 2.1 オープンソースとは何か

オープンソースソフトウェア，と聞いてその特徴，性質を正しく捉え，他人に対して詳細な説明をすることができる人はそう多くないだろう．だが，それは無理もない．このオープンソースソフトウェアという言葉がこの世に誕生したのはわずか数年前，1998年のことである．しかし，名前の響きこそ新しいものだが，オープンソースソフトウェアの持つ性質，概念といったものは遥か昔，それこそコンピュータ創世記から脈々と流れてきているものである．字面を見れば「ソースがオープンになっているソフトウェア」であるということは漠然と理解できるだろうが，その先の段階に理解が進む人は稀であろう．では，そもそもソースとは何か．これはプログラミング言語と呼ばれる高水準の言葉で書かれている．これはあらゆるソフトウェアを作るにあたり，その設計図とでも言うべきものである．ソフトウェアを作るにはそのプログラムを作らなければならない．しかし，人間とコンピュータでは理解できる言語が異なり，例えば英語や日本語でプログラムを入力してもそれをコンピュータに理解させることは不可能である．では，どうすればいいのかというと，ここでプログラミング言語が使われる．プログラミング言語とは人間の理解できる言語とコンピュータが理解できる言語の中間に位置する言葉であり，主なものとして C 言語や JavaScript といったものがある．プログラマはこのプログラミング言語を使いプログラムを作成する．この段階ででき上がる物がソースコードと呼ばれるものである．このソースコードはプログラミング言語を読むことができない人には何が書いてあるのか分からないだろうが，その言語を学んだ人であれば手間と時間をかければ解読することができる．つまり，このソースコードは一見コンピュータに読ませるための言語に見えるが，まだ人間向けの言語であり，コンピュータがそれを理解することはできない．そこで，このソースコードをコンパイルすることでやっとプログラムとして完成するのである．

コンパイルとは翻訳することを指し，人間向けの言語で書かれたソースコードをコンピュータが理解できる言語に変換することである．このコンパイルを経てソースコードはバイナリコード呼ばれるものに変化する．バイナリコードとは実行プログラムのことである．ソフトウェアを購入するなりダウンロードするなりして手に入れた場合，大概のソフトにはセットアッププログラムがあり，それを起動することでソフトウェアはコンピュータにインストールされる．この際に使われるセットアッププログラム，これがバイナリコードである．このバイナリコードは通常，「0」と「1」の羅列，つまり２進法により記述されており，この段階になりやっとコンピュータに理解できる状態になる．これらを踏まえた上でオープンソースソフトウェアの説明をすると，オープンソースソフトウェアとは「設計図たるソースコードが公開されたソフトウェア」ということになる．ここで注目する点はソースコードを公開する，という点である．前述のとおり，ソースコードとはソフトウェアの設計図にあたる．このソースコードを手に入れた人は技術さえあればそのコードを改変することで元の機能を持った類似製品を低コスト，短

時間で作成したり，またはそれを元にしたまったく新しいソフトウェアを作成したりすることができる．つまり，このソースコードは開発者側から見れば本来死守すべきものなのである．では，なぜこのソースコードを公開するのだろうか．詳しい説明は第4章で述べるためここでは避けるが，ソースコードを公開することには開発者側，ユーザ側双方にメリットがある．ユーザ側のメリットをあげるとおおよそ三つのメリットがある．プログラム保守，プログラム移植，機能追加の三点である．第一にプログラム保守とは，バグ等を発見した際，開発者の修正を待たずに自身の手でその修正をすることができる．第二にプログラム移植とは同じソフトウェアを他の OS 環境にあるコンピュータに移植する場合，例えば Windows環境から Mac環境へ移植する場合に通常ならそれぞれの機種に対応しているソフトウェアを別途購入しなければいけないが，ソースコードがあれば必要な部分のみ訂正すれば新たに購入する必要はない．第三に機能追加はユーザによる自由なカスタマイズを可能にするためである．

## 2.2 オープンソーススフトウェアを定義する

オープンソースという言葉を最も広義に使う場合，これはソースコードの公開の有無のみを指し，それ以外の条件付けはされていない．逆に，最も狭義の意味で使われる場合にはオープンソースの定義が正しく守られるように管理しているオープンソースイニシアティブ（Open Source Initiative 以下，文中ではOSIと表記）[1]による定義（Open Source Definition以下，文中では OSD と表記）[1]による「OSI認定マーク」の付いたソフトウェア，または OSI認定のライセンスを使用しているソフトウェアのことを言う．また，これ以外にも OSIの認定は受けていないがこれら「オープンソースの定義」に準拠しているソフトウェアもある．このように，様々な解釈のされているオープンソースだが，本論文中で「オープンソース」，または「オープンソースソフトウェア」という言葉を使った際にはこの最も狭義の意味で使うものとする．では，OSI による OSD とはどのようなものなのか．これは現在では全十か条の定義となっており，

1. Free Redistribution（自由な再頒布）

2. Source Code（ソースコードの開示）

3. Derived Works（派生著作物に関するライセンス）

4. Integrity of The Author's Source Code（著作者のソースコードの完全性）

5. No Discrimination Against persons or Groups（個人やグループに対する差別禁

止）

6. No Discrimination Against Fields of Endeavor（利用分野に対する差別禁止）

7. Distribution of License（ライセンスの継承）

8. License Must Not Be Specific to a Product（特定製品に特化したライセンスの

禁止）

9. License Must Not Restrict Other Software（他のソフトウェアを制限するライセンスの禁止）

10. License Must Be Technology-Neutral（テクノロジーニュートラルなライセンス）

となっている．以下では上記の各項目について説明をしていく．

1. Free Redistribution（自由な再頒布）

この項目では完全に自由な再頒布を許諾することで，オープンソースソフトウェアの普及を目指している．そのため，販売することも許諾しなければならないが，手元を離れたオープンソースソフトウェアに対して更なる使用料や報酬を求めてはならない．

2. Source Code（ソースコードの開示）

OSIライセンスではすべての製品に対して適切な価格での提供を定めている．これに加え，オープンソースソフトウェアを頒布する際には必ずソースコードを含まなければならないということと，ソースコードでの頒布の許可を認めることが必要である．また，この他にもソースコードを意図的にわかりにくくすることは許されていない．これはオープンソースソフトウェアの普及という考えが背景にある．改変を困難にすることでこの普及が妨げられることを防ぐためである．

3. Derived Works（派生著作物に関するライセンス）

OSIライセンスでは以下の項目を遵守することを前提にソフトウェアの改変と派生ソフトウェアの再頒布を認めている．それはソフトウェアに対する自由な改変の許諾，及び派生ソフトウェアに対してもとのソフトウェアと同条件での頒布を許諾しなければならない，というものである．これはソースコードの公開のみではオープンソースとは言えず，自由改変，自由再頒布を認めなければならない，という前提があるためである．

4. Integrity of The Author's Source Code（著作者のソースコードの完全性）

OSIライセンスでは著作者はオリジナルのソースコードと派生版との見分けをつけられるように，「ソースコード+パッチファイル」という形で頒布を認める場合はオリジナルのソースコードと派生版との相違点を明確にし，製作者の名前を記述し， 元のソフトウェアが何かということを分からせなければならない，というものである．これはオリジナルと同名のソフトウェアが出回ると区別がつきにくくなってしまうからである．これにより改悪版などが作られた場合，著作者の評判を落とすことを防ぐことができる．

5. No Discrimination Against persons or Groups（個人やグループに対する差別禁止）

国家によっては外国為替法によりソフトウェアの輸出制限を行っている場合があるが，OSIライセンスではそのような制限をかけることは許されない．これはオープンソースの進化のためには多くの人が触れる必要があり，それを締め出すことは禁止されている.

