Special Feature 1 関数型プログラミングに強くなる
Special Feature 2 通信の安全性を高める
[ソフトウェア デザイン]
OSとネットワーク、
IT環境を支える
エンジニアの総合誌

Software Design

2015年8月18日発行
毎月1回18日発行
通巻364号
（発刊298号）
1985年7月1日
第三種郵便物認可
ISSN 0916-6297
定価 本体 1,220円 +税

2015 August 8

総力特集
大増ページ

Lispより始めよ、
されば救われん!

special feature 1

なぜ関数型プログラミングは難しいのか?

Lisp
Scala
Haskell
Elixir
Python
Clojure
関数型
のエッセンスを学習する

special feature 2

安全な通信を確保する

SSL/TLSの教科書

インターネットの通信セキュリティを
確保するしくみをマスターしよう!

extra feature

エンタープライズJavaの進化

AWSで始めよう! モダンなJavaアプリケーション開発 第1回

JavaとDockerで始める
実践Elastic Beanstalk入門

OSとネットワーク、
IT環境を支えるエンジニアの総合誌

Software Design

毎月18日発売

PDF電子版
Gihyo Digital
Publishingにて
販売開始

年間定期購読と電子版販売のご案内

1年購読（12回）

14,880円（税込み、送料無料）1冊あたり1,240円（6%割引）

PDF電子版の購入については

Software Design ホームページ
http://gihyo.jp/magazine/SD

をご覧ください。
PDF電子版の年間定期購読も受け付けております。

・インターネットから最新号、バックナンバーも1冊からお求めいただけます！
・紙版のほかにデジタル版もご購入いただけます！
　デジタル版はPCのほかにiPad/iPhoneにも対応し、購入するとどちらでも追加料金を支払うことなく読むことができます。
　※ご利用に際しては、／～＼Fujisan.co.jp（http://www.fujisan.co.jp/）に記載の利用規約に準じます。

Fujisan.co.jpからのお申し込み方法

1 >> ／～＼Fujisan.co.jp クイックアクセス
http://www.fujisan.co.jp/sd/

2 >> 定期購読受付専用ダイヤル
0120-223-223（年中無休、24時間対応）

2015
August
8
Software Design

contents

第1特集

なぜ関数型プログラミングは難しいのか?

Lispより始めよ、されば救われん!

Lisp、Scala、Haskell、Elixir、Python、Clojure、関数型のエッセンスを学習する

2015
August
8
Software Design

contents

第2特集

安全な通信を確保するSSL/TLSの教科書

短期連載

Catch up trend

Inside View

アラカルト

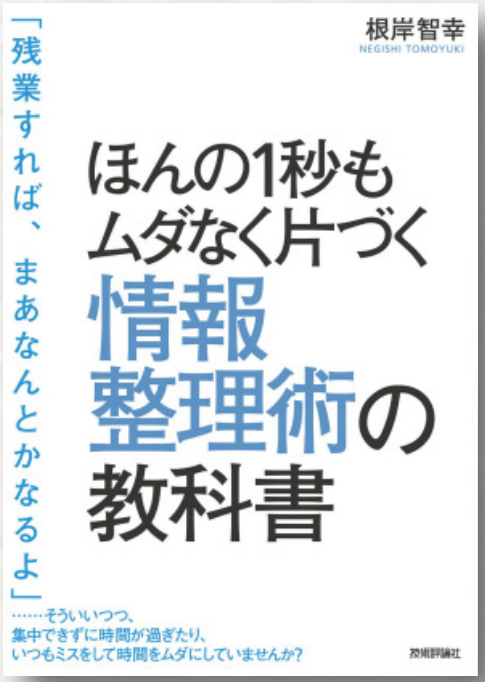
Column

Development

OS/Network

［ロゴデザイン］
デザイン集合ゼブラ+坂井 哲也
［表紙デザイン］
藤井 耕志（Re:D）
［表紙写真］
(c) VGL/a.collectionRF /amanaimages
［イラスト］
フクモトミホ
［本文デザイン］
＊岩井 栄子
＊ごぼうデザイン事務所
＊近藤 しのぶ
＊SeaGrape
＊安達 恵美子
＊轟木 亜紀子、阿保 裕美、佐藤 みどり（トップスタジオデザイン室）
＊伊勢 歩、横山 慎昌（BUCH+）
＊森井 一三
＊藤井 耕志（Re:D）
＊石田 昌治（マップス）

この個所は、雑誌発行時には広告が掲載されていました。編集の都合上、
総集編では収録致しません。



ほんの1秒もムダなく片づく情報整理術の教科書

根岸智幸　著

「必要なデータがどこにいったかわからなくなった……」
「やるべきことがあれこれあって、収集がつかない……」
そんなことが常態化していませんか?
30年間、雑誌編集長、Webサイトのディレクター、プログラマなどITの最前線でさまざまな仕事を手がけてきた著者が、効率的に仕事を進めるために欠かせない最新の整理の技術を集大成。パソコンのファイル、メール、スケジュール、ToDo、ノートやメモ、ネットの情報まで、あらゆる整理のポイントがこれ1冊でわかる!

A5 判／328ページ
定価（本体1780円+税）
ISBN978-4-7741-7409-9

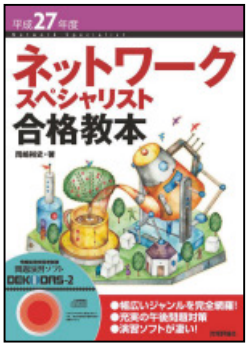
たった1日で即戦力になるExcelの教科書

吉田拳　著

「Excelぐらい、まあなんとかなるよ」
…そういいつつ、作業に何時間もかかってイライラしたり、いつもミスをして時間をムダにしていませんか?
そんなExcel地獄に陥らないための、今日から即役立つテクニックから、社会人10年目でも意外とわかっていない「なぜ、それが必要なのか?」といった目的意識まで、実務直結の知識を最小限の時間でマスター。
50社以上、のべ2000名以上の指導実績に裏打ちされたノウハウが満載!

A5 判／328ページ
定価（本体1780円+税）
ISBN978-4-7741-6808-1

技術評論社　当社書籍・雑誌のご購入は全国の書店、またはオンライン書店でお願い致します。
〒162-0846　東京都新宿区市谷左内町 21-13　販売促進部　TEL.03-3513-6150 FAX.03-3513-6151



岡嶋裕史　著
A5判／624ページ
定価（本体2980円+税）
ISBN978-4-7741-7207-1

岡嶋裕史　著
B6判／320ページ
定価（本体1680円+税）
ISBN978-4-7741-6942-2

左門至峰・平田賀一　著
A5判／352ページ
定価（本体2380円+税）
ISBN978-4-7741-7294-1

エディフィストラーニング株式会社　著
B5判／384ページ
定価（本体2980円+税）
ISBN978-4-7741-7208-8

あなたを合格へと導く一冊があります!

定平誠・須藤智　著
A5判／560ページ
定価（本体1680円+税）
ISBN978-4-7741-6820-3

山本三雄　著
B5判／568ページ
定価（本体1480円+税）
ISBN978-4-7741-7439-6

岡嶋裕史　著
A5判／656ページ
定価（本体2880円+税）
ISBN978-4-7741-6937-8

エディフィストラーニング株式会社　著
B5判／392ページ
定価（本体2980円+税）
ISBN978-4-7741-7456-3

金子則彦　著
A5判／680ページ
定価（本体3300円+税）
ISBN978-4-7741-6941-5

大滝みや子・岡嶋裕史　著
A5判／736ページ
定価（本体2980円+税）
ISBN978-4-7741-6924-8

大滝みや子　著
B6判／384ページ
定価（本体1480円+税）
ISBN978-4-7741-6710-7

加藤昭・高見澤秀幸・矢野龍王　著
B5判／456ページ
定価（本体1880円+税）
ISBN978-4-7741-7440-2

技術評論社　当社書籍・雑誌のご購入は全国の書店、またはオンライン書店でお願い致します。
〒162-0846　東京都新宿区市谷左内町 21-13　販売促進部　TEL.03-3513-6150 FAX.03-3513-6151

第80回

ITエンジニア必須の

最新用語解説

TEXT：㈲オングス 杉山 貴章 SUGIYAMA Takaaki
takaaki@ongs.co.jp

テーマ募集
本連載では、最近気になる用語など、今後取り上げてほしいテーマを募集しています。sd@gihyo.co.jp 宛にお送りください。

Project Brillo

新 IoT プラットフォーム「Project Brillo」

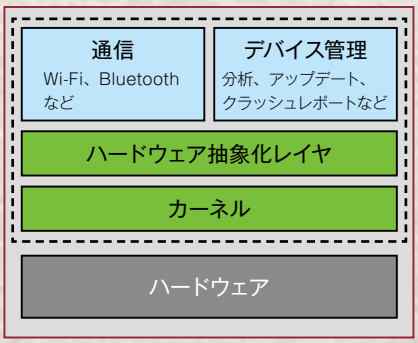
「Project Brillo」（以下、Brillo）は、2015年6月にGoogleが発表したIoT（Internet of Things：モノのインターネット）のための新しいプラットフォームです。Brilloは、PCやスマートフォンだけでなく、家電製品や車載機器、ひいては家のドアの鍵まで、あらゆる“モノ”を端末としてセットアップすることができ、これらの端末同士やクラウドサービスをシームレスに連携させられるようになります。

BrilloはAndroid OSをベースとして開発され、最小限のスペックでも動作するように洗練された超小型OSになるとのことです。おもな特徴としては次のようなものが挙げられています。

- 要求仕様が最小限で、極めて小さなデバイスでも動作する
- 幅広いハードウェアおよびプロセッサをサポート
- 豊富な接続性やセキュリティ仕様を備える
- 必要最低限の要素をあらかじめ備えているため、対応デバイスの開発が容易
- クラッシュレポートやアップデートサービスなど、Webコンソールからアクセス可能な管理機能を備える

図1はGoogleが発表したBrilloの基本的な概念図になります。カーネルの上にハードウェア抽象化レイヤ（HAL）がある点はAndroidと同様で、その上にネットワーク接続とデバイス管理のためのレイヤが存在しています。注目すべき点としては、Brillo単体でWi-FiやBluetoothによる通信手段を備えており、スマートフォンやほかのデバイスに接続できるようになっていることが挙げられます。また、メッシュネットワーク規格のThreadもサポートされる予定となっています。一方で、アプリケーションの実行環境やライブラリはこの中には含まれておらず、Brilloがあくまでも“モノをネットワーク端末にすること”に特化したプラットフォームであることがわかります。

▼図1　Brilloのアーキテクチャ（破線部）



JSONベースの通信プロトコル「Weave」

Brillo端末と、他のBrillo端末やスマートフォン、クラウドサービスとのコミュニケーションは、Brilloと同時に発表された「Weave」と呼ばれる新しいプロトコルによって行われます。Weaveは、IoT端末が外部の端末やサービスとシームレスに接続・連携するためのスキーマセットを提供します。WeaveはJSONをベースとしており、BrilloやAndroidだけでなくiOSや既存のPC、クラウドサービスなどもサポートしたクロスプラットフォームな通信レイヤになるとのことです。

従来のIoTデバイスでは、どのような手段で外部とコミュニケートするかが大きな課題になっていました。BrilloとWeaveはその悩みを解決するモデルとして注目されています。とくに、JSONをベースとしていることから取り扱いが容易で、クラウドサービスとの親和性も極めて高い点が大きな強みと言えます。

また、Weaveは家庭用サーモスタットの「Nest」とも互換性があります。Nestは通信機能や学習機能などを備えた高機能サーモスタットで、IoTのハブになり得る存在として注目を集めていました。GoogleはNestを開発したNest Labsを2014年に買収しており、Brilloへの取り組みはこのNest Labsのチームが中心となって進めているとのことです。したがって、Brillo/WeaveとNestの親和性が高いのは必然とも言えます。前触れなく登場したようにも思えるBrilloですが、圧倒的なシェアを誇るAndroidや、すでにIoT分野で実績のあるNestなどと結びつくことで、かなり現実味のある未来が見えてくるのではないでしょうか。

Googleでは、Brilloの開発者向けプレビューを2015年第3四半期にもリリースする予定だとしています。SD

Project Brillo
https://developers.google.com/brillo/

DIGITAL GADGET

Volume 200

安藤 幸央
EXA Corporation
[Twitter] »@yukio_andoh
[Web Site] »http://www.andoh.org/

200回を振り返る: SF映画とデジタルガジェット

200回を振り返る

筆者の出身高校の校訓は「継続は力なり」でした。高校生の頃はピンと来ませんでしたが、今なら実感できます。1998年末から始まった本連載「デジタルガジェット」は、一度も休むことなく今回で200回となりました。16年以上、もうすぐ17年目に入ります。支えていただいた読者の皆さん、編集部の皆さん、いろいろなデジタルガジェットを生み出し続けている世界中の皆さんに感謝したいと思います。

以前、マイクロソフトリサーチでユーザインターフェースの研究をしているビル・バクストン氏の講演を聞いたとき、「現在、最新だと思っている技術は、だいたい20〜30年前に創られたものである」とのコメントが強く印象に残っています。タッチパネルも、スマートウォッチも、タブレット端末も、VRも、現在最新技術だと思っているデジタル技術は、最初に登場し、研究開発され始めたのは、だいたい20年から30年前だということです。とても納得するとともに、そう考えると、現在研究開発中でまだまだ実用化にはほど遠いと思えるような最新技術が、20年後、30年後には、ごく普通に多くの人に使われている可能性を示唆しています。

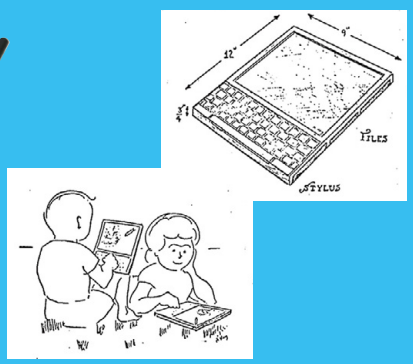
1998年、連載第1回目のタイトルは『Javaの現在と未来——「Java Demo」って何?』でした。1994年に初めて一般にお披露目されたプログラミング言語Javaは、現在もAndroidアプリのプログラミングや、エンタープライズ分野でのサーバサイドの大規模プログラミングなど、形を変えて活用されています。

今から過去を振り返り、20年から30年ほど生き残っているテクノロジには何があるでしょうか? QWERTYキーボードでの入力も、変わっていないものの1つかもしれません。

- **MIDI:** 電子楽器の演奏データを機器間でデジタル転送する標準規格
- **OpenGL:** グラフィックスハードウェアをコントロールするためのAPI。OpenGL ES、WebGLへと派生
- **RenderMan:** フルCG映画などの表現力豊かなコンピュータグラフィックスを描くためのPixarの技術／ファイル仕様

スター・トレック テレビシリーズに登場した通信機器(のオモチャ)。フタを開けて通信するフリップタイプ(1966年)

約30年後、左の通信機器を彷彿させる携帯電話「StarTAC」がモトローラから登場。日本でもIDOから発売された(1996年)



Dynabookのイラスト。このイラストが掲載されている論文の中では、最適な重量が680gと明記されており、初代iPadの重量も680gだったことから、真偽のほどはわからないが、強く影響を受けていると考えられている(出典:『A Personal Computer for Children of All Ages』(Akan C. Key, 1972)より)

1968年代、主流だったDEC社製の大型コンピュータ「PDP-10」(撮影:Scott Francis氏、クリエイティブコモンズライセンス)

※本記事で紹介しているものは国内未発表・未発売のものを含んでおります。

●**x86アーキテクチャ:**CPUの基本原理、基本構造
●**TCP/IPなど:**各種ネットワークプロトコル
●**HTML:**構造化言語としではなく、Webコンテンツを表現するための手段としてのHTML

CG映画「トイ・ストーリー」の制作で知られるPixarが映画製作に使っているCGソフトウェア「RenderMan」の基本アルゴリズムは、1987年頃に発表された論文に由来しています。当時は縦横512画素程度の画素数の、たった1枚の画像生成に、数億円する最新のコンピュータでも数時間かかっていた時代です。その当時からすでに20年後のコンピュータの性能を予想し、設計していたそうで、その先見の明には強い信念が感じられ、将来を見通す目の確かさに驚くばかりです。そして、RenderManは今もCG映画制作の第一線で活躍し続けているのです。

今主流の各種プログラミング言語は20年後、30年後も使われ続けているでしょうか？　そもそもプログラミングするという仕事は、30年後も残っているのでしょうか？　何十年も生き残り、今後もしばらくは活用し続けられるであろう長命のテクノロジには、共通するポイントがあります。勝手に解釈すると、それらは次のとおり。

●基本となる仕様が大きすぎず、複雑すぎず、必要十分な部分のみ規定されていること
●標準として強固な規定があり、業界団体または1社によって矛盾がなく、解釈がぶれない規定がなされていること
●拡張し続ける要素、余地があること
●拡張したものを基本部分に取り入

1998

「アルマゲドン」「ロスト・イン・スペース」「トゥルーマン・ショー」

現代の延長線上で少しだけ未来を描いている作品の多かった年。

» アルマゲドン
●発売元:ウォルト・ディズニー・スタジオ・ジャパン
●価格:2,381円＋税(Blu-ray)

1999

「アンドリュー NDR114」「イグジステンズ」「マトリックス」「スター・ウォーズ エピソード1／ファントム・メナス」

仮想世界をより具体的に描いたり、擬人化されたロボットが生活に融け込んできた年。

» アンドリュー NDR114
●発売・販売元:ソニー・ピクチャーズ エンタテインメント
●価格／発売日:1,410円＋税／発売中

2000

「シックス・デイ」「インビジブル」「ザ・セル」「X-メン」

ハイテクデバイスに囲まれていたり、クローン人間が登場するが、現代を描いた年。

2001

「トゥームレイダー」「A.I.」「猿の惑星(リメイク版)」「ジュラシック・パークIII」

人に作られた知性、人工知能の概念、恐竜のクローニングと、地球の遠い将来に目が向いた年。

2002

「マイノリティ・リポート」「劇場版カウボーイ・ビバップ」「リベリオン」「クローン」「メン・イン・ブラック2」「リターナー」「シモーヌ」「タイムマシン」

2054年を描いたマイノリティ・リポートは綿密な調査のうえで未来が描かれている。未来的UIの礎となった作品の年。

» マイノリティ・リポート
●発売元:20世紀フォックス ホーム エンターテイメント ジャパン
●価格／発売日:1,905円＋税(Blu-ray)／発売中

2003

「マトリックス リローデッド」「マトリックス レボリューションズ」「ベイチェック」「ターミネーター3」

仮想世界と現代世界との対比や、記憶を操作できることなどが描かれた作品など、テクノロジへの恐怖感が描かれた年。

» マトリックス リローデッド
●発売元:ワーナー・ブラザース・ホームエンターテイメント
●価格／発売日:2,381円＋税(初回限定版スペシャル・パッケージ Blu-ray)／2015年3月18日

2004

「サンダーバード(実写版)」「エージェント・コーディ2」「アイ,ロボット」「イノセンス」

実世界におけるデジタルグッズ、現代の技術である程度実現可能なデジタルグッズに囲まれた作品の年。

» アイ,ロボット
●発売元:20世紀フォックス ホーム エンターテイメント ジャパン
●価格／発売日:2,381円＋税(Blu-ray)／発売中

2005

「アイランド」「イーオン・フラックス」「銀河ヒッチハイク・ガイド」「宇宙戦争」

臓器のためのクローニング、延命などに焦点をあてた作品。現代の延長線上としての未来が描かれた年。

2006

「M:i:III」「007カジノ・ロワイヤル」「スキャナー・ダークリー」「パプリカ」「ウルトラヴァイオレット」

テクノロジの進化によって、必ずしもバラ色の未来が来るというだけではないことが描かれた年。

» M:i:III
●発売元:パラマウント ジャパン
●価格／発売日:1,429円＋税(1枚組)／2007年6月22日発売(発売中)

