

## 目次

第1章 緒論.....	1
1.1 本章の構成.....	1
1.2 研究背景.....	1
1.3 研究目的.....	2
1.4 プロジェクトマネジメントとの関連.....	2
第2章 スポーツ界の現状.....	4
2.1 本章の構成.....	4
2.2 マネー・ボールとは何か.....	4
2.3 マネー・ボールが野球界に与えた影響とその後の結果.....	5
2.4 日本野球に与えた影響と代表的な事例.....	5
2.5 マネー・ボールがサッカー界に与えた影響.....	6
2.5.1 選手の獲得方法について.....	6
2.5.2 野球にあってサッカーにはあまりないものについて.....	7
2.5.3 マネー・ボール理論を取り入れ成功した例.....	7
第3章 オープンソースソフトウェアの概要.....	9
3.1 本章の構成.....	9
3.2 オープンソースとは何か.....	9
3.3 オープンソースの定義.....	10
3.4 オープンソースソフトウェアの種類.....	13
3.5 オープンソースの今後.....	15
第4章 オープンソースソフトウェアの現状.....	20
4.1 本章の構成.....	20
4.2 商用ソフトウェアの比較.....	20
4.3 オープンソースを活用したプロジェクトのメリット.....	22
4.3.1 開発者のメリット，デメリット.....	22
4.3.2 ユーザ側のメリット，デメリット.....	24
4.4 ノマドワーキングとは.....	26
4.4.1 ノマドワーカーの事態.....	26
4.4.2 ノマドワーキングのメリット，デメリット.....	26
第5章 GitHub について.....	30
5.1 本章の構成.....	30
5.2 GitHub とは何か.....	30
5.3 バージョン管理システム.....	32

5.4 GitHub の機能について .....	33
5.4.1 Git について .....	33
5.4.2 リポジトリについて .....	34
5.4.3 コミットについて .....	35
5.4.4 リポジトリの共有について .....	37
5.4.5 変更履歴のマージについて .....	39
5.4.6 変更履歴の統合「競合の解決」 .....	40
5.4.7 ワークツリーとインデックスについて .....	41
5.5 API について .....	42
5.5.1 API を公開する・公開されるメリット .....	42
第 6 章 開発・調査 .....	44
6.1 開発・調査について .....	44
6.2 研究方法 .....	44
6.3 研究ツール .....	44
6.3.1 Ubuntu とは .....	44
6.3.2 Ubuntu の意味 .....	45
6.3.3 Ubuntu の日本語環境 .....	45
6.3.4 Ubuntu の特徴 .....	46
6.3.5 Ubuntu サーバーの特徴 .....	48
6.3.6 調査環境構築 .....	50
6.4 研究対象 .....	53
6.5 調査方法 .....	53
第 7 章 調査結果・考察 .....	60
7.1 本章の構成 .....	60
7.2 調査結果 .....	60
7.2.1 square / spoon .....	60
7.2.2 stanfy / spoon-gradle-plugin .....	62
7.2.3 spoonapps / spoonme .....	63
7.3 考察 .....	66
7.4 謝辞 .....	66

## 図目次

図 1 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例 .....	34
図 2 コミットがリポジトリに格納されている状態 .....	35
図 3 ローカルリポジトリから <b>Push</b> を実行した例 .....	37
図 4 ローカルリポジトリに <b>Pull</b> を実行した例 .....	38
図 5 <b>Push</b> が拒否された例 .....	39
図 6 マージを行わないと <b>Push</b> が拒否される例 .....	40
図 7 インデックスに登録してコミットする流れ .....	41
図 8 検索画面 .....	53
図 9 プロジェクトのメイン画面 .....	54
図 10 メンバー一覧画面 .....	55
図 11 個人のコミットの履歴画面 .....	56
図 12 プロジェクトのメイン画面 .....	57
図 13 全体のコミットの履歴画面 .....	57
図 14 コミットで追加・修正・削除した画面 .....	58
図 15 メンバがコミットで修正・追加した行数の割合 .....	61
図 16 メンバがコミットで修正・追加した行数の割合 .....	62
図 17 メンバがコミットで修正・追加した行数の割合 .....	63
図 18 メンバがコミットで修正・追加した行数の割合 .....	64
図 19 メンバがコミットで修正・追加した行数の割合 .....	65

## 表目次

表 1 オープンソースソフトウェアの配布条件 .....	10
表 2 オープンソースソフトウェアの種類 .....	13
表 3 Ubuntu の特徴一覧 .....	46
表 4 Ubuntu サーバーの特徴一覧 .....	48
表 5 コマンドの解説 .....	52
表 6 プロジェクト一覧 .....	53

# 第 1 章

## 緒論

## 第 1 章 緒論

### 1.1 本章の構成

第 1 章では，本論文の緒論を述べる．研究背景，研究目的，プロジェクトマネジメントの関連について記述する．

### 1.2 研究背景

人材マネジメントに統計分析を活用する試みがスポーツ界で広まっている．

野球界で最も有名なものとしてマネー・ボール[1]が挙げられる．その中で登場するアスレチックスという球団は，他球団に比べて資金面で劣っていたため，有名な選手を補強できずにいた．そこで，選手の成績を統計的に分析して少ない資金でチームの戦術に合う選手を獲得した．その結果，全球団の中で最高の勝率を記録した．しかし，この構成方法はあくまで確率論によるため，長期戦においては効果を発揮するが，プレーオフなどの短期決戦においては選手の力量が勝因を決めることも多いため課題はある．

サッカー界では，チームを統計的な手法で強化するのは，野球チームを統計的な手法で強化するよりも難しかった．その理由として，サッカーは野球ほど分析すべきデータが単純ではないためより高度な分析が必要となる．さらに，野球の場合は統計上の知識／分析力があればある程度は容易に分析できるが，サッカーの場合は分析力とサッカーの知識の双方において理解することが不可欠なのが特徴的である．サッカーは時代と共に戦術が変化し，必要なスキルもそれに合わせて変化することから，選手の評価基準も柔軟に変更する必要がある．

その中でセイバーメトリクスを利用して成功を収めているチームがある．それは，プレミアリーグ（イングランドのリーグ）のニューカッスルである[2]．プレミアリーグは全 20 チームが所属していてその中にマンチェスター・ユナイテッドやチェルシー，アーセナルといった経営規模が大きいクラブが多数いる．今までのニューカッスルの用いた戦術は，前線の長身の選手にロングパスを出し競り合ったこぼれ球を拾い攻撃することが多かった．しかし，その主力を引き抜かれたことと監督が変わったことにより，その戦術が大いに変化した．その戦術は，4-3-3 というコンパクトなシステムである．コンパクトにすることにより，中盤の選手が活きよりポゼッションの高い戦術に変化した．新たに定められたチームの編成基準は，「ロングパスによるチャンスメーク」よりも「ショートパスによるチャンスメーク」である．選手の基準において「ロングパスによるチャンスメーク」よりも「ショートパスによるチャンスメーク」を重視したのは，ロングパスによるチャンスメークは正確性がなくゴールに繋がりにくい，ショートパスによるチャンスメークは正確性がありゴールに繋がる確率が高くなるからである．そこで補強した選手は，怪我をしたために他のクラブの興味を引かなくなっていた元有名選手である．その結果，チームはリーグ戦 7 位という好成績を収め，EL（ヨーロッパリーグ）への出場権を獲得した．トップ 10 が目標

だったチームにはEL 出場という結果は、成功と言っても良いだろう。

本研究では、スポーツ界で行われているような統計解析手法を用いた人材マネジメントの、ソフトウェア開発の現場への導入を検討する。

現在、オープンソースソフトウェア開発は GitHub 上で行われていることが多い。

オープンソースソフトウェアとは、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。その特徴として企業、個人など参加形態を問わずに誰でもプロジェクトに参加することができる。

過去に GitHub 上で行われているプロジェクトの各メンバの活動ログを収集し、役割分担の実態を明らかにする研究が行われていた[3]。この研究で、Push する行為とりポジトリにスターを付ける行為は別のメンバが行っていることが多いことが明らかになった。

そこで今回は、プロジェクトのコミット総数と個人がコミットの追加、修正で書いた行数が何行かを調べて、プロジェクトへの貢献度を調査する。

### 1.3 研究目的

GitHub 上で多く行われている OSS 開発のプロジェクトを用いて、活動ログを統計解析手法で分析する。その結果から個人のプロジェクトへの貢献度を調査する。

### 1.4 プロジェクトマネジメントとの関連

チームスポーツにおいて選手を評価する客観的な方法確立することは、プロジェクトにおいてメンバを評価する客観的な方法の確立につながる。

本研究で検証する手法は、PM の人的資源マネジメントに役立つだろう。

#### 参考文献

- [1] マイケル・ルイス／中山宥訳. マネー・ボール 奇跡のチームをつくった男. ランダムハウス講談社. 2004.
- [2] 山中忍. プレミアリーグ版“マネー・ボール”？清貧クラブのニューカッスルが躍進. 2012.  
<http://number.bunshun.jp/articles/-/216542>(参照 2014-5-7)
- [3] 関口元基. オープンソースソフトウェア開発における役割分担の実態調査. 千葉工業大学, 2013, 卒業論文.
- [4] GitHub Developer.  
<http://developer.github.com/v3/activity/events/types/#gollumevent>(参照 2014-5-15)

## 第 2 章

### スポーツ界の現状



## 第2章 スポーツ界の現状

### 2.1 本章の構成

本章ではスポーツ界の現状や統計的手法を取り入れたことでチーム構成に与えた影響について記述する。

### 2.2 マネー・ボールとは何か

ビリー・ビーンはかつて超高校級選手としてニューヨーク・メッツから1巡目指名を受けるほどのスター候補生だった。当時のスカウトの言葉を信じ、名門のスタンフォード大学の奨学生の権利を蹴ってまでプロの道を選んだ。しかし、自身の性格も災いして泣かず飛ばずの日々を過ごすことになった。その後様々な球団を転々として、挙句の果てに引退した。第二の野球人生としてスカウトに転進し歩み始める物語である。

2001年のポストシーズンにオークランド・アスレチックスは、ニューヨーク・ヤンキースの前に敗れ去った。その結果、オフにはチームの核となるジアンビーなどの3選手がFA（他球団とも選手契約を締結できる権利を持つ選手のこと）による移籍が決定的となり、補強が迫られていた。アスレチックのゼブラルマネージャーとなっていたビリーは、2002年シーズンに向けて戦力を整えるべく補強資金を求めるも、スモールマーケットのオークランドを本拠地とし、資金に余裕のないオーナーの返事はずれなく、補強資金は例年通りしか出なかった。ある日、イエール大学を卒業したピーター・ブランドに出会った。ピーターは各種統計から選手を客観的に評価するセイバーメトリクスを用いて、他のスカウトとは違う尺度で選手を評価していた。その理論に興味を抱いたビーンは、その理論をあまり公にできず肩身の狭い思いをしていた彼を自身の補佐として引き抜き、他球団から評価されていない埋もれた戦力を発掘し、低予算でチームを改革しようと試みた。

野球統計学から高い評価を得た選手たちを獲得するも、ビリーの思うように勝てず、また監督もその意図に反し、自身の考えに基づいた起用をした。その結果14連敗を喫し、周囲から批判を受けた。業を煮やしたビリーは監督が起用していた選手をトレードに出し、自身が推し進める野球に合った選手を起用させた。またそれと同時にビリーは選手とも積極的に交流し、自身が野球界に革命を起こすであろう野球理論を選手に啓発した。その結果、MLB史上初の20連勝という大記録を達成した。しかし、地区予選で敗戦し、ワールドシリーズへの出場はできなかった。野球統計学はあくまで確率論によるため、長期戦においては効果を発揮するが、プレーオフなどの短期決戦においては選手の力量が勝因を決めることも多く、個々の選手の力量が低くチーム力で勝つタイプのアスレチックスはまけてしまう。だが、その野球理論に目を付けたボストン・レッドソックスは、彼らの理論が使えると判断し、ビリーに野球史上最高額ともいえるビッグオファーを提示するも、ビリーはそれを断り、アスレチックスでワールドチャンピオンを目指すべく、今も挑戦している。

そして皮肉なことに巨額な資金を使えるレッドソックスが、彼らの野球統計学の理論を

用いたことでワールドチャンピオンになり、その「セイバーメトリクス」の有用性は一気に野球界へと広がった。

## 2.3 マネー・ボールが野球界に与えた影響とその後の結果

野球統計学をチーム運営に応用する場合、以下のようなことを行う。

- 「出塁率」などおおくのスカウトが着目しない指標により選手を評価し、当時の球界では低評価だった選手を獲得し、チームも好成績を上げた。
- トレードでも年棒と適正評価（適正価値）が釣り合わない選手は、容赦なくトレード要員として放出し、その代わりチームに必要で若くて年棒の安い選手を獲得、FA は年棒高騰に繋がるため基本的に引き止めない。
- 費用対効果の高い選手に関しては年棒が低い時期に長期複数年契約をすることで年棒の抑制を図る。
- ドラフトでも「将来性」という不確実性を排除し、即戦力になる大学生を優先的に獲得する方針にする。また、選手の身辺調査を行い、須高に問題があり、なおかつ更生不可なら選手としての能力が高くても獲得はしない。

こういったものが挙げられる。しかし、これらの手法はあくまでも資金の貧しい球団が資金が豊富にある球団と互角に戦えるために行っていた「弱者の戦略」でしかない。こういった手法がチーム運営やスカウティングに広まるに連れて、出塁率が高い選手の獲得競争が激化し安価で獲得できないなど、ビリーが行っていたマネー・ボールの優位性は薄れてしまう。しかしながら、マネー・ボール自体も発展途上のため、様々な変化や発展があり、その中でチームごとにカスタマイズされていくのではないだろうか。また、投手力がウリのチームはそれに合ったスカウティングをし、野手育成に定評のあるチームはそれに合ったスカウティングをする、その際にこれまでの理論を応用しているのではないだろうか。

## 2.4 日本野球に与えた影響と代表的な事例

日本において最も有名なのが「野村 ID 野球」がこの野球統計学（セイバーメトリクス）に当たる。野村監督が行ったセイバーメトリクスは、「プロのスカウトのみによる主観的な選手評価に頼るのではなく、客観的なデータに基づき、多額の投資を行うリスクを回避する」ことである。