6. No Discrimination Against Fields of Endeavor（利用分野に対する差別禁止）

オープンソースコミュニティでは学術・商業利用のためのユーザの参加も奨励している．このため，ライセンスで使用分野に制限をかけることを禁止している．

7. Distribution of License（ライセンスの継承）

OSIライセンスによるソフトウェアに付随する権利は再頒布された人すべてに対して適用されなければならない．

8. License Must Not Be Specific to a Product（特定製品に特化したライセンスの禁止）

ここでは特定の頒布物に対して特化した特長をつけることを禁止している．これは Linuxベースのマシンではオープンソースになるが，その他の OSマシンでは使えない，ということを防ぐためである．これは，オープンソースはあらゆる環境においてオープンソースでなければならないからである．

9. License Must Not Restrict Other Software（他のソフトウェアを制限するライセンスの禁止）

これはオープンソースとその他ソフトウェアを同時に利用した場合，その他ソフトウェアの機能を制限するようなことがあってはならない，ということである．

10. License Must Be Technology-Neutral（テクノロジーニュートラルなライセンス）

OSIライセンスではいかなる項目もその技術に対して制限を設けることはできない．これは FTPダウンロードや CD-ROM での頒布，ミラーサイトからの頒布といったクリック・ラッピングとの相性の悪い頒布方法あり，またこのライセンス方法を共用することで再頒布を足止めしてしまう結果を生む危惧が発生してしまう．そのため，OSIに準拠するライセンスは特定の技術に固執することなく，技術的に中立なライセンスを保持する必要があるのである．

さて，ここまで OSD の十項目のライセンスを提示してきたが，ここで注目すべき点が一つある．これは，Linux カーネルなどを筆頭に一般に思われている「オープンソース」像を崩すものである．それは，この OSD は「頒布に際してそのソフトウェアは無料でなければならない」という項目を持たない，という点である．オープンソースソフトウェアの定義の中には有償か無償かという考えは含まれてはいないのである．事実，Linux カーネルをはじめとした多くのオープンソースソフトウェアは無償で頒布されている．しかし，これは「無償で頒布しなければならないもの」だからではなく，頒布元や再頒布元の判断で無償での頒布を行う，と決めているためである．つまり，逆に言えば頒布元，再頒布元が「有償での頒布を行う」と判断を下せば有償頒布を行うことができ，また，今現在では無償頒布されているソフトウェアが将来的に有償になる可能性も常に秘めているのである．また，この Linux カーネルにしてもそのベースとなるオリジナルのものは無償頒布されているが，Red Hat社をはじめとする多くの企業がディストリビュータとして有償による頒布を行っている．これらの企業から頒布を受ける場合には Windows OSとさして変わらない料金を提示されることになる．これは，そのディストリビュータはあくまでもビジネスとして頒布を行い，本来は自らで行う必要があるバージョンアップや修正パッチの確認を企業側が行い，それに付随して各種ソフトウェアとの組み合わせ販売や独自のサポートを行うことで付加価値をつけ，有償での頒布を行える道を作ったのである．

以上のことから，ユーザはオープンソースソフトウェアを無償で手に入れる確証を得ているわけではない，ということになるのが分かるだろう．

## 2.3 オープンソーススフトウェア誕生の背景

独特の性質をもつ OSS だが，その成り立ちはどのようなものであったのだろうか．OSS の誕生の背景を語るうえで欠かすことができないのが，フリーソフトウェアの存在である．コンピュータがごく一握りの人々にしか使われていなかった 1970 年代までは，ソフトウェアの権利は至極曖昧なものであった．当時は，ソフトウェアは共有し，相互利用されるのが一般的で，特に，研究者の間では相互利用が盛んに行われていた．しかし 1980 年代に入ると，ソフトウェアを専有化し，そこから収益を得ようという流れが誕生する．そうした商業ソフトウェアの登場により，ソフトウェアの所有権は明確なものとなり，ソースコードも秘匿され，自由な配布や改変を行うことは不可能なものとなった．そうした状況で誕生したのが，先に挙げたフリーソフトウェアというものだ．フリーソフトウェアは，ソフトウェアの専有化の潮流とは逆に，コンピュータ黎明期のような相互利用を促すためにつくられた概念である．フリーソフトウェアは，ソースコードを公開し，自由に配布や改変を行うことができる．さらに，フリーソフトウェアを改変して生まれたソフトウェアも，フリーソフトウェアでなければならない，というコピーレフトの性質をもっている．このフリーソフトウェアの概念をつくったのが，フリーソフトウェア財団の代表であるリチャード・ストールマンである．ストールマンは The Open Source Definition を提唱しているオープンソースイニシアティブの創設者でもある．フリーソフトウェアは派生ソフトウェアも必ずフリーソフトウェアとしなければならないという，厳密なコピーレフトの性質を持ち，当時の商業ソフトウェアのビジネスモデルとの間に大きな軋轢を生んでいた．そこでストールマンは，フリーソフトウェアの定義をゆるめ，権限の移譲の方式に多様性を認めた．そうして新たにうまれた概念がオープンソースソフトウェアである．オープンソースソフトウェアは，2.2章のとおり，著作者の考え方に合ったライセンスを選ぶことで，コピーレフトをどこまで厳密に採用するかを決定することができる．このように，OSS は，商業ソフトウェアとフリーソフトウェアとの折り合いをつけるために生まれたのだ．

## 2.4 ソフトウェア産業におけるオープンソース

これまでは OSS の定義と歴史を洗い，その概要に触れてきた．では，その実態はどのようなものであるのだろうか．2.3 章においても触れたように，OSS はそもそも個人が作成し，公開していたフリーソフトウェアを起源としているから，一般的に企業との関係性は薄かった．しかしながら，OSS の発展が進むにつれ，OSS を自らのビジネスに取り入れようとする企業が登場する．その先駆けとなったのが Red Hatである．Red Hat は Linux ディストリビューションである Red Hat Linux の提供を行なった．Linux ディストリビューションとは，オープンソースで開発されている OSである Linux を，一般的な利用者が導入・利用しやすいように様々なソフトウェアを組み合わせ，パッケージングしたもののことを言う．Linux ディストリビューションには様々なものがあるが，Red Hat Linux は，ネットワークサーバやワークステーションといった，主に企業を対象とした構成になっている．Red Hat Linux の最も特徴的なところは，そのビジネスモデルにある．構成するソフトウェアのほとんどが OSSである Red Hat Linux は，基本的に無料での頒布が行われた．その一方で，サポートを有料にし，そこから収益を生み出していた．90 年代の後半では，こうした，OSSの関連商品・サービスを販売して収益を得るというビジネスモデルが多く存在した．その一方で，2000 年代中頃に入ると，オープンソースの開発形態を自社製品に取り込むことで，イノベーションコストを削減することを目的とする企業が登場した．その最も顕著な例が Google である．Google は検索事業からの広告収入を主な収益源としているが，その主幹事業を間接的に支えるために，インターネットブラウザであるGoogle Chrome やモバイルオペレーションシステムである Android といったソフトウェアの開発・提供を行っている．ブラウザや OS といった汎用的なソフトウェアは，多様なユーザニーズに逐一応えていると非常にコストがかさんでしまう．そこでGoogle は，この 2 つのソフトウェアの一部をオープンソースとすることで，ユーザ自身が追加機能の開発や，カスタマイズを行えるようにし，多様なユーザニーズに対応した．

参考文献

[1] オープンソースイニシアティ. 2010. http://www.opensource.org/

[2] オープンソースの定義. 2012. http://www.opensource.org/docs/definition.php

**第3章**

**オープンソーススフトウェアプロジェクトの現状**

# 第3章 オープンソーススフトウェアプロジェクトの現状

## 3.1オープンソースソフトウェアの種類

現在，オープンソースソフトウェアの数は増加の一途にある．しかし，オープンソースソフトウェアの開発者は自らの興味や目的によって開発が行われるため，ソフトウェアの種類や分野には多くの偏りがある．そのため，ユーザは自らの判断でニーズにあったソフトウェアを見つけなければならない．この節ではこの数あるソフトウェアの中からメジャーなソフトウェアをいくつか紹介する．

1. Linux

Linuxは 1991 年，フィンランドの学生であった Linus Torvaldsを中心に開発されたオープンソースの UNIX 系カーネルである．この Linux はインターネットサーバ用 OSとして大きなシェアを占めており，頒布条件は GNU GPLによって公開されている．Linuxの特徴はバザール方式を始めて採用して開発されたオープンソースソフトウェアである，ということである．Linuxはこの開発方式によって多くのハッカを集めることができ，またそのハッカから提供されたドライバソフト（周辺機器を利用できるようにするのに必要なソフトウェア）を組み込むことができたため，急速に普及していったのである．

2. FreeBSD

FreeBSD はオープンソース UNIX 系 OSとして 1993 年に発表された．カリフォルニア大学（UC）の Berkeley 校で開発された長い歴史を持つ UNIX の BSD（Berkeley software distribution）版を Intel社のマイクロプロセッサ搭載パソコンで動作するように開発された OSである．日本では当初，IBM互換機でしか動作しない Linux よりも BSD 系 OSのほうが普及していた．この BSDはソフトウェアの商用利用に関する制限が最も少ないライセンスの一つであり，カスタムソフトウェアを商用ソフトウェアとして販売することも可能である．また，この FreeBSDは Apple社の最新 OSである Mac OS Xのベースになったことも知られている．

3. Apache

Apacheは現在世界で最も高いシェアを持っている Webサーバである．このオリジナルは NCSA（National Center for Supercomputing Applications）[1]が開発したフリーソフトウェアである．1995 年にこの主要な開発スタッフが Netscape 社に移籍後，オリジナル版のユーザであったハッカが開発を続け，1999 年に Apache Software Foundation [2]を設立．オープンソースソフトウェアの成功例として現在に至っている．

現在，オープンソースソフトウェアの数は増加の一途にある．しかし，オープンソーンソースコミュニティの賛同を得ることができず，開発は大きく遅れることとなった1998年のMozillaプロジェクトの発表から4年がたった2002年になってMozilla 1.0がリリースされた．