れるしくみがあること。亜流や枝分かれが増えない規定があること

- 利用すること、使うこと自体は安価であるか、無料であり、広まる要素を持つこと
- 現在のハードやネット環境に適合するだけでなく、将来登場するであろう高性能な機器、環境をも想定していること

SFから学び取る、これから20年後も生き残る技術

1968年に公開された「2001年宇宙の旅」には、iPadのようなタブレット端末が出現します。1人1人の宇宙飛行士が個人のパーソナルな端末として、文書を見たり、ビデオ映像を見たりしています。iPadが発売されたのは2010年、iPadの登場に少なからず影響を与えていると言われているアラン・ケイの論文に、Dynabookが登場するのは1972年のことです。

1968年というと、現在のようなコンピュータは存在せず、設置に巨大な部屋が必要であった時代です。その1台の大型コンピュータを大勢で共有して利用していました。マウスが初めて登場したのもこの頃です。そんな時代に、未来を夢想して、タブレット端末が必要な世界、活用されている世界を想像できたことには感嘆しかありません。

今あるものにイノベーションをもたらす、想像力の限界を超える、拡げる、引き延ばすのが空想の世界ではないでしょうか？　実際に、最先端のデバイスやユーザインターフェースを専門で研究開発している研究者の中にもSFファンは多く、未来を想像し、作り出していくためにはSFは不可欠なのです。SD

2007
「アイ・アム・レジェンド」「サンシャイン 2057」「トランスフォーマー」「バイオハザードIII」
地球や人類が滅びる可能性が描かれた年。

2008
「イーグル・アイ」「アイアンマン」「ウォーリー」
テクノロジによって、体躯や知能が強化されることが描かれた年。

2009
「アバター」「2012」「スタートレック（リメイク版）」「第9地区」「月に囚われた男」「サロゲート」
身体と精神とクローン的なものについて描かれた年。

≫アバター
●発売元:20世紀フォックス ホーム エンターテイメント ジャパン
●価格／発売日:1,905円＋税（Blu-ray）／発売中

2010
「トロン:レガシー」「アイアンマン2」「インセプション」
テクノロジによって作り上げられた仮想世界が描かれた年。

≫トロン:レガシー
●発売元:ウォルト・ディズニー・スタジオ・ジャパン
●価格:2,381円＋税（Blu-ray）

2011
「ミッション・インポッシブル・ゴーストプロトコロル」「スパイキッズ」「アジャストメント」「宇宙人ポール」「ミッション: 8ミニッツ」「X-MEN: ファースト・ジェネレーション」「スーパーエイト」「カウボーイ&エイリアン」「タイム」
物理法則から言ってあり得ないことも描き出していた年。

≫スパイキッズ
●発売元:ワーナー・ブラザース・ホームエンターテイメント
●価格／発売日:2,381円＋税（Blu-ray）／2013年8月7日

2012
「ハンガーゲーム」「アベンジャーズ」「プロメテウス」「トータルリコール（リメイク版）」「クラウドアトラス」「アイアン・スカイ」「ジャッジ・ドレッド（リメイク）」「ルーパー」「メン・イン・ブラック3」
近代と古代の風習が混在した世界が描かれた年。

≫プロメテウス
●発売元:20世紀フォックス ホーム エンターテイメント ジャパン
●価格／発売日:1,905円＋税（Blu-ray）／発売中

2013
「エンダーのゲーム」「パシフィックリム」「アフターアース」「アイアンマン3」「オブリビオン」「エリジウム」「ゼロ・グラビティ」「her/世界でひとつの彼女」
テクノロジが極限まで進化したときの地球、人類の有様が描かれた年。

2014
「ガーディアンズ・オブ・ギャラクシー」「ベイマックス」「ダイバージェント」「オール・ユー・ニード・イズ・キル」「インターステラー」
ごく近い近未来を描いているが、地球ではないどこか並行宇宙の話を描いている年。

≫ガーディアンズ・オブ・ギャラクシー
●発売元:ウォルト・ディズニー・スタジオ・ジャパン
●価格:4,000円＋税

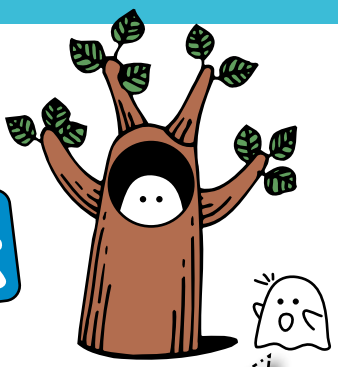
2015
「ゼロの未来」「エクスマキナ（公開未定）」「チャッピー」
人工知能、人工生命、すぐそこにある、少し怖いディストピア感のある未来が描かれている年。今年後半はわかりません。

≫チャッピー　アンレイテッド・バージョン プレミアムエディション
●発売・販売元:ソニー・ピクチャーズ エンタテインメント
●価格／発売日:5,695円＋税（初回生産限定　Blu-ray）／9月18日発売

結城 浩 YUKI Hiroshi
http://www.hyuki.com/
Twitter：@hyuki
図版イラスト●フクモトミホ

27

結城浩の再発見の発想法



Fail-Safe

Fail-Safe——フェイル・セイフ

フェイル・セイフとは

フェイル・セイフ(Fail-Safe)とは、**システムが壊れるときには安全側に倒れるようにする**という設計方針のことです。

フェイル・セイフのフェイル(fail)は失敗や故障という意味で、セイフ(safe)は安全という意味です。上で「壊れる」と書きましたが、これは非常に広い意味で使っています。ソフトウェアであれ、ハードウェアであれ、何らかの原因でうまく動作しないことはあるものです。これはいわばシステムが「壊れた」状態です。そのとき、システムはどんなふうに壊れるのが良いでしょうか。安全側に倒れる、すなわち「システムが壊れても関係者に被害が及ばないようにする」のが、フェイル・セイフという設計方針なのです。

たとえば、フェイル・セイフの簡単な例として踏切の遮断機を考えましょう。遮断機が故障したときに遮断機が上がってしまうなら、電車が通過しようとしているのに人や車両が線路を横断してしまう危険性があります。ですから、遮断機は故障したときには下がったままになっているべきです(**図1**)。

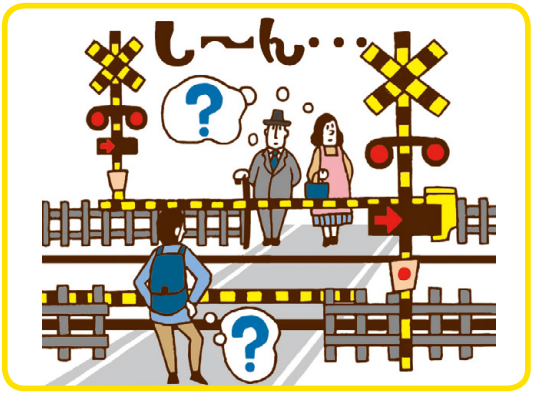
フェイル・セイフの例

フェイル・セイフの身近な例に、電気系統のブレーカーがあります。家中に張り巡らされている電気系統のどこかでショートが起き、大電流が流れるという異常事態が起きたとき、ブレーカーが落ちて電気の供給を止めます。電気が供給されないのはシステムとしては壊れたわけですが、大電流を流し続けていると火事になる危険性がありますから、電気の供給を止めて、安全側に倒しているわけですね。

地震発生時ストーブが消えることや、上からものが落ちてきたときに電源スイッチが切れるような方向にトグルスイッチの向きを決めるのも、フェイル・セイフの一種でしょう。

筆者は学生時代の高電圧実験で、「電源がオフになっているとわかっていても、電線に最初に触れるときには《手の甲》を使うように」と指導されたのを覚えています。万一電源がオフになっていなかった場合《手のひら》の側で電線に触れると、電流のために筋肉が収縮して電線をつかんでしまい、自分の意志で離せなくなって

▼図1　壊れた遮断機は下りたままが安全



しまう危険性があるとのこと。これもまた、フェイル・セイフに基づいています。

電車の運転ではデッドマン装置というしくみが使われています。これは、ハンドルを握り続けていないと、電車が止まってしまうしくみです。デッドマン(死者)という名前のとおり、電車の運転士が死亡したり、気を失ったりした場合に自動で電車が止まるようにするためです。これも、フェイル・セイフの一種です。

フェイル・セイフの前提条件

フェイル・セイフに基づく設計のためには、大きな前提条件を認める必要があることに注意してください。それは**このシステムは壊れる可能性がある**という前提条件です。フェイル・セイフは「壊れるときには安全側に倒れるようにする」という設計方針ですから、壊れることを認められないなら、フェイル・セイフの設計を行うことは絶対にできません。

すなわち、「この機械は壊れません」や「事故が起きる可能性はゼロです」という主張はフェイル・セイフと相反することになります。

セイフの定義

フェイル・セイフにおけるセイフ(安全)は、よく考えると難しい問題です。システムごとに定義しておく必要があります。

たとえば、コンピュータが組み込まれた金庫を想像してみてください。表面にあるボタンで暗証番号を正しく入力すると金庫は開きますが、誤った入力では開きません。この金庫が壊れるときには、どういう振る舞いをすればフェイル・セイフと言えるでしょうか(**図2**)。

壊れた金庫は、けっして開いてはいけないのでしょうか。それともすぐに開くべきでしょうか。けっして開かないとしたら、中のものが盗まれる心配はありませんが、緊急時の取り出しができなくなります。すぐに開いてしまったら、緊急時の取り出しができますが、中のものが盗まれてしまう危険性があります。どちらが安全なのか、一概には言えません。

「金庫なんだからすぐに開いちゃ困るよ」というのなら、銀行の扉はどうでしょう。火災発生時に銀行の扉が開かなくなってしまったら、財産は安全ですが、中にいる人が閉じ込められてしまう危険性があります。なかなか難しい問題であることがわかります。

日常生活とフェイル・セイフ

私たちの日常生活で、フェイル・セイフという発想はどのように役立つでしょうか。

災害時とは、日常というシステムが壊れた状況です。家族の間で、災害時の集合場所や連絡方法を決めておくのは、日常生活をフェイル・セイフにする方法の1つかもしれません。

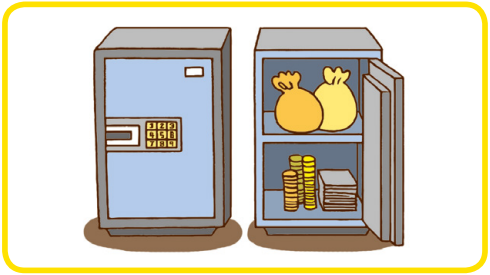
「自分の手帳に自分の住所を書く」のはフェイル・セイフの観点からどうでしょう。手帳は大切なものですから、万一落としたときに自分に連絡が来るようにと、自分の住所を書くことには意味があります。しかし、悪意のある人がその手帳を入手した場合、住所が書いてあったばかりに大きな被害を受ける危険性があるかもしれません。何が正しいか、ここには正解はありません。本人が何を安全と定義するかが大事になります。

◆ ◆ ◆

あなたの周りを見回して、「絶対に《これ》は壊れない」と思っているものはありませんか。また「万一《これ》が壊れたときには安全側に倒れるか」を考えてみてください。そのときの「安全」とはなんでしょうか。

ぜひ、考えてみてください。SD

▼図2　壊れた金庫はどうなるべきか



耽溺せよ電子工作

おとなラズパイリレー

第10回

「IoTをやってみよう（後編）」

江草 陽太

おとなラズパイリレーは、Raspberry Piを文字どおり「リレー」し、好奇心旺盛なITエンジニアが電子工作をするという企画です。前編で構想を練り、後編で実装します。1年を通してどんなデバイスができあがるのか？……今回は、さくらインターネット㈱のホープ、江草さんによるIoT工作です。

Writer 江草 陽太（えぐさ ようた） さくらインターネット㈱ プラットフォーム事業部 サービス開発チーム
写真撮影：三村 朋広（さくらインターネット㈱ プラットフォーム事業部）

前回のおさらい

「IoTをやってみよう！」企画、前回は、使用するパーツを選定し、GPSと携帯回線の動作テストを行いました。後半である今回は、いよいよアプリケーションを開発し、Raspberry Pi(以下、発信機)から位置情報を送信し、スマートフォン・タブレットから位置が確認できるようにします。開発するアプリケーションですが、僕は普段からPython 3と、Webフレームワークであるを利用しているので、今回もこちらを利用しました。

開発アプリケーションは、次の3つです。

- DBに位置情報を収集し、APIを通して位置情報の登録と取得ができるサーバサイドプログラム
- Raspberry Pi上で動作し、GPSから得た位置情報をサーバにHTTPで送信するプログラム
- ブラウザ上で動作し、位置情報をサーバサイドから取得して、地図上に表示するシングルページプログラム

ちなみに、この記事のコードはすべてGitHub[注1]に公開しています。

サーバは弊社のIaaSサービスである、「さくらのクラウド」を利用しました。

サーバサイドのアプリケーション

データベース

スーパーママチャリグランプリには、弊社からは1台だけでなく、複数台の自転車が出場しているので、それらすべての位置情報を管理する必要があります。そのため、発信機を管理するモデル(テーブル)Transponderと、ある時点での位置情報を管理するモデルWaypointを用意しました(**リスト1**)。

API

APIは、Raspberry Piから位置情報を登録するためのAPIと、ブラウザに対して、すべての発信機の位置情報を渡すためのAPIが必要になります。

APIは、発信機名、時刻、緯度および経度をPOSTすると位置情報が保存され、GETすると最新の位置情報を含む発信機の一覧をJSONがGETで取得できるエンドポイントを用意しました。

発信機の一覧をGETすると**リスト2**のようなレスポンスが返ってきます。

注1） https://github.com/y-egusa/softwaredeisign-rpi-gps

▼リスト1　ソースコードサンプル（Transponder、Waypoint）

```
class Transponder(models.Model):
    class Meta:
        verbose_name = verbose_name_plural = "発信機"
    name = models.CharField("名前", max_length=20, unique=True)
    marker = models.ImageField("マーカー", upload_to="uploads/markers", blank=True)
    marker_disabled = models.ImageField("マーカー（無効時）", upload_to="uploads/markers", blank=True)

class Waypoint(models.Model):
    class Meta:
        verbose_name = verbose_name_plural = "位置情報"
    transponder = models.ForeignKey(Transponder)
    created_at = models.DateTimeField("時刻")
    latitude = models.DecimalField("緯度", max_digits=10, decimal_places=5, blank=True, null=True)
    longitude = models.DecimalField("軽度", max_digits=10, decimal_places=5, blank=True, null=True)
```

▼リスト2　発信機の位置情報を示すJSON例

```
{
  "GPS01": {
    "latitude": 35.69287,
    "longitude": 139.69417833333333,
    "updated_at": "2015-06-22T16:02:52"
  },
  "GPS02": {
    "latitude": 35.694493333333334,
    "longitude": 139.69567333333333,
    "updated_at": "2015-06-22T16:02:53"
  }
}
```

このとき、注意しないといけないことが1点あります。緯度経度の表記には、10進法（度）と60進法（度分）の2種類があります。GPSモジュールは60進法、後ほど利用するGoogle Maps APIは10進法を利用しているため、どこかで変換する必要があります。今回は位置情報をブラウザに渡す時点で変換をしました。

発信機の登録などの作業はAPIを用意せず、Django標準の管理サイトを利用しています。

Raspberry Piのアプリケーション

こちらもサーバサイドと同様にPython 3で実装しました。

市販されているほとんどのGPSモジュールは、NMEA 0183という規格に沿った文字情報を一方的に送信してきます。このプロトコルは、1行が1つのセンテンスになっており、センテンスにはGPRMC、GPGGA、GPGSVなど、さまざまな種類があります。これらの情報には、位置だけではなく、受信状態、正確な時刻、受信できている衛星の情報といったさまざまな情報が含まれています。今回の用途では、時刻と位置情報だけが必要なので、両方が1つのセンテンスに含まれているGPRMCだけを利用しました。

通信状況が悪くなってもシリアルポートからの読み込みが停止しないように、次のマルチスレッド構成にしました。

シリアルポートから位置情報を取得

pyserialを利用すると、簡単にシリアルポートにアクセスできます。Python 3で利用するためには、pip3のインストールとpyserialのインストールが必要です.

```
$ sudo apt-get install python3-pip
$ sudo pip-3.2 install pyserial
```

APIでのリクエスト

HTTPのリクエスト発行は標準ライブラリ

であるurllibを利用しました。

先ほどのシリアルポートから位置情報を受信するスレッドによって、キューにパースされた位置情報が入っています。このスレッドではこのキューに溜まっている位置情報をただひたすら送信し続けます。

自動起動

Raspberry Pi上にプログラムを実行するためのシェルスクリプトrun.shを設置し、/etc/rc.localに記載することで、起動後に自動で実行されるようにしました(**リスト3**)。

▼リスト3 /etc/rc.local設定例

```
_IP=$(hostname -I) || true
if [ "$_IP" ]; then
printf "My IP address is %s¥n" "$_IP"
fi

/bin/sh /home/pi/softwaredeisign-rpi-gps/rpi/run.sh &

exit 0
```

ブラウザ上のアプリケーション

jQueryを使って開発しました。Ajaxを使って定期的に位置情報を取得し、Google Maps APIを利用して地図上にプロットしています。サーバサイドアプリケーションの一部として含まれているため、個別のデプロイは不要です。

Google Maps APIは、利用規約上、認証をかけたページなどに設置することはできません。もしも閉じた環境で地図を表示する必要がある場合には、Leaflet.js[注2]のような、タイルマップを表示するためのライブラリと、OpenStreet Map[注3]のような、オープンソースの地図データを組み合わせて使うと良いと思います。

動作試験

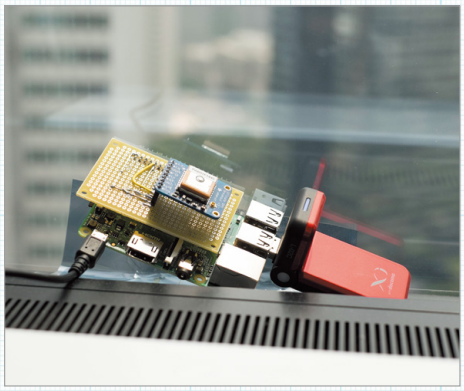
さて、これで必要な準備は終わりました。いよいよ動作実験です。

今回は、弊社の東京支社から近くの新宿中央公園まで散歩しながら試験をしてきました(**写真1**)。

東京支社の窓際でバッテリーをRaspberry Piにつなぎ電源ON！(**写真2**)しばらくすると、GPSの衛星捕捉が完了し、地図上に旗が表示されました。これで一安心です。片手にRaspberry Pi、片手にiPadというあやしい格好(**写真3**)で

▼写真1 著者近影

▼写真2 オフィスで動作確認



注2) http://leafletjs.com/
注3) https://www.openstreetmap.org/

▼写真3　移動中も気になる……

▼写真4　都庁前

◀写真5　位置もぴったり

▼写真6　ビルの間ではズレることも（新宿警察署前）

（笑）、新宿中央公園に向けて出発進行。公園内をぐるぐる回ってみましたが、大きなズレはほとんどなく、実際の地図ともピッタリ一致しています（**写真4**、**写真5**）。これならレースでも充分、活用できそうです。と、ここまでは大満足の出来だったのですが、帰り際に通った新宿警察署前では、位置が大きくズレるという問題も発生（**写真6**）。高層ビルが多いところでは、やはり位置が正確に出ないこともあるようです。とはいえ富士スピードウェイには障害物もあまりないので、心配する必要はないでしょう。