この野球統計学を日本で初めて本格的に導入し、大きな成功を収めた球団が、「北海道日本ハムファイターズ」である。ベースボール・オペレーション・システム[1]（球団の保有している選手の能力・立ち位置を見えるか（数値化）して、分類している IT データベースのこと）に多額の投資をし、選手の能力を出来るだけ細分化して数値評価に置き変えている。ただこれはあくまでもシステムであり、全てのチームがこれを導入しても必ずしも上手いくとは限らない。大事なものは「活用できる力」である。

2013 年度のペナントレースは低迷していたが、その問題として考えられるのが「監督」

だろう。これは、原作の中でも実際にあった。ゼブラルマネージャーが獲得してきた選手だとしても最終的な起用の権限は監督にある。その問題を解決するために日本ハムは「監督に最小限の権限」しか与えない。日本ハムが目指すチームとして「編成部が導き出したチームや選手の育成プランをきちんと実践し、なおかつコーチや監督がプラスアルファの効果を生み出し、そして監督やコーチに依存することのないチーム」が挙げられる。

## 2.5 マネー・ボールがサッカー界に与えた影響

### 2.5.1 選手の獲得方法について

まず、選手の獲得について説明する。

プロ野球の場合は、ドラフト制度により所属チームが決まる。また、外国人選手の登録枠も制限がある。それに対して、Jリーグの場合は、試合に出場できる人数制限はあるものの、外国人選手も含めてすべて自由競争での獲得となっている。そのため、選手が所属するチームを決めることができる。選手がチームを選ぶ基準として以下の例が挙げられる。

- ・ チームの基盤がしっかりしている
- ・ 選手の質が高く、魅力的なサッカーをする
- ・ 自分を高く評価してくれる（高年俸）

等の理由により、将来有望な選手はJ1の強豪チームに入るのが自然の理となっている。

そこで、マネー・ボール理論で考えてみる。選手獲得の際に重要とされている資質の評価方法を、従来とは違う視点から見るということである。サッカーというのは、「点を取られなければ負けないスポーツ」、「試合終了時に1点でも多く得点していれば勝つスポーツ」という原点がある。その原点に立ち返り、今まで評価してきた判断基準（つまり、背が高い、足が速い、テクニックがある）を強豪チームを作るにあたって解釈を変えてみる。

野球を例にしてみる。硬式野球の経験が全くないソフトボールの選手をドラフトで指名した日本ハムが、「従来の常識を一度疑ってみて、素質で選手を獲得する」というスタンスを取ってチーム編成を行っていた。現代のような情報化社会では、どのチームも同じようなスカウティングのノウハウや世界中にちらばる選手の情報を「データ」として持っているもおかしうはないだろう。

しかし、その「データ」をどの角度から眺めるかはそれぞれのチームのセンスで、大きな違いが生まれるのは間違いないだろう。チームが飛躍する時、それは「ローコスト・ハイパフォーマンスが実現した」時である。簡単に説明すると、「安く獲得した選手が、思った以上にチームに貢献してくれる」状態が、一度に複数起こった時に、そのチームは台風の目と呼ばれる。逆に言えば、「今まで実績で選手を獲得したものの、すでにその力が落ちている」ような場合、「ハイコスト・ローパフォーマンス」という状態になってしまう。

### 2.5.2 野球にあってサッカーにはあまりないものについて

以心伝心という言葉がサッカーでいうとアイコンタクトと言い換えることができる。ゲームが動いている瞬間瞬間の（次のプレーを判断する時間）短い時間の中では、たとえ有名な選手であったとしてもパスミスをするのは当たり前のことである。ただ、サッカーの試合では、常にボールがピッチの中で動いている訳ではなく、リスタートの瞬間というのが多々あります。その例としていくつか挙げる。

- ・ 試合開始時のボールが静止している瞬間
- ・ 点が入ったり、取られたりした後のキックオフ
- ・ ファウルを取られた後の直接・間接フリーキックの場面
- ・ PK の場面
- ・ ボールがタッチラインを割って、スローインをする場面
- ・ ボールがゴールラインを割って、ゴールキックをする場面

等が挙げられる。このような場面に野球のようなサインプレーでリスタートすることによりチャンスが広がるのではないだろうか。

### 2.5.3 マネー・ボール理論を取り入れ成功した例

マネー・ボール理論を取り入れて成功を収めたチームがいる。それは、プレミアリーグ（イングランドのリーグ）のニューカッスルである。プレミアリーグは全 20 チームが所属していてその中にマンチェスター・ユナイテッドやチェルシー、アーセナルといった経営規模が大きいクラブには足元にも及ばない。会計・監査法人大手のデロイト社による「クラブ長者番付[2]」によれば、ニューカッスルの昨季売り上げは約 9800 万ユーロ（約 105 億円）である。今季のリーグ優勝を争うマンチェスターの両雄や、残る CL 出場 2 枠を争うロンドンの 3 強の数字には一桁足りないのである。加えてニューカッスルは、3 シーズン前に 2 部リーグに降格まで経験している。それにもかかわらず、今季のニューカッスルは 7 位という結果で終わり、資金面では上にいるリバプールよりも良い成績で終えることができた。

ニューカッスルは、リバプールに多くの主力選手を引き抜かれながら、リバプールが補強に費やした資金の 5 分の 1 以下しか費やさなかった。リバプールの米国人オーナーは、野球界で成功した「マネー・ボール理論」のコンセプトの信望者として知られている。そのモットーは、「過小評価されている選手に本来の価値を見出す」ことである。ニューカッスルは、この点においてもリバプールを凌いでいると言えるだろう。

## 参考文献

[1] BOS (ベースボールオペレーションシステム) - Let it be - JUGEM. 2013.

[https://www.google.co.jp/url?sa=t&rct=j&q=&esrc=s&source=web&cd=3&cad=rja&uact=8&ved=0CCsQFjAC&url=http%3A%2F%2Fletitbe.jugem.cc%2F%3Fleid%3D1531&ei=dwo\\_VIPaleHUmG\\_W\\_voLgAg&usq=AFQjCNGvU7yL2RN2wRtshsBRXNUUgvq3Q&sig2=tX-6O9GnyozTEbNWzCn0iw](https://www.google.co.jp/url?sa=t&rct=j&q=&esrc=s&source=web&cd=3&cad=rja&uact=8&ved=0CCsQFjAC&url=http%3A%2F%2Fletitbe.jugem.cc%2F%3Fleid%3D1531&ei=dwo_VIPaleHUmG_W_voLgAg&usq=AFQjCNGvU7yL2RN2wRtshsBRXNUUgvq3Q&sig2=tX-6O9GnyozTEbNWzCn0iw)(参照 2014-7-15)

[2] デロイト・フットボール・マナー・リーグ

<http://ja.wikipedia.org/wiki/%E3%83%87%E3%83%AD%E3%82%A4%E3%83%88%E3%83%BB%E3%83%95%E3%83%83%E3%83%88%E3%83%9C%E3%83%BC%E3%83%AB%E3%83%BB%E3%83%9E%E3%83%8D%E3%83%BC%E3%83%BB%E3%83%AA%E3%83%BC%E3%82%B0>(参照 2014-8-2)

## 第3章

### オープンソースソフトウェアの概要

## 第3章 オープンソースソフトウェアの概要

### 3.1 本章の構成

本章では本論文の調査対象であるオープンソースソフトウェアの概要について記述する。

### 3.2 オープンソースとは何か

オープンソースとは、プログラムのソースコードを無償で一般に公開し、誰でもそのソフトウェアの改良、再配布の自由を認めることを指す。多数の個人が浸透性をもってソフトウェアの開発、流通に携わることにより、より高品質で柔軟かつ廉価なソフトウェアの開発を実現するための概念である[1]。オープンソースは、**Open Source Initiative**（オープンソース・イニシアティブ）によって策定された定義によれば、プログラムのソースコードが入手可能であることやソフトウェアの変更や派生品を許可すること、特定グループを差別したり利用分野を制限しないことなどの条件を満たす必要がある。パブリックドメインソフトウェアとは異なり、著作者の著作権を保つことができる。オープンソースの思想では、プログラムのソースコードを多くの人々に吟味させて、改良や拡張を促すことで、ソフトウェアの機能・性能が向上されていくことが期待できる。

オープンソースで開発・提供されるソフトウェアは、オープンソースソフトウェアと呼ばれ、利用形態まで含めれば、「利用者が一定の条件のもとで、自由にソースコードを利用、複写、改変、再頒布できるソフトウェア」と定義されるだろう。オープンソースソフトウェアでは、オリジナルのソースコードは無償で一般に公開されるが、OSS 開発者によって定められた条件下であれば、ソースコードの入手者はソースコードを自由に利用、複写、改変、再頒布することが可能である。このオープンソースソフトウェア開発者により定められた条件は、オープンソースソフトウェアライセンスと呼ばれ、オープンソースソフトウェアの流通過程において非常に重要な役割を果たしている。

また、オープンガバメントやオープンデータなど、個人や機関が保有するデータやアプリケーションなどのリソースを「オープン」にすることにより、リソースの多様な活用を促し、イノベーションを創出しようとする取り組みが各所で行われているが、こうした取り組みの根底にあるものもオープンソースであり、オープンソースソフトウェアはこの概念が速く適用された代表的なものと言えるだろう。

### 3.3 オープンソースの定義

オープンソースは、単にソースコードへのアクセスを意味するものではない。オープンソースソフトウェアの配布条件は、以下の基準を満たす必要がある。

表 1 オープンソースソフトウェアの配布条件

	内容
Free Redistribution (自由な再頒布)	ライセンスは、複数の異なるソースからプログラムを含む集計ソフトウェア配布の構成要素としてのソフトウェアを販売したり配ったなら任意のパーティーを制限してはならない。ライセンスは、そのような販売のための使用料やその他の手数料を必要としないものとする。
Source Code (ソースコードの開示)	プログラムは、ソースコードが含まれている必要があり、ソースコード内の配布だけでなく、コンパイルされた形式を許可する必要がある。製品のいくつかのフォームは、ソースコードと一緒に配布されない場合は、無料でインターネットを介してダウンロードし、好ましくは合理的な再生産原価を超えないためのソースコードを取得する広く報道手段があるに違いない。故意難読化されたソースコードは許可されない。そのようなプリプロセッサや変換の出力のような中間形式は許可されない。
Derived Works (派生著作物に関するライセンス)	ライセンスは、改変された派生作品を許可する必要がある、それらが元のソフトウェアのライセンスと同じ条件の下で配布されるようにする必要がある。



<p><b>Integrity of The Author's Source Code</b> (著作権のソースコードの完全性)</p>	<p>ライセンスは、改変された形態で配布されてからソースコードを制限することができる唯一のライセンスである。ビルド時にプログラムを修正することを目的としたソースコードに「パッチファイル」の配布を許可している。ライセンスは、明示的に変更されたソースコードから構築されたソフトウェアの配布を許可する必要がある。ライセンスは、オリジナルのソフトウェアとは異なる名前やバージョン番号を選ぶために派生作品を必要とする場合がある。</p>
<p><b>No Discrimination Against Persons or Groups</b> (個人やグループに対する差別禁止)</p>	<p>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可されていない。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。</p>
<p><b>No Discrimination Against Fields of Endeavor</b> (利用分野に対する差別禁止)</p>	<p>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。</p>
<p><b>Distribution of License</b> (ライセンスの継承)</p>	<p>プログラムに付随する権利は、プログラムがそれらの当事者による追加ライセンスの実行を必要とせずに再配布されすべてに適用されなければならない。</p>
<p><b>License Must Not Be Specific to a Product</b> (特定製品に特化したライセンスの禁止)</p>	<p>プログラムに付随する権利は、特定のソフトウェア配布のプログラムの幸福の一部に依存してはならない。</p>

<p><b>License Must Not Restrict Other Software</b>  (他のソフトウェアを制限するライセンスの禁止)</p>	<p>ライセンスは、ライセンスされたソフトウェアと一緒に配布される他のソフトウェアに制限を置かなければならない。プログラムは、その分布から抽出され、プログラムの使用許諾契約の条項内で使用または配布されている場合は、プログラムが再配布されすべての関係者は、元のソフトウェアの配布と伴わせて付与されているものと同じ権利を持つべきである。</p>
<p><b>License Must Be Technology-Neutral</b>  (テクノロジーニュートラルなライセンス)</p>	<p>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 での分布、ミラーサイトからの分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があり、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険が発生してしまう。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。</p>

代表例に、LinuxOS などがある。Linux の場合、インターネットなどで取得し、対価を支払わずに利用できるなどの利点がある反面、商業利用を目的に改変した場合、改変部分の開示を求められることがある。

### 3.4 オープンソースソフトウェアの種類

オープンソースソフトウェアは、今や非常に幅広い領域をカバーしており、その種類は多様である。代表的なオープンソースソフトウェアには、Ruby, Perl, Python などのプログラム言語から、Linux や Solaris といった Operating System, Apache HTTP Server や nginx などの Web サーバーソフトウェア, Apache Tomcat や JBoss などのアプリケーションサーバソフトウェア, そして Mozilla Firefox や Apache Open Office などのエンドユーザアプリケーションがあり、これを見るだけでもオープンソースソフトウェアは多数存在することがわかるだろう。また、オープンソースソフトウェアである Linux をとってみても、Red Hat ブランドの Linux ディストリビューションや Android など Linux から派生したオープンソースソフトウェアも存在し、その種類は非常に多い。

以下では、代表的なオープンソースソフトウェアを分野別にまとめた。

表 2 オープンソースソフトウェアの種類

分野	オープンソースソフトウェアの種類
プログラム言語	Java, Ruby, Perl, Python, PHP, R (R 言語) など
OS (Operating System)	Linux (Fedora, CentOS, Ubuntu 含む) Solaris, Android など
ツールキット, フレームワーク	GNOME, KDE, LXDE など
仮想化環境	Qt, GTK+など
統合開発環境 (IDE)	KVM, Xen など (デスクトップ仮想化にも対応)
Web サーバー	Eclipse, NetBeans, WideStudio など
アプリケーションサーバー	Apache HTTP Server, ngx など
データベースサーバー	MySQL, PostgreSQL, Firebird, OpenLDAP, Apache Derby など
データ処理	Hadoop (データ分散処理技術), HBase, Cassandra, Redis (非構造型データベース)
拡張機能	Iptables, OpenSSL (SSL アクセス認証) ip6tables (パケットフィルタリング/NAT) TCP Wrapper (ネットワークフィルタリング)