## 3.2商用ソフトウェアとの比較

オープンソースソフトウェアと商用ソフトウェアという選択肢がユーザに用意されている．これらの違いは開発工程やソースコードの公開の有無だけではない．使用に当たっての責任の所在や導入に当たってのコストなど，双方の間には大きな違いがある．ここではその相違点を挙げる．

1. 無保証の提供条件

オープンソースソフトウェアは基本的に無保証である．著作権者のみならず，頒布を行っている関係者も含め，ソフトウェアの利用に関して一切の保証を付けずに提供している．

2. 環境への適合性への保証

オープンソースソフトウェアは開発者側で一定の環境の元，動作チェックは行っているが，それはユーザに対して保証するものではない．

3. 品質や性能に関する責任

開発者はオープンソースソフトウェアの品質や性能に関する責任を負わない．ソフトウェアに欠陥があったとしても著作権者は修正の義務を負わず，ユーザが自ら修正や訂正を行うことが求められている．また，そのためにかかるコストはすべてユーザの負担である．

4. 使用結果の責任

ソフトウェアを利用した結果，ユーザに何らかの損害があったとしてもそのことに対する保証は一切行われない．この場合の損害とはデータの損失や他プログラムとの相性問題などを指す．これに対し，商用ソフトウェアはライセンシーをとっての販売を行う代わりにマニュアルに沿った行動に対しての品質保証を行っている．その保証内容は以下のとおりである．

1. マニュアルに記載のある内容に従って動作することを保証

2. マニュアルに記載のある内容のサポートを行うことの保証

3. ソフトウェアが保証規定を満たさない場合は製品の交換，保証，代金の交換のいずれかを行う これらの内容は記載のある事柄に関しての保証を行うことを確約するものではあるが，それは裏を返せばマニュアルに記載のないことに関しては一切の保証を行うものではない，ということを意味している．これは商用ソフトウェアに関する問題のひとつとなっている．また，そのソフトウェアを利用して事業を行った結果いかなる損失を企業が負ったとしてもその補償は一切ない．しかし，例外条件として法律にのっとり損失の一部を補填するシステムは存在する．オープンソースソフトウェアと所要ソフトウェアを比較する場合，真っ先に出てくるのはコストの問題である．オープンソースソフトウェアには無償でなければならないという規定はないが，多くのソフトウェアは無償で行われている．例えば雑誌の付録やボランティアのディストリビュータによるダウンロード頒布などである．これに比べ，商用ソフトウェアは有償での販売を行っている．

しかし，TCO（Total Cost of Ownership 購入から保守・運用までにかかる総コスト）から見た場合，オープンソースソフトウェアは本当に割安なのだろうか．オープンソースソフトウェアを使用する上で発生するコストにはどのようなものがあるのだろうか．

ここではサーバシステムの構築を例に挙げる

1. システム構築にかかるコスト

サーバを構築するに当たってはアクセス権の限定や不要ソフトウェアのインストールを行わないなどの注意が必要であり，これらを行うためには初期段階での厳密なシステムの設計が必要となる．ここでそのためのコストがかかってくるのである．

2. インストールコスト

Linuxは Windowsに比べてメーカによる動作確認情報が不足している．この ため，Linux をインストールする場合，使用するパソコンにインストールできるのか，また何かしらの設定を行う必要があるのか，といったことを自ら調べないとならないのである．さらにいえば Linux関係の技術者は雇用コストが一般の技術者に比べ高くつくことが多い．

3. バージョンアップ時のコスト

オープンソースソフトウェアの基本は「早めに公開，頻繁に公開」にある．そのため，ユーザにも頻繁なバージョンアップが要求される．これはつまり，その度に技術者に対する人件費がかかる，ということになる．また，バージョンアップ後にしっかりとソフトウェアが起動するかを確かめる動作確認にもコストがかかる．また，これらの費用以外にも外部サポートを採用している場合にはそのための費用もかかり，さらに言えば企業では社員が Linux環境になれていない場合には社員教育のための費用もかかることになる．では，これらの点を踏まえた上でユーザはオープンソースソフトウェアと商用ソフトウェアのどちらを選ぶのが有益なのだろうか．FSFの Richard M Stallman を始めとした「フリーソフトウェア」信者は「商用ソフトウェア＝悪」，「フリーソフトウェア＝正義」という信念を持っている．彼らはソースコードの公開を行わない商用ソフトウェアを販売している企業を「悪の権化」とし，OS環境をすべてフリーソフトウェアで統一しようとしている．彼らはあらゆる商用ソフトウェアの開発手段を批判している．しかし，この手の批判は穴が多く，的確なものではない．例えば商用ソフトウェアにはバグが多く，ソースコードが複雑である，という批判がある．しかし，まずバグに関しては現在のコンピュータ工学の技術ではバグを完全になくすことは不可能とされている．またソースコードが複雑である，という点に関しては商用ソフトウェアにはバックワードコンパチビリティ（前バージョンとの互換性を持たせること）の必要があるため，ソースコードが複雑にならざるを得ない．つまり，ユーザ本位の開発を行えば，必然的にソースコードは複雑になってしまうのである．このように「反商用ソフトウェア」派の意見は容易に反論ができる．また，一般的なユーザにとってはソースコード公開の有無はソフトウェアの有益性を語る上で重要視されることはなく，Windows 環境における動作保証がなされ，購入当初の契約でサポートがしっかりと約束されている商用ソフトウェアのほうが安心して利用できる，という面がある．このため，現実的な側面から見ればオープンソースソフトウェアやフリーソフトウェアよりも，商用ソフトウェアを選んだほうが有益であるだろう．

## 3.3 オープンソースを活用したプロジェクトのメリット

ほとんどの人がそれなりにはわかっているようですが，きちんと理解していないためにトラブルになるケースもあるようです．ここでは， オープンソースソフトウェア の メリット ， デメリット を説明し，導入するための参考にしていただきたいと思います．

### 3.3.1開発者側のメリット，デメリット

開発者にとってオープンソースのメリットはソフトウェアを自由にカスタマイズできることや，目的にあった環境に移植できることにある．これにより，システム構築やカスタマイズソフトウェアの開発などのビジネスも可能となる．このオープンソースを使ったソフトウェア開発を行うに当たって有効な点が二点ある．まず一点目がオープンソースソフトウェアを使ったソフトウェア開発が行えること，二点目に既存のソフトウェアを二次利用できる，という点である．既存のソフトウェアを利用したいと思った場合，そのソフトウェアが自分の持っている環境に合わなければ使うことは不可能であった．Macintosh と Windows の互換性の問題などがこれにあたる．Macintoshは Windows が普及する以前から GUIに適応した環境を提供していたため，DTP（Desk Top Publishing）やデザインなどの業界で多く利用されていた．こういったソフトウェアは多くが Macintosh用として開発されたため，Windowsでは利用できない，ということが数多くあった．逆に，ワープロや表計算ソフトといったオフィスソフトは Windows が主流であったため，Macintosh では利用できなかった．このため，コンピュータ技術者はこれらのソフトウェアを双方に利用できるものに作り変えたいと思っても，商用ソフトウェアは使用が認められているのはバイナリコードのみであり，ソースコードは非公開であったため，改変をすることは非常に困難であった．さらにもうひとつの大きな障害が著作権の問題である．1980 年代に各国でソフトウェアに著作権が認められたため，特定の条件以外では第三者は勝手に改変や再頒布を行うことはできなくなっていたのである．

しかし，これがオープンソースソフトウェアの場合，この問題は解決する．オープンソースソフトウェアはソースコードが公開され，最初から移植を含むあらゆる改変の自由が認められている．商用ソフトウェアでは開発会社以外にできない移植もOSSでは決して簡単な作業ではないが，技術と時間があれば誰でも自由に行うことができるのである．また，まったく新しいアプリケーションソフトウェアを開発したとして，そのソフトウェアを Macintoshないし Windows上で開発した場合，そのソフトウェアを提供したとしてもユーザはそのソフトウェアに対応した OS を購入する必要がある．しかし，元から Linux 上で開発をしていれば，そのソフトウェアと Linux をセットにして頒布することが可能であるため，ユーザに対して余計な費用負担を強いることはない．開発者，ユーザ双方にとって思い通りの環境が低コストで用意することが可能となる．このように，自由な改変やバグの修正が自らで行えるといった利点がコンピュータ技術者に受け入れられ，利用者が広がる中で商用ソフトウェアを抜きディファクトスタンダードとなったソフトウェアも少なくない．特に，Linux に代表されるサーバコンピュータ用のソフトウェアはオープンソースソフトウェア抜きには考えることができない規模にまで浸透している．特にシェアの大きなソフトウェアとしては Web サーバソフトウェアである「Apache」は全体の 66.04％（2002 年），電子メール配信ソフトウェアである「Sendmail」は全体の 42％（2001 年），DNS（Domain Name System）用ソフトウェアである「BIND」にいたっては全体の 95％（2000 年）と，他の追随を許さないほど普及している．また，Linux 自体も Windows の 49.6％に対して 29.6％（2002年）のシェアを持っている[1]．これらのソフトウェアは現在では Linux のディストリビューションによって Linuxのカーネルとともに頒布されているが，元をただせばこれらのソフトウェアは Linux 用に開発されたものではない．元々は UNIX 用ソフトウェアとして開発されたものがLinux用に移植され，Linux とともに普及をしていったのである．では，この Linux が普及していった敬意はどのようなものなのか．Linux は現在普及している OS の中では特に歴史の浅い OS である．この歴史の浅い OS が現在の地位に上り詰めた主要な要因は二つあげることが出来る．一つは多くのボランティアの協力がある．プロジェクトに協力したハッカは様々なドライバを開発し，Linux は多くの環境に適応可能な OS となった．この結果，Linux はFreeBSD などそれまで主流であった UNIX 系 OS を押しのけ，UNIX 系 OS のディファクトスタンダードとなった．二つはそれまでの GNUや UNIXの資産と相乗効果が挙げられる．これはすでに多くのユーザの支持を得ていた GCCや Apache，Sendmail やBIND といったソフトウェアを利用できたため，サーバ用 OS としての地位を得た．また，Linuxが普及したことによって付随する形でこれらのソフトウェアもディファクトスタンダードとして広がっていったのである．現在，FSF では Linux のことを「GNU／Linux」と呼んでいる．これは，Linux のカーネル部分と Linuxパッケージを区別するための名称であり，「GNU／Linux」とはツール群を含むパッケージ全体のことを指している．