今後やりたいこと

時間の関係上、妥協している点や、実装できていない機能がまだいくつかあります。来年の1月のスーパーママチャリGPまでに、次の機能を実装したいと思っています。

1. POSTのAPIのJSON対応（手間の関係上できなかったので）
2. 最新の位置情報だけでなく、過去の通過地点も取得可能に！
3. 通知機能（発信機に「もう1周！」「交代したい！」ボタンを付ける）

これらの機能を実装して、スーパーママチャリGPに挑みます!! SD

● 執筆協力

RSコンポーネンツ㈱ Raspberry Piに興味のある方は次のサイトをチェック!
http://jp.rs-online.com/web/generalDisplay.html?id=raspberrypi

軽酔対談

かまぷの部屋

第13献 ゲスト：鹿野 恵子さん

㈲ユニバーサル・シェル・プログラミング研究所
鎌田 広子（かまた ひろこ）
Twitter：@kamapu

鹿野 恵子（かの けいこ）さん

宮城県仙台市出身。早稲田大学法学部卒。学生時代はバンド活動やアルバイトに夢中になる。大学卒業後、アスキーに入社。PC書の書籍営業職に携わる。その後、流通小売業系の出版社、IT企業などを経て、オライリー・ジャパン Maker Faire Tokyo事務局で広報、スポンサー担当を務める。

2013年のMaker Faire Bay Areaでの1枚→

（鎌田）鹿野さんはオライリー・ジャパンでMaker Faire Tokyoの事務局を担当されているそうですね。まずはご出身や学生時代のことを教えていただけますか。

（鹿野）生まれは宮城県仙台市で、高校までは仙台に住んでいました。中学時代は演劇部、高校時代は軽音部とクイズ研究会に所属していました。

いろいろなことをしていたんですね。演劇といっても脚本や役者など、さまざまな役割がありますが、何をされたのですか？

演劇部では演出を担当していました。そのころから全体をプロデュースしたり、人を盛り上げることが好きだったんだと思います。

クイズ研究会というのも意外です。

実は、十うん年前の高校生クイズ選手権で宮城県代表だったんですよ。全国大会では、東京のスタジオで早朝から翌日までぶっ通しで収録があって刺激的でした。

大学時代はどんなことをしていましたか？

編集プロダクションでアルバイトをしていました。企画を立てたり、週刊誌の仕事で、著名な方の取材に立ち会う機会をいただいたり、公務員宿舎の張り込み取材をしたり……、いろいろな経験をしました。

しかしまったく技術の影が見えませんね……。

そんなことはないですよ。小さなころから技術的なものにも憧れがあって、小学3年生のときにおじいちゃんに「MSX2+」を買ってもらったのがこの道に入ったきっかけだったのかもしれません。

MSXですか！　しかも小学校3年生！

中学のときは、アマチュア無線の免許も取りましたよ。

男子のようですね。どうして無線を触ることになったのですか？

もともと海外の人とコミュニケーションを取ることに興味があったところに、当時の技術の先生が、アマチュア無線を勧めてくれたのです。中2のときにはアマチュア無線技士4級免許を取りました。コールサインは「JL7FCV」です。

しかし鹿野さんは理系ではなく文系なんですよね。法学部に入って編プロから出版社、IT企業に……現在はMaker Faireと。一貫性があるようでないような。プログラミングのご経験はあるのですか？

MSXでは、BASICマガジンに掲載されているコードをそのまま入力して遊んでいましたが、原理はあまり理解していませんでした。編集職につきたくてアスキーに入社したのですが、配属が営業だったので、流通小売業系の業界誌出版社に転職して編集の仕事をしばらくしていました。その後、流通系のSIerに転職してマーケティングの仕事をしたのですが、そこでLinuxを触った経験が、コンピュータの原理を知るきっかけだったと思います。

8月1、2日にビッグサイトで開催される、Maker Faire Tokyoのお話を聞かせてください。そもそもどのようなイベントなのでしょうか？

アメリカでオライリー・メディアが刊行している「Make:」という雑誌




が土台にあるイベント[注1]で、2006年にスタートしてから、今年で10周年を迎えた世界最大規模のDIYイベントです。電子工作から3Dプリンタやレーザーカッターなどのデジタルファブリケーション、アート、クラフト、宇宙、乗り物、音楽……新しいものづくりの道具を使った、ありとあらゆる作品が展示されています。現在、世界131ヵ所の国や地域で開催されていて、78万人以上が来場しています。日本ではオライリー・ジャパンの主催で年に1回東京で開催していまして、それ以外にも山口、大垣でMini Maker Faireというイベントも開催されています。

今回のMaker Faire Tokyoはどのくらいの規模なのですか？

来場者数は1万5千人を見込んでいます。出展者数は約350組です。混雑が予想されますのでこの記事をご覧の来場予定の方には、スムーズに入場できるよう前売りチケットをぜひご購入いただければと思います。それと夏真っ盛りなので、熱中症対策をお忘れなく。

冬場に開催された去年と違って今年は夏ですものね。見どころを教えてください。

目玉の1つは、アメリカのEepy Birdさんというアーティストさんの「メントスコークショー」でしょうか。コーラにメントスを入れると、コーラが噴き出すのですが、このショーでは1回に130本のコーラゼロを音楽に合わせて噴出させます。家族連れの方に楽しんでいただけるように、会場のいたるところでお子さんが参加できるようなワークショップが開

催されているのもこのイベントの特徴です。今年はVR(仮想現実)モノの展示も多いです。また、ドローンを飛ばすためのエリアの設置も予定しています。

楽しそうですね！　Maker Faireの裏話などはありますか？

イベントの運営もなるべく、自分たちでできることは、自分たちでやっているところでしょうか。以前、東工大で「Make: Tokyo Meeting」という名称でイベントを実施をしていたころは、発電機の調達や運用、机を並べるところまで事務局でやっていたそうです。

どういう方が出展をされているのですか？　本誌の読者も出展できるのでしょうか？

本職でハードウェアや組み込みのエンジニアをしている人が多いですが、ソフトウェアエンジニアの方ももちろん、教師の方、アート活動をしている人など、本当にいろいろな方が出展されています。ソフトウェアエンジニアの出展者さんは、お仕事以外で趣味として触ることができるものを作りたいという気持ちがあるようです。技術的には素人のような私からすると、彼・彼女らは魔法使いのような人たちで、なおかつ来場する方をおもしろがらせようとするサービス精神に溢れた素敵な出展者さんが多いと思います。

Maker Faireのおもしろさというのはどういうところにあるのでしょうか？

多様性の幅広さでしょうか。いたるところで、見たこともないような、おもしろいものを作っている人がいて、それを楽しそうに説明してくれる。ものすごく最先端のテクノロジを使ったハイセンスな作品から、超アナログなユーモアのある作品まで、会場のすべてを1日で見て、理解するのは難しいぐらい、いろいろなものがあります。この『多様性の幅広さ』言い換えれば『懐の深さ』がMaker Faireの味わいの1つなのではないかな、と個人的には思っています。

懐の深さというと、鹿野さんの生き方と似ているところがあるように思いますね(笑)。ユニークな生き方、Maker Faireの見どころポイント、今日はたくさんのことを教えていただきました。ありがとうございました。 SD

注1）現在は、分社化したMaker Mediaにより刊行。



ツボイの なんでもネットに つなげちまえ道場

第二回

LED点滅の極意（前編）

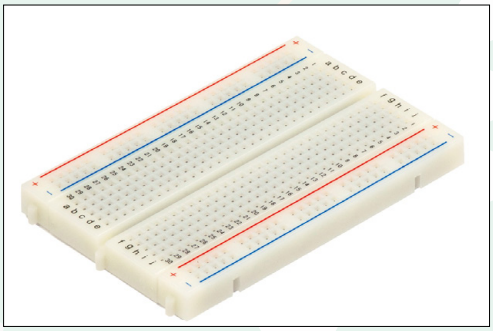
Author 坪井 義浩（つぼい よしひろ） Mail ytsuboi@gmail.com @ytsuboi
協力：スイッチサイエンス

ブレッドボード

前回は、電圧と電流、抵抗やLEDの話をしました。次は、LEDをマイコンで点灯させてみることにしましょう。その前に、部品同士をつなげる方法を紹介したいと思います。電子機器の組み立てというと、まず、基板にはんだづけが連想されると思います。しかし、ちょっと部品をつなげて実験をするにはブレッドボード（**写真1**）を使うのが手軽です。

ブレッドボードは、はんだづけを行わずに、電子部品や配線（ジャンパ線）を挿し込むだけで電子回路を組むことができる道具です。本来はソルダーレス・ブレッドボードと呼ぶのが正しいのですが、たいていは略してブレッドボードと呼びます。ブレッドボードには、電子部品についているリード線を挿し込むための穴が2.54mm間隔で空いています。この穴どうしは、ブレッドボードの内部で決まったルールで電気的に接続されています（**図1**）。このルールをうまく活用することで、ブレッドボードを使って配線を簡単に行えるようになっています。

▼写真1　ブレッドボード



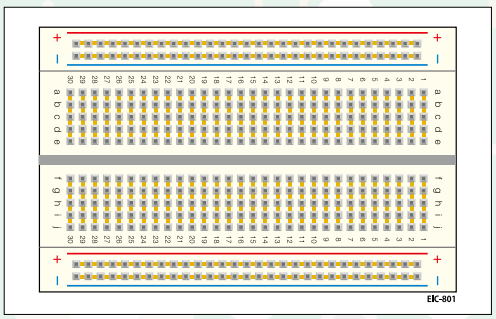
スルーホール実装

2.54mmというのは0.1インチです。最近では少なくなってしまいましたが、リード線などの長い端子が付いた電子部品の多くは、端子の間隔が0.1インチの倍数であるためです。こういった長い端子は、基板の穴（スルーホール）に挿し込んではんだづけをするためのものです（**写真2、3**）。この方法はスルーホール実装と呼ばれます。一方、スルーホール実装と比較して基板のスペースを取らない、表面実装という方法が近年では一般的で、たいていの電子部品は表面実装用に作られています。

表面実装というのは、基板の表面のはんだづけをしたい個所にペースト状のはんだを塗り、その上に部品の端子を置いてはんだづけ[注1]する方法です。スルーホール実装用の部品のようにリード線がないため、そのままではブレッドボードに挿し込むことができません。こういった電子部品をブレッドボードに挿し込めるように変

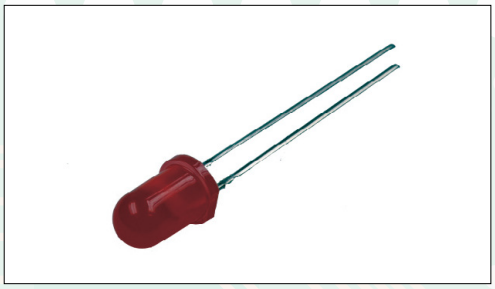
注1）表面実装のはんだづけは、リフロー炉と呼ばれるオーブンで熱を加え、塗ったはんだを融かして行います。

▼図1　ブレッドボードのスルーホールの内部接続



※図中黄色線の部分が内部で結線されています

▼写真2　スルーホール実装用のLED



換基板に実装した、Breakout（ブレークアウト）という製品がいろいろと販売されています（**写真4**）。ブレッドボードで電子回路を組むときには、Breakoutを使ったりスルーホール実装用の部品を使ったりします。

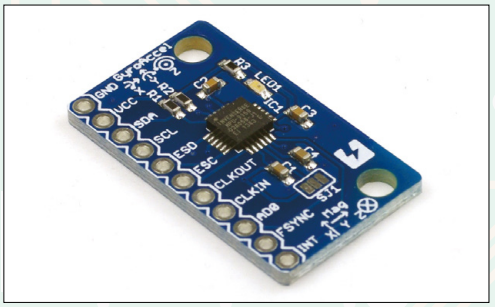
マイコンボード

次にLEDを点滅させるために使うマイコンボードを選びましょう。今日メジャーなプラットフォームとしては、Arduino（アルドゥイーノ）、mbed、Raspberry Pi（ラズベリーパイ）（**写真5**）、Edison（エジソン）などが挙げられます。

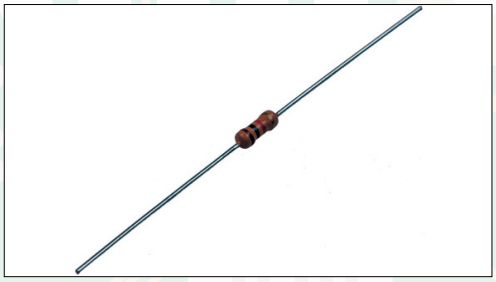
Raspberry PiやEdisonはLinuxが動きますから、IPを喋らせるのがとても楽ちんです。また、PythonやPerl、あるいはRubyなどのスクリプトを走らせられます。本誌読者層には馴染みやすい環境だとは思いますが、せっかくIoT的な連載をしていくのですから、こういったリッチなマイコン[注2]ではなく、低消費電力な組み込みの世界に近い環境を紹介していきたいと思い

注2）　これらのボードは世代交代が早いので、本連載の間に型が変わってしまう可能性が大きいという事情もあります。

▼写真4　ブレークアウトの例（表面実装部品が搭載されている）



▼写真3　スルーホール実装用の抵抗

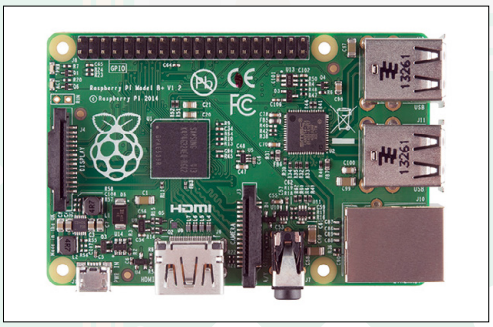


ます。

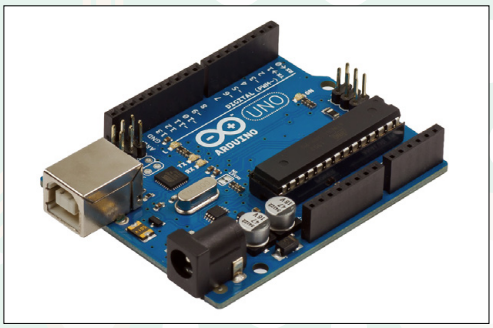
Arduinoは最もメジャーなプラットフォームです。使い始めるのが容易で、メジャーがゆえにインターネットにも情報が最も豊富です。しかし最もメジャーなArduinoであるArduino Uno（**写真6**）など、Arduinoは8bitのプロセッサが主流で、それ単体でIPを喋らせるのが困難です。Arduinoは、TCP/IPのスタックをハードウェアとして搭載しているチップを使ってネットワークに接続できるようにすることもできますが、少々面倒でもあります。

mbedは、マイコンの設計図を販売しているイ

▼写真5　Raspberry Pi



▼写真6　Arduino Uno



ギリスのARMという会社のプロジェクトで、学生などが同社のARMアーキテクチャのマイコンを利用したプロトタイピングを可能にするために始まったものです。mbedにも複数種のマイコンボードがあるのですが、当初から販売されているmbed LPC1768(**写真7**)が最も一般的で、Ethernetも搭載されています。この連載では、当面mbed LPC1768を例に進めたいと思います。

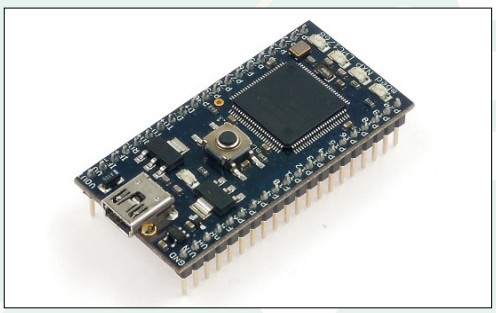
クラウド開発環境

mbedの特徴の1つに、mbedのプログラムを開発する環境がクラウドで提供されており、Webブラウザを使ってコードの記述からコンパイルまで行えることが挙げられます。Webブラウザでアクセスして書いたコードのコンパイルに成功すると、マイコンに書き込むためのバイナリファイルのダウンロードが自動的に開始されます。一般的には、コンパイルして生成したマイコンのバイナリ(実行ファイル)を書き込むには、各マイコンに応じたアダプタが必要です。mbedの場合は、ボードに書き込みアダプタの機能も搭載されており、Webブラウザでダウンロードしたファイルを、USBフラッシュドライブに書き込むようにドラッグ&ドロップするだけでマイコンへの書き込みが完了します。

つまり、マイコンの開発をするときに一般的に必要な、コンパイラなどのツールやマイコンへの書き込みを行うツールをmbedの場合は用意する必要がありません。自分が書いたコードを含めて開発環境はクラウドに存在するため、Webブラウザでmbedのwebサイトにログインするだけで、どのパソコンからでも、いつも同じ環境を使うことができます。

mbedの開発は、今時のWebブラウザとUSBを使うことができればよいので、開発マシンのOSを選びません。比較的新しいChromeやFirefox、あるいはSafariやInternet Explorerが動けばよいので、WindowsでもOS XでもLinuxでも開発が可能です。

▼写真7　mbed LPC1768



アカウントを作る

mbedの開発環境を使うため、まずはアカウントを作りましょう。`https://developer.mbed.org`にアクセスしてください。

アカウントを作るには、まずページの右上にある「Login or signup」と書かれたボタンをクリックします(**図2**)。次の画面で「Signup」と書かれた紺のボタンをクリックすると、アカウント作成ページに移動できます(**図3**)。メールアドレス、ユーザ名、名前などを聞かれます。ユーザを作成すると、ログインした状態になります。

画面上部に「Platforms」というボタンがありますので、ここをクリックします。このページにはmbedのクラウド開発環境で開発できるボードが、この記事執筆時点で60種類以上登録されています。mbedのクラウド開発環境が対応しているボードは、ものすごい勢いで増え続けています。ここでは、一番左上の「mbed LPC1768」をクリックします。

mbed LPC1768のページ(**図4**)には、このボードを使うために参考になる情報がいろいろと掲載されています。このボードを自分のクラウド開発環境で使えるようにするには、「Add to your mbed Compiler」というボタンをクリックしてください。なお、その下にある「Buy Now」というボタンをクリックすると、mbed LPC1768を売っているオンラインストアの一覧ページを見ることができます。ボードの入手に困っていたら、このページの下のほうにある「Asia

Pacific」というところを参考にするとよいでしょう。

コンパイルしてみる

「Add to your mbed Compiler」をクリックして自分の環境にmbed LPC1768が追加されると、ボタンが「Open mbed Compiler」に変わります。このボタンをクリックすると、クラウド開発環境が起動し、サンプルプログラムを環境に追加するダイアログが開きます（**図5**）。「OK」ボタンを押してサンプルプログラムを追加してみましょう。

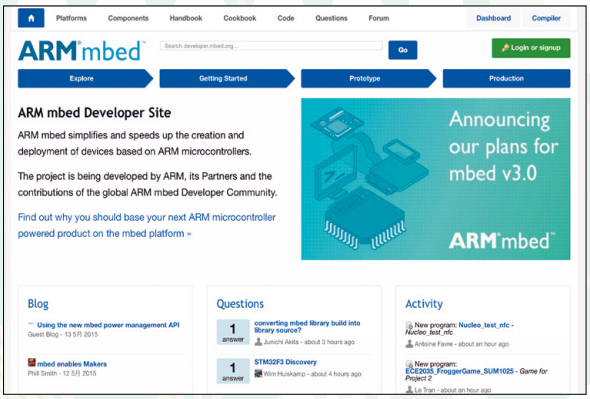
追加した状態では、今回追加をした「mbed_blinky」というプログラムが選択されています。ここでツールバーにある「Compile」ボタンをクリックすると、コンパイルが実行され、生成されたバイナリのダウンロードが始まります。ダウンロード先はWebブラウザの設定によって決まっているので、Webブラウザでダウンロードしたときにいつも保存されるフォルダを確認してください。「mbed_blinky_LPC1768.bin」というファイルがダウンロードされているはずです。

mbedをUSBでパソコンに接続するとき、開発に使うパソコンがWindowsの場合のみ、USBシリアルのドライバ（**https://developer.mbed.org/handbook/SerialPC**）をインストールする必要があります。mbed LPC1768をパソコンに接続し、認識されたドライブにこのファイルをドラッグ＆ドロップしてコピーします。