特定アプリケーション向けサーバー	Exim, Sendmail, Postfix (電子メールサーバ／メール転送エージェント) Samba (ファイルサーバー) Squid (プロキシサーバー) ZFS (ファイルシステム)
アプリケーション連携機能	Amanda (バックアップ), Liferay, JBoss Portal, eXo (ポータル) LISM (アカウント ID 管理) Open AM (SSO : シングルサインオン) Pentaho, Plone, Drupal, Jaspersoft (データ解析／BI : ビジネスインテリジェンス) DotNetNuke (検索エンジン) Selenium (ウェブアプリケーションテスト)
アプリケーション	Apache OpenOffice, Compiere, eGroupWare, LibreOffice (オフィススイート) ADmpiere (EPR アプリケーション) Scalix Zimbra (グループウェア) dotProject (プロジェクト管理) Mozilla Firefox (ウェブブラウザ) Mozilla Thunderbird (電子メールアプリケーション) osCommerce (E コマースサイト構築) SugarCRM (CRM アプリケーション) SalesLabor (CRM アプリケーション) WordPress (ブログプラットフォーム) VLC (メディア再生プレイヤー)
その他	Git (ソースコードバージョン管理) jQuery (JavaScript ライブラリ)

上記にある通り、オープンソースソフトウェアはその種類の多さもさることながら、今では特定企業が開発したプロプライエタリとしてのソフトウェアに代わり、業務に利用されるようなケースも多くみられるようになっており、システム開発の様々な分野においてそのプレゼンスは強まっている。

なお、ソフトウェアではないが、ウェブサイト上のコンテンツをオープンソース化するという取り組みがある。代表的な例として、インターネット上の百科事典サイト Wikipedia

であり、同サイト上のテキストコンテンツは、Creative Commons というライセンスのもと、誰でも自由に編集したり、複製したり、頒布したりすることができる形となっている。他に、地図データベースをオープンソースで作成するプロジェクト OpenStreetMap[2] も知られている。

### 3.5 オープンソースの今後

オープンソースは、システム開発、運営負担が大きい大規模システムを中心に、引き続き主要なソフトウェアリソースとして活用されていくと考えられる。特に現在の IT 業界のトレンドであるクラウド、ビッグデータ、オープンガバメント、そしてソーシャルメディアなどの分野では、大量のデータの利用・管理などが重要な課題となっており、それを支えるシステム基盤は大規模化する一方であることから、オープンソースソフトウェアの活用が拡大していく可能性が高いだろう。ただし、その一方で企業側に「オープンソースは使いにくい」といった意識がある点是否めないだろう。以前問題視されていた「無償ソフトウェアのためサポートが得られない」という問題は、商用サービスの展開により解決しつつあるが、もう一つの問題点として、オープンソースソフトウェアの利用の際には「オープンソースソフトウェアライセンス」に従う必要があり、このライセンスの遵守や取り扱いが煩雑であるため、利用しづらいと考える企業が増える可能性がある。

ここで、オープンソースソフトウェアとオープンソースソフトウェアライセンスについて再確認してみる。オープンソースソフトウェアとは「ソフトウェアの利用者が一定の条件のもとで、自由にソースコードを利用、複製、改変、再頒布できるソフトウェア」でのことであり、ここで言う一定の条件にあたるのがオープンソースソフトウェア開発者の定めるオープンソースソフトウェアライセンスとなる。

オープンソースソフトウェアライセンスは、基本的に各ソフトウェア開発者が自由に作成・適用できるものであり、GNU General Public License, BSD License, Apache License, Commn Public License など、現在 100 以上存在していると言われている。しかし、ライセンスの内容は非常に複雑であり、個人レベルの開発者が独自のライセンスを作成することは難しいのだ。そのため、多くの開発者は、オープンソース文化の啓蒙を目的に設立された国際非営利組織 Open Source Initiative（以下 OSI）やオープンソースソフトウェアを推進する非営利組織 Free Software Foundation（以下 FSF）が認定したライセンスを流用するケースが多くなっている。

現在のオープンソースソフトウェアライセンスのほとんどは、「コピーレフト」と呼ばれる概念を何らかの形で含んでいることが多いのだ。「コピーレフト」とは、著作権者が開発した（オープンソースソフトウェアやソースコード）に対する権利を保有したまま、利用者に著作物を複製、改変、再頒布する自由を与えるものであるが、その一方で、著作物が複製、改変、再頒布される形で派生物（二次的著作物）の頒布者に対しても、もともとと全く同じ条件で派生物を頒布することを義務付けるものである。著作物が頒布され続ける限

り、制限なく適用され続けるほか、ソフトウェア利用者に対して、利用者がソースコードを改変した際に、改変部分のソース公開までを義務付けたり、ソフトウェア利用者がソースコードを他のソフトウェアのコードと組み合わせた際に、他のソースコードの公開までが必要となったりと、著作物の流通過程における扱いを厳格に定める概要となっている。

代表的なコピーレフトライセンスとして、Free Software Foundation が作成した GNU General Public License (GPL) がある。以下に GPL v3 (以下 GPL) の概要をまとめる[3]。

- ライセンシーは、頒布するオープンソースソフトウェアに GPL を適用しなければならない。
- ライセンシーは、オープンソースソフトウェアをオブジェクトコード形式で頒布する際、対応するソースコードを頒布先に対して後悔しなければならない。
- ライセンシーは、オープンソースソフトウェアに改変を加えて頒布する際、改変を加えた事実及び日付を明確にしなければならない。
- ライセンシーは、オープンソースソフトウェアをソースコード形式で頒布する際、GPL の本文を明示しなければならない。
- ライセンシーは、追加的条項を加えることにより、GPL に例外を設けることが出来る。
- ライセンサは、頒布するオープンソースソフトウェアに自身の特許が含まれる場合、ライセンシーに対して当該特許を無償でライセンス付与しなければならない。
- ライセンシーは、頒布先に対して、頒布するオープンソースソフトウェアに含まれる自身の特許に関する特許侵害訴訟を起こしてはならない。
- ライセンサが差別的な特許契約を締結した際、ライセンシーにも当該特許契約が付与される。
- ライセンサは頒布するオープンソースソフトウェアに関して、いかなる保証も提供しない。
- ライセンサは頒布したオープンソースソフトウェアが引き起こす損害に対して、一切責任を持たない。
- ライセンシーは、オープンソースソフトウェアを頒布する際、著作権および無保証の旨が記載された定型文を明示しなければならない。

逆に、このコピーレフトの概要を含まないライセンスもある。例えば Apache License などは、オープンソースソフトウェアの利用者に改変部分のソースコードの公開までは義務付けておらず、ライセンシーには、利用するソースコードを他のソフトウェアのソースコードと組み合わせた際に他のソースコードを公開する義務も発生しない。

同ライセンスの概要は以下の通りである。

- ライセンシーは、オープンソースソフトウェアをソースコード形式で頒布する際、ライセンス本文・著作権・特許・商標・帰属についての周知を明示しなければならない。
- ライセンシーは、オープンソースソフトウェアに改変を加えて頒布する際、改変加えた事実をわかりやすく周知しなければならない。
- ライセンシーは、オリジナルオープンソースソフトウェアに帰属周知が含まれている際、同周知をオープンソースソフトウェアに含まなければならない。

- ライセンサは、頒布するオープンソースソフトウェアに自身の特許が含まれる場合、ライセンシーに対して当該特許のライセンスを付与しなければならない。
- ライセンシーが誰かを特許違反で訴えた場合、ライセンシーのライセンサより付与された特許ライセンス権利が失効する。
- ライセンサは頒布するオープンソースソフトウェアに関して、いかなる保証も提供しない
- ライセンサは頒布したオープンソースソフトウェアが引き起こす損害に対して一切責任を持たない。
- ライセンサは、オープンソースソフトウェアを頒布する際、著作権および無保証の旨が含まれた定型文を表示しなければならない。

なお、今後、オープンソースソフトウェアの利用拡大が見込まれるクラウド、ビッグデータ、オープンガバメントの分野についてみると、関連オープンソースソフトウェアのライセンスは、次のような状況となっている。

- クラウドコンピューティング分野: CloudStack (Apache License), OpenStack (Apache License), Eucalyptus (GPL), CloudFoundry (Apache License), OpenShift (Apache License)
- ビッグデータ分野: Hadoop (Apache License), Cassandra (Apache License), HBase (Apache License)
- その他: Git (GPL), Drupal (GPL)

このように、企業戦略に関連するシステムの構築に利用されるオープンソースソフトウェアの多くは比較的制限が緩やかな Apache License が適用されており、企業にとって利用上の障害が引き下げられていると言える。一方で、オープンガバメントで利用されているような Git や Drupal といったコンテンツ管理向けオープンソースソフトウェアについては、適用される対象がオープンな取り組みであることから、コピーレフトが明確に規定される GPL のような厳格なライセンスでも問題にならないものと考えられる。

また、開発しやすさとは相反するが、オープンソースソフトウェアライセンスの厳しさは訴訟のリスクを下げる可能性がある。プロプライエタリなソフトウェア開発企業が基本的な特許を押さえているデータベースを複製していないことを保証する手立てではなく、オープンソースソフトウェアの（商業）利用を行う企業は、常に特許・著作権侵害の訴訟リスクを負うことに留意すべきである。

これに対して、Apache License の場合、オープンソースソフトウェアをもとにある事業者が自由に改変し、商業化したソフトウェアについて、特許権又は著作権違反の訴えを受けた場合、その事業者がもっぱら訴訟リスクを引き受けることとなるが、これと比較するとソフトウェアをそのままの形で再配布することを義務付ける GPL の場合、訴訟リスクはそのソフトウェアを利用した企業だけにとどまらず、開発コミュニティにも及ぶことから、自然と GPL ライセンスのコミュニティにおいては、他者の知的財産権侵害に対して慎重な開発体制をとらざるを得ないものと想定される。最も、GPL の場合は、対象オープンソースソフトウェアについて商業利用の禁止が規定されることも多い。

オープンソースソフトウェアがもつ訴訟リスクへの対策として、開発者個々がオープンソースソフトウェアライセンスを使い分けていくというも考えられるが、個人でライセンスの内容を吟味する労力は多大であり、かつ開発コミュニティごとにライセンス形態が決まっていることが多く、変更は困難であると想定されることから、それには時間がかかると見込まれるだろう。



#### 参考文献

[1] The Open Source Definition. 2013.

<http://opensource.org/osd>(参照 2014-8-12)

[2] OpenStreetMap.

<http://www.openstreetmap.org/>(参照 2014-8-19)

[3] GNU オペレーティング・システム

<http://www.gnu.org/licenses/gpl.html>(参照 2014-9-4)

## 第4章

### オープンソースソフトウェアの現状

## 第4章 オープンソースソフトウェアの現状

### 4.1 本章の構成

本章では本論文の調査対象であるオープンソースソフトウェアの現状について記述する。

### 4.2 商用ソフトウェアの比較

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。使用にあたっての責任の所在や導入にあ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、領布を行っている関係者を含め、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もと、動作チェックは行っている。しかし、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ぬ。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。これに対して、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通りである。

- 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、ということの意味している。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保証は一切ない。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あ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。さらに言えば Linux 関係の技術者は雇用コストが一般の技術者に比べて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 環境に慣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、これらの例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

「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方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方が有益であるだろう。

#### 4.3 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。

##### 4.3.1 開発者のメリット、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、目的に合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い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である。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の多くが 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オープンソースソフトウェア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の負担を強いることはない。開発者とユーザの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、元をたどれ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 用に試食され、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挙げることができる。一つ目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。二つ目はそれまでの 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見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り開発期間、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できるのである。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をすべて開発者に覆いかぶさるのである。

#### 4.3.2 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に対し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利用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済む。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用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。

#### ・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、いざという時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い
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言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稀である。

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レク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に触る機会は滅多にないと考えて構わないだろう。

#### 4.4 ノマドワーキングとは

オープンソースソフトウェア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、町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には、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。このように社外に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。

##### 4.4.1 ノマドワーカーの事態

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。

##### 4.4.2 ノマドワーキングのメリット、デメリット

###### ・メリット

###### 1. 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。

## 2. 一瞬のひらめきも逃さない

アイデアをメモするだけでなく、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。

## 3. 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湧いてきやすく、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。

## 4. 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。

## 5. 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。

## 6. 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、ちょっとした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挙げてきたが、反対にデメリットを挙げていく。

### ① 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。

### ② 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ものとなってくると、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広報の作業だったり文章を書く仕事をし

ていたり、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うか。

#### 参考文献

[1] NCSA. National Center for Supercomputing Applications.

<http://www.ncsa.uiuc.edu/> (参照 2014-9-9)

[2] Apache Software Foundation .