次に既存ソフトウェアの二次利用であるが，代表的なものではやはり Linux やApache，またデータベースソフトウェアの MySQL や PostgreSQL などがある．これらのソフトウェアは商用ソフトウェアと比べて決して見劣りするものではなく，また市場シェアからみても大きなシェアを持っている．このような安定性の高いソフトウェアを使えるシステムでは初期コストを大きく削減することができるというメリットがある．また，特定の目的に適合したカスタムソフトウェアを開発する際にも同様の機能を持ったオープンソースソフトウェアを利用することによって開発期間，及びコストを大幅に削減できるのである．この開発期間の削減とはシステム設計，デバッグ，そしてメンテナンスの期間を短縮することが出来るのである．しかし，開発者が気を付けなければならない点としてオープンソースソフトウェアには動作保証がない，ということである．商用ソフトウェアであれば当然指定された環境であれば動作保証があるが，オープンソースソフトウェアにはこういった保証は一切ない．オープンソースソフトウェアはユーザが自由に利用できる代わりに一切の動作保証はなく，ソフトウェアのメンテナンスはすべてユーザが負うことになる．また，中にはいかなる環境においても動作保証自体をまったくしていない，というものもある．このため，開発者は完成したソフトウェアに何らかの問題が起きた場合はその責任すべて開発者に覆いかぶさるのである．

### 3.3.2 ユーザ側のメリット，デメリット

オープンソースソフトウェアに対して一般ユーザが感じるメリットは開発者が感じるメリットと大きくかけ離れていることが多い．これは基本的に一般ユーザはソフトウェアを元のまま利用し自ら手を加える，といったことを行わないからである．彼らがオープンソースソフトウェアを利用する理由はソースコードが公開されているからでもなく，また改変の自由が認められているからでもない．商用ソフトウェアに匹敵する高性能ソフトウェアを無償で利用できるからである．これまでにも述べたようにオープンソースソフトウェアは必ずしも無償で頒布されているわけではないため，「オープンソース＝無償のソフトウェア」という考えは誤りである．しかし，多くのユーザから見ればオープンソースソフトウェアは無償頒布されていて当たり前，と認識されているのである．確かに入手からセッティングまでの肯定を考えればそれまでにかかるコストはディストリビュータに支払うソフトウェアの代金のみであり，企業単位での使用の際もひとつ分の料金を支払えば後は複製の自由があるため商用ソフトウェアに比べはるかに安価ですむ．しかし，後々の保守・運営のことまで考えるとそのコストはユーザの利用目的や技術者の技術によって異なる．ユーザによってはオープンソースソフトウェアを利用することによって余計な手間とコスト負担が増える可能性があるのである．このため，オープンソースソフトウェアを利用するに当たっては初期導入コストのみではなく，その後のサポートコストを含む調査の必要がある．では，ユーザがオープンソースソフトウェアを利用する際にある無償以外のメリットは何があるだろうか．それでは以下にメリット，そしてデメリットを挙げる[2]．

・メリット

1. コストを節約できる．特にデータベースでは大きく節約できる

2. 安定性がある（メジャソフトウェアに限る）

3. セキュリティ・パッチ対応が早い．ソースコードが公開されているため，いざというときは自分で対応できる

4. 機能の追加を自由に行える

・デメリット

1. 自分でサポートを行う場合，技術が伴わない場合がある

2. マイナソフトウェアの場合，開発者がサポートを行わなくなると自分で修正を行わなければならなくなる

3. ディストリビュータが販売している製品まで無償と誤解されることがある

4. Linuxなどに詳しいエンジニアが必要

これを見ると，メリットとしてはコストの削減，安定性の確保，ソースコードの公開と多方面から見たメリットが挙げられる．しかし，逆にデメリットを見ると3.のオープンソースに対する誤解を除けば残りはサポートに関するものばかりである．このことから，目先のコスト削減やソースコードの公開といった自由度の高さを理由に採用し，安易に採用した結果サポート作業に苦心する，といったことが多いようである．また，個人ユーザに関していえば一般的にソースコードを使用することはない．彼らがオープンソースソフトウェアを使用する最大の理由はひとえに「無償」という点にあり，自らソフトウェアを改変し，カスタムソフトウェアを作る，といった高い知識を必要とすることを行うユーザは極めてまれである．そんななか，個人ユーザがソースコードを利用せざるを得ない機会がひとつだけある．それは，ソフトウェアのアップデートの際である．ソフトウェアをアップデートするには通常，パッチとして頒布されるバイナリコードを利用する．しかし，場合によってはこのバイナリコードの使用ができないことがある．Linuxを例にすると，GNU／Linuxではオープンソースプロジェクトが推奨しているディレクトリ構造とディストリビューションのディレクトリ構造が異なる場合がある．このような場合，バイナリコードを使ったアップデートは行えず，ユーザがソースコードを入手し，自らそのソースコードをコンパイルしなければならないのである．しかし，通常は頒布元のディストリビュータが元のソースコードをコンパイルしたプログラムを提供しているので，やはり個人ユーザがソースコードを触る機会は滅多にないと考えて構わないだろう．

## 3.4 ノマドワーキングとは

OSS プロジェクトを行う場合，場所や時間を問わないこともあり，街中のカフェでノートパソコンを広げ，作業に取り組む．こうしたワークスタイルを取るのは，今まではフリーで仕事をする人や企業の営業担当者のような外出が多い人が中心だった．ところが2011年3月の東日本大震災以降，事業継続を目的に社外でも仕事ができる環境を整えることが急務になってきた．このように社外でまるで遊牧民のように場所を移動しながら，社事をするスタイルをノマドワーキングと呼ぶ．

### 3.4.1 ノマドワーカーの実態

事業継続以外にも，ノマドワーキングには仕事の生産性を上げるメリットが期待されている．通勤時間を無くしたり，電話対応などで中断されずに仕事に集中できたりするからである．週に1度程度の在宅勤務を認め，その日を創造的な仕事に充てるといった使い方が多い．ノマドワーキングを実施するうえでは，IT（情報技術）の支援が不可欠．リモートアクセス環境を社員に提供し，ノートパソコンやスマートフォンなどからVPN（仮想プライベートネットワーク）経由で会社のパソコンやサーバにアクセス．企業の情報システムやパソコン内のデータを利用できる環境を整える企業が増えている．ただし，自宅以外でのノマドワーキングには慎重な企業も少なくない．端末の紛失による情報漏えいを防ぐ必要があり，会社から遠隔で紛失したスマホのデータを消去できるといった工夫が求められている．また，カフェや交通機関など人目の多いところでは機密性の高い情報を扱わないといった運用ルールを，事前に社員に浸透させておくことも重要である．

### 3.4.2ノマドワーキングのメリット，デメリット

メリット

・必要な道具や情報を見定めるようになる

オフィス資料や，デザイン資料など，バンバン開くという方はハイスペックなノートPCが必要である場合や，文章さえ書ければいいという方は軽くて安いノートで，もしくはタブレットで十分なんて人もいる．すると，結果的に不要なアイテムを切り捨て，その分情報も切り捨てることになる．必要なガジェットを見極めるだけでなく，チェックすべきブックマーク，チェックすべき資料，などオフィスでよくあるようにPCを開きっぱなしにしながら，タブレットで長時間動画サイトや掲示板などを見たりすることも少なくなる．

・一瞬のひらめきも逃さない

アイデアをメモするだけではなく，一瞬思いついたときにそのひらめきを逃すことはなくとっさに書いたりすることで，気づく間に良いプレゼン資料ができたりする．これはモバイルとも相まって強力な武器になるのだ．