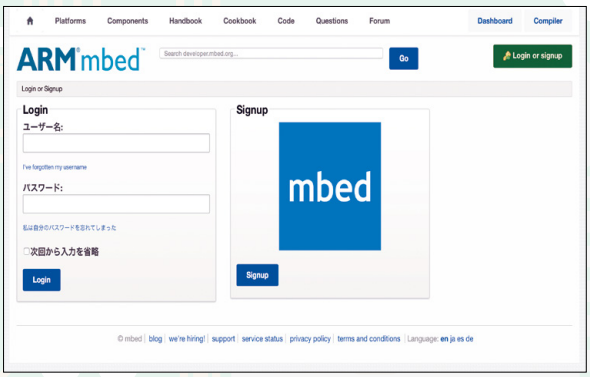
コピーが終わったら、mbed LPC1768についているリセットボタンを押します。するとボードについているLEDが点滅を始めます。

次回は、ブレッドボードを使って、外付けしたLEDを点滅させてみましょう。SD

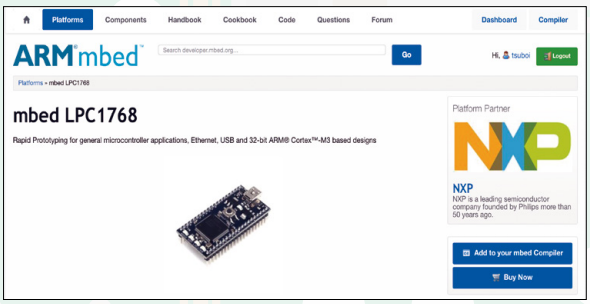
▼図2　mbed.org



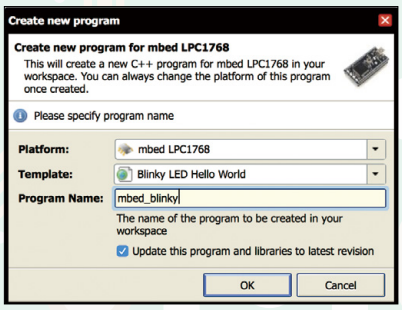
▼図3　Login or signup画面



▼図4　mbed LPC1768のページ



▼図5　Create new programダイアログ

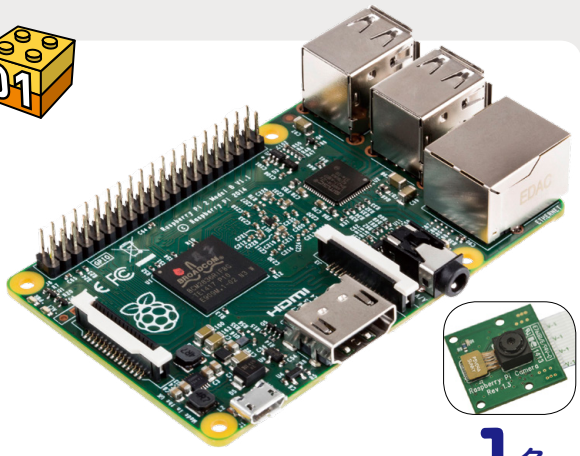


読者プレゼントのお知らせ

『Software Design』をご愛読いただきありがとうございます。本誌サイト**http://sd.gihyo.jp/**の「読者アンケートと資料請求」からアクセスし、アンケートにご協力ください（アンケートに回答するにはgihyo.jpへのお名前と住所のアカウント登録が必要となります）。ご希望のプレゼント番号を記入いただいた方の中から抽選でプレゼントを差し上げます。締め切りは**2015年8月17日**です。プレゼントの発送まで日数がかかる場合がありますが、ご了承ください。

ご記入いただいた個人情報は、プレゼントの抽選および発送以外の目的で使用することはありません。アンケートの回答は誌面作りのために集計いたしますが、それ以外の目的ではいっさい使用いたしません。記入いただいた個人情報は、作業終了後に責任を持って破棄いたします。

01 「Raspberry Pi 2 Model B」&「Camera module」セット



1名

旧型「Raspberry Pi Model B+」から、速さ最大6倍（クワッドコアARM Cortex-A7）・メモリ容量2倍（1GB RAM）と格段にパワーアップしました。さまざまなデバイスと接続して自分だけのガジェットが作れます。今回はRaspberry Pi本体と接続できる、5Mピクセルセンサー搭載の「カメラモジュール」とセットでのご提供です（ロゴ入りのペンケースもお付けします）。

提供元 アールエスコンポーネンツ　http://jp.rs-online.com

02 iPhone用モバイルバッテリー Power Tube 3000

Lightningケーブル・コネクタ、USB入力ポート内蔵で、これ1つでiPhoneに給電、本体を充電できます。専用アプリ「JuiceSync」をiPhoneに入れることで、バッテリー残量、気温、iPhoneとの距離をモニターできます。容量は3,000mAh。赤と黒を1名様ずつ提供します。

提供元 MIPOWジャパン
http://www.mipow.co.jp

2名

03 Linux Foundation／Red Hat ノベルティTシャツ

LinuxCon Japan 2015で配られたLinux FoundationとRed Hatのノベルティ Tシャツです。『Project Atomic』は“Dockerに最適化されたRHEL”と言われる「Atomic Host」の開発プロジェクトです。どちらか1枚を提供します。

提供元 レッドハット
https://www.redhat.com/en/global/japan
Linux Foundation
http://www.linuxfoundation.jp

3名

04 その数式、プログラムできますか？

著：アレクサンダー・A・ステパノフ、ダニエル・E・ローズ

データ形式に依存しないプログラミング手法「ジェネリックプログラミング」をテーマに、数学の歴史を遡る本です。歴史上の数学的証明・アルゴリズムを、C++11のコードを載せながら解説しています。

提供元 翔泳社
http://www.shoeisha.co.jp

2名

05 ITプロジェクトの英語

著：塚本 俊、小坂 貴志

企画・設計・開発から、保守・管理・評価まで、ITプロジェクト全体の流れを解説しながら、その場面々々で役立つ英語表現を紹介します。無料でダウンロードできる英語音声を聴きながら勉強できます。

提供元 ジャパンタイムズ
http://bookclub.japantimes.co.jp

2名

06 Scalaファンクショナルデザイン

著：深井 裕二

プログラミング言語Scalaについて、その関数型機能に焦点を絞って解説した1冊。簡潔で短いコード例と豊富な図で、Scalaにおける関数型機能の使い方を効率的に学べます。

提供元 三恵社
http://www.sankeisha.com

2名

07 ゲームプログラマのためのコーディング技術

著：大圖 衛玄

「わかりやすいコード」「効率よく機能を追加できる設計」といった、ゲームプログラマはもちろん、それ以外のプログラマにも求められるコーディング技術を学べます。サンプルコードはC++11です。

提供元 技術評論社
http://gihyo.jp

2名

第1特集

Lispより始めよ、されば救われん！

なぜ関数型プログラミングは難しいのか？

Lisp、Scala、Haskell、Elixir、Python、Clojure、関数型のエッセンスを学習する

「なぜ関数型プログラミング」は難しいのか。その問いに応えるべく総力特集を組みました。まず原点に戻るためLispの基礎を解説しました。歴史を振り返りながら、数学の関数との違い、Lispの特徴紹介とその使いこなしをまとめました。

そして現代です。チャットワーク㈱様ではPHPアプリケーションをScalaに移植しました。その過程で開発者達の間で起きたさまざまな知見を公開します。㈱はてな様では多くのサービスをScalaに直してきた実績から、小さな部品を組み合わせ大きなプログラムを作っていくためのノウハウを解説いただきました。人気のHaskellは数学と物理での利用例をベースに、どのように数式をコード化するのか実例を紹介します。そしてWeb開発系での本命とも言われるElixirは導入方法から学習方法まで一気に解説！

Pythonも関数機能をベースにその特徴を紹介します。最後はJava上のLispであるClojureです。ライブコーディングをはじめとして、楽しく今風に関数型を学ぶやり方を公開します。

Contents

第1章

気軽に試してみよう!

今こそLisp入門
関数型プログラミングとは何か

本章では、関数型プログラミングの源流であるLispの世界に触れ、その基本的な文法やしくみを学びます。決して特殊なスタイルで閉じた世界のコンピュータプログラミング言語ではないことがわかるでしょう。そして後の言語にいろいろな影響を与えてきたこともわかるようになるでしょう。恐れず気軽にLispを試してみましょう。

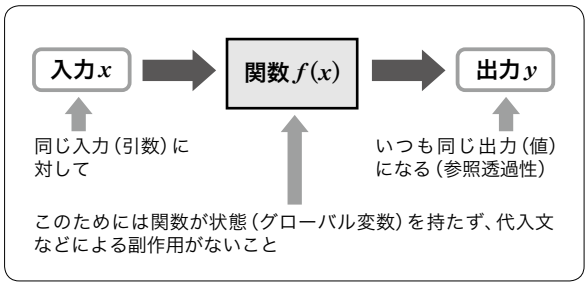
Author 五味 弘(ごみ ひろし) 沖電気工業㈱ mail gomi@gomi.info

関数型プログラミングとは? それがなぜ難しいのか?

Lispを知る前に、まず今回の特集である関数型プログラミングとは何かを見ていきましょう。そして、この関数型プログラミングがなぜ難しいのか、または難しいと思われているのかを考えてみます。

最初に関数とは、$y=f(x)$のような数学でお馴染みのものです。この関数の考えをそのままプログラミングするのが**関数型プログラミング**です。このように考えると関数型プログラミングは数学を習った一般の人にとっては特別なものではありません。ここで、数学でいう関数とは、**図1**のように同じ入力に対して、いつも同じ結果を返すものです。数学を習った方には、これは当たり前だと思えるでしょう。

▼図1 数学の関数



でも考えてみてください。プログラミングの世界では、これが当たり前と言えるでしょうか。たとえば、**リスト1**のプログラムを考えてみます。

数学の関数は状態を持たず副作用も生じませんが、プログラミングの世界では**リスト1**のように「普通」に状態を持ち、副作用を簡単に生じさせてしまい、同じ入力でも違う結果が出てしまいます。これは数学の関数ではありません。

ここで副作用と書いたため、それ自体が悪いように感じたかもしれませんが、プログラミングの世界ではグローバル変数や代入文は基本的な技術であり、強力な武器です。もっと言えば、プログラム格納方式であるノイマン型コンピュータにとっては、根本的な機能です。関数型プログラミングではこの根本的な機能を使いません。つまりプログラマにとっては今までの副作用を積極的に使うプログラミングと違うので、難しく感じるのです。

ここで少し振り返ってみましょう。今までのプログラミングは、データ構造として配列を多用し、グローバル変数を気の向くまま使い、繰り返し文でプログラムを制御し、代入文で好きなようにグローバル変数や配列の要素の値(プログラムの状態)を変えて副作用を積極的に使う、いわゆる俗世のプログラミングでした。

▼リスト1 状態を持ち副作用が生じるプログラム

```
int z = 0;          //このグローバル変数で、このプログラムは「状態」を持ってしまう
int f(int x){
     z = z + x;     //この代入文で zの値が変わるという「副作用」を与えてしまう
     return z;      //この結果、同じ入力に対しても、違う結果を返してしまう
}                   //これは、もはや数学の「関数」ではない
```

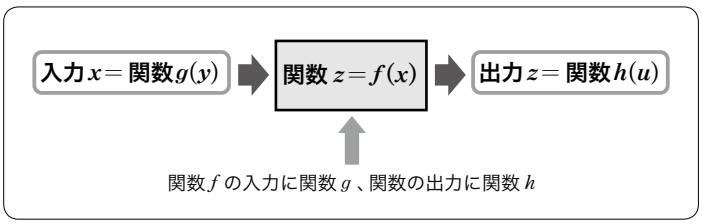
▼リスト2　繰り返し文で制御し、状態を持ち、その副作用を利用した階乗のプログラム

```
int fact(int n){
    int ret = 1;                    //現時点の計算結果の状態を持たせるための変数
    for (int i = 1; i <= n; i++){   //繰り返し文によってプログラムを制御する
        ret *= i;                   //変数に現時点の計算結果の状態を代入し、retに副作用させる
    }
    return ret;
}
```

たとえば、階乗の計算プログラムは**リスト2**のように書いていました。

この俗世のプログラミングからグローバル変数禁止や代入文禁制、副作用禁則の戒律を堅く守る修行僧のような厳しいプログラミングの世界に拉致されてしまうのではないかという恐怖から、関数型プログラミングはプログラマから恐れられているかもしれません。

▼図2　高階関数



また数学では、**図2**に示すように関数を普通の値と同じように、関数と演算子の引数(入力)にでき、さらに関数の結果(出力)にもできる「**高階関数**」の考えがあります。

関数型プログラミングでも同様に関数を普通のもの(これを**ファーストクラスオブジェクト**と格好良く呼んでいますが、簡単に言えば、普通のオブジェクトのこと)として扱えるようになっています。つまり関数も1や"abc"のような普通のオブジェクトのように、引数やリターン値として使えるのです。このように関数型言語では高階関数があることから、何か高級なことや難しいことをしているように思われているかもしれません。いつも難しいことを考えている学者のような人がプログラミングしていると思われているのでしょう。

でも大丈夫です。Cであれば関数ポインタを使うことにより、関数を普通の値と同じような感覚で使えます。Java には昔は匿名クラス、今はラムダ(lambda)式があり、普通の値として使えます。決して関数型言語だけが高級なことをしているわけではありません。

そして安心してください。代入文を使わず、副作用がない清らかな世界だけど、プログラマにとって住みにくい世界に拉致することはしません。今回の特集では、関数型プログラミングをするために、決して、難しいことを考える学者になるのでもなく、いろいろと禁欲する修行僧になることを勧めているのではありません。本特集を読んでもらえれば、関数型プログラミングが楽しみながら組めることになると思います。

関数型プログラミングを勧めるこれだけの理由

前の節で関数型プログラミングとはどういうものかと、どうして関数型プログラミングが難しいと思われているのかを紹介しました。ここでは逆になぜ今、関数型プログラミングをするのか、そしてなぜ流行っているのかということを考えてみます。関数型プログラミングを勧めるにはこれだけの理由があります。

①関数型プログラミング向きの問題がある

筆者たちは多くの種類の問題に対してプログラミングしていますが、そのなかには従来のプログラミングよりも関数型プログラミングに適した問題もあります。状態を持たずに再帰的な構造をしている問題がそうです。たとえば、情報処理技術者試験によく出てくる探索や整列プ

ログラムは、再帰プログラミングを使った関数型プログラミングがぴったりでしょう。将棋プログラムの思考ルーチンで先読みするところも再帰がぴったりでしょう。

②人間は再帰的思考が向いている

人間の思考は再帰的に考えることに向いているというのがあります。大きい問題を同形の小さな問題に分解して考える数学的帰納法が自然だとするものです。たとえば、年間計画を期別計画に落とし、月間計画、そして週間計画にするというのもそうでしょう。社長から部長へ指示を出し、部長から課長、課長から社員、そして最後に社員が仕事をして、その結果を課長、部長経由で最後に社長まで連絡するのもたぶん再帰的でしょう。しかしプログラマ脳では再帰よりも繰り返しで考えるのが理解しやすいかもしれませんが。

③並列処理が自然にできる

状態を持たず副作用がない関数型プログラミングでは、並列処理が自然に行えます。同期処理や排他処理をする必要がありません。クラウドコンピューティングやビッグデータ処理などでは有効になる技術です。

④動的なプログラミングが自然にできる

関数を引数にしたり、リターン値にすることができ、関数自身もデータとして扱えるので、動的に関数を生成したり変更することができます。これは動的で柔軟なサービス指向のコンピューティングに向いています。

⑤抽象的なプログラミングが自然にできる

上記の動的なプログラミングとも相関がありますが、抽象的なプログラミングができるというのもあります。抽象的にプログラミングできますので、仕様変更に強いプログラムを作ることができます。さらに状態を持ちませんので、これからも仕様変更に強い柔軟なプログラミングができます。

⑥グローバル変数や副作用を使う機会が減るのでバグが少なくなる

グローバル変数や代入文などによる副作用は、バグの温床になっています。諸悪の根源です。知らない隙に誰かが勝手にグローバル変数の値を滅茶苦茶にしていることがあります(多くの場合は犯人は自分自身ですが)。さらに悪いことに、その愚行を見つけるのは、砂漠の中の砂粒を見つけるくらいに非常に困難です。関数型プログラミングではグローバル変数や副作用はいざ鎌倉というときにしか使いませんから、品行方正なプログラミングができます。

⑦型推論が行える静的型付け言語では、コンパイル時にバグが多く取り除ける

関数型プログラミングの一般的な性質ではありませんが、型推論が行える言語では、コンパイル時に型に関する多くのバグが取り除けるメリットがあります。

⑧動的型付けで型宣言のわずらわしさから解放される

これも関数型プログラミングの一般的な性質ではありませんが、プログラマがメモリ管理から解放されたGC(ガーベッジコレクション)の機能と同様に、動的型付けを行う関数型言語では、型宣言のわずらわしさから解放されます。

◆ ◆ ◆

上記の理由には、やや我田引水的な点もありましたが、このように多くの関数型プログラミングを勧める利点があります。とくに並列処理に向いていることや、逆に欠点であった多くのリソースを使うことが最近のコンピュータの高機能化によりあまり問題にならなくなったことから、関数型プログラミングが今、流行ってきています。さらに関数型プログラミングを勧める理由は上記のほかに、⑨数学的で何かかっこよさそうとか、⑩古いけど何か新しいそうだというものがあります。むしろ、こちらのほうが受けがいいかもしれません。

Lispを勧めるには理由がある

前の節では関数型プログラミング言語を勧める理由を紹介しました。この節では関数型プログラミングの第一歩として、Lispをお勧めしています。もちろんLispには関数型プログラミングのための機能があることは当然の理由になります。ここではほかの関数型プログラミング言語ではなく、Lispを勧める5つの理由を紹介します。Lispの基礎は次の節で紹介しますので、ここではLispの細かいところは気にせずにLispの良いところを覚えてください。

①Lispは覚えることが少ない

Common Lispのように大きなLisp処理系もありますが、一般的にはLispの言語仕様は小さいので、覚える量が少なくて済みます。まさに関数型プログラミングの入門として、最適な言語です。

②Lispはインタプリタですぐに部分的に実行できる

Lispはインタプリタを持っていますので、コンパイルすることなく、すぐに実行できます。またJavaのようにすべてを完成させてから実行する必要はありません。作ったところから実行できます。たとえば(+ 1 2 3 4)とすれば、すぐに10が返ります。簡単な電卓としても使えます。これをJavaで書くと、たいへんな目にあうでしょう。Javaプログラミングをせずに電卓を使うことになるでしょう。なお(+ 1 2 3 4)は、1＋2＋3＋4をLisp で書いたもので、次の節で説明しますので、ここでは気にしないでください。

③Lispには呪文は不要

Java でいつも必要となる呪文(たとえば`public static void main(String[] args)`のような呪文)は不要です。Lispはプログラミングするために、こんな呪文を覚える必要はありません。実行したいところだけ書けばいいのです。

④Lispには歴史がある

ほかの関数型言語にはない歴史があります。このため、Lisper(Lispを信奉し伝道する人)は多くいます。ほかの関数型言語をやっている人もきっと元Lisperか隠れLisperです。このため、Lispを語り合える仲間が多くいます。

またGC(ガーベッジコレクション)やラムダ(Lambda)式などの最近のプログラミング言語の流行は、Lispから始まっています。

⑤Lispは動的言語である

Lispは動的型付けをするために、いちいち型を宣言する必要がありません。簡単で便利な機能です。そしてこれは数学の関数で型をあまり宣言せずに使うことにも通じる自然なものです。関数はLispのデータとして扱えるために動的に生成や変更も容易です。