<http://www.apache.org/>(参照 2014-9-14)

## 第 5 章

### GitHub について

## 第 5 章 GitHub について

### 5.1 本章の構成

本章では本研究で個人の活動ログを調査するにあたり、利用するバージョン管理システムについて、それを利用することが出来る GitHub についての基本知識、またそれらの機能について記述する。

### 5.2 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今週 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、いや、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 とは何なのか？ 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、といった伝聞を聞いたことはあるかもしれない。どちらの説も正しいけど、しかしいずれも、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 を使うためのハブだ。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、中央の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ースを提供する。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)。今年のために 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定利用できる製品だ。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。しかも GitHub が将来、アレクサンドリア図書館と同じ運命を辿ったとしても、世界中に分散している多くのデベロッパたちのラップトップ上にあるローカルなフォークから再建できる。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、驚異的な遺産だ。引用文献[1]

### 5.3 バージョン管理システム

Git とは Git リポジトリというデータの貯蔵庫にソースコードなどを入れて利用する。この Git リポジトリを置く場所をインターネット上に提供しているのが GitHub である。つまり、GitHub で公開されているソフトウェアのコードはすべて Git でされている。Git について理解しておくことは GitHub を使う上で重要である。



## 5.4 GitHub の機能について

### 5.4.1 Git について

「Git」とは、プログラムソースなどの変更履歴を管理する分散型のバージョン管理システムのことである。これは、もともと Linux の開発チームが使用して、徐々に世界中の技術者に広まっていった。

「Git」の最大の特徴として挙げられるのは、分散型の名の通り、ローカル環境（自分のパソコンなど）に、全ての変更履歴を含む完全なリポジトリの複製が作成されることである。これは、各ローカル環境がリポジトリのサーバーとなれるということである。分散型ではない、これまでのバージョン管理システムでは、サーバー上にある 1 つのリポジトリを、利用者が共同で使っていた。このため、利用者が増えると変更内容が衝突したり、整合性を維持するのが大変である。

「Git」では、ローカル環境にもコードの変更履歴を保持（コミット）することができるので、リモートのサーバーに常に接続する必要がなく、ネットワークに接続しなくても作業を行うことが出来るのだ。

こういったメリットが受けて、近年のバージョン管理システムの主流になっている。

#### 5.4.2 リポジトリについて

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す。保存された状態は、内容の変更履歴として格納されています。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ます。

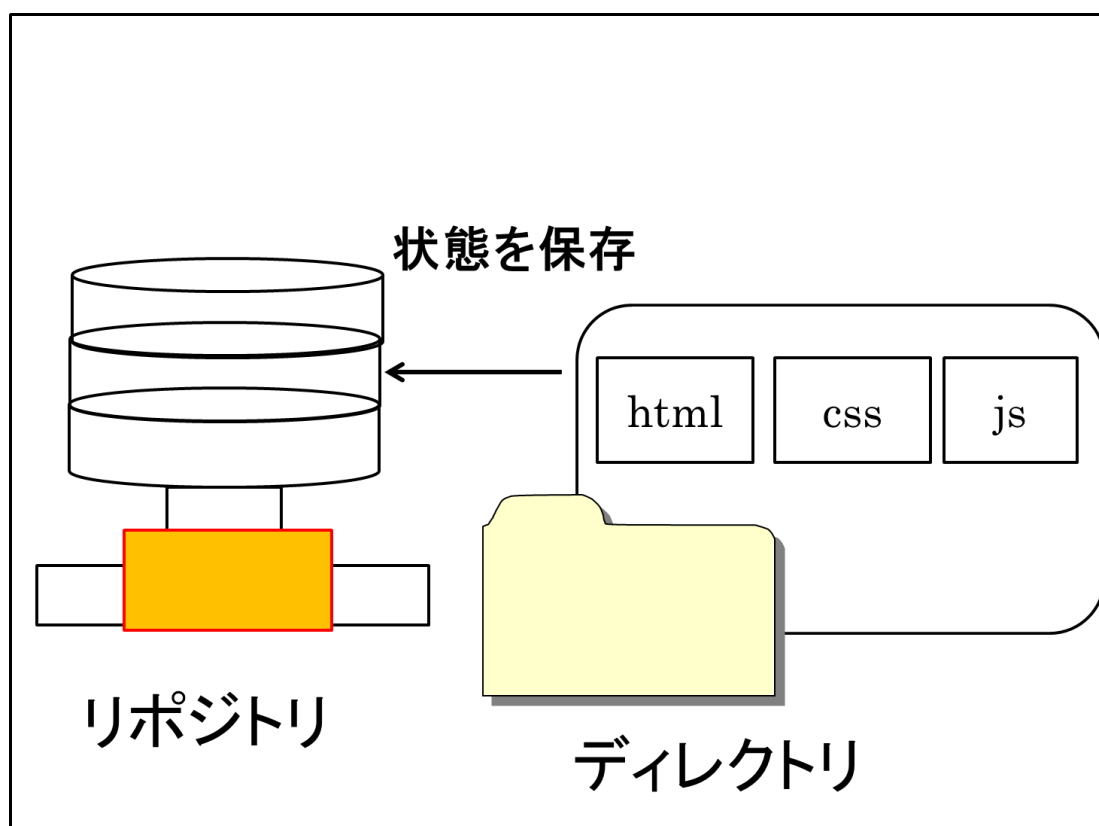


図 1 ディレクトリ内のファイルやディレクトリの変更履歴を記録する例 参考文献[2]

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 種類に分けられます。

- 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です。

- ローカルリポジトリ

ユーザー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です。

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ます。

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す。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ます。引用文献[2]

#### 5.4.3 コミットについて

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います。

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ます。

このコミットは、次の図のように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ます。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ます。

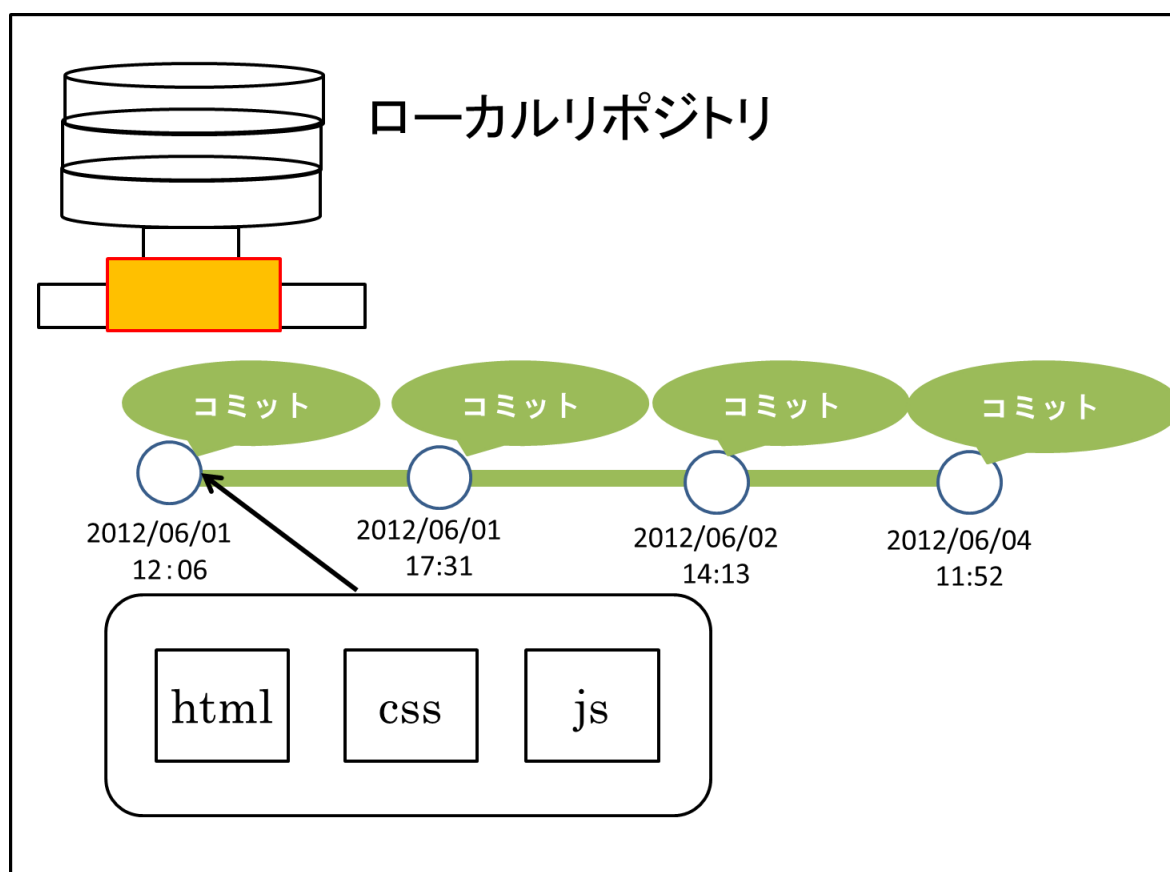


図 2 コミットがリポジトリに格納されている状態 引用文献[2]

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います。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ます。

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にしましょう。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ります。

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ます。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します。

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心がけましょう。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きます。引用文献[2]

#### 5.4.4 リポジトリの共有について

- **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ります。そのために、**Git** ではプッシュ(**Push**)という操作を行います。**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ります。引用文献[2]

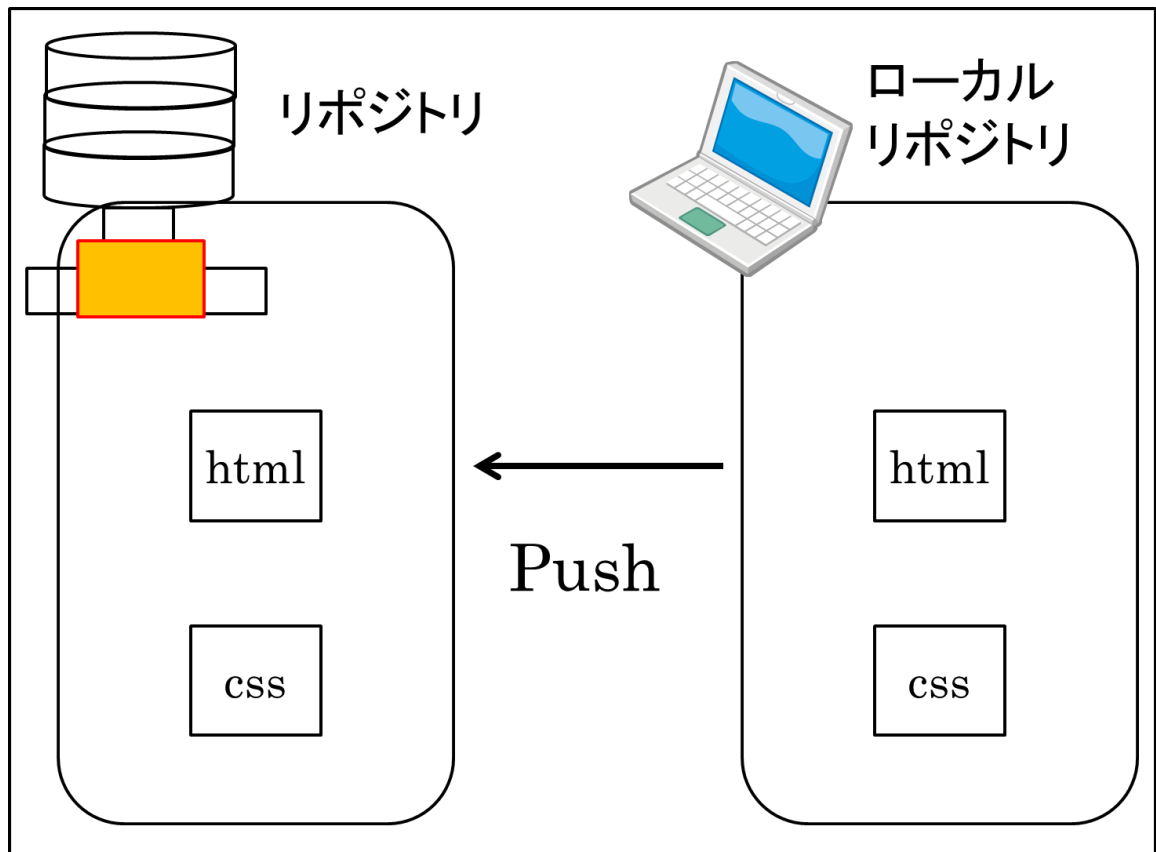


図 3 ローカルリポジトリから **Push** を実行した例 引用文献[2]

- **Clone** : 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ます。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ます。[2]
- **Pull** : 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**)という操作を行います。**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みます。引用文献[2]

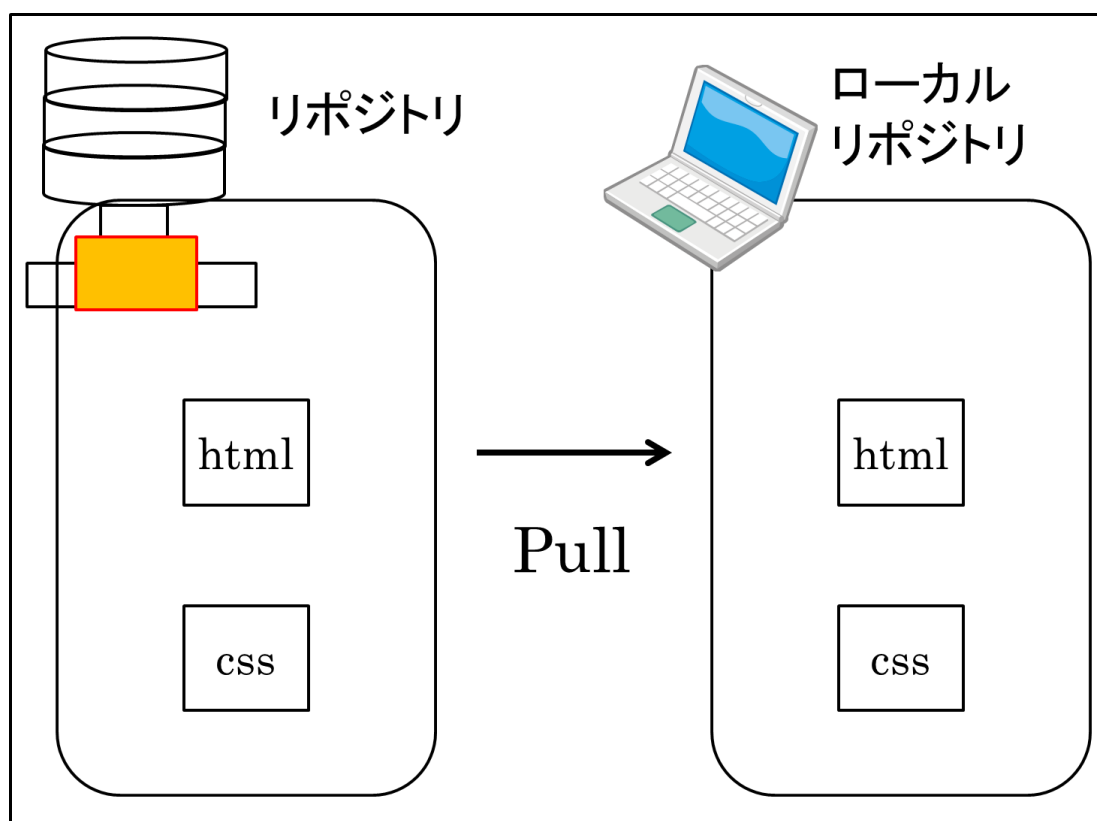


図 4 ローカルリポジトリに **Pull** を実行した例 引用文献[2]