・気分転換につながる

一番のメリットである．少し風景が違うだけでも新しいアイデアが沸いてきやすく，ふと煮詰まったときに，周りを見渡せば，老若男女いろんな人がめいめい好きなことをしており，そんなことからも新しいヒントが見つけ出せたりする．私も本論を執筆中の気分転換には海岸に来て作業をしている．

・時間的制限による集中力

電源がないところでは，PCのバッテリーなどの物理的に制限がある．私の場合は2時間くらいしか持たないのだが，これがむしろちょうど良い集中を呼んでくれている．2時間や作業したら，強制的に一休みというサイクルはかなり効率が良いと考える．

・カフェ，逃げ場に詳しくなる

よく行く駅やビルなどのカフェのメニューや電源・電波状況にやたら詳しくなる．

・ココロの余裕

いつでもすぐ作業に移れるということは，急な用件にも対応でき，ちょっとの時間をこまめに使うと意外と多くの作業をこなすことができる．

ここまでメリットを上げてきたが，反対にデメリットもある．

・常に遊んでいるように見られる

フリーランスの場合はこの弊害はあまり多くないが，机にかじりついてないと，サボっていると思われがちなのも日本の組織にありがちなことである．いい仕事には気分転換や，ある意味遊びも重要である．

・職場の理解

成果が見えやすい職種じゃないとなかなか厳しいの．となってくると，管理者の判断やマネジメント基準が不明確になってしまう．

ノマドワーカーになり，生活していくには，提供できる成果物の質が第一になる．実際私の知り合いにも，オフィスを持たずに完全にノマドワーキングな働き方をしている知人が何人かいる．彼らは輸入雑貨を手掛けていたりセミナー講師だったり文章を書く仕事をしていたりと，フィールドとする業界はさまざま．しかし彼らに共通しているのは，「個人」として働くことのできるスキルに加え，企業から仕事を任せられるくらいのクオリティを担保する能力があるということ．彼らと会うたびに，圧倒的な力の差を感じている．単に外で仕事をしたいという理由でノマドワーカーになりたいというなら，セキュリティの問題さえクリアすれば，いつでもノマドワーカーにはなれるのだ．ただ，本当にフリーランスとしてノマドをしたいのであれば，それなりの覚悟が必要．もしノマドワーカーという働き方に漠然とした憧れのようなものを持っているのであれば，自分がフリーランスとしてのノマドをしたいのか，それとも部分的にノマドがしたいのかをはっきりとさせたほうが良いのだ．ノマドワーキングが欧米のプログラマたちの間から出てきたというのは，成果が見えやすく・管理者の理解を得やすい職種という点で自然だったのかもしれない．今後こういった職種からでも，日本でも理解を得られるようになっていき，成果とリラックス・自由が両立するようになっていけば，無駄なストレスを減らし効率的に作業ができるのではないだろか．

次章ではweb上でプロジェクトを運用するために必要なツールが揃うホスティングサイトを述べる．

参考文献

[1] NCSA. National Center for Supercomputing Applications.

http://www.ncsa.uiuc.edu/

[2] Apache Software Foundation .

http://www.apache.org/

**第4章**

**ホスティングサイトの台頭**

# 第4章 ホスティングサイトの台頭

## 4.1 ホスティングサイトとは

技術的な観点からプロジェクトのウェブサイトを立ち上げることについては， それほど語ることはない．ウェブサーバを起動し，ウェブページを書くことはかなり単純な仕事である．ウェブサイトの主な機能は，明快にプロジェクトの概要を提供し，(バージョン管理システム, バグ追跡システムなどの)他のツールをウェブサイトと結びつけることだ．たとえウェブサーバを設定して起動する技量がなくても，その作業をやってくれる人を探すことは普通難しくない．とはいえ，時間と労力を節約するために，プロジェクトを運営するためのツールが一通り揃っているホスティングサイトがよく好んで使われる．

## 4.2 匿名性とプロジェクト参加

厳密にはホスティングサイトに限った問題ではないが，ホスティングサイトで最もよく見られるのが，ユーザーログインの機能に関する苦情だ．ログイン機能自体は十分単純でウェブサイトでは，訪問者が自分のユーザ名とパスワードを登録することができる．登録するとユーザのプロフィールが保存され，プロジェクトの管理者は，ユーザにリポジトリへのコミット権限のような特定の権限を与えることができる．この機能は非常に有用で，ホスティングサイトの主な利点のひとつ．問題は，未登録のユーザにも許可されるべきタスク，特にバグ追跡システム内のファイルアップロードや，既存の問題にコメントをつけるときに，往々にしてログインが必要になってしまっている点にある．こうしたタスクにログインを必要としてしまうと，本来迅速で便利であるべきタスクに参加する敷居が高くなってしまう．勿論，バグ追跡システムにデータを登録した人に連絡を取りたい人もいるが，その場合は(本人が望んだ場合に)電子メールアドレスを入力できるフィールドを設けておけば可能．新しいユーザがバグを発見して報告したいと思ったとして，バグ追跡システムに入力する前にアカウント作成フォームを入力しないといけないとわかれば難しい．ユーザを管理する利点は，通常は欠点に勝るものだが，ユーザを匿名で行動させる選択肢があるなら，全ての読み取り専用のアクションだけでなく，特にバグ追跡システムや，持っているならwikiページのデータ入力についても，ログインしていないユーザに許可するように心がけた方がよい．

## 4.3 ツールが一通り揃ったホスティングサイト

一通りのものが揃ったホスティングサイトには，主に二つの利点がる．ひとつめは，サーバのディスク容量とネットワーク帯域の太さ．つまり，超高速なネットワーク上に，複数のサーバマシンが巨大なラックに収納されている．プロジェクトがどれだけ成功しても，ディスク容量を使い切ったり，ネットワーク接続が使い物にならなくなることはないだろう．ふたつめは,サイトの維持が簡単なところ．ホスティングサイトは，バグ追跡システム，バージョン管理システム，メーリングリスト管理システム，アーカイバや,プロジェクトのウェブサイトを運営するのに必要ものを全て選べる．また，それらは既に設定済みであり，蓄積される全てのデータのバックアップにも注意を自動的に払ってくれるので，多くの決断をする必要はなくなる．フォームに入力し，ボタンを押しさえすればよいのだ．そうすれば巨大なプロジェクト用ウェブサイトを容易に運用することができるようになる．しかし，他のツールでよいものがあったとしても，ホスティングサイトが 選択したツールや設定を受け入れなければならないという欠点もあり，ホスティングサイトは，設定できるパラメータの範囲を普通狭くしているので柔軟性がない．自前でウェブサイトを立ち上げてサーバへの完全な管理権限を持っていた場合にできるなど，きめの細かい制御は厳しい．このことのよい例が，自動生成されるファイルの扱いだ．あるプロジェクトのウェブページは，自動生成されたファイルかもしれない．たとえば，FAQのデータを編集しやすいマスターフォーマットに保存し，それからHTMLやPDF，その他表示用のフォーマットを生成するシステムがあるとする．自動生成されたフォーマットではなく， マスターファイルだけをバージョン管理したいと考える．しかしウェブサイトが他人のサーバに置いてある場合は，マスターファイルが変更された場合にFAQのHTML版を再生成するカスタムフックを設定できないかもしれない．唯一の回避策は，ウェブサイトで表示できるように自動生成されたフォーマットもバージョン管理することだ．もっと大きな影響があるかもしれない．自身が望むほどウェブサイトの見た目を変える権限がない可能性もあり,ホスティングサイトの中には，ウェブページをカスタマイズすることを許可しているが 結局はデフォルトの配置がうんざりするようなやり方で表示されてしまうものもある．たとえば，SourceForgeというホスティングサイトがある．ここでのプロジェクトの中には，完全にカスタマイズしたホームページがあるが，詳しい情報を参照させるために "Sourceforge上のページ" に開発者を誘導しているものがある．SourceForge のページは，プロジェクトのホームページになるはずのものだが，ユーザに自前のホームページを使わせないようにしています．SourceForge のページには，バグ追跡システムやCVSリポジトリ，ダウンロードサイトへのリンクなどがあるが，不幸なことに，SourceForge のページにはたくさんのプロジェクトとは無関係なリンクも含まれている．ページの一番上部にはバナー広告があるが，アニメーション画像であることもよくある．左側にはプロジェクトに興味がある人には殆ど関係がないリンクが垂直に配置されている．右側にも別の広告がよく配置されており，ページの中央部分だけが本当にプロジェクトに特有の事項に専用の場所になっているが， これらもわかりにくい方法で配置されているので，訪問者が次にどこをクリックしたらいいのかわからなくなってしまうことがよくある．SourceForge のページ設計の裏には，きっと広告のように，SourceForgeの立場からすれば尤もな理由があるのだろう．しかし，プロジェクトの立場から見れば，その結果が理想のウェブページとはかけ離れたものになるかもしれない．私はSourceForgeを非難するつもりで言っているのではなく，似たような懸念は多くのホスティングサイトにも当てはまる．重要なのは，トレードオフが存在するということであり，プロジェクトのウェブサイトを維持するための技術的な重荷から解放されるが，他人のやり方を受け入れることとひきかえにはじめて恩恵を受けられるのだ．ホスティングサイトがプロジェクトに最適かどうかを決められるのは自分自身であり，仮にホスティングサイトを選んだ場合，プロジェクトの"ホームとなるURL"に自前のドメインを使うことで，自前のサーバに移行する余地を残しておくことも重要である．そうすることで自前のURLをホスティングサイトに転送することもでき，公開されているURLに完全に手を加えたホームページを置くことで，ユーザを洗練された機能を持ったホスティングサイトに誘導することもできる．ウェブサイトのホスティングについて後に別の解を選んだとしても，プロジェクトのURLは変えないように確実に準備をしておくようにしたい．