MLのような型推論をする静的型付けの言語とLispの動的型付け言語は、いつもどちらが優れているかという宗教戦争をしています。しかし後の章で静的型付け言語の紹介もありますので、ここでは動的言語の宣伝はこれだけにします。

◆ ◆ ◆

大きな声では言えませんが、Lispにはデメリットもあります。①括弧が多くて見にくいとか、②前置記法が見にくいとか、③動的型付けのために多くのメモリと実行時間が必要だとか、④その実装が面倒であるとかがあります。でも上記のメリットを読めばわかるように些細なことです。そうですよね、読者の皆様。

ところでLisp処理系は多くのものが無料で公開されています。ここではISO規格になったISLispの処理系の1つを紹介します。

・OKI ISLisp(http://islisp.org/index-jp.html)

以降の節では、このISLispを例にとって、Lispを紹介していきます。

CやJavaプログラマでもわかるLispの基礎

ここからいよいよLispの心髄を紹介していきます。まずこういう言葉があります。「プログラミング言語には2種類ある。この2種類とはLispとそれ以外の言語である」という言葉です。これぐらいLispは、ほかの言語と比較して特徴がある言語です。

最初に前の節で紹介した階乗のプログラムを再帰プログラムに変更したものを紹介します(**リスト3**)。

(0)再帰プログラミングの考え方とそのコツ

Lispの説明に入る前に再帰プログラミングの考え方を紹介します。再帰プログラミングは**図3**に示すように、再帰プログラミングする関数f(x)を(i)f(x-1)を使って記述し、次に(ii)ベースになるf(0)を記述します。こうすると、f(5)を実行すると、(i)を使ってf(4)を呼び出し、f(4)の中でf(3)を呼び出し、最後にはf(1)の中でf(0)を呼び出し、(ii)を使ってベースのf(0)は具体的な値を返します。そのf(0)の値をf(1)が受け取りf(1)の値が決まり、f(2)に返されます。最後f(5)の値が返されます。階乗のときの流れは**図3**を参照してください。

この再帰プログラミングは数学的帰納法と同じ考えになります。この再帰プログラミング(と数学的帰納法)のコツはずばり「再帰」を見つけることです。この再帰の発見のコツは「同じことをしている」部分を見つけることです。階乗であれば、「5の階乗は4の階乗の結果に5を掛けること」という再帰を見つけることです。

これを5の階乗は1×2×3×4×5のように繰り返しであるという考えは駄目です。手続き型の考えに毒されています。この再帰の考え方に慣れてください。

次に**リスト3**のプログラムをLispで書いてみると**リスト4**のようになります。**リスト4**で紹介した最初のLispプログラムはどうでしたでしょうか。この**リスト3**と**リスト4**のプログラムを比較すれば、Lispのだいたいの感覚はつかめるかと思います。そしてそんなに難しくないものだとわかるかと思います。これで安心して、次からLispの特徴を説明できます。

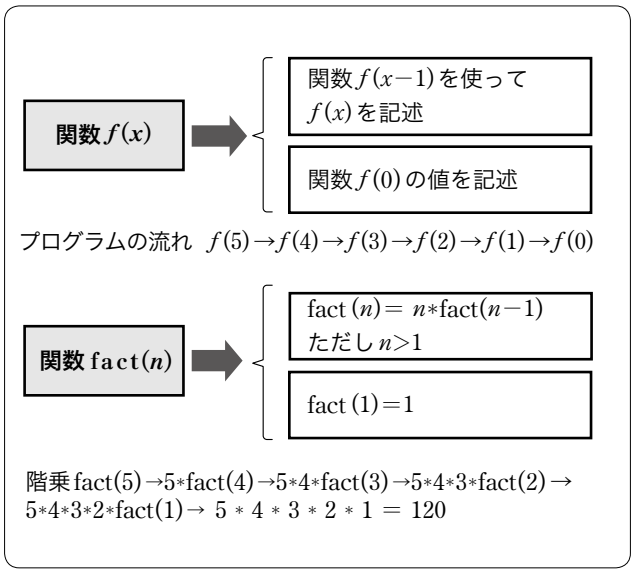
▼リスト3　再帰呼び出しによる階乗のプログラム

```
int fact(int n){
  if (n <= 1) return 1;          //nが1以下であれば、1を返す
  else return n * fact(n - 1);  //nが1より大きければ、再帰呼び出し
} // このプログラムは状態を持たず副作用もない関数プログラム
```

▼リスト4　Lispによる階乗の計算プログラム

```
(defun fact (n)
  (if (<= n 1)
     1                          ; nが1以下であれば、1を返す
     (* n (fact (- n 1))) ))  ; nが1より大きければ、再帰呼び出し
```

▼図3　再帰プログラミング



(1)Lispは1 + 2と書かずに(+ 1 2)のように書く前置記法

小学校で算数を習っているときから、四則演算などの演算子を引数と引数の間に置く中置記法(代数記法)を使っています。たとえば、1と2の足し算は

算数の時代から1＋2と書いていました。もちろん、CやJavaもそのように記述します。一方、関数の表記は$f(x)$や$\sin\theta$のように、数学でもCやJavaでも前置記法です。一方、Lispでは四則演算のような演算子もすべて関数になります。演算子がなく関数だけのLispはすべてこの前置記法になります。つまりLispでは1＋2を(+ 1 2)と表記します。この表記がほかの言語と一線を画すものになります。

前置記法を採用しているため、演算子の優先順位を気にする必要はありません。優先順位を制御する(x + y) * (z - u)のような括弧も使う必要はありません。逆に言えば、演算子の優先順位は(* (+ x y) (- z u))のようにプログラマが括弧で全部書く必要があります。

(2)Lispは型を気にせずにプログラミングできる動的型付け

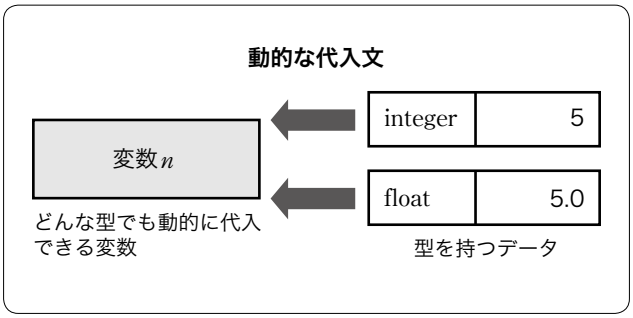
リスト4のLispプログラムでは引数宣言(n)のところで型宣言がないことに気づくと思います。そうなのです。Lispでは処理系が実行時に動的に型を処理していて、プログラマが気にする必要はありません。これを動的型付けと呼んでいます。

たとえば、**リスト4**の関数で(fact 5)を実行するとnの値は整数型の5になり、実行結果として整数型の120が返り、(fact 5.0)を実行するとnの値は浮動小数型の5.0になり、120.0が返ります。つまりLispは内部では型を持っていて、その型に最適なコードになっていますが、それをLisp処理系自身が動的に割り付けていますので、プログラマに意識させることはありません。この様子を**図4**に示します。

(3)階乗計算をするための関数や特殊形式の紹介

リスト4で使っていたLisp の関数や特殊形式を紹介します(**図5**)。

▼図4　動的型付け



▼図5　階乗計算をするときに使った関数

```
(defun 関数名（引数1 引数2 ...）関数本体 ...）――関数定義
(if 条件 then節 else節）――分岐
(* 引数1 引数2 ...）――乗算（他の四則演算も同様）
(<= 引数1 引数2）――比較（他の比較も同様）
(関数 引数1 引数2 ...）――関数呼び出し
```

defunは関数定義のdefine-functionの省略形になりますが、これ以外の関数はCやJavaプログラマでもすぐに推理できるかと思います。前置形式に惑わされなければ、Lispはそんなに掛け離れた言語でもありません。

上記の関数を使って、たとえばフィボナッチ数の計算プログラムは

```
(defun fib (n)
    (if (<= n 1)
        1
        (+ (fib (- n 1)) (fib (- n 2))) ))
```

のようになります。括弧の多さに惑わされないでください。

(4)Lispの評価と特殊形式quote、function

(foo (bar ...) (baz ...))を実行するときは、括弧の先頭にあるシンボルfooを関数名として、その関数を引数(bar ...)や(baz ...)とともに実行します。このときにbarやbazのリストも同様に実行されます。

Lispではプログラムを実行することを評価(eval)すると呼んでいます。(+ 1 2)を評価すると3になります。また引数を評価せずにそのまま返す特殊形式quoteがあります。(quote (1

▼図6　リスト操作の代表的な関数

・(car リスト)――「リストの先頭要素を取り出す」

例.(car '(1 2 3)) → 1

・(cdr リスト)――「リストの先頭以外の残りのリストを取り出す」

例.(cdr '(1 2 3)) → (2 3)

・(cons 要素 リスト)――「要素をリストの先頭に加えたリストを生成する」

例.(cons 1 '(2 3)) → (1 2 3), (cons (car list) (cdr list)) → list
(cons 3 nil) → (3) → nil
「nilはリストの末尾を示すシンボル」
(cons 1 2) → (1 . 2)
「リストの末尾がnilでないものはドット対と呼ばれる」

・(append リスト ...)――「リストを連結する」

例.(append '(1 2 3) '(4 5 6)) → (1 2 3 4 5 6)

・(list 要素 ...)――「要素のリストを返す」

例.(list 1 2 3) → (1 2 3)
これは (cons 1 (cons 2 (cons 3 nil)))と同じ

・(length リスト)――「リストの長さを返す」

例.(length '(1 2 3))→3

・(member 要素 リスト)――「要素がリストの中にあるかどうかを返す」

例.(member 2 '(1 2 3)) → (2 3), (member 4 '(1 2 3)) → nilは偽を表すシンボルとしても使われる

2 3))を評価すると、何もせずに引数(1 2 3)がそのまま返ります。

quoteがないと、(1 2 3)は1を関数名として評価することになり、そのような関数はないというエラーが出ます。また(quote (1 2 3))の略記法として、'(1 2 3)のように書きます。このquoteと似ているものにfunctionがあります。これはシンボルの関数を返す特殊形式で、たとえば(function +)とすれば、関数+の実体が返ります。また(function +)の略記法として、#'+のように書きます。

(5) Lispのデータ型の中心はリスト

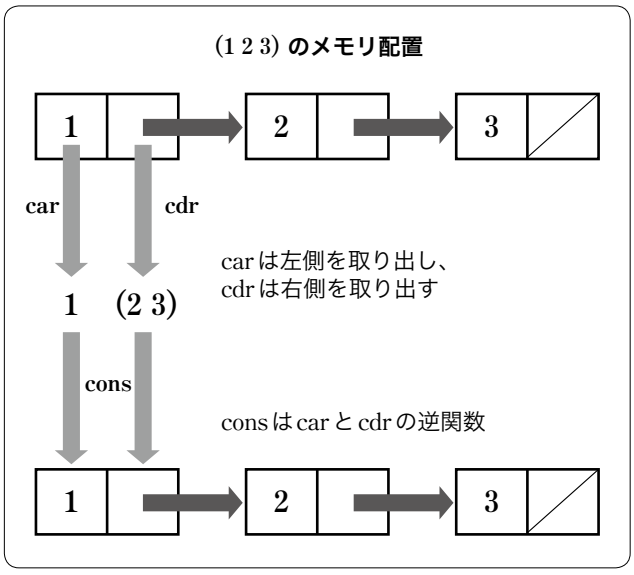
Lispは一般の言語で多用される整数型、文字や文字列型などももちろんありますが、データ型の中心はリストです。Lispの名前がリストプロセッシング(List processing)から来ていることからもリスト処理が中心になっている言語です。ここではLisp特有のデータ型を紹介します。

(i) リスト

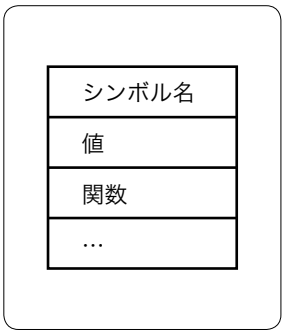
リストは複数のデータを要素として1列に並べるデータ型です。リストの例としては(1 2 3)や(1 (2 3) 4)、(foo (bar 1 2.0 #¥a "abc") '(1 2 3))があります。また()は要素がない空リストです。そして重要なことはLispのプログラム自身もリストで表現されます。つまりプログラムもデータと同じ形式で、両者を区別せずに同等のものとして扱えます。これからLispは柔軟な構造と振る舞いを持つ言語になります。リストを処理する代表的な関数として、**図6**のものがあります。

図7にリスト(1 2 3)のメモリ上の表現と、リスト関数car(カー)、cdr(クダー)、cons(コンス)の流れを紹介します。**図7**の1個の箱が1個のデータを表し、2個の箱の組をセルまたはコンスと呼んでいます。斜め線が入っている箱がリストの末尾nilを表

▼図7　リスト関数のcar、cdr、cons



▼図8　シンボルのメモリ配置



▼リスト5　2引数版のappend

```
(defun append2 (list1 list2)
  (if (null list1) ; nullは引数のリストが空のときに真になる関数
      list2
      (cons (car list1) (append2 (cdr list1) list2)) ))
```

▼図9　述語関数の代表的な関数

・(eq 引数1 引数2)――「ものが同じか(アドレスが同じか)どうかを判断する。等しいときはt(真)を、違うときはnil(偽)を返す」

```
例.(eq 1 1) --> t
   (eq 'abc 'abc) --> t
   シンボルは名前が同じときは同じアドレスに同じものとしてインターンされる
   (eq '(1 2 3) '(1 2 3)) --> nil
   内容は同じでもアドレスが違う別物なのでnilになる
```

・(equal 引数1 引数2)――「内容が同じかどうかを判断する(深い比較をする)」

```
例.  (equal '(1 2 3) '(1 2 3)) --> t, (equal 1 1) --> t
```

・(null リスト)――「リストが空のときにtを返す」

```
例. (null '(1 2 3)) --> nil, (null ()) ---> t
```

しています。nil(ニル)はラテン語の「無」の意味で、ニヒルな単語です(古語のニヒルnihilが変化してnilになっています)。

関数appendの2引数版の関数append2は、**リスト5**のように再帰プログラミングで実装できます。

簡単な例(append2 '(1) '(2 3))を頭の中で想像して、ちゃんと(1 2 3)が生成されることを確認してみてください。最初は(cons 1 (append2 () '(2 3))となり、次に(cons 1 '(2 3))となり、最終的に(1 2 3)が生成されることを確認してみてください。

(ii) シンボル

シンボルは**図8**のように関数や値などを格納できるもので、Lispでリストと双対を成すデータ型です。たとえば、シンボルに実行中に動的に関数を代入することもできますので、柔軟なプログラムが書けます。シンボルが括弧の先頭にあると関数として扱われ、先頭以外にあると変数として扱われます。

また'abcや(quote abc)とすると、関数や変数として扱われず、シンボルとして扱われます。さらにシンボルはインターン、つまり同じ名前のシンボルは同じものとして同じアドレスに配置されています。'abcと'abcを評価したものは同じものになります。

(iii) 真理値と述語関数

t(**真**)とnil(**偽**)で表現します。また約束として空リスト()とnilは同値です。**図9**に比較や等価性を判断する述語関数(真理値を返す関数)を紹介します。

(iv) 関数

関数もデータ型の1つで、ほかのデータ型と同じように引数やリターン値として使えます。詳しくは次の(6)を参照してください。

(6) 関数プログラミングに使う関数の紹介

関数プログラミングで使うLispの関数を紹介します(**図10**)。

(lambda (x) (+ x 1))は「**ラムダ式**」と呼ばれ、これを評価すると、**図12**に示すような

▼図10　関数プログラミングのための関数

```
・(lambda (引数1 引数2 ...) 関数本体 ...)――「ラムダ式の定義」
・(apply 関数 引数リスト ...)――「関数適用」
・(mapcar 関数 引数リスト ...)――「マップ関数(他のマップ関数も同様)
```

▼図11　クロージャの生成とその評価

・(i) (lambda (x) (lambda () (+ x x)))

これは引数xを入力として、無引数の匿名関数(lambda () (+ x x))を返す関数です。これを評価すると変数xの値を環境に閉じ込め、それを無引数の匿名関数の荷物(環境)として持ち運ぶことができます。

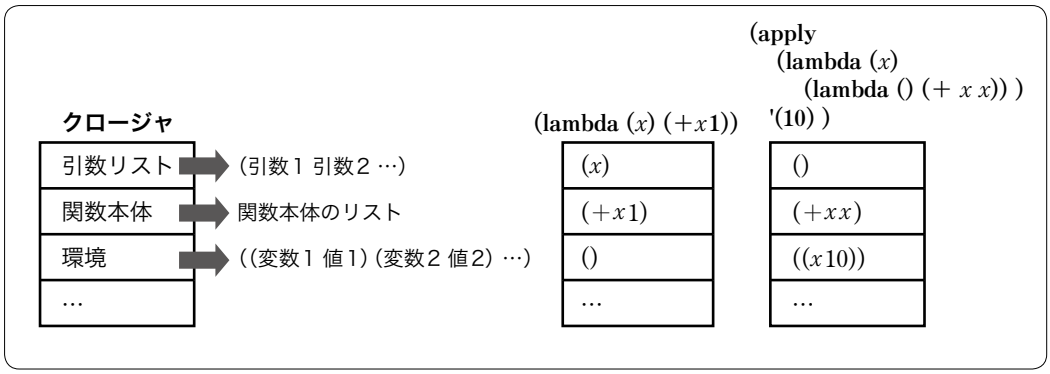
・(ii) (defglobal fun (apply (lambda (x) (lambda () (+ x x))) '(10)))--- 環境((x 10))を生成

xに10の値を束縛させて適用すると無引数の関数が生成され、それがfunに格納されます。funに格納された関数はxの値が10である環境((x 10))を持ち歩いています。defglobalはグローバル変数の定義に使います。これについては次の(7)を参照してください。

・(iii) (apply fun ())--- 環境((x 10))を持ち運んで、関数適用

funには(ii)で生成された無引数の匿名関数が束縛されています。このfunを上記のように無引数で適用すると、持ち運んだ環境からxの値は10なので、(+ x x)が評価され、20を返します。

▼図12　クロージャのメモリ配置



関数(引数はxの1個で、その引数を1つ加算する関数)を生成します。lambdaに続くリストが引数リストになり、この引数の値を使って関数本体が実行されます。Cの関数ポインタやJavaの匿名クラス、ラムダ式と同じものです。

そしてこの関数が普通のオブジェクトとして、どこでも気楽に使えます。(apply (lambda (x) (+ x 1)) '(10))を実行すると、10がxに束縛(xの値として10がセットされる)され、ラムダ式の中の(+ x 1)が実行し、11が返ります。2引数以上のときは、たとえば、(apply #'+ 1 2 3 '(4 5 6))とすると21が返ります。最後の引数がリストになるところがポイントです。

どうでしたでしょうか。ラムダ式はわかりましたでしょうか。関数型言語の説明でこのラムダ式を中心に解説しているものが多いですが、無理に使うことはありません。関数型プログラミングに慣れてきて、ラムダ式の便利さや必要性がわかってから使い始めてもまったく問題がありません。

ここで少し難しい話をします。どうしても関数型プログラミングでは「**クロージャ**」を出さなくては格好がつかないのでLispらしく括弧を付けて、少し触れるようにします。そして、関数型プログラミングをしているという見栄を張るためには必要な知識です。でも安心してください。クロージャがわからなくても関数型プログラミングはできます。きっぱりと言い切れます。