#### 5.4.5 変更履歴のマージについて

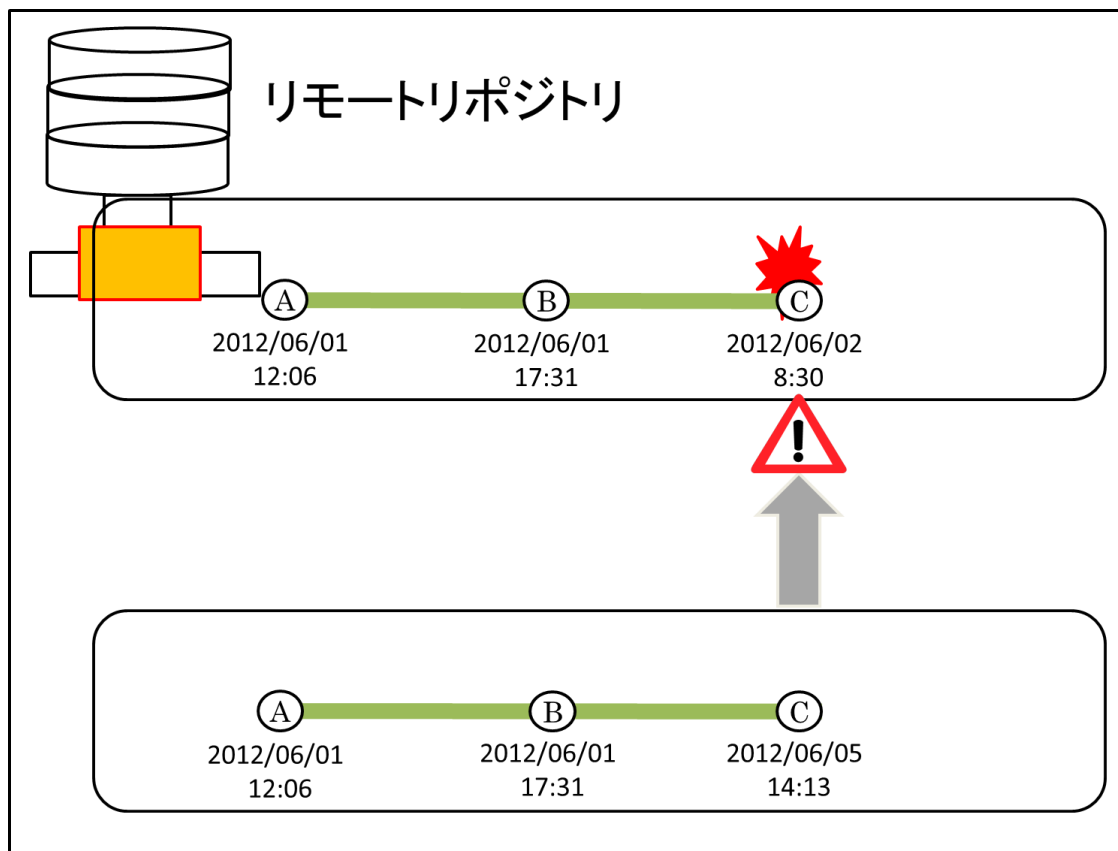


図 5 Push が拒否された例 引用文献[2]

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 が拒否されてしまいます。  
引用文献[2]

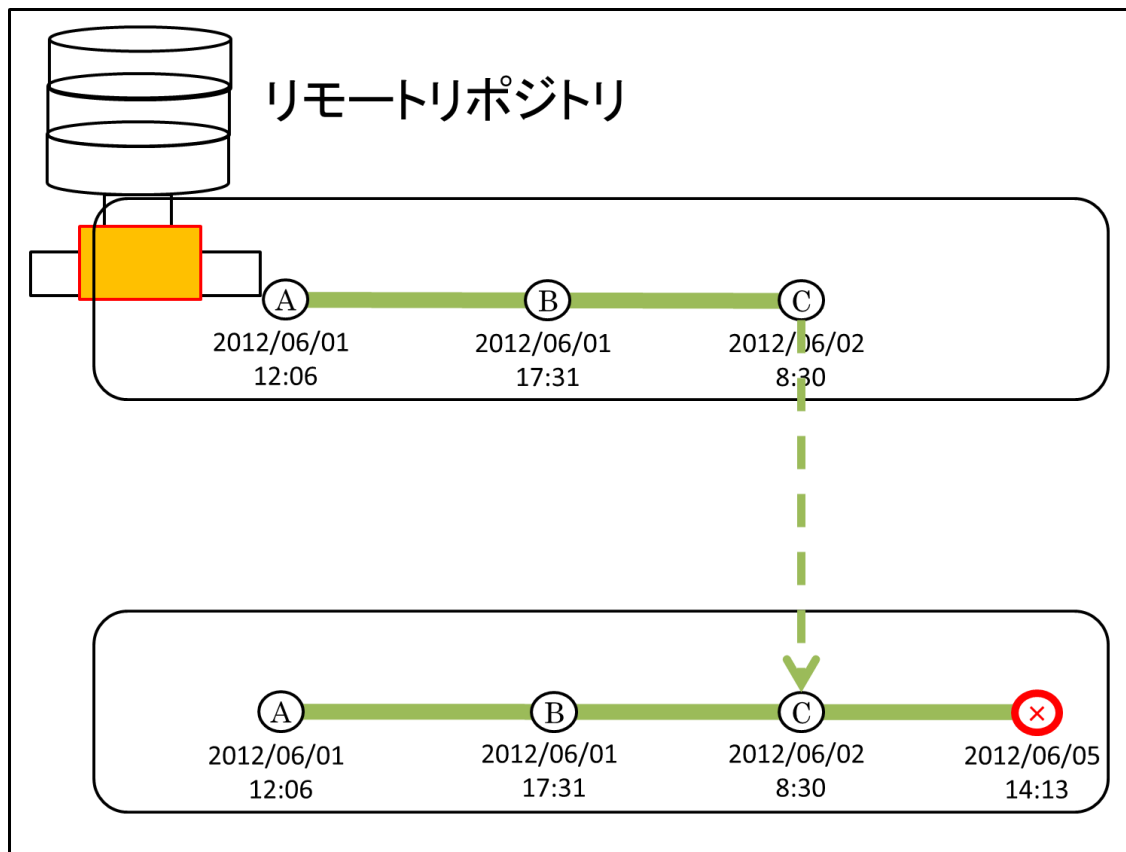


図 6 マージを行わないと Push が拒否される例 引用文献[2]

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 は拒否されます。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です。引用文献[2]

#### 5.4.6 変更履歴の統合「競合の解決」

前ページで説明したように、マージを行うと Git が変更箇所を自動的に統合します。しかし、自動で統合できない場合もあります。

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す。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ます。

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ります。引用文献[2]



#### 5.4.7 ワークツリーとインデックスについて

Git では、Git の管理下に置かれた、みなさんが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びます。

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ます。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です。

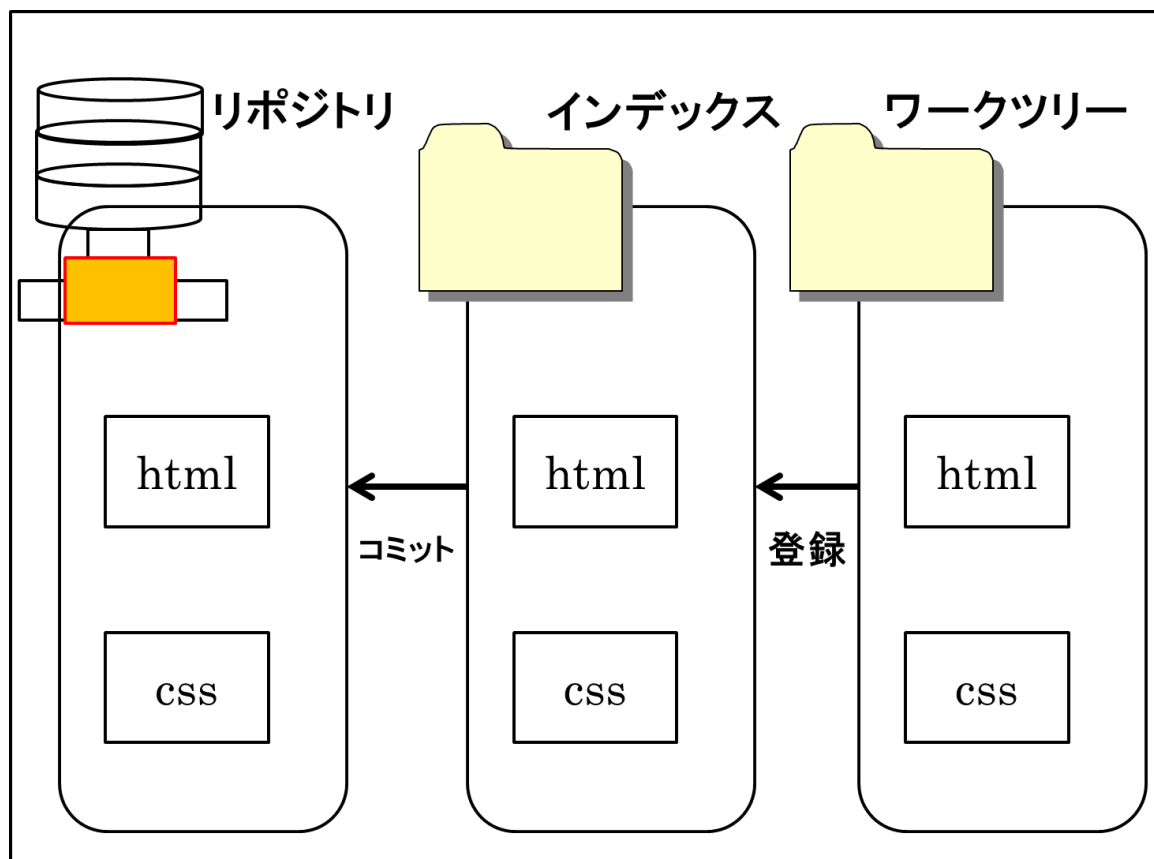


図 7 インデックスに登録してコミットする流れ 引用文献[2]

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ます。

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ります。

このように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ます。引用文献[2]

## 5.5 API について

API (Application Programming Interface, アプリケーション・プログラミング・インターフェース) とは、アプリケーションから利用できる、オペレーティングシステムやプログラミング言語で用意されたライブラリなどの機能の入り口となるものである。主に、ファイル制御、ウィンドウ制御、画像処理、文字制御などのための関数として提供されることが多いのである。

つまり、アプリケーションをプログラムするにあたって、プログラムの手間を省くために、非常に簡潔にプログラムができるように設定されたインタフェースのことである。

### 5.5.1 API を公開する・公開されるメリット

Web 製作者にとっては、他社の膨大なデータベースや機能を無料で利用できるため、Web サイトの開発のコストを大幅に削減できるので、効率的に制作できる。このため、個人や小資本の会社でも、人気の Web サイトを低コストで効率的に作成されることが可能になったのである。

また、API 提供会社にとっては、自社のみでは考え付かないような Web サービスや自社のみでは実現できないような Web サービスを外部の人間が作成してくれるため、API 利用サイトから自社のサイトへユーザーが流入してくるのを期待できる。さらに、自社のデータベースや機能を間接的に自社のサイトを利用してくれるユーザーが増加し、それが結果としてデータを集積するスピードが速まり、競争優位に立つことを期待できるのである。

#### 参考文献

[1] そもそも GitHub とは一体何か？－TechCrunch. 2012.

<http://jp.techcrunch.com/2012/07/15/20120714what-exactly-is-github-anyway/>(参照 2014-10-6)

[2] サルでもわかる Git 入門～バージョン管理を使いこなそう～.

<http://www.backlog.jp/git-guide/>(参照 2014-10-27)

[3] GitHub. GitHub Developer, /, 2013-10-10.

<http://developer.GitHub.com/v3/issues>(参照 2014-10-30)

## 第 6 章

### 開発・調査

## 第6章 開発・調査

### 6.1 開発・調査について

本章では、これまでの調査を含めた研究方法と、現状の進捗における研究状況を報告する。

### 6.2 研究方法

本研究の流れを以下に記す。

1. GitHub 上で行われている OSS 開発のプロジェクトのコミット総数を調査する。
2. 1 で調査したプロジェクトメンバの個人がコミットの追加・修正をした行数を調査する。
3. 2 で得られた活動ログを統計解析手法で分析する。
4. 3 で得られた結果から個人のプロジェクトへの貢献度がどのくらいかなのかを考察する。

### 6.3 研究ツール

#### 6.3.1 Ubuntu とは

Ubuntu (ウブントゥ) とは、コミュニティにより開発されているオペレーティングシステムです。ラップトップ、デスクトップ、そしてサーバーに利用することができます。Ubuntu には、家庭・学校・職場で必要とされるワープロやメールソフトから、サーバーソフトウェアやプログラミングツールまで、あらゆるソフトウェアが含まれています。

Ubuntu は現在、そして将来に渡って無償で提供されます。ライセンス料を支払う必要はありません。Ubuntu をダウンロードすれば、友達や家族と、あるいは学校やビジネスに、完全に無料で利用できます。

私たちは、新しいデスクトップおよびサーバーを 6 ヶ月ごとにリリースすることを宣言しています。これにより、オープンソースの世界が提供する最新の優れたアプリケーションを常に利用できるようにしています。Quantal Quetzal 12.10 2012年10月18日 2014年4月

Ubuntu は、セキュリティに配慮して設計されています。デスクトップおよびサーバーの無償セキュリティアップデートが、少なくとも 9 ヶ月間に渡って提供されます。長期サポート(LTS)版を利用すれば、5 年間に渡りセキュリティアップデート提供されます。もちろん、LTS 版を利用するために追加の費用は必要ありません。すべての人が無償という同じ条件で、私たちの精一杯の成果を利用することができます。Ubuntu を新しいバージョンにアップグレードする場合も、常に無償です。

Ubuntu のインストールイメージにはデスクトップ環境がひと通り含まれています。さらに、オンラインでソフトウェアを追加することができます。

グラフィカルインストーラにより、素早く簡単にインストールして使い始めることができます。標準的なインストールにかかる時間は 10～20 分未満です。十分に高速な環境であれば、インストールが 5 分程度で終了することもあります。

一度システムをインストールすれば、インターネット、ドローイング、グラフィックス、そしてゲームといったアプリケーションがすぐに使えるようになります。

Ubuntu サーバーでは、ユーザーがセットアップしたものだけが動作し、それ以外はインストールされません。引用文献[1]

### 6.3.2 Ubuntu の意味

Ubuntu は、アフリカの単語で「他者への思いやり」や「皆があつての私」といった意味を持ちます。Linux ディストリビューションである Ubuntu は、Ubuntu の精神をソフトウェアの世界に届けます。引用文献[1]