## 4.4 ホスティングサイトを選ぶ

2013年現時点で，自由につかえるホスティングサイトとして定評のあるのは次の三つだ．これらはすべて，オープンソースライセンスで公開するプロジェクト用に無料で使うことができ，バージョン管理システムやバグ追跡システム，そして Wiki も用意されている (バイナリのダウンロード機能など，その他の機能を用意しているところもある)．

・GitHub 対応しているバージョン管理システムはGit だけだがGitHubはGit を使ったプロジェクトの世界の中心となっており， 各種サービスも Git と統合されている．また，GitHub には，プログラムからそのサービスを利用するための API も完備されている．メーリングリストはないが，メーリングリストのサービスは他にもいろいろあるのでたいした問題にはならない．

・Google Code Hosting

対応しているバージョン管理システムは Subversion と Mercurial (少なくとも現時点では Git は未対応) で，Wiki やダウンロードエリア，そしてよくできたバグ追跡システムも用意されている．また，API もある．バージョン管理システムには当然ながらそれ自身の API があり，問題追跡システムにも 専用の API がある．Wiki のコンテンツは バージョン管理システムと連動 しているので，スクリプトから編集することができる．また，ダウンロードエリアにも スクリプトによるアップロードという名の API が用意されている．また，メーリングリストとしてGoogle Groupsが使える．

・SourceForge

これは最も古くからあるホスティングサイトで，古くからあるだけあって今でも規模は最大．他の二つのサイトが用意する機能をすべて備えており，インターフェイスも非常に使いやすいものだ．しかし，プロジェクトのページに表示される広告ブロックが気になるという人もいる．現時点では，ホスティングサイトの三巨頭の中で唯一，主要なバージョン管理システム (Git，Subversion，Mercurial そして CVS) のすべてに対応している．SourceForge にはメーリングリストサービスも用意されているが， もちろん他のメーリングリストサービスを使うこともできる．

Apache Software Foundation のように，自分達の任務と既にあるプロジェクトのコミュニティーにうまく合っているオープンソースプロジェクトを無料でホスティングしている組織もある．もう少し詳しく言うと，もしバージョン管理システムに Git を使っているのなら GitHub を，それ以外を使っているのなら Google Code Hostingを使うと良いと考えられている．

フリーなホスティングサイトの多くは，そのサイト自身を構成するソフトウェアをフリーソフトウェアのライセンスで公開していない(公開しているサイトとしては Launchpad やGitorious そしてGNU Savannah がある)．個人的には，サイト自身を構成するソフトウェアのすべてのコードにもアクセスできるのが理想だと考えている．自分のデータをエクスポートしたり，自分のデータの操作を自動化させたりできるということが重要であり，そのようなサイトであれば決してそこにロックインされることがなくなり，さらにプログラムを通じて拡張可能になる．ホスティングサイトを動かすためのすべてのコードをオープンソースで入手することにはそれなりの価値がある．とはいえ，実際にそのコードを運用するということはほとんどのユーザにとって難しい．これらのサイトには何台ものサーバやカスタマイズしたネットワーク，そして専任スタッフが必要になってくる．いずれにせよ，単にコードを持っているだけでは，サービスを複製したり「フォーク」したりするのは厳しいものがある．大事なのは，自分のデータがしばられてしまわないのを確実にするということだ．

Wikipedia に ホスティングサービスの比較 (日本語) の記事がある．オープンソースプロジェクトのホスティング環境の最新情報については，まずここを見るとよい．また，Haggen So は，博士論文 Construction of an Evaluation Model for Free/Open Source Project Hosting (FOSPHost) sites の研究の一環としてさまざまなホスティングサイトを評価した．発表されたのは2005 年だが，2007 年に新たに更新されている．この評価基準は今後も長く使えるものである．

彼の調査結果は http://www.ibiblio.org/fosphost/ にあり，

比較結果の表は http://www.ibiblio.org/fosphost/exhost.htm

possv2 24 March 2013: If you're reading this note, then you've encountered this chapter while it's undergoing substantial revision; see producingoss.com/v2.html for details.

poss2 todo: Note the possibility of mixing services from different canned hosting sites. Also, talk about the importance of APIs for an exit export. Mention the Google Groups conundrum. Maybe replace some of the above commentary with a reference to 付録A Canned Hosting Sites.

今回の研究で使用したホスティングサイトで，プロジェクトホスティングサービスで高い成長率と普及率で注目を集めるGithubについて以下に述べる．

## 4.5 Githubとは何か

Andreessen Horowitzの発表によると，1億ドルという途方もない額がGitHubに投資された．GitHubはその金で一体何をするつもりか，Andreessen Horowitzにとって良い投資だったのか，そんな巨額の投資はGitHubにとって良いことなのか，さまざまな意見や憶測が，Webの至るところに出現した．しかしそもそも，GitHubとはどのようなものなのか，デベロッパにとってなぜそんなに人気があるのか，GitHubは「コードを共有したり公開するためのサービスだ」とか，「プログラマのためのソーシャルネットワーキングサイト」だ，として通説されているがいずれも，GitHubが特別な存在である理由を説明していない