クロージャとは関数に「関数定義時」の変数の値を閉じ込めた「環境」を持ち運ぶことができる便利なしかけです。実行時でなく関数定義時というのが大事な点です。これはLispには昔からある考えで、Smalltalkのブロック文などにも導入された便利なしかけです。関数型言語を作る側から見ると面倒なしかけなのですが。**図11**にクロージャの例がありますので見てください。これらのクロージャの実装例を**図12**に示します。実際の実装では、環境をリストで実

▼リスト6　1引数版のmapcarを再帰プログラム

```
(defun mapcar1 (lambda list)
 (if (null list)
     nil
     (cons (apply lambda (list (car list)))   ; mapcarはこのようにcarを実行
           (mapcar1 lambda (cdr list)))))
```

装するのは遅いので配列で実装されているでしょう。

マップ関数は関数を引数として実行する高階関数(今回は2階関数)で、たとえば、次のように実行されます。

```
(mapcar (lambda (x y) (+ (* x 10) y))
            '(1 2 3) '(4 5 6)) → (14 25 36)
```

mapcarのリターン値が(14 25 36)になったことから、mapcarの機能を、推理してみてください。そしてmapcarの便利さを実感してください。ここで試しに1引数版のmapcar関数mapcar1を再帰プログラミングしてみます。mapcar1は関数を引数lambdaに受け取る高階関数になります(**リスト6**)。

さっそく、この関数を使ってみましょう。(mapcar1 (lambda (x) (+ x 1)) '(1 2 3))を評価すれば、(2 3 4)が返ってきます。

ラムダ式やマップ関数を使えば、いかにも関数プログラミングぽくなり、きっと見栄を張ることができるでしょう。もちろん、実利もありますので、見栄を張って使ったあとでは、それをじっくりと堪能して、自分のプログラミング技術の1つにしてください。

(7)Lispの手続き型機能

ここでは逆にLispが持つ手続き型機能を**図13**に紹介します。つまりLispは純粋な関数型言語ではなく、手続き型言語の面も持ちます。これはLispの暗黒面でなく実用的な面です。

手続き型の機能は**図13**以外にも多数用意されています。これはLispは修行僧のような純粋関数型言語でプログラミングをするのではなく、手続き型のいいところは取り入れている実用的な関数型言語なのです(自画自賛が30%ほど入っています)。副作用を起こすプログラムは禁止だとか、代入文はいっさい使わないだとか、グローバル変数は世界を破滅させるから使うなという縛りはLispにはありません。

(8)Lispは柔軟な言語

今まで紹介してきたように言語は小さく、動的な型付けがあり、関数を普通のオブジェクトとして扱え、関数はデータと同じリスト構造をしているなど、非常に柔軟な言語と言えます。たとえば自分自身を拡張することが容易な言語になっています。このため、プログラミング言語に新規の機能を入れる実験にも使われています。たとえば、オブジェクト指向機能は早期にLispで実装されました。

▼図13　Lispの手続き型機能

・(defglobal シンボル 初期値)――「グローバル変数の定義」

```
例. (defglobal my-name "GOMI Hiroshi")
→Lisp のシンボルは - なども使える
```

・(setq シンボル 値)――「代入文」

```
例. (setq pi 3.14159) , (setq pi "円周率")→どんな型でも代入できる
```

・(let ((引数 初期値) ...) 本体 ...)
――「引数を初期値に束縛して、本体を評価」

```
例. (let ((x 10) (y 20)) (+ x y)) → 30
これは (apply (lambda (x y) (+ x y)) '(10 20))と同じ結果になる
```

・(for ((ステップ変数 初期値 ステッパ) ...)(終了条件 終了値) 繰り返し...)
――「繰り返し」

```
例. (for ((i 0 (+ i 1))) ((> i 10) i) )
→CやJavaのfor(int i = 0; i <= 10; i++){}とほぼ同じ
繰り返し文には、ほかにもwhileがある
(Common Lisp の場合は dotimes や dolist、loopマクロがある)
```

(9) Lispは関数型言語

今さらな紹介になりますが、Lispは関数型言語です。今まで見てきたように関数型を破壊する関数もありましたが、基本的には関数型言語です。とくにcar, cdr, cons, eq, atom(シンボルや数型、文字型、nilなどの値型をアトムと言い、その判断を行う関数)の基本5関数 とif, quote, lambda, defunだけを使った「純Lisp」は純粋な関数型言語です。純Lispでない、手続き型操作も含んだLispもその根底に流れる思想は関数型言語になっています。

そのほかにもLispには強力なマクロ機能があり、Lisp自身の拡張も容易にできます。以上がCやJavaと比較してのLispの特徴で、これをLispの基礎として紹介しました。

(10) Lispのサンプルプログラム

ここでいくつかのLispのサンプルプログラムを紹介します。

(i)「クイックソート」

最初に情報処理技術者試験によく出てくるプログラム「クイックソート」を再帰プログラミングしてみます。クイックソートはピボットと呼ぶ基準値をデータから選び、そのピボットを使って与えられた比較関数で2種類に分けます。2種類に分けたデータに対して同様にこの操作を繰り返します。これをデータが1個になるまで行うことでソートするプログラムです。このプログラムを**リスト7**に示します。

どうだったでしょうか。比較関数を引数にすることにより、昇順でも降順でもほかの順序でも自由にソートできるところが便利だと感じてもらえればと思います。

次にこのクイックソートをピボットで振り分けるところを繰り返し文にしたハイブリッドなプログラムで作ってみます(**リスト8**)。実行結果は**リスト7**と同じになります。

この繰り返し文と再帰のハイブリッドプログ

▼リスト7 「クイックソート」のLispプログラムとその実行

```
(defun qsort (lambda list)
   (if (null list)
       list
      ; クイックソートのピボット(基準値)は先頭の値(car list)にする
       (qsort2 lambda (car list) (cdr list) nil nil) ))
;; lambda比較関数、pピボット、listデータ、left比較でtrue、
;; rightそうでないもの
(defun qsort2 (lambda p list left right)
   (if (null list)
       (append (qsort lambda left) (cons p (qsort lambda right)))
;; ピボットpよりもlambdaなものをleftに、そうでないものをrightに入れる
       (if (apply lambda (car list) (list p))
           (qsort2 lambda p
                   (cdr list) (cons (car list) left) right)
           (qsort2 lambda p
                   (cdr list) left (cons (car list) right)) )))
```

・実行

```
ISLisp>(qsort #'< '(3 1 5 2 4))
(1 2 3 4 5)
ISLisp>(qsort #'> '(3 1 5 2 4))
(5 4 3 2 1)
```

▼リスト8 「クイックソート」の繰り返し文と再帰プログラムのハイブリッドプログラミング

```
(defun iqsort (lambda list)
   (if (null list)
       list
       (let ((p (car list))
             (left nil)
             (right nil))
         ;; この繰り返し文によってピボットでデータを振り分ける
         ;; Common Lisp では(dolist (n list) ...)になる
         (for ((n (cdr list) (cdr n)))
              ((null n) nil)
              (let ((e (car n)))
                 (if (apply lambda e (list p))
                     (setq left (cons e left))
                     (setq right (cons e right)) )))
         ;; 以下は再帰プログラミングにする
         (append (iqsort lambda left)
                 (list p)
                 (iqsort lambda right)) )))
```

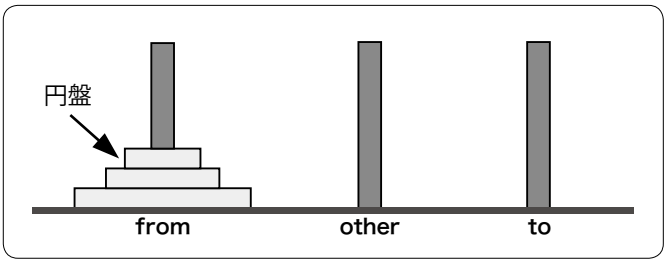
ラミングはいかがでしょうか。Lispだから再帰ばかり使う必要はありません。ピボットの振り分けは繰り返し文のほうがわかりやすいと思えば、気楽に繰り返し文を使ってください。そして再帰が必要なところは再帰を使ってください。ハイブリッドプログラミングこそ、実用的な関数型プログラミングです。

(ii)Lispのサンプルプログラム「ハノイの塔」と「4本ハノイの塔」

Lispで書いたハノイの塔のプログラムを例として紹介します。ハノイの塔は**図14**のようにfromの柱にあるすべての円盤を1枚ずつtoの柱へ移動するものですが、そのときに小さな円盤の上に大きな円盤は置けません。

このハノイの塔のプログラムを**リスト9**に紹介します。このプログラムが理解できたら、次は4本ハノイの塔を作ってみることにします。4本ですから、3本よりも世界が滅びるのが早くなることがわかります(**リスト10**)。

▼図14　ハノイの塔



▼リスト9　「ハノイの塔」のLispプログラムとその実行

```
(defun hanoi (n) (hanoi3 n 'from 'to 'other) ) ; nは円盤の枚数、fromからtoへ移動させる
(defun hanoi3 (n from to other)                ; fromは移動元、toは移動先、otherはワーク用の柱
    (if (= n 1)
        (cons (cons from to) nil)              ; 円盤が1枚ならfromからtoへ移動させて終了
        (append                                ; 円盤が2枚以上のときは
     (hanoi3 (- n 1) from other to)            ; 最初にfromからotherへ
          (hanoi3 1 from to other)             ; 次にfromからtoへ
          (hanoi3 (- n 1) other to from) )))   ; 最後にotherからtoへ移動させる
                                               ; appendはリストを連結させる関数
```

・実行

```
ISLisp>(hanoi 3)    ; 上記のプログラムを実行
                    ;; シンボルは大文字でインターン(登録)される
                    ;; (cons 1 2)は(1 . 2)のようなドット対と呼ばれるデータになる
((FROM . TO) (FROM . OTHER) (TO . OTHER) (FROM . TO) (OTHER . FROM) (OTHER . TO) (FROM . TO))
                    ;; 最初の1手は(FROM . TO)、つまり FROMにある大きさ1の円盤をTOへ移動する
                    ;; これを繰り返すと、FROM にあった3枚の円盤がすべてTOへ移動する
```

▼リスト10　「4本ハノイの塔」のLispプログラム

```
(defun hanoi4 (n from to other1 other2)
   ;; 2枚以内であれば、3本ハノイの塔と同じ
   (if (< n 3)
       (hanoi3 n from to other1)
       (if (= n 3)
           ;; 3枚のときは最初の1枚をother1に移動し、後は円盤2枚の3本ハノイと同じで、
           ;; 最後に最初の1枚をother1からtoへ移動して移動完了
           (append (list (cons from other1)) (hanoi3 2 from to other2) (list (cons other1 to)))
           ;; 4枚以上あるときは、いくつかのアルゴリズムがあるが、ここでは次の
           ;; アルゴリズムにしている。これを解読して、さらに改良してください
           (append (hanoi4 (- n 3) from other1 other2 to)
                   (hanoi3 3 from to other2)
                   (hanoi4 (- n 3) other1 to other2 from) ))))
```

(iii)Lispのサンプルプログラム「エイトクイーン」

最後にエイトクイーンのパズルを解くLispプログラムを紹介します。エイトクイーンとはチェスのクイーンの駒をチェス盤に、ほかのクイーンの利いている位置に置かないようにして8個配置するパズルです(**図15**を参照)。チェス盤は8×8のサイズで、クイーンは縦と横、斜めに盤の端まで移動できます。8人の女王様の機嫌を損ねることなく、ちゃんとご配置しなければなりません。

リスト11で紹介するプログラムは8人の女王だけでなく、n人の女王を配置できるように拡張したnクイーンの解法プログラムになっています。

実行結果のリストの要素(例.(4 2 7 3 6 8 5 1))が解の1つになります。要素のリストは女王のy座標を示していて、それが逆順のx座標に対応するリストとして格納されています。実行結果の最初の(4 2 7 3 6 8 5 1)が**図15**に配置した図に相当します(*x*座標はリストの先頭が8で最後が1であることに注意してください)。リストの長さが92個であることから、エイトクイーンでは92通りの解があることがわかります。

次に斜めチェックの関数diagonalを繰り返し文で作ったプログラムidiagonalを**リスト12**に示します。nクイーンは基本的なところでは再帰で作成しますが、それ以外を繰り返し文で作ったハイブリッドプログラミングにしても

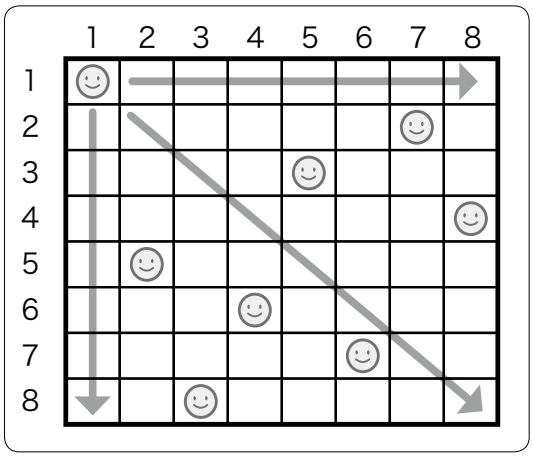
▼リスト11 「nクイーン」のLispプログラム

```
(defun nqueen (n)
  (nqueen2 n 1 nil) )

;;; 引数n --- 女王の人数(盤の大きさ)
;;;       y --- 配置しようとする女王の縦位置
;;;       board --- 盤(縦位置(y座標)を要素とするx座標のリスト(x座標は降順))
;;; 返り値 --- y以上n以下のy座標に配置できる全パターンをリストにして返す
(defun nqueen2 (n y board)
  (if (> y n)
      nil
    ;; (member y board)は横報告に他の女王がいるかどうか
    ;; (diagonal 1 y board)は斜め方向に他の女王がいるかどうか
    ;; 縦方向は女王を1個ずつしか配置しないことでチェック不要
     (if (or (member y board) (diagonal 1 y board))
         (nqueen2 n (+ y 1) board) ; 他の女王がいたときは次のy座標にする
         (append ; 以下の2つのリストを連結する
                 ;; yの位置で配置できるパターン
                 (if (= (length board) (- n 1))
                    (list (cons y board))   ; 最後の女王が配置できたとき
                    (nqueen2 n 1 (cons y board)) ) ; 次の女王を配置する
                 ;; y+1からnの位置で配置できるパターン
                 (nqueen2 n (+ y 1) board) ))))

;;; queenの位置に女王が置けるかどうか
;;; 駒が置けなければTを返す
;;; boardの長さ分のチェックをする
(defun diagonal (x queen board)
  (if (null board)
      nil
      (if (= (abs (- (car board) queen)) x) ; 斜めチェック
          t                ; absは絶対値を取る関数(例. (abs -3) ---> 3)
          (diagonal (+ x 1) queen (cdr board)) )))
```

・実行

```
ISLisp>(nqueen 8)
((4 2 7 3 6 8 5 1) (5 2 4 7 3 8 6 1) (3 5 2 8 6 4 7 1) ... (5 7 2
6 3 1 4 8)) ; 92個の配置パターン
```

▼図15 エイトクイーン



▼リスト12 「nクイーン」のプログラムの一部を繰り返し文にする

```
(defun idiagonal (x queen board)
  (block nil                                    ; return-fromでこのblockを抜け出す
    (for ((i x (+ i 1)))
         ((null board) nil)                     ; nilを返す
      (if (= (abs (- (car board) queen)) i)     ; 女王が斜めにいたら
          (return-from nil t)                   ; tを返す(CやJavaのbreak文に相当)
          (setq board (cdr board)) ))))
```

女王様はお怒りにならないでしょう。

Lispの歴史

前の節ではLispの基礎を紹介してきました。その中でCやJavaプログラマでもお馴染みのものや、いかにも関数型プログラミングならではのものも紹介してきました。ここではLispそのものの知識から離れて、Lispの歴史を紹介していきたいと思います。Lispの誕生から現在のLisp事情までの歴史になります。これは関数型プログラミングの歴史にもなり、基本を学ぶのに役立つことはもちろん、Lisperとして見栄を張るのに使えます。

(1)ジョン・マッカーシーによるLispの黎明期(1950年代~1960年代)

Lispを語るには、ジョン・マッカーシーを真っ先に挙げなければなりません。ジョン・マッカーシーが1950年代にLispに関する研究を行ったのがLispの始まりです。LISP1やLISP1.5が作られました。ちなみに後に続くCommon Lispなどが大文字でシンボルを規定しているのは、この時代は大文字しか使えなかった理由によるものです。

(2)Lispの戦国時代(1970年代)

その後、Lisp はその柔軟で動的な言語であったために、新しいプログラミング言語やそのしかけを作るのに利用されていました。このため、1970年代にはLispには多くの方言が出てきて激しい派閥争いが行われていました。Schemeもこの時代に生まれました。

(3)Common LispによるLispの統一(1980年代~1990年代前半)

1980年代になると人工知能(AI)の研究が流行し、このための言語として柔軟で動的なLispが注目されました。AIマシンとして、Lispマシンも開発されていました。一方、Lispに多くの方言があるのは不便でしたので、これを統一する目的で1980年代から1990年代にかけ、ガイL.スティールジュニアが中心となって、Common LispがANSI規格として制定されました。

(4)新たな戦乱と統一への息吹(1990年代後半~2000年代前半)

統一されたかに見えたCommon Lispですが、Common Lispは巨大な言語仕様になってしまい、不便さを感じる場面もありました。そこでCommon Lispの核の部分を再構成して、小さな共通のLisp処理系を設計するという動きが日本発として出てきました。伊藤貴康や湯淺太一らによる日本案をベースに1997年にISO規格として承認されました。日本発のプログラミング言語がISO規格になるのは初めてです。しかしISLispは学習目的や実験目的にしていることもあり、普及したとは言い難い面があり、小さなLisp処理系はその後も多く作られてきています。

(5)そして現在のLisp新時代へ

現在はCPU性能が向上し、メモリが大量に搭載されてきて、Lispを動作させるための環境が整っています。そしてクラウドコンピューティングやビッグデータの解析、IoTの進展で並列コンピューティングの必要性や動的実行が必須になっています。このため、関数型プログラミングの必要性が高まってきていることもあり、Lispの世界もまた脚光を浴びています。第7章のClojureもその代表的なものです。この状況は後段の章に譲ることにします。

▼図16　関数型プログラミングのメリットとデメリット

関数型プログラミングのメリット

(1) 問題が再帰プログラミング向きのものが多い
(2) 人間の考え方は再帰プログラミングに向いている
(3) 並列処理に向いている
(4) 動的なプログラミングができる
(5) 抽象的なプログラムが作りやすい
(6) グローバル変数やその副作用によるバグが少なくなる
(7) 型推論ができる(静的言語のとき)
(8) 型宣言をしなくてよい(動的言語のとき)

関数型プログラミングのデメリット(と対処法)

(1) 問題が繰り返し向きのものが多い――繰り返しも強力な武器です、使ってください
(2) 副作用(代入文)のようなプログラミングの強力な武器が使えない――使いましょう
(3) 関数型プログラミングはスタックを大量に消費する――現在は気にしなくていいです
(4) 値指向のデバッグ(特定アドレスの値を中心としたデバッグ)が行えない――スタックでデバッグしましょう
(5) 過去の手続き型プログラミングで培った技術が使えない――すべてはスタックです
(6) 手続き型言語では関数型プログラミングができないという思い込みがある――迷信です
(7) 手続き型から関数型へのパラダイムシフトが必要(敷居が高い)――気楽にシフトです

Lispを使いこなすにはコツがある――Lispのメリットとデメリットを考えて

この節ではLispを使いこなすコツ、そして関数型プログラミングと付き合うコツを紹介します。Lispと、そして関数型プログラミングといい付き合いをするようにしてください。

(1)関数型プログラミングと手続き型プログラミングのいい関係

純粋な関数型プログラミングを目指すべきではありません。過去の莫大な、そして重要な資産である手続き型プログラミングとのトレードオフが重要です。教科書的な書き方をすれば、「関数型プログラミング向きの問題と手続き型プログラミング向きの問題を見極め、どちらにするかを決めてください」となります。同じような議論が形式手法でも聞いたことがあるかもしれません。アジャイル開発でも同じ注意を受けたかもしれません。要するにいいバランスを取ることが大事になります。