### 6.3.3 Ubuntu の日本語環境

Ubuntu は、できる限り多くの言語に対応すべく国際化が進められており、もちろん日本語での利用も可能です。Ubuntu Japanese Team では、Ubuntu 日本語サポートをより良いものとする活動を進めています。しかしながら、他の言語環境に悪い影響を与えてしまう変更が必要であるなどの理由で、現段階ではオリジナルの Ubuntu に含めることが難しい修正が必要な場合もあります。そこで Japanese Team では、現在のところ Ubuntu に追加できていない修正を加えたパッケージ、および日本語環境に必要とされるパッケージを収録した Remix イメージと仮想ハードディスクイメージを作成・配布しています。この Japanese Team のパッケージを含むイメージを、オリジナルの Ubuntu と区別するために「日本語 Remix」や「日本語 Remix 仮想ハードディスクイメージ」と呼んでいます。引用文献[1]

#### 6.3.4 Ubuntu の特徴

Ubuntu を使って、Web を閲覧したり、メールを読み書きしたり、文書や表計算をしたり、画像を編集したり、その他さまざまなことができます。Ubuntu のデスクトップ CD には、早くて簡単なグラフィカルインストーラが搭載されています。一般的なコンピュータの場合、25 分以内でインストールが完了します。

表 3 Ubuntu の特徴一覧 参考文献[1]

シンプルなデスクトップ	初期状態で、デスクトップにはアイコンがひとつもありません。デフォルトのデスクトップテーマは、目にやさしいものを使用しています。
オフィスアプリケーション搭載	オープンオフィスには、他のオフィススイートと似たユーザインタフェースと機能が備わっています。また、よく使われる主要なデスクトップアプリケーションが含まれています。
ワープロ	ちょっとした手紙から書籍まで、さまざまな文書を編集することができます。
表計算	計算、分析、数表やグラフの作成のためのツールです。
プレゼンテーション	印象的なプレゼンテーションを作成するための、簡単で優れたツールです。
さまざまな形式のファイルを編集可能	マイクロソフトオフィス、ワードパーフェクト、KOffice、StarOffice のファイルを開いて編集することができます。
簡単で手軽なアップデート	アップデートが利用できる時には、タスクバーのアップデートエリアに表示されます。単純なセキュリティフィックスから、完全なバージョンアップグレードまで、このエリアで通知されます。アップデート作業は簡単なので、マウスで何度かクリックするだけでシステムを最新の状態に保つことが可能で
非常に充実したフリーソフトウェアライブラリ	もっと別のソフトウェアが必要ならば、カタログにある何千というソフトウェアパッケージから選ぶことができます。すべての利用可能なソフトウェアは、クリックしていくだけでダウンロードとインストールが可能です。もちろん、これらはすべて完全にフリーです。

ヘルプとサポート	<p>メニューから [システム] - [ヘルプとサポート] を選択するか、<a href="https://help.ubuntu.com/">https://help.ubuntu.com/</a> にアクセスすることで公式ドキュメントを参照することができます。もし、Ubuntu の使い方について質問があるならば、誰かが既に質問していないか調べると良いでしょう。Ubuntu コミュニティでは、ドキュメントを整備しており、あなたの質問の答えを含んでいるかもしれません。あるいは、どこを参照したら良いか分かるかもしれません。</p> <p>Ubuntu コミュニティのチャットやメーリングリストにおいて、多くの言語でフリーサポートを受けることができます。あるいは、Canonical 社や各地のサービスプロバイダより、有償サポートを購入することもできます。</p>
国際化とアクセシビリティ	<p>Ubuntu は、できるだけ多くの人が利用できることを目標としています。そのため、Ubuntu にはフリーソフトウェアコミュニティが提供できる最大限の国際化とアクセシビリティ機構が含まれています。</p>
システム要件	<p>Ubuntu は x86 PC、64 ビット PC で利用することができます。少なくとも 1 ギガバイトの RAM と 5 ギガバイトのストレージスペースが必要です。</p>



### 6.3.5 Ubuntu サーバーの特徴

Ubuntu サーバーは、堅牢なサーバーとして定評のある Debian をベースとし、高い信頼性とパフォーマンス、そして定期的なアップグレードを提供します。

表 4 Ubuntu サーバーの特徴一覧 引用文献[1]

統合されたセキュアなプラットフォーム	<p>ビジネスが成長するにつれ、ネットワークも大きくなります。より多くのアプリケーションが必要となり、多くのサーバーが要求されます。Ubuntu サーバーは、いくつかの共用設定をサポートし、一般的な Linux サーバの配置プロセスを単純化します。これにより、メール、Web、DNS、ファイル共有あるいはデータベースといった一般的なインターネットサービスをうまくまとめたプラットフォームを、早く簡単に新しいサーバーにセットアップすることができます。</p> <p>Debian から受け継いだ鍵となる教訓は、デフォルトでセキュリティを保つことです。Ubuntu サーバーはインストール後どのポートも開いておらず、セキュアなサーバの構築に必要な不可欠なソフトウェアのみが導入されます。</p>
自動 LAMP インストールによる TCO 削減	<p>Ubuntu サーバーのインストールは約 15 分以内で終了し、LAMP(Linux, Apache, MySQL, PHP)サーバが起動して利用可能な状態にできます。Ubuntu サーバーだけのこの機能は、インストール時に利用できます。</p> <p>LAMP オプションにより、LAMP の各コンポーネントを別々にインストールして設定する必要がなくなります。各アプリケーションのインストールおよび設定方法を知っている方が数時間かかる作業を、自動的に行うことができるのです。LAMP オプションを使えば、十分なセキュリティ、インストール時間の削減、設定ミスリスクの軽減を実現することができ、その結果コストの削減につながります。</p>

<p>ワークステーションをアップグレードするコストの削減</p>	<p>Ubuntu サーバーには LTSP(Linux Terminal Server Project)を利用したシンククライアントサポートが含まれています。最新版である LTSP-5 により、単純なインストールと簡単なメンテナンスが提供されています。すべてのデータはサーバーに蓄積され、個々のワークステーションをアップデートし、セキュリティを保つよりも大幅にコストを削減することができます。Ubuntu のシンククライアントサポートによる利点には以下のようなものがあります。</p> <ul style="list-style-type: none"> <li>• 簡単な管理：一つのシステムからすべてのクライアントを管理できます。新しいソフトウェアのインストール、それらの設定変更、サーバー上のあたらしいバージョンへのアップグレード、そしてすべてのクライアントをすぐに最新の状態にすることさえ可能です。全てのクライアントのバックアップを1箇所でもとることができます。</li> <li>• 完全に自動化されたインストールとセットアップ：シンククライアントサーバーをインストールするのは、デスクトップシステムのインストールと同じぐらい簡単です。一度インストールを終えれば、サーバー上で管理作業を行うことなく新しいクライアントを加えることができます。</li> <li>• 共通リソースにより TCO 低減：ハイパワーなデスクトップワークステーションを何台も電源を入れ、一日中アイドル状態にしておくとコストがかさんでしまいます。ハイパワーなサーバーと、低コストのシンククライアントを使えば、パフォーマンスが良い上コストも節約できます。さらにパフォーマンスを上げたいならば、サーバー一台をアップグレードするだけで、クライアントのパフォーマンスも改善されます。</li> <li>• 高速な障害復旧：クライアントシステムが故障しても、他のシステムを使って仕事を継続することができます。特別な設定は必要なく、すべてのユーザーデータと設定は保持されています。</li> <li>• ローカル接続されたデバイス：ユーザーは、シンククライアントに接続されたプリンタ、カメラ、iPod、USB メモリやその他のデバイスに直接アクセスできます。</li> </ul>
----------------------------------	---

### 6.3.6 調査環境構築

本研究では、調査環境の OS は Ubuntu を使用する。

1. curl をインストールする。

```
sudo apt-get install curl
```

Ubuntu に curl をインストールする。

2. GitHub のログイン情報を省略

```
echo 'ユーザ名:パスワード'> github.passwd
```

ログイン情報を入力する作業を省略するため、ユーザ名とパスワードをファイル github.passwd に記述する。

```
chmod 600 github.passwd
```

ファイル github.passwd に読込権限と書き込み権限を付加する。これを行うことにより本人以外はファイル github.passwd を操作することができなくなる。

確認するために以下を行う。

```
cat github.passwd
```

GitHub のユーザ名とパスワードを確認することができる。

3. コミットのデータを取得する。

```
curl -s -u ユーザ名:パスワード "https://api.github.com/repos/ユーザ名/リポジトリ名/commits"
```

4. ユーザ名、パスワードを省略したものを利用するためには以下のように実行する。

```
curl -s -u $(cat github.passwd) "https://api.github.com/repos/ユーザ名 /リポジトリ名/commits"
```

5. API を呼び出し結果の内容を確認するには以下のようにパイプ | を利用する。

```
curl -s -u $(cat github.passwd) "https://api.github.com/repos/ユーザ名/リポジトリ名/commits" | grep title
```

6. Python と HTTP アクセスのための requests をインストールする.

```
sudo apt-get install python python-setuptools
sudo easy_install requests
```

7. api.Py を作業ディレクトリにダウンロードする.

api.Py とは以下のようなプログラムで, データを所得し, ファイルに保存するプログラムである.

```
#!/usr/bin/python
# coding: UTF-8

import sys, json, requests

tmp = open('github.passwd').readline().rstrip('\n').split(':');
username = tmp[0]
password = tmp[1]
#print >> sys.stderr, username, password

url = sys.argv[1]

count = 0
while (url is not None):
    #print url
    r = requests.get(url, auth=(username, password))
    print >> sys.stderr, r.headers['status'],
    for item in r.json():
        count = count + 1
        print json.dumps(item)
    if (r.links.has_key('next')):
        url = r.links['next']['url']
    else:
        url = None
    print >> sys.stderr, count
```

8. JSON 形式のデータを扱うために jq のバイナリを作業ディレクトリにコピーし, jq を実行できるようにするため, 以下を行う.

```
chmod +x jq
```

9. コマンドでグラフを描くために Gnuplot をインストールする.

```
sudo apt-get install gnuplot-x11
```

Gnuplot とは 2 次元または 3 次元のグラフを作成するためのコマンドラインアプリケーションソフトウェアのことであり, 入力した数式等をもとにグラフを作成することができる.

表 5 コマンドの解説

コマンド	解説
curl	プロトコルを用いてデータを通信するコマンドラインツール.
cat	ファイルの内容を閲覧する.
echo	因数に与えられた文字列を表示する.
>	ファイルに出力する.
>>	ファイルの末尾に追記する.
chmod	ファイルやディレクトリのアクセス権を変更する. 600 →所有者にのみ読み込み書き込みの権限がある. +x →実行権限を付加する.
-s	行分割だけを行い、行の結合は行わない.
-u	単語間のスペースを 1 つに、文の間はスペース 2 つに減らす.
	コマンドの出力を次のコマンドの入力として渡す.
grep	文字列を検索する.
awk	Awk 言語使用の宣言.

## 6.4 研究対象

本研究の対象は以下の 5 つのプロジェクトを調査対象とした。

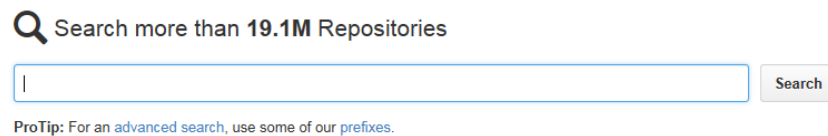
表 6 プロジェクト一覧

プロジェクト名	開発メンバ	総コミット数
square / spoon[2]	30 名	487
stanfy / spoon-gradle-plugin[3]	5 名	81
spoonapps / spoonme[4]	13 名	123
seatgeek / soulmate[5]	10 名	105
spoon / library[6]	16 名	406

## 6.5 調査方法

今回は上記のツールを使用する時間がなかったため以下のように行った。

1. プロジェクトを検索する。



The image shows a search interface with a magnifying glass icon and the text "Search more than 19.1M Repositories". Below this is a search input field containing a vertical bar "|" and a "Search" button. At the bottom, there is a "ProTip" message: "ProTip: For an advanced search, use some of our prefixes."

図 8 検索画面

検索したいプロジェクト名を記入し，検索する。

## 2. プロジェクトメンバの確認をする.

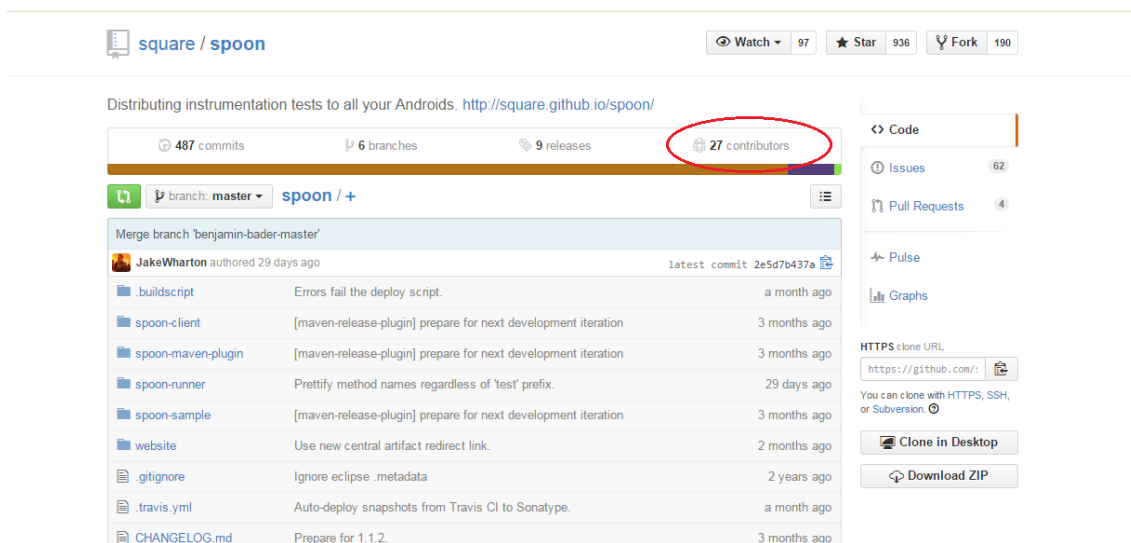


図 9 プロジェクトのメイン画面

プロジェクトに何名参加しているかを確認する.