GitHubはその名のとおり，Gitを使うためのハブである．GitはLinuxの作者Linus Torvaldsが始めたプロジェクトで，GitHubの訓練士Matthew McCulloughの説明によると，Gitは昔からあるバージョン管理システム(version control system, VCS)の一種であり，プロジェクトの改訂履歴を管理し保存する．プログラムのコードに利用されることが圧倒的に多いが，実はWordのドキュメントでもFinal Cutのプロジェクトでも，どんなタイプのファイルでも管理できる．とにかく，コンピュータのプログラムにかぎらず，どんなドキュメントでも，すべての段階の草案やアップデート履歴を保存し管理できるファイルシステムなのだ．Git以前によく使われたCVSやSubversionといったバージョン管理システムでは，一つのプロジェクトの複数の参加者に対し，中央的な“リポジトリ”（repository, 保存庫）が一つだけあった．一人のデベロッパがどこかを書き換えると，その変更が直接，中央的リポジトリにも及ぶ．しかしGitは分散バージョン管理システムなので，一人一人がリポジトリ全体のコピーを保有し，そのコピーに対して変更を行う．そしてその変更がOKとなったら，中央的サーバにその変更を“チェックイン”する．コードを書き換えるたびにサーバに接続しなくてもよいから，これまでよりもきめ細かい変更とその共有が，やりやすくなる．GitHubは， いろんなプロジェクトのためにGitのリポジトリをホスティングするサービスだ．独自の便利な機能がたくさんある．Gitはコマンドラインツールだが，GitHubはWeb上でグラフィカルなユーザインタフェイスを提供する．セキュリティのためのアクセス制御もあり，また各プロジェクトにwikiやベーシックなタスク管理ツールなど，コラボレーションのための機能も提供する．GitHubの最大の目玉機能が“フォーク(forking)”だ．フォークとは，食器のフォークの先端のように，一つのプロジェクトが複数に分派していくこと．それをGitHubでは，誰かのリポジトリをほかの人がコピーすることによって行う．オリジナルに対するライト(write, 書き込み)アクセスがなくても，それを自分のところで改変できる．自分が行った変更をオリジナルに反映したかったら，オリジナルのオーナーに“プルリクエスト(pull request)”と呼ばれる通知を送る．そしてオーナーは，ボタンをクリックするだけで，その人のリポジトリに対して行われた変更を自分のリポジトリにも取り入れることができる．人のコードを自分の（メインの）コードに導入することを，デベロッパ界隈の用語でマージする(merge, 併合)と言う．マージは複数のデベロッパが関わるプロジェクトにおいて最重要な工程であり，GitHubではそれが安全確実迅速にできる．これら三つの機能…フォークとプルリクエストとマージ…があるために，GitHubはきわめて強力なサービスなのだ．Gitを学ぶクラス﻿TryGitを最近始めたばかりの，Code SchoolのGregg Pollackの説明によると，GitHub以前には，オープンソースのプロジェクトに寄与貢献しようと思ったら，まずそのプロジェクトのソースコードを手作業でダウンロードし，変更をローカルに行い，そして“パッチ(patch, つぎあて)”と呼ばれる変更内容の一覧をプロジェクトのメンテナにメールする．メンテナはそのパッチを評価し，OKとなったらそれらの変更をマージする．パッチは，まったく未知の人から突然送られてくることも多い．しかし，GitHubではネットワーク効果が絶大な威力を発揮する，とPollackは説明を続ける．あなたがプルリクエストを送ると，プロジェクトのメンテナはあなたのプロフィールを見る．そこには，あなたがGitHub上で行った寄与貢献(contributions, コントリビューション)がすべて書かれている．あなたのパッチが受け入れられたら，あなたはオリジナルのサイト上で信任を取得し，そのことがあなたのプロフィールにも載る．メンテナがあなたのプロフィールを見るときそれは，あなたの評判を知るための履歴書のようなものだ．GitHub上にたくさんの人間とたくさんのプロジェクトがあればあるほど，プロジェクトのメンテナが優れた寄与貢献者(contributors, コントリビュータ)をより見つけやすくなる．これぞまさに，ネットワーク効果の典型だ．パッチは，特定のメンテナの手に渡るだけでなく，公開の場で議論することもできる．GitHubのインターフェイスを使わないメンテナにとっても，GitHubの存在はコントリビューションの管理を容易にしてくれる．オープンソースのサーバサイドJavaScriptプラットホーム，Node.jsのメンテナIsaac Schlueterによると，“ぼくは最終的にはとりあえずパッチをダウンロードしてコマンドラインでマージしている．GitHubのマージボタンは使わない．しかしGitHubは，人びとがパッチを議論できる“町の中心の広場”として，かけがえのない存在だ”，という．敷居を低くして誰もが気軽に参加できるため，オープンソースの開発がより民主化され，若いプロジェクトの成長を助ける．“GitHubがなければ，今日の姿のNode.jsもなかっただろう”，とSchlueterは言う．GitHubには，一般に公開されるオープンソースのリポジトリ集という顔のほかに，企業にプライベートのリポジトリを売ったり，そのソフトウェアの企業向けのオンプレミスのインスタンスを売るビジネスとしての顔がある．これらのソリューションにはもちろん，オープンなGitHubのネットワーク効果というアドバンテージはないが，コラボレーションの機能はそっくりそのままある．ただしビジネスとしてのそれには，敵もいる．Atlassianが2010年に買収したBitBucketも，競合製品の一つだ（VCSないしSCMS(source code management system)としてGitだけでなく﻿同じく好評なMercurialも使える）．今年の初めにAtlassianが立ち上げたStashは，プライベートでオンプレミスなGitリポジトリをBitBucket的/GitHub的なコラボレーション環境で設営利用できる製品だ．Atlassianはこのほか，バグトラッカーのJiraやwikiソフトのConfluenceなど，デベロッパのためのコラボレーションツールも売っている．Atlassianは2010年にAccel Partnersから6000万ドルを調達している．この6000万ドルという数字を見れば，GitHubの1億ドルの意味もおよそ分かるだろう．同社が未来に向けて何を志向しているのかも分かる．たとえばSchlueterは，GitHubの問題追跡機能はある面でJiraと競合するだろう，と言っている．お金はプライベートでオンプレミスなホスティングにあり，愛はパブリックなリポジトリ集にある．しかし，おそらくいちばん重要なのは，GitHubがコードに関して現代のアレクサンドリア図書館になっていることだ．Gitを使うときめ細かい粒度の小さい変更の記録ができるので，まったくの初心者でもあるいはエキスパートでも，世界のもっとも偉大なデベロッパたちの足跡を踏みしめることができ，彼らがプログラミングの難題をどうやって解いたかを知ることができる．今回の投資の結果がどう出ようとも，それこそが，GitHubが今後の世界の歴史に残す，驚異的な遺産である．

以降はGitHubの代表的な機能を詳しく述べる．

### 4.5.1バージョン管理システム「Git（ギット）」

Github上で主に使用する管理システムの一つである．Gitとは，分散型バージョン管理システムの一つで，もともとLinuxのソースコードを効果的に管理するために開発された．

Gitでは，ファイルの状態を好きなときに更新履歴として保存しておくことができる．そのため，一度編集したファイルを過去の状態に戻したり，編集箇所の差分を表示したりすることができる．また，古いファイルを元に編集したファイルで，他人の編集した最新ファイルを上書きしようとすると，サーバにアップロードした時に警告が出るため，知らず知らずのうちに他人の編集内容を上書きしてしまうといったリスクは起こらない．

そのためGitでファイルを管理すると，更新履歴がGitに保存されて行く．手元にバックアップ用のファイルのコピーは置く必要がない．

### 4.5.2 履歴を管理する「リポジトリ」

リポジトリとは，ファイルやディレクトリの状態を記録する場所である．保存された状態は，内容の変更履歴として格納されている．変更履歴を管理したいディレクトリをリポジトリの管理下に置くことで，そのディレクトリ内のファイルやディレクトリの変更履歴を記録することができる．まず，Gitのリポジトリは，リモートリポジトリとローカルリポジトリの2種類に分けられる．

「リモートリポジトリ」

専用のサーバに配置して複数人で共有するためのリポジトリ．

「ローカルリポジトリ」

ユーザ一人ひとりが利用するために，自分の手元のマシン上に配置するリポジトリだ．リポジトリをリモートとローカルの2種類に分けることで，普段の作業はローカルリポジトリを使って全て手元のマシン上で行うことができる．自分のローカルリポジトリで作業した内容を公開したい時は，リモートリポジトリにアップロードして公開し，また，リモートリポジトリを通してほかの人の作業内容を取得することもできる．

### 4.5.3 変更を記録する「コミット」

ファイルやディレクトリの追加・変更を，リポジトリに記録するにはコミットという操作を行う．

コミットを実行すると，リポジトリの内では，前回コミットした時の状態から現在の状態までの差分を記録したコミット(またはリビジョン)と呼ばれるものが作成される．このコミットは，時系列順につながった状態でリポジトリに格納されている．このコミットを最新の物から辿ることで，過去の変更履歴やその内容を知ることができるようになっている．

これらのコミットには，コミットの情報から計算された重複のない英数字40桁の名前が付けられており，この名前を指定することで，リポジトリの中からコミットを指定することができる．

バグ修正や機能追加などの異なる意味を持つ変更は，できるだけ分けてコミットするよう心掛けたい．後から履歴を見て特定の変更内容を探す時に探しやすくなるからだ．また，コミットの実行時にはコミットメッセージの入力を求められる．コミットメッセージは必須となっているため，空のままで実行するとコミットが失敗する．コメントは，他の人がコミットの変更内容を調べる場合や，自分で後から履歴を見直す際に大切な情報となるので，変更内容のわかりやすいコメントを書くようにすることが重要になってくる．

Gitでは標準的に，

1行目 : コミットでの変更内容の要約

2行目 : 空行

3行目以降 : 変更した理由

という形式でコメントを書く．

### 4.5.4 リポジトリに変更を記録する「ワークツリー」と「インデックス」

Gitでは，Gitの管理下に置かれた，実際に作業をしているディレクトリのことをワークツリーと呼ぶ．そして，Gitではリポジトリとワークツリーの間にはインデックスというものが存在している．インデックスとは，リポジトリにコミットする準備をするための場所のことを指す．

Gitでは，コミットを実行した時にワークツリーから直接リポジトリ内に状態を記録するのでなく，その間に設けられているインデックスの設定された状態を記録するようになっている．そのため，コミットでファイルの状態を記録するためには，まずインデックスにファイルを登録する必要がある．インデックスを間に挟むことで，ワークツリー内の必要ないファイルを含めずにコミットを行ったり，ファイルの一部の変更だけをインデックスに登録してコミットすることができる．

### 4.5.5 リポジトリの共有「プッシュ」と「クローン」と「プル」

ここまでは，ローカルリポジトリの基本的な使い方を説明してきた．ここからは，リモートリポジトリを使ってローカルリポジトリの変更履歴を共有する方法について説明する．

「Push」

リモートリポジトリで自分の手元のローカルリポジトリの変更履歴を共有するには，ローカルリポジトリ内の変更履歴をアップロードする必要がある．

そのために，Gitではプッシュ(Push)という操作を行う． Pushを実行すると，リモートリポジトリに自分の変更履歴がアップロードされて，リモートリポジトリ内の変更履歴がローカルリポジトリの変更履歴と同じ状態になる．

「Clone」

誰かの変更履歴の入ったリモートリポジトリがあれば，それをまるごと複製して自分の手元で作業をすることができる．リモートリポジトリを複製するには，クローン(Clone)という操作を行います．クローンを実行すると，リモートリポジトリの内容をまるまるダウンロードしてきて，別のマシンにローカルリポジトリとして作成できる．クローンしたローカルリポジトリは変更履歴も複製されているので，元々のリポジトリと全く同じように履歴の参照やコミットをすることができる．

「pull」

リモートリポジトリを共有して複数人で作業すると，リモートリポジトリにプッシュしていく．すると，自分のローカルリポジトリに，ほかの人がプッシュした変更内容を取り込む必要がある．リモートリポジトリからローカルリポジトリを更新するにはプル(Pull)という操作を行う．Pullを実行すると，リモートリポジトリから最新の変更履歴をダウンロードしてきて，自分のローカルリポジトリにその内容を取り込むことができる．

### 4.5.6 変更履歴の統合「変更履歴のマージ」

最後にpullを実行してから次のpushをするまでの間に，ほかの人がpushをしてリモートリポジトリを更新してしまっていた場合には，自分のpushが拒否されてしまう． この場合，マージという作業を行なって他の履歴での変更を取り込むまで自分のpushは拒否される．これは，マージを行わないまま履歴を上書きしてしまうと，ほかの人がpushした変更(図中のコミットC)が失われてしまうからだ．マージでは，Gitが変更箇所を自動的に統合してくれるが，自動では統合出来ない場合もある．次の4.5.7では，手動で統合する方法を説明する.