このためには何が関数型プログラミング向きで、関数型プログラミングのメリットはどこで、デメリットはどこにあるかを理解する必要があります。メリットは「関数型プログラミングを勧めるこれだけの理由」の節で紹介しましたが、ここではデメリットとともに紹介します(**図16**)。

結局、関数型プログラミングでは、このメリットを使い、デメリットを使わない戦略が必要になりますが、無理をする必要はありません。副作用を積極的に使ったほうがいい場面では、手続き型でプログラミングをしてください。

(2)Lispのデメリットに付き合って、メリットを活かす

Lispのメリットとデメリットは(1)の関数型プログラミングのメリットとデメリットと多くの部分で重なりますが、Lispはすでに紹介したように純粋な関数型プログラミング言語ではありません。多くの手続き型の機能を持っています。これを利用しない手はありません。両者をうまく使い分けてください。

Lisp特有のメリットとデメリットについては「Lispを勧めるには理由がある」の節で紹介したものになりますが、もう一度まとめてみます(**図17**)。

Lispのメリットは小さくて、直ぐに実行でき、動的であることです。このメリットを活かせるかどうかがLispを使うコツになります。デメ

▼図17　Lispのメリットとデメリット（克服法込み）

Lispのメリット（関数型プログラミングのメリットは除く）

(1) Lispは小さい
(2) Lispはインタプリタですぐに実行できる
(3) Lispには呪文は不要
(4) Lispには歴史がある
(5) Lispは動的言語である

Lispのデメリットと克服法

(1) 括弧が多くて見にくい――インデントと空白で見やすくします
(2) 前置記法が見にくい――慣れです、慣れてしまえば勝ちです
(3) 動的型付けのために多くのメモリと実行時間が必要――最近は問題にならないです
(4) 動的型付けの実装が面倒――使う側はそんなことは知ったことではありません
(5) 型推論によるエラー検出が弱い――それよりも動的型付けのメリットのほうが重要です
(6) Lispには方言が多い――自分の使っているLispが一番えらいのです

リットはこのメリットが有効である場合はカバーできるものと信じています。

(3) 関数型プログラミングは、「構えて」使うものではなく、「気楽」に使うもの

たとえば、関数を通常のデータと同じように引数や関数のリターン値に使える高階関数は、前にも書きましたが、これはCでは関数ポインタを使えば、よく似たものができます。

また、Javaでは過去の匿名クラスや今のラムダ式を使えば、同等のことができます。逆にLispでも手続き型言語の機能を多く持っています。代入文もありますし、グローバル変数もあります。この意味では手続き型言語でも関数型プログラミングをでき、その逆に(純粋関数型言語以外の)関数型言語でも手続き型プログラミングができます。

関数型プログラミングは構えて使うものではなく、気楽に使ってもいいものです。今までのCやJavaで関数型プログラミングするのもありで、関数型プログラミングを小さく、早く始めるのが吉です。

◆　◆　◆

ここまで関数型プログラミングとは何か、なぜそれを勧めるのかから始め、関数型プログラミングの第一歩としてのLisp入門を紹介してきました。途中で代入文を使うなとか、高階関数やクロージャあたりで面倒で小難しいことが書いてあったかもしれません。このため、関数型プログラミングは難しいと思われたかもしれませんが、そんなことは気にせずに気軽に関数型プログラミングを楽しんでください。「再帰プログラミングはおいしい、そして副作用は今まで何気なしに使ってきたけれど、実は不思議なものだった」ということと、「繰り返し文による制御も悪くない、代入文や副作用は強力な武器だ」という、2つの相反する考えを自分の中で、うまく折り合いを付けて、関数型プログラミングを楽しんでいってください。このときにLispを一番にお勧めします。きっとLispで関数型プログラミングが楽しく学べることでしょう。

この後に続くいろいろな関数型言語もきっと楽しく関数型プログラミングの世界へ導いてくれることでしょう。Lispについては後の章でもいろいろな観点で紹介されていますので、ぜひ参考にしてください。またLisp以外の関数型言語もおいしいことが後の章で紹介されていますので、それも楽しんでください。関数型プログラミングの世界はあなたをお待ちしております。SD

第2章

サービス改善への解答 PHPエンジニア、Scalaを学ぶ!

チャットベースのコミュニケーションWebサービスを提供するチャットワークでは、1年ほど前から開発言語をPHPからScalaに切り替えました。数ある関数型プログラミング言語からなぜScalaを選んだのか、関数型への移行にはどのような難しさや魅力があったのかをPHPエンジニアの視点で紹介してもらいます。

Author 安達 勇太(あだち ゆうた) ChatWork㈱ Twitter @UAdachi

チャットワーク、なぜPHPからScalaへ?

筆者の勤務するチャットワーク(ChatWork)[注1]はPHPで書かれたWebサービスで、約4年間、増改築を続けてきました。ですが次のような理由により、これ以上PHPで増改築していくことが難しくなってきました。

- メンテナンス面:コードの複雑度が非常に高くなり、機能の追加をしようとしても予期せぬバグやリソースの消費を招く
- パフォーマンス面:もともと大規模サービスを想定したコードやアーキテクチャではなかったため、ユーザ急増への対応が困難

そこでチャットワークのアーキテクチャとコードを刷新する方針が決まり、またその際に、言語やフレームワークについても見直しをしたいというエンジニアの要望もあったため、これらを選定するための合宿を2泊3日で行いました。そのとき選考対象になったのが、PHP、Python、Scalaの3つの言語です。エンジニアを3チームに分け、各チームでチャットワークのAPIを実装し、合宿終了時に各言語・フレームワークの特徴や、使ってみた感想を発表するという流れでした。この合宿の結果、コンパイル時に型検査される静的型付き言語で、並行処理ライブラリが整っているScalaを採用しようと決まりました。

本章では、PHPエンジニアだった筆者が実務を通じて1年間Scalaを学んだ経験をもとに、Scalaの強力な機能や関数型プログラミングの考え方について説明し、そしてこれらをどのようにチームで共有してきたのか、その取り組みを紹介したいと思います。

Scalaって難しい?

Scalaとは

ScalaはJava仮想マシン上で動作するプログラミング言語であり、大きな特徴として、JavaやPHPのようなオブジェクト指向言語と関数型言語の両方の特性を持つということが挙げられます。筆者がそうだったのですが、「関数型プログラミング」と聞くと難しそう、というイメージが先行してしまうかもしれません。もちろん関数型プログラミングの概念を深く学ぼうとすると時間はかかりますが、難しいことを知らなくても強力なパターンマッチや失敗の可能性を表現できる型など、便利な機能を使って楽しくプログラミングをすることができます。

ほかにもScalaには次のような特徴があります。

- 静的型付き言語
- 並行処理
- Javaの資産を利用できる

静的型付き言語

静的型付き言語では、変数や関数の引数や戻り値の型がコンパイラによってあらかじめ検証されます。よって型に起因する不具合は実行す

注1) http://www.chatwork.com/ja/

るまでもなく、コンパイル時に発見できるのでバグを作り込みにくいです。

並行処理

標準ライブラリに、コレクションに対する操作を並列に実行できる「並列コレクション」というデータ型が用意されています。また、処理を並列に実行し、その結果を非同期で取得するための「Future」という型も用意されています。そのほかにも標準ライブラリではありませんが、アクターモデルを採用した「Akka」という負荷分散と耐障害性を兼ね備えたスケーラブルな並行処理のためのライブラリも利用できます。

Javaの資産を利用できる

Scalaの実行環境はJava仮想マシンなので、ScalaからJavaで書かれたプログラムを利用することができます。たとえばAWS(Amazon Web Services)のSDKはScala版が用意されていませんが、Java版のSDKをScalaから利用できます。

PHPとScalaのコードを比較してみる

細かいScalaの機能の説明は抜きにして、「文字列中に含まれる単語数を数える」処理を例に、PHPで書かれたコード(**リスト1**)を、Scalaで書き直すとどのようになるのか(**リスト2**)イメージをつかんでいただきたいと思います。

PHPの例も簡潔なコードになるように心がけてみましたが、Scalaの例はとても短く書けました。コード量だけでなく、そもそもスタイルが違うようです。Scalaの例では小さな関数をつなぎあわせているようにも見えます。むしろ筆者がScalaを始めたばかりのころは、あまりに簡潔過ぎてこのようなコードはすぐに理解できませんでした。

また、静的型付き言語なのに、引数や無名関数の型は書かなくて良いのか？という疑問を抱いた方もいるかもしれません。しかし、後述するScalaの高階関数や関数リテラル、プレースホルダ構文や型推論を理解するだけで、すぐに読みやすいコードだと思えるようになります。

ところでこの問題、実はPHPのほうが簡単に書けます。というのも`array_count_values`という関数が用意されていて、それを呼び出すだけで良いからです。

Scalaの実行環境を用意する

実際に手を動かしたほうが理解しやすいかと思いますので、これから紹介するソースコードを動作させるための環境を用意しておきましょう。下記の手順に進む前に、OSによらずJavaのインストールが必要ですので、Java SE Development Kit 8 Downloads[注2]よりダウンロード後、インストールをしてください。

ここでは、Scalaの対話型評価環境(REPL：Read Eval Print Loop)を用意します。REPLでは、Scalaのコードが入力単位ごとにJavaのバイトコードに変換され、対話的なプログラミングができます。スニペットなど、ちょっとしたコードの動きを確認するときに便利です(2015年6月1日時点でのScalaの最新安定版は2.11.6です)。

▼リスト1　文字列中に含まれる単語数を数える(PHP版)

```
function wordCount($text) {
    $result = [];
    foreach(explode(" ", $text) as $word) {
        if (array_key_exists($word, $result))
            $result[$word] += 1;
        else
            $result[$word] = 1;
    }
    return $result;
}
```

▼リスト2　文字列中に含まれる単語数を数える(Scala版)

```
def wordCount(text: String): Map[String, Int] = {
  text.split(" ").groupBy(identity).mapValues(_.size)
}
```

注2) http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html

Windows

Scalaの公式Webサイト[注3]から対応するtgzファイルをダウンロードして展開します(本稿では「C:¥scala-2.11.6」)。展開後、ユーザー定義環境変数[注4]に「SCALA_HOME」という環境変数を新しく追加し、環境変数「PATH」とともに次の設定を入力してください。

・SCALA_HOMEの値:C:¥scala-2.11.6
・PATHの値(追加):%SCALA_HOME%¥bin

OS X

OS Xをお使いの方は、Homebrew[注5]を使うと簡単にインストールできます。

```
$ brew install scala
```

REPLの使い方

REPLという簡単にコードを試せる実行環境の使い方を説明します。まずはコンソール(コマンドプロンプト)を開いて、scalaコマンドを実行してください。すると次のようなプロンプトが表示されるはずです。

```
scala>
```

試しに変数を定義してみましょう。

```
scala> val name = "Software" ↵
name: String = Software
```

変数はvarとvalというキーワードで定義できます。valは再代入ができないので、この場合正確には「変数に代入する」ではなく「値(value)に束縛する」と呼びます。

複数行にわたるコードをペーストする

:pasteと入力すると、ペーストモードへ移行します。移行後にコードをペーストし、Ctrl+Dを押下してペーストモードを終了します。

終了方法

:qでREPLを終了し、コンソールに戻ることができます。

```
scala> :q
```

PHPエンジニアだった私たちが感じたScalaの魅力

Scalaの便利な機能や関数型プログラミングの考え方について説明をします。その多くがPHPエンジニアだった筆者にとって新鮮なもので、最初はなかなか慣れなかった高階関数や副作用の概念も紹介していきます。

条件分岐を柔軟に記述できるパターンマッチ

まずはモダンな関数型言語に多く見られる、パターンマッチという機能を紹介します。パターンマッチを利用すると、if文やswitch文を使うよりずっと柔軟に条件分岐を記述できます。ここではケースクラスを使ったパターンマッチというものがどのようなものか、ソースコードを見てイメージしてみましょう(**リスト3**)。

最初にUserというトレイトを定義します(**リスト3:02〜04行目**)。Userはpasswordという文字列(String)型のメンバを持っています。そして、そのUserを継承したAdministratorとGuestという2つのクラスを定義します(**07、08行目**)。親のUserトレイトにsealedというアクセス修飾子を付け加えていますが、これによりパターンマッチの際に"すべてのサブクラスを網羅できているか"をコンパイラがチェックしてくれるようになります[注6]。

また、User型のインスタンスを受け取ってBoolean(真偽値)型を返すloginというメソッド

注3) http://www.scala-lang.org/download/
注4) 「コントロールパネル」−「システムとセキュリティ」−「システム」−「システムの詳細設定」−[詳細設定]−[環境変数]から設定。
注5) http://brew.sh/index_ja.htmlを参照。
注6) sealedはUserトレイトが同一ファイル内からしか継承できないような制約を加えます。

を定義しました(10〜16行目)。matchキーワードの前にマッチ対象の値を記述し、各caseの後ろにパターン(条件)、そしてその後ろに処理を記述するのですが、この処理結果の値がパターンマッチの結果となります。

試しにUser型のインスタンスをいくつか定義して、loginメソッドに渡してみましょう(**図1**)。**リスト3**では、

①Administratorクラスのインスタンスで、かつpasswordが"secret"の場合にマッチし、trueを返す
②Guestクラスのインスタンスで、任意のパスワードにマッチし、trueを返す
③①、②以外の場合にfalseを返す

となるようにパターンを記述していて、上から順に評価され、「=>」の後に記述された式の評価結果が返り値となります。またアンダースコア「_」がワイルドカードとして扱われ、任意の条件にマッチします。

パターンマッチの力をもう少しご紹介しましょう。先ほどの例では、Guestはパスワードを入力しなくてもログインが可能でした。これを、1文字以上のパスワードを入力しないとログインできないように変更してみましょう。と言っても、**リスト3：13行目**のcase Guest(_) => trueのパターンを次のように書き換えるだけです。

```
case Guest(password) if password.length > 0 => true
```

パターンの中でpasswordにGuestのパスワードを束縛し、またこのようにifを続けて書くことで追加条件を与えることもできます。これをパターンガードと呼びます。

パターンマッチは柔軟な条件分岐を記述することができる機能で、今回紹介したパターンの書き方以外にもコレクションや正規表現などに対するパターンを記述することができます。

▼リスト3　ケースクラスを使ったパターンマッチの例

```
01: // traitは、PHPのトレイトやインターフェースのように利用します
02: sealed trait User {
03:     val password: String
04: }
05:
06: // ケースクラスは、コンストラクタの引数がクラスのメンバになります
07: case class Administrator(password: String) extends User
08: case class Guest(password: String) extends User
09:
10: def login(user: User): Boolean = {
11:   user match {
12:     case Administrator("secret") => true ❶
13:     case Guest(_) => true ❷
14:     case _ => false ❸
15:   }
16: }
```

▼図1　リスト3の実行例

```
scala> val admin = Administrator("secret") ↵
admin: Administrator = Administrator(secret)

scala> login(admin) ↵
res0: Boolean = true

scala> val badAdmin = Administrator("invalid") ↵
badAdmin: Administrator = Administrator(invalid)

scala> login(badAdmin) ↵
res1: Boolean = false

scala> val guest = Guest("guest") ↵
guest: Guest = Guest(guest)

scala> login(guest) ↵
res2: Boolean = true
```

関数型の機能

次に、関数型言語としての機能をいくつか紹介していきます。

関数を整数や文字列と同じように扱える

Scalaでは関数を整数型(Int)や文字列型(String)と同じように、変数に代入したり、関数の引数に渡したり、関数の戻り値にすることができます。また、整数型や文字列型と同様に、関数を定義するためのリテラルが用意されています。試しに関数リテラルを使って簡単な関数を定義し、値に束縛してみましょう。

```
scala> val twice = (n: Int) => n * 2 ↵
twice: Int => Int = <function1>

scala> twice(2) ↵   変数に束縛した関数オブジェクトを呼び出すこともできます
res0: Int = 4
```

twice: Int => Int = <function1>の部分は、Int型の引数を1つ受け取り、Int型の戻り値を返す関数であることを意味し、この関数を"Int => Int"型の関数と呼びます。ということは、twice変数はval twice: Int => Intのように型注釈を付けて宣言する必要があるように見えます。しかしこの例はそうなっていません。というのも、Scalaコンパイラの型推論によって型注釈を省略できるからです。もちろん明示的に書くことも可能ですが、人間が簡単に推論できる型なら省略したほうが見た目がすっきりすると思います。

高階関数

関数を引数に取ったり、戻り値として返す関数を高階関数もしくは高階メソッドと呼びます[注7]。ここではリスト[注8]に対する操作の例を紹介します。

図2の例では、リストの各要素を2倍し、5より大きい要素から構成される新しいリストを返しています。引数が関数内で一度しか使われない場合に限り、引数名を書く代わりにプレースホルダ_を使用でき、コードを簡潔に書ける便利な構文です。プレースホルダ構文を利用することで、引数の名前を考えたり、不適当な名前の引数や変数に悩まされる時間も減っていきます。

mapメソッドは、リストの各要素を変換する高階メソッドで、変換のための関数を引数として受け取り、新しいリストを返します。filterメソッドは、引数で受け取った条件(この場合はInt型の引数をとってBoolean型の戻り値を返す関数)を満たす要素だけで構成される新しいリストを返す高階メソッドです。

▼図2　リストを使った高階関数の例

```
scala> List(1, 2, 3, 4, 5).map(_ * 2).filter(_ > 5) ↵
res0: List[Int] = List(6, 8, 10)
```

副作用と参照透過性

筆者がScalaを勉強し始めた当初は、どうしてvarではなく再代入のできないvalを使うのか、またどうしてwhileループやPHPのコレクションを反復処理するforeachのようなものがScalaにもあるのに、mapやfilterといった高階関数をつなぎあわせてコードを書くほうが良いのか理解していませんでした。

関数やメソッドは与えられた引数をもとに処理をし、その結果を値として返します。つまり「値を返すこと」が関数やメソッドの主な仕事です。しかし、foreachのような処理結果を返さないメソッドやwhileで書かれた反復処理は、foreach内の処理結果をforeachの外側へ伝えるため、またはループの終了条件を満たすために、メソッドやループの外側の世界の変数を書き換える必要があります。そのためプログラマは変数の状態を常に意識しながらプログラムを書く必要があり、これが予期せぬ不具合を作りこんでしまう原因になりかねません。

関数やメソッドが主な仕事以外の処理を行い、その外の世界の状態が変わってしまうことを副作用と呼びます。副作用の例として次のようなものが挙げられます。

- 変数への再代入
- 入出力(標準入出力、ファイルシステム、ネットワーク通信など)
- 例外の発生

また、関数が副作用を持たず、戻り値が引数によってのみ決まるとき、その関数は参照透過性があると呼びます。Scalaにおいてはvarやforeachのような戻り値を返さない関数の利用を避けるだけでも、無駄な副作用の発生を防ぐことができます。

とはいえ、アプリケーション開発では

注7) Scalaではdefを使って定義するものをメソッドと呼び、それ以外を関数と呼びます。
注8) ScalaのList型は単方向リンクの線形リストです。

データの永続化において副作用を完全になくすことは難しいと思います。関数型プログラミングの設計手法をより深く学び、実践していくことで、避けることができない副作用の影響を局所化していくことができます。

そのほかのScalaの便利な機能

本記事では紹介しきれませんが、Scalaにはまだまだ便利な機能が用意されています。たとえば、「値が存在しない(nullになる)可能性がある」ことを表現できるOption型や「処理に失敗する可能性がある」ことを表現できるTry型やEither型が用意されています。これらを用いることで宣言的で読みやすいコードを記述することができます。

どのように学んでいったのか?