### 3. プロジェクトメンバのコミット履歴を調査する.

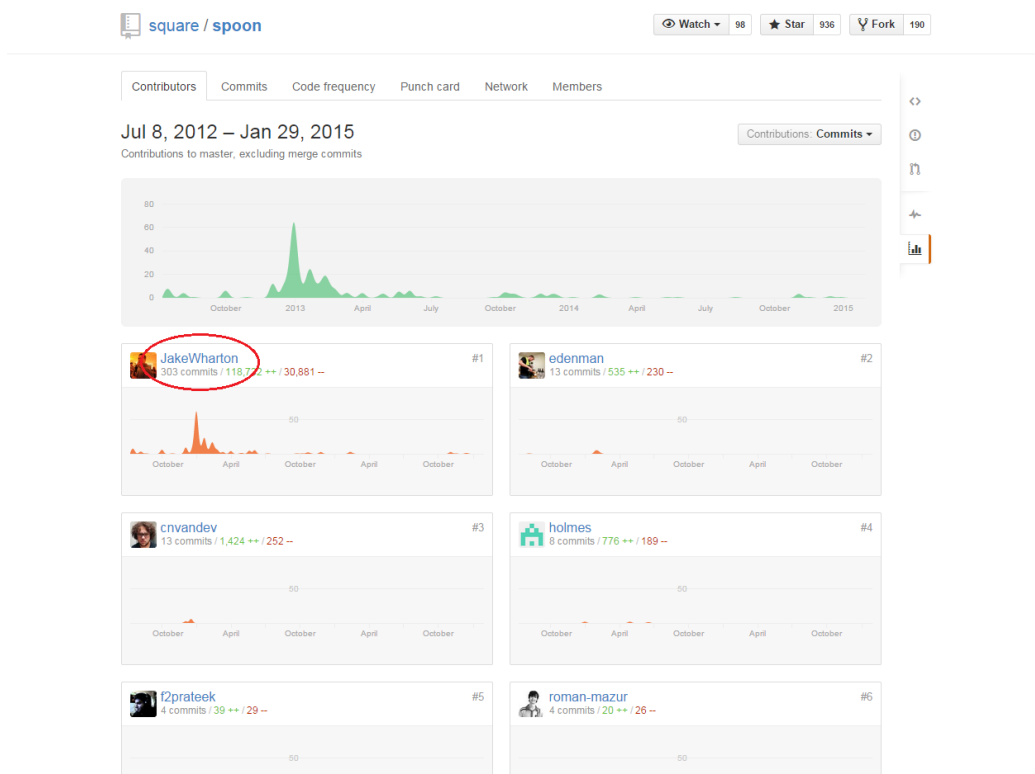


図 10 メンバー一覧画面

○で囲ってあるところから個人がコミットした日付と行数を調査する.



#### 4. 個人のコミット履歴を調査する.

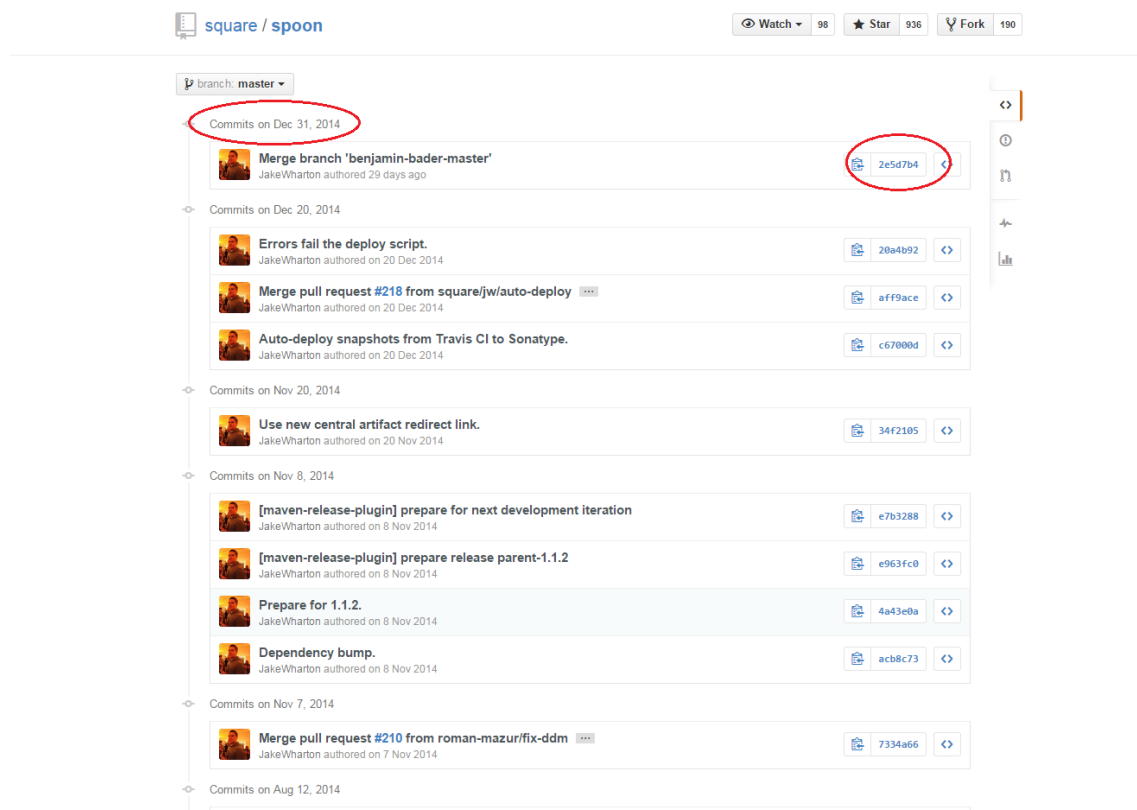


図 11 個人のコミットの履歴画面

○で囲ってあるところからコミットした日付と行数を調査する.

別のやり方もある。

5. プロジェクトのコミット履歴を調査する。

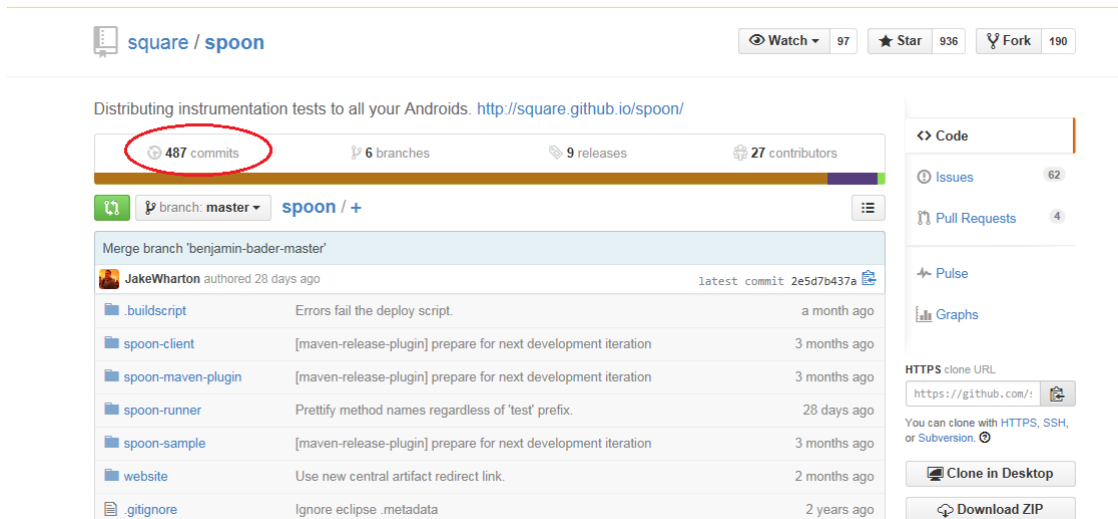


図 12 プロジェクトのメイン画面

○で囲ってある場所から調査する。

6. プロジェクトメンバ個人のコミット履歴を調査する。

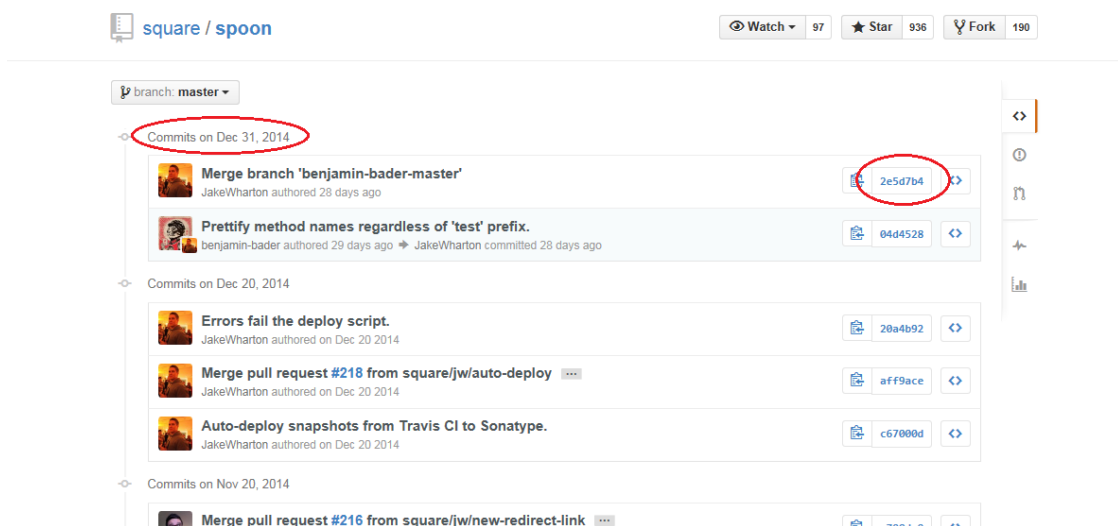
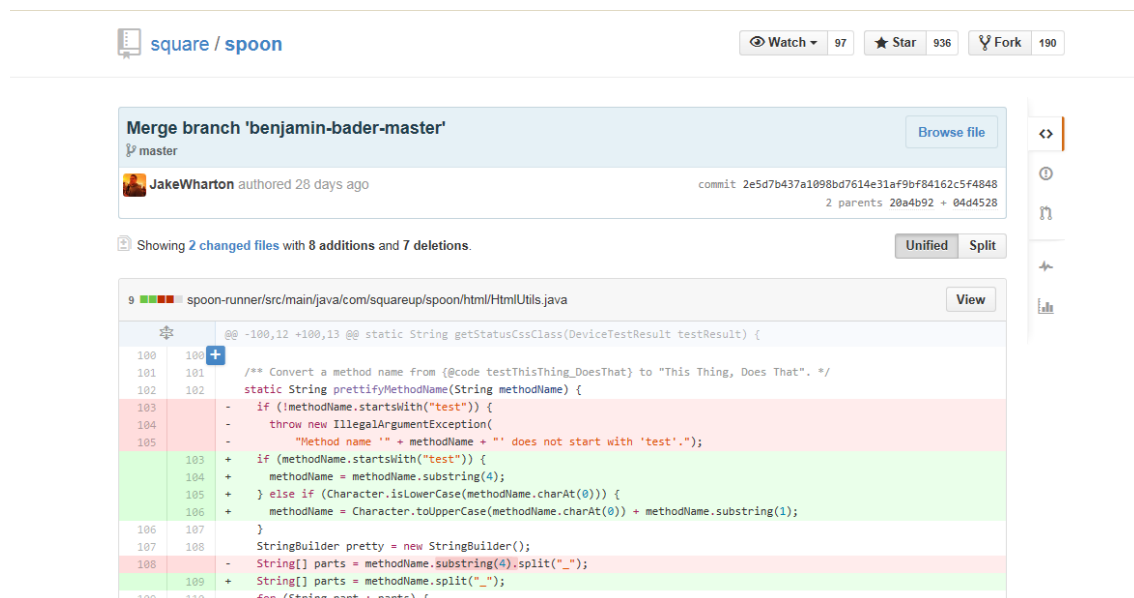


図 13 全体のコミットの履歴画面

○で囲ってあるところからコミットした日付と行数を調査する。

## 7. メンバ個人がコミットで追加・修正・削除した履歴を調査する.



The screenshot shows a GitHub commit diff for the 'spoon' repository. The commit is titled 'Merge branch 'benjamin-bader-master'' and was authored by JakeWharton 28 days ago. It shows changes to the file 'spoon-runner/src/main/java/com/squareup/spoon/html/HtmlUtils.java'. The diff highlights 2 changed files with 8 additions and 7 deletions. The code shows a method 'prettifyMethodName' with changes to its logic, including adding a new condition and removing an old one.

図 14 コミットで追加・修正・削除した画面

削除した行は赤く表示され、追加・修正した行は青く表示される。今回は追加・修正した行数だけを調査する。

コミットで追加・修正・削除した履歴を一人一人でまとめたい場合は、3.4 で収集した方がまとめやすい。

## 参考文献

[1] Ubuntu Japanese Team: Homepage.

<https://www.ubuntulinux.jp/>(参照 2014-11-19)

[2] square / spoon.

<https://github.com/square/spoon>(参照 2014-12-12)

[3] stanfy / spoon-gradle-plugin.

<https://github.com/stanfy/spoon-gradle-plugin>(参照 2014-12-12)

[4] spoonapps / spoonme.