### 4.5.7 変更履歴の統合「競合の解決」

4.5.6で説明したように，マージを行うとGitが変更箇所を自動的に統合する．しかし，自動で統合できない場合もある．それは，リモートリポジトリとローカルリポジトリでファイル内の同じ箇所を変更していた場合で，この場合，どちらの変更を取り込むか自動では判断できないのでエラーが発生してしまう．競合が発生した箇所は，Gitがファイルの内容を次の図のように修正するので，手動で修正する必要がある．

次章ではGitHub内で行われているプロジェクトを調査する，

**第5章**

**オープンソーススフトウェアプロジェクトのマネジメント実態調査**

# 第5章 オープンソーススフトウェアプロジェクトのマネジメント実態調査

今回の研究にあたり参照にした今回GitHub上で行われている「Ruby on Rails3アプリ開発プロジェクト」「CMS開発プロジェクト」の二点に着眼し，研究調査した．

## 5.1 調査方法

研究方法は以下の3通りである．

1. OSSのプロジェクトマネジメントの実態把握のための調査項目の決定をする データの収集方法を検討する．
2. データを収集し，解析する．

① プロジェクトの規模を調べる．第一段階では実態把握のため調査項目を独自に制定した．

１．何人規模のメンバで機能しているのか

２．リリース（バージョン１への到達）までにかかった時間

３．ソフトウェアの規模

４．コミット，リポジトリ，イシューなどの数

データマイニング的な手法で解析

で集められた情報を元にデータ化する．

グラフにまとめ，可視化する

でまとめたデータを元にグラフにまとめる．

## 5.2「Ruby on Rails3アプリ開発プロジェクト」

### 5.2.1Rubyとは

Rubyは，手軽なオブジェクト指向プログラミングを実現するための種々の機能を持つオブジェクト指向スクリプト言語で，本格的なオブジェクト指向言語であるSmalltalk，EiffelやC++などでは大げさに思われるような領域でのオブジェクト指向プログラミングを支援することを目的としている．もちろん通常の手続き型のプログラミングも可能である．

Rubyはテキスト処理関係の能力などに優れ，Perlと同じくらい強力でシンプルな文法と，例外処理やイテレータなどの機構によって，より分かりやすいプログラミングが出来る．

長所として，

・シンプルな文法

・普通のオブジェクト指向機能(クラス，メソッドコールなど)

・特殊なオブジェクト指向機能(Mixin，特異メソッドなど)

・演算子オーバーロード

・例外処理機能

・イテレータとクロージャ

・ガーベージコレクタ

・ダイナミックローディング(アーキテクチャによる)

・移植性が高い．多くのUNIX上で動くだけでなく，DOSやWindows，Mac，BeOSなどの上でも動く

### 5.2.2 Ruby on Railsとは

Webアプリケーション向けのMVCフレームワークである．実際に様々なサービスで用いられている．たとえば，Twitterなどがその典型例で(現在はすべてRailsではなくScalaをバックエンドやインフラに用いているが)，他にもメジャーどころではクックパッド・食べログなどかなり大規模なサービスでも運用実績がある．Ruby on Railsはいくつかのコンポーネントの集合体で構成されており，Ruby on Railsとは無関係に使用することも可能ではあるが，Ruby on Railsとともに使用することが多い．

つまり，特にデータベースを利用したWebアプリケーション開発において，新たに書く必要のあるコードを大幅に減らし開発期間を短縮できる．その中でRuby on Rails3には

Webフレームワークに必要なモジュールがすべてバンドルされてリリースされるが，それは，あくまでも参照実装的な位置付けでもちろん，そのまま使っても十分に優れたものだが，自由にモジュールを入れ替えられており，ORM層だけでなく，コントローラやビュー，JavaScript，Generator API，テストフレームワークなど，各コンポーネントのインターフェイスがきちんと決められいて（APIに対してSPIと呼ぶこともある），交換可能となっている．そのため，アプリ開発などに利用されており，今回の研究の題材にする．

## 5.3「Ruby on Rails3アプリ開発プロジェクト」の概要

主に4つのプロジェクトに着眼し，統計した．

### 5.3.1「Gitlab」

・開発メンバ14名

・Vｒ

・総コミット数　421

・概要GitHubクローンの結構大きなプロジェクト．RedisやGitoliteが必要であるため，自分でインストールして動かすのはやや手間がかかるが，なGitHubクローンなので，GitHub利用者なら，app/models 以下を見ても，どのモデルが何がおき，どのような作業しているのかが分かりやすい．

https://github.com/gitlabhq/gitlabhq

### 5.3.2「WhiteHall」

開発メンバ　67名

Vr

概要　英国政府のポータルサイト「Gov.uk」は，オープンソースのRails3アプリで，GitHub上で活発に開発されている．ポータルサイトのような，かつてならテンプレート言語で対応していたものであっても，今ではアプリとして実装することができる．手元で動作させるのは難しいが，Gov.ukに行けば実物が見れるので，参考になる．したがってポータルサイトであるため，そう複雑なことはなく，やっていることはユーザーやコメントの管理，イベント情報などの管理という感じだ．コミットを見ていると，意外にダメなコードが入って誰かがそれを直したりしているような様子も伺え，興味深い．

https://github.com/alphagov/whitehall

### 5.3.3「Spree」

開発メンバ　67名

Vr

　オープンソースのECサイト向けアプリです．かなり大きいRailsアプリですが，なんといっても，リアルに運用されている商用サイトが多いという意味で，これ以上の実用アプリはないかもしれません．サポートビジネスを展開しているだけあって，ドキュメントも充実しています．Spreeは海外サイトでしかメジャーでないと思っていましたが，国内にも採用事例はあるようです．ちなみにメガネ・サングラス通販サイトのOh My Glassesは，Spreeを使って3カ月で構築されたサイトだ．

https://github.com/spree/spree/

### 5.3.4「Shapado」

開発メンバ　67名

Vr

StackOverflow.comのクローン実装です．MongoDB（Mongoid）を使っていたり，今どきのソーシャルっぽい機能があれこれ実装されていて，参考になるかもしれません．以前VPSで動かしてみたことがありますが，Mongoさえ入れれば動かすのは簡単でした．使っている Gem の一覧（Gemfile ）を見ると，割りと良く見かける

https://github.com/ricodigo/shapado

## 5.4「cms開発プロジェクト」

### 5.4.1 CMSとは

## 5.5「cms開発プロジェクト」の概要

### 5.5.1「RefineryCMS」

名前はCMSですが，カスタマイズが前提で，CMS Frameworkと名乗っているプロジェクトです．テストコードはRSpecで書かれていますが，spec/ではなく，./\*/spec に分散していたり，何かと素直なRailsアプリではないので，参考にするにはビミョウかもしれません．私はCMSとして注目しています．

https://github.com/resolve/refinerycms

**第6章**

**結論**

# 第6章 結論

## 6.1実証分析結果

### 6.1.1 総合統計結果

## 6.2プロジェクトマネジメントありかた．

GitHubでの目標は「誰も辞めない職場づくりを」だ．

実績として，GitHub本社では，

「創業から今までに辞めた社員が1人もいない」，「従業員約150名（2013年2月時点）のうち，マネジャーは1人もいない」，「勤務時間は完全フリー」，「2/3がリモートで働く」この発想はマネジメント技術の賜物ではないだろうか．こうした希有な体制を支えているのは，同社内で築かれた優れたチームワークである．彼らがいかにしてGitHubの開発メンバの存在価値を最大化し，それを維持し続けているのか．そんな疑問を通じて，「プログラマーのクリエイティビティを最大限に引き出すために必要な要素」「すべてをオープンにする」ことで社員の創造性を引き出す．

## 6.3オープンソーススフトウェアプロジェクトの抱える課題と今後のありかた

# おわりに