<https://github.com/spoonapps/spoonme>(参照 2014-12-20)

[5] seatgeek / soulmate

<https://github.com/seatgeek/soulmate>(参照 2014-12-20)

[6] spoon / library

<https://github.com/spoon/library>(参照 2014-12-27)

## 第 7 章

### 調査結果・考察

## 第7章 調査結果・考察

### 7.1 本章の構成

本章では前章で記述した調査方法を用いて、GitHub 内のプロジェクトからプロジェクトメンバの活動ログの調査を行った結果を記述し、その結果に対する考察を記述する。

### 7.2 調査結果

実際に調査したプロジェクトにコミットされている行数をどのような割合でメンバが書いているのか円グラフで解説する。

#### 7.2.1 square / spoon

プロジェクトメンバ	コミット回数	コミットで修正・追加した行数
JakeWharton	303	27459
Eric Denman	13	2339
Chris Vandeveld	13	1871
Jason Holmes	8	1852
Dimitris Couchell	4	1475
Edward Dale	3	94
Ian Warwick	1	88
Prateek Srivastava	4	39
Matt Oakes	2	35
Felix Schulze	2	27
brien Colwell	2	24
Peter Miklosko	2	24
Veonua	1	23
Roman Mazur	4	20
James Wald	1	19
Riccardo Civati	1	11
Krzysztof Bielicki	1	9
Ben Bader	1	8
Xavier Ducrohet	1	5
Daniel Lubarov	1	5
Brennan Taylor	1	5
Brian Hoffmann	2	3
matt swanson	1	3

Jesse Wilson	1	3
Simo Kinnunen	1	2
Jonas Alves	1	1
Roman Tsukanov	1	1
Dustin Lam	1	1
Lucas Lampietti	1	1
Jack Harrison	1	1

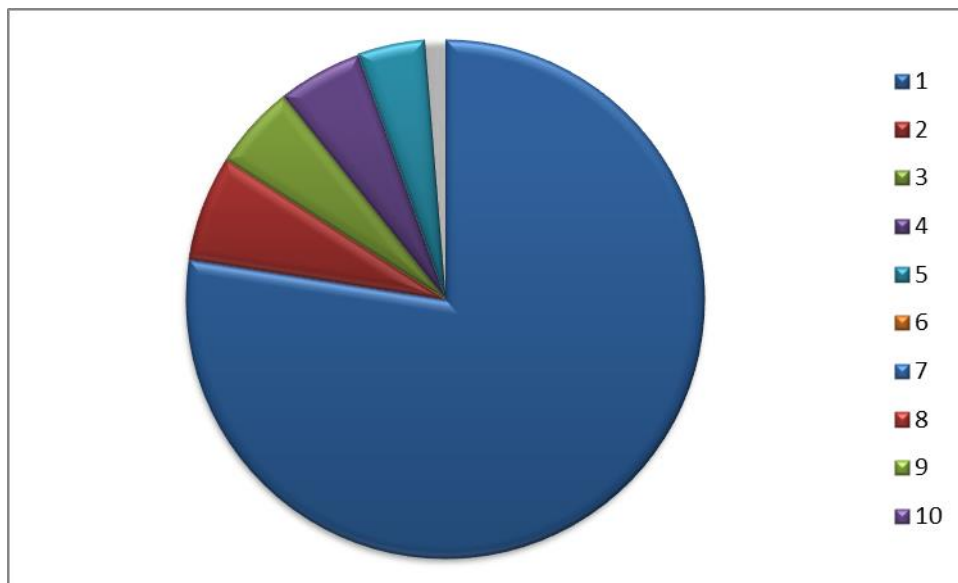


図 15 メンバがコミットで修正・追加した行数の割合

編集回数が多い Jake Wharton は 1 回のコミットで平均約 91 行の追加・修正をしていた。  
編集回数が少ない Jack Harrison は 1 回のコミットで 1 行の追加・修正しかしていなかった。

### 7.2.2 stanfy / spoon-gradle-plugin

プロジェクトメンバ	コミット回数	コミットで修正・追加した行数
roman-mazur	56	1714
Mindaugas Kuprionis	6	253
Ivan Carballo	3	28
acortelyou	1	1
Tomasz Rozbicki	1	1

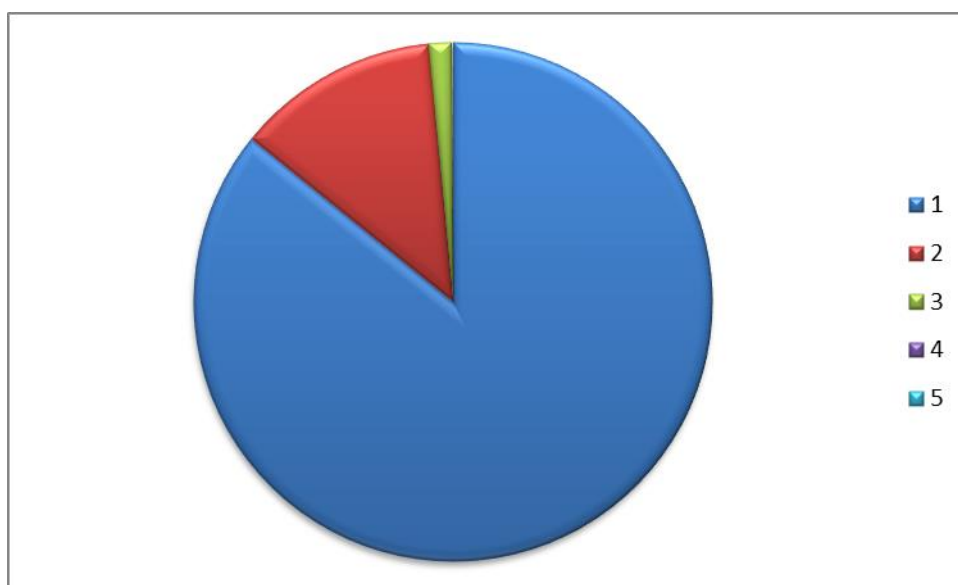


図 16 メンバがコミットで修正・追加した行数の割合

編集回数が多い roman-mazur は 1 回のコミットで平均約 31 行の追加・修正をしていた。編集回数が少ない acortelyou と Tomasz Rozbicki は 1 回のコミットで 1 行の追加・修正しかしていなかった。



### 7.2.3 spoonapps / spoonme

プロジェクトメンバ	コミット回数	コミットで修正・追加した行数
olegsomphane	57	2255
Lee Murphy	3	547
Roman Stoffel	20	259
matt-black2	6	259
Kenji Obata	4	209
Edi Weissmann	5	190
Denny Ferrassoli	1	133
johnsu	1	82
Bartosz Brachaczek	1	75
Victor Churchill	5	72
Edward Evans	1	54
Matt Wrock	4	52
Richard Astbury	1	51

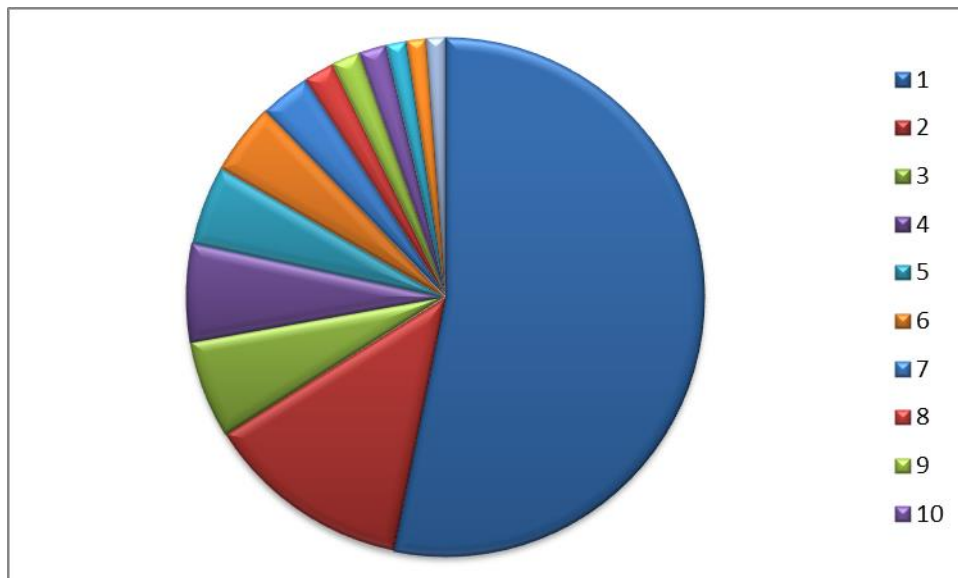


図 17 メンバがコミットで修正・追加した行数の割合

編集回数が多い olegsomphane は 1 回のコミットで平均約 40 行の追加・修正をしていた。編集回数が少ない Richard Astbury は 1 回しかコミットしておらず、その 1 回で 51 行の追加・修正しかしていなかった。

#### 7.2.4 seatgeek / soulmate

プロジェクトメンバ	コミット回数	コミットで修正・追加した行数
Eric Waller	76	1262
Hung YuHei	6	102
David Czarnecki	1	69
Thomas Rix	5	46
dmix	2	20
Will Cosgrove	1	8
theghostwhoforks	5	5
Daniel Gomez Sierra	2	2
John Gadbois	2	2
Jose Diaz-Gonzalez	1	1

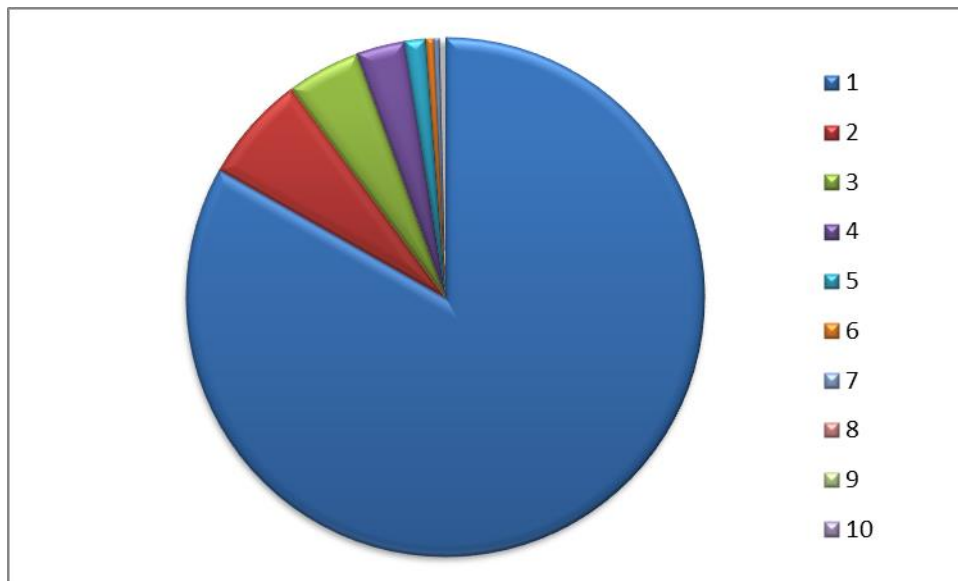


図 18 メンバがコミットで修正・追加した行数の割合

編集回数が多い Eric Waller は 1 回のコミットで平均約 17 行の追加・修正をしていた。編集回数が少ない Jose Diaz-Gonzalez は 1 回のコミットで 1 行の追加・修正しかしていなかった。

### 7.2.5 spoon / library

プロジェクトメンバ	コミット回数	コミットで修正・追加した行数
Davy Hellemans	208	12817
Annelies Van Extergem	3	3965
Tijs Verkoyen	15	2237
Matthias Mullie	26	1508
Alberto Vena	1	344
Jeroen Desloovere	12	270
Jan Moesen	16	230
Per Juchtmans	6	177
Dieter Vanden Eynde	10	140
Bramus!	5	54
SerkanYildiz	1	9
Sam Tubbax	3	6
Michaël Rigart	1	5
Jelmer Snoeck	2	4
Johan Ronsse	2	2
Tadas Juozapaitis	1	1

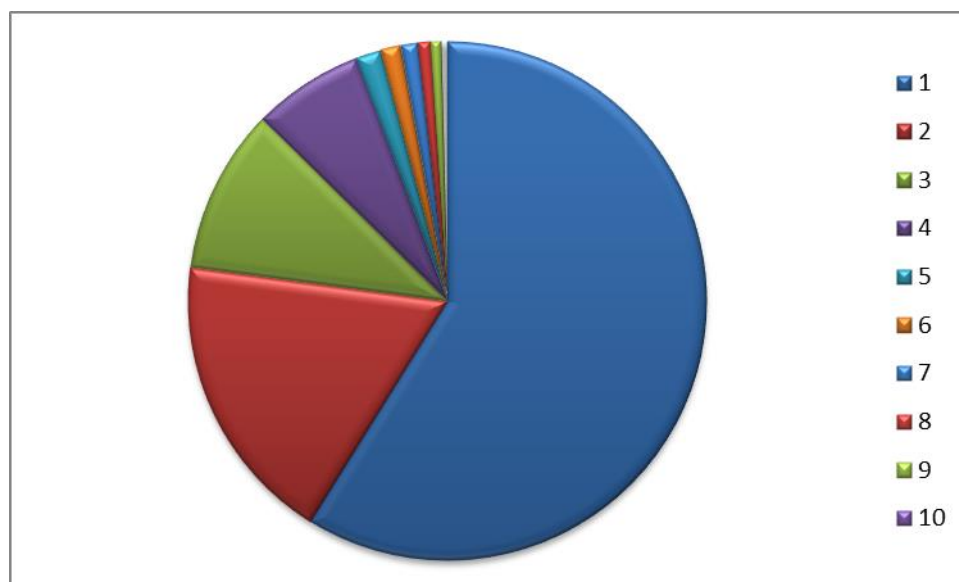


図 19 メンバがコミットで修正・追加した行数の割合

編集回数が多い Davy Hellemans は 1 回のコミットで平均約 62 行の追加・修正をしていた。編集回数が少ない Tadas Juozapaitis は 1 回のコミットで 1 行の追加・修正しかしていなかった。

### 7.3 考察

- square / spoon に参加していたのは 30 名だった。そのメンバがコミットで追加・修正した行数は合計で 35,449 だった。そのうちの 31,669 の行数は 3 名によって書かれていた。このことからこのプロジェクトの全体の行数の約 8 割はメンバの約 2 割が書いている結果となった。
- stanfy / spoon-gradle-plugin に参加していたのは 5 名だった。そのメンバがコミットで追加・修正した行数は合計で 1,997 だった。そのうちの 1,714 の行数は 1 名によって書かれていた。このことからこのプロジェクトの全体の行数の約 8 割はメンバの約 1 割が書いている結果となった。
- spoonapps / spoonme に参加していたのは 13 名だった。そのメンバがコミットで追加・修正した行数は合計で 4,238 だった。そのうちの 3,719 の行数は 6 名によって書かれていた。このことからこのプロジェクトの全体の行数の約 8 割はメンバの約 5 割が書いている結果となった。
- seatgeek / soulmate に参加していたのは 10 名だった。そのメンバがコミットで追加・修正した行数は合計で 1,517 だった。そのうちの 1,364 の行数は 2 名によって書かれていた。このことからこのプロジェクトの全体の行数の約 8 割はメンバの約 2 割が書いている結果となった。
- spoon / library に参加していたのは 16 名だった。そのメンバがコミットで追加・修正した行数は合計で 21,769 だった。そのうちの 19,019 の行数は 3 名によって書かれていた。このことからこのプロジェクトの全体の行数の約 8 割はメンバの約 2 割が書いている結果となった。

以上の結果により、一般に言われているパレートの法則（80:20 の法則）が OSS 開発の現場でも成り立つ可能性が示唆される。

### 7.4 謝辞

卒業論文を進めるにあたり、ご指導を頂いた矢吹太郎准教授に感謝いたします。また、矢吹研究室の同期や後輩のご協力に感謝いたします。