社内や他社との勉強会

これまで紹介してきたような実務で使えそうなScalaの機能などを学ぶ良い機会となったのが、社内やBizReachさんと一緒に開催した勉強会でした[注9]。勉強会では事前に課題を用意し、当日は参加者が持ち寄った解答を順次発表し、レビューしました。課題自体の難易度を上げてしまうとアルゴリズムの検討などScalaに関係ないところで時間を使ってしまうので、「文字列中の単語数をカウント」、「フィボナッチ数を計算する関数を実装」、「2分木データ構造を実装」などのあまり難しくない課題を用意しました。毎回いろいろな解答が出そろい、パターンマッチの使い方や再帰の書き方などを学び、またチーム全体で知識を共有できた良い機会でした。

プロジェクトを進めながらペアプログラミング

プロジェクトを進めながらも、ペアプログラミングを通してコレクションの高階メソッドを覚えたり、効率の良いデバッグ方法やIDEの使い方も知ることができました。少し行き詰まったら、1人で悩んで時間を無駄にしてしまう前に、ほかのチームメンバーとペアプログラミングする習慣がチームにあると良いかもしれません。

ChatWorkとして変わってきたこと

PHPからScalaへの移行を始めてそろそろ1年経ちますが、良い方向に大きく変わってきたことがあります。

- 型推論や高階関数、柔軟なパターンマッチなどを使って、簡潔で読みやすいコードになった
- 副作用を局所化したり、変数への再代入や可変コレクションの使用を抑えることでテストが書きやすくなった
- 型に関する不具合をコンパイル時に発見できるようになった

このような大きな効果があった一方、コード量が増えていくにつれてコンパイルに時間がかかったり、Scalaでの開発経験者はPHPと比べるとまだまだ少ないため、採用活動には苦労をしています。そこでチャットワークではScalaのことはまだ良くわからないけど、関数型プログラミングに興味があるエンジニアを積極的に募集しています[注10]。

本章のまとめ

オブジェクト指向同様、関数型プログラミングの世界は広くてさまざまな考え方や法則があり、本稿ではその一部を紹介しただけです。しかし最初から難しく考えずに、便利だと思ったものから使っていけば良いと思います。その点でScalaは今まで慣れ親しんできたオブジェクト指向をベースに、関数型プログラミングのエッセンスを少しづつ取り入れていくことができるので、関数型プログラミングの入門に適した言語だと感じています。SD

注9) http://c-note.chatwork.com/post/87584062960/bizreach-chatwork-scala
注10) http://recruit.chatwork.com/ja/developer.html

第3章

機能を最大限に活かすコーディング術

Scalaで始める、型安全な関数型プログラミング

小さな部品を組み合わせ、大きなプログラムへ

関数型プログラミングは「簡単で小さな副作用のない部品」を組み合わせて「複雑で大きなプログラム」を作るという性質上、部品の再利用・部品ごとの修正が効きやすいという利点があります。本章では、関数型プログラミングをサポートするScalaの機能を紹介していきます。手元でひとつずつ試しながら、関数型プログラミングに慣れていきましょう。

Author 伊奈 林太郎(いな りんたろう) ㈱はてな　Twitter @oarat　URL http://d.hatena.ne.jp/tarao/

Scalaの魅力

ScalaはJava VM上で動作するアプリケーションを書くための言語で、Javaの標準・非標準のライブラリをそのまま使える、静的型付けのオブジェクト指向言語です。その一方で、副作用のない式を基本とした関数型プログラミングを強く推奨し、そのための言語機能も多く備えています。普段オブジェクト指向プログラミングに慣れ親しんでいれば、オブジェクト指向の伝統的な書き方をしながら徐々に関数型プログラミングを身につけることができ、関数型プログラミングの入門言語としては最適と言えるでしょう。また、動的型付けのスクリプト言語に精通していれば、静的型付けと聞いて、型をいちいち書いて面倒だというイメージを持つかもしれません。実はまったく逆で、Scalaはスクリプト言語のように柔軟に書くための工夫をふんだんにとり揃えています。本稿では、スクリプト言語のようになめらかで生産性の高いコーディングを維持するための工夫にも注意を払いながら、静的型付けによる型安全性のメリットを最大限に活かす、関数型プログラミングのための言語機能を紹介していきます。

インストールとREPLの実行

Scalaの処理系はオフィシャルサイトからダウンロードしてインストールできます(第2章参照)。インストール後、scalaコマンドを実行すると対話型実行環境(REPL)が起動するので、本稿のコード例を実際に実行して試せます。コード例はScala 2.11.6で動作確認しています。

関数型プログラミングのための言語機能

まずは、関数型プログラミングを実現するScalaの機能を見ていきましょう。

不変な値と可変な変数の区別

関数型プログラミングでは、簡単で小さな、副作用のない部品を組み合わせていくことで、複雑で大きなプログラムを完成させます。もし小さな部品に副作用がたくさん出てくると、ある部品の動作で生じた副作用が、同じ部品を用いた別の個所でも影響を及ぼしてしまい、部品の再利用が困難になります。逆に、副作用なしにうまく抽象化された良い部品があれば、そのシンプルな組み合わせだけで複雑なことも整然と実現できるはず、というのが関数型プログラミングの信念といえます。

非関数型プログラミングでもっとも頻繁に登場する副作用は、おそらく変数への破壊的代入でしょう。再利用性の高い部品を作り上げるには、破壊的代入をなるべく排除するのが得策です。しかし一般的には、破壊的代入がないようにプログラマが気をつけるのは難しく、また誰か別の人が書いたコードに破壊的代入があるかどうかを確かめるのも簡単ではありません。

Scalaでは、変数を宣言するときに代入可能な変数(var)かそうでない(val)かを必ず書か

なければなりません。valは宣言時のみ値を指定でき、再代入できません。valしか使われていなければ破壊的代入のないことが誰の目にも明らかですので、なるべくvalを使うことが推奨されています。

```
val i = 3
i = 5 // コンパイルエラー

var j = 3
j = 5 // OK
```

そうはいっても、普段から破壊的代入が当たり前になっていれば、急にそれなしでコードを書けと言われても難しいかもしれません。とくにオブジェクトに対するアクションを、破壊的代入をしないでどうやって表現するのか疑問に思うことでしょう。発想の転換は必要ですが、オブジェクト指向のコードであろうと、副作用のない形で表現することは可能です。

たとえば、**リスト1**は移動可能な2次元座標点を表すクラスPointをvarを用いて書いた例です。これをvalのみの形に書き直すと**リスト2**のようになります。関数move()では、オブジェクトのフィールドに再代入する代わりに変更後のフィールドを持った新たなオブジェクトを作っています。これでPointクラスからは副作用がなくなりましたね。おっと、move()の結果を受けとるptがvarで宣言されていました。これもvalにしてしまいましょう(**リスト3**)。

varで宣言された変数ptに再代入する代わりに、move()する前の値をval pt1に、move()後の値をval pt2に束縛しています。インチキくさいでしょうか。しかしこれが関数型プログラミングの考え方です。オブジェクトの変化を副作用によって表現するのではなく、変化の前の値を残したまま変化の後の値を新たに作っていきます。

▼リスト1　副作用のあるクラス

```
case class Point(var x: Int, var y: Int) {
  def move(dx: Int, dy: Int) {
    x += dx
    y += dy
  }
}

var pt = new Point(10, 20)
println(pt) // Point(10,20)
pt.move(5, 5)
println(pt) // Point(15,25)
```

▼リスト2　副作用のない純粋オブジェクト指向のクラス

```
case class Point(val x: Int, val y: Int) {
  def move(dx: Int, dy: Int): Point =
    new Point(x + dx, y + dy)
}

var pt = new Point(10, 20)
println(pt) // Point(10,20)
pt = pt.move(5, 5)
println(pt) // Point(15,25)
```

必ず値を返す式

全体を副作用なく書いていくと、ほかの動作の結果である新たな値を受けとり続けなくてはなりません。あらゆるメソッドはほかのメソッドの返り値を使って何かを計算し、そして自身の計算結果の値を返すものです。これはメソッドに限らず、条件分岐やループについても同様です。Scalaではifやforも値を返します。

```
val x = if (true) 3 else 5
// => 3
val y = for (i <- 1 to 5) yield(i*i)
// => Vector(1, 4, 9, 16, 25)
```

不変なデータ構造

計算結果の値を次々に返し、受けとっていくやり方をプログラムの隅々まで行き渡らせ、あらゆるデータ構造やクラスに対して適用するのは、容易ではないと思うかもしれません。幸い、Scalaではよく使うデータ構造については、は

▼リスト3　リスト2を修正、全体を通して副作用をなくす

```
val pt1 = new Point(10, 20)
println(pt1) // Point(10,20)
val pt2 = pt1.move(5, 5)
println(pt2) // Point(15,25)
```

じめから不変なスタイルのものが用意されています。

リスト4は不変なスタイルで実装されたList[]とMap[]の使用例です。コレクション値を更新する::や+といった演算子メソッドは、更新された新しいコレクション値を返します。コレクションクラスには要素の参照・更新のためのメソッドだけでなく、全体を走査するためのmap()やfoldLeft()などのメソッドも豊富に用意されています。

独自のデータ構造を定義したい場合には自分でクラスを定義することもできますが、簡単な構造ならタプルで済む場合も多いでしょう。タプルは2つ組や3つ組を表すデータ構造で、組の要素の型はバラバラでも良く、要素の型や個数が違えばまったく別の型として扱われるようなデータ構造で、また次のように入れ子にもできます。タプルのn番目の要素は_nメソッドで参照できます。

```
val t = (1, "foo")
val u = (t, (2, "bar", "baz"))
t._1 // => 1
u._2 // => (2, bar, baz)
u._2._3 // => baz
```

パターンマッチ

不変な値を作って操作するには、単に値を作ったりその内部の要素にアクセスしたりするだけでなく、複雑な構造をいったん分解して、中身を調べていく必要もあります。このときに便利なのがmatch式によるパターンマッチです。

リスト5はパターンマッチを使ってList[]中の任意の同じ値の連続を1つにまとめるメソッドcompress()を実装した例です。まず引数listに対してmatch式でパターンマッチしていきます。最初のcaseはlistが2要素以上の場合にマッチし、その2要素(aとb)と残りの要素(rest)に名前を付けて、条件処理や再帰呼び出しの引数に利用しています。2つめのcaseは要素が1つだけだった場合にマッチします。マッチした部分はとくに使わずにlistをそのまま返せばよいので、パターン中では名前を付けずに_としています。_は任意の構造にマッチするパターンです。最後のcaseはリストが空だったときにマッチします。Nilのように定数もパターンとして使えます。

タプルもパターンマッチで分解できます。また、パターンはcaseだけでなく次のようにvalの宣言時にも使えます。

```
val pair = (1, "foo")
val (fst, snd) = pair
fst // => 1
snd // => "foo"
```

このようにパターンマッチは構文だけとっても十分に便利なものですが、パターンマッチが好まれる特筆すべき理由として「網羅性チェック」があります。たとえば**リスト5**の例で最後のcase Nil => Nilを書き忘れると、Scalaコンパイラは次のような警告を表示します。

▼リスト4　不変なコレクションの使用例

```
val l1 = List(3, 2, 1)
val l2 = 5 :: 4 :: l1
l2.head // => 5
l2(2) // => 3
l1.map { i => i * i }
// => List(9, 4, 1)

val m1 = Map(1 -> "foo", 2 -> "bar")
val m2 = m1 + (3 -> "baz")
m2(2) // => "bar"
m1.keys // => Set(1, 2)
```

▼リスト5　連続する値をまとめる、パターンマッチの利用例

```
def compress[A](list: List[A]): List[A] =
  list match {
    case a :: b :: rest =>
      if (a == b) compress(a :: rest)
      else a :: compress(b :: rest)
    case _ :: Nil       => list
    case Nil            => Nil
  }

compress("aaaabccaadeeee".toList)
// => List(a, b, c, a, d, e)
```

```
warning: match may not be exhaustive.
It would fail on the following input: Nil
```

分岐の漏れがコンパイル時に検出されるので、実行時エラーにおびえる必要はなく、安心感がありますね。静的型付けするのであれば当然ほしい機能をきちんと備えています。

ケースクラス

より複雑なデータ構造を表現するためにmatchで分岐可能な型を自分で定義したいこともあるでしょう。そのような型を簡単に定義する方法としてケースクラスが用意されています。たとえば枝(Branch[])もしくは葉(Leaf[])からなる二分木構造(Tree[])を定義するには**リスト6**のようにします。

Tree[]をsealed traitとして宣言することによって、Tree[]にはLeaf[]とBranch[]以外のサブクラスがないことがコンパイラに伝わり、matchの際の網羅性チェックが働くようになります。

Option型

ここまでで、簡単な部品を使って複雑なものを組み立てられそうな手応えが感じられたでしょうか。副作用もなく、メソッドは必ず値を返し、すべての値にきちんと型がついた世界では実行時エラーが差し挟まる余地はありません。

▼リスト6　ケースクラスによるツリー構造

```
sealed trait Tree[A]
case class Leaf[A](value: A)
    extends Tree[A]
case class Branch[A](left: Tree[A], right: Tree[A])
    extends Tree[A]

def sum(tree: Tree[Int]): Int = tree match {
  case Branch(b1, b2) => sum(b1) + sum(b2)
  case Leaf(v)        => v
}

val tree = Branch(
  Branch(Leaf(1), Leaf(2)),
  Branch(Leaf(3), Leaf(4)))
sum(tree) // => 10
```

とはいえ、現実的なプログラムでは値を返せない状況も発生します。ScalaはJavaとの互換性のためにnullや例外クラスをサポートしていますが、少なくともnullに関しては絶対に使わないほうが良いでしょう。nullを使いつつもNullPointerExceptionが発生しないことを、注意深く設計された型の性質によって保証できる状況というのは存在しますが、型安全性のエキスパートだと公言できるのでもなければやめておくべきです。さもなければ、せっかくの静的型付けも、不変な値で組み上げたきれいな世界も台無しになってしまいます。

では値を返せない場合にScalaではどうするかというと、Option[]型を使います。これは、なんらかの値があることを表すSome(_)か、値がないことを表すNoneのいずれかの値をとる型です。**リスト7**は最初に見つかった偶数を返すメソッドfirstEven()の実装例で、返り値にOption[]型を使っています。

Option[]型の値を受けとったときは、それがSome(_)だったのかNoneだったのかを次のようにパターンマッチで調べられます。ここでもやはり、どちらかのcaseを書き忘れるとコンパイラに警告されます。

```
firstEven(...) match {
  case Some(n) => println(s"Even: $n")
  case None    => println("Not found")
}
```

Option[]型の値に依存した計算結果を返す方法はほかにもあります。まず、getOrElse()を使うとNoneだったときのデフォルト値を与える

▼リスト7　最初に見つかった偶数を返す、Option[]型の例

```
def firstEven(list: List[Int]): Option[Int] =
  list match {
    case a :: rest =>
      if (a % 2 == 0) Some(a)
      else firstEven(rest)
    case Nil       => None
  }

firstEven(List(1 ,2, 3)) // => Some(2)
firstEven(List(1 ,3, 5)) // => None
```

ことができます。

```
firstEven(List(1, 3, 5)).getOrElse(0)
// => 0
```

また、map()を使うと、Some(v)だったときにはvを使った計算結果をふたたびOption[]型にして返し、Noneだったときは Noneのまま返せます。map()に渡した関数の中では、vの値があったものとして計算を進められます。

```
firstEven(List(1, 2, 3)).map { v => v * v }
// => Some(4)

firstEven(List(1, 3, 5)).map { v => v * v }
// => None
```

もしmap()のような操作を何段も繰り返すなら、for式を使うこともできます。**リスト8**のsqOddEvenは、リストから奇数と偶数をとり出してそれぞれ二乗して返します。firstOdd()とfirstEven()はともにOption[Int]型を返し、それぞれSome(_)だった場合の値が<-の左辺の変数に束縛されます。<-の右辺がNoneになった場合はそれ以降の行は計算されず、for式全体がNoneを返します。

このようにして、値を返さないことがある処理をOption[]型で表現し、そのような処理を組み合わせた複雑な結果もまたOption[]型で表せます。

▼リスト8　forによるOption[]型の操作の例

```
def firstOdd(list: List[Int]): Option[Int] =
  list match {
    最初に見つかった奇数を返す実装(略)
  }

def sqOddEven(list: List[Int]): Option[(Int, Int)] =
  for {
    o <- firstOdd(list)
    e <- firstEven(list)
  } yield( (o * o, e * e) )

sqOddEven(List(1, 2, 3))
// => Some((1,4))

sqOddEven(List(1, 3, 5))
// => None
```

Either型

同様にして、エラーになることがある処理も例外を使わずに表現できます。Either[]型を使うと、Left(_)かRight(_)のどちらかの値をとる型を表現できます。Left(_)でエラーのときの情報を、Right(_)で結果の値を表すのが慣例です。Either[]型のインスタンスはLeft()やRight()メソッドでも作れますが、ここではOption[]型から変換してみましょう(**リスト9**)。

toRight()メソッドを呼ぶと、Some(_)だったときには値がRight()に入り、Noneだったときには引数で指定された値がLeft()に入ったEither[]型が返ります。Either[]型の値のright()メソッドを指定すると、<-でRight(_)の値をとり出せるようになります。途中でエラー(Left(_))になった場合はfor式全体の結果がLeft(_)になります。この例の場合、最終的にeをとり出すところで失敗してエラーが返ったことがわかります。

常に不変な値を返すスタイルでも、複雑に部品を組み合わせることが可能となるだけでなく、エラー時の処理までできることがわかりました。入門ですのでこのあたりにとどめておきますが、この先は状態の管理が本質的に必要とされる場合の方法論なども学んでいくと良いでしょう。

そのほかの言語機能

ここまでの例でScalaは、静的型付けの割にはすべての変数宣言に型を書くようなことはせ

▼リスト9　forによるEither[]型の操作の例

```
val list = List(1, 3, 5)
for {
  o <- firstOdd(list).toRight {
    "Odd number not found"
  }.right
  e <- firstEven(list).toRight {
    "Even number not found"
  }.right
} yield( (o * o, e * e) )
// => Left(Even number not found)
```

ず、非常に柔軟な書き方ができる言語だということが見てとれたかと思います。ほかに、自由度を上げるのに役立つScalaの機能をいくつかピックアップして紹介したいと思います。

未実装部分の型付け

静的型付言語に対してよく挙がる不満として、書きかけの部分を残したままではコンパイルが通るかどうかを確かめられないというものがあります。Scalaの場合は、書きかけの部分にはひとまず`???`と書いておくことで、それ以外の部分の型チェックを走らせることができます。

```
def balanced[A](elements: List[A]): Tree[A] =
  ???
```

`???`はどんな型が要求される場所でも型が付く式で、この式を実行すると`NotImplementedError`という例外が発生します。

ブロック

メソッドの引数型を`=> A`や`A => B`にすると、`{ ... }`や`{ x => ... }`の形でRubyのブロックに似た引数を受けとることができます。正確には前者は名前呼びで、後者は関数を引数に渡すことで実現されているものです。

```
def when[A](cond: Boolean)(block: => A) =
  if (cond) Some(block) else None
when(true) { println("foo") }  // foo
when(false) { println("foo") } // 印字なし
```

受けとったメソッド内で実際に使うまではブロック内の計算は実行されないので、独自の制御構文のような使い方もできます。まずは使う方からなれていき、だんだんと自分でも定義できるようになっていきましょう。

文字列補完

例の一部にも出てきましたが`s"..."`のように文字列リテラルの前にプレフィックスを付けることで、リテラル中に`$name`の形で変数名を指定して、その値を埋め込めます。

```
val name = "foo"
val value = 3
s"$name is $value" // => foo is 3
```

詳しくは説明しませんが、この`s`に相当するものは自分で定義することもでき、変数埋め込み可能なDSLを自由に定義できるようになっています。

暗黙変換

`implicit`という言語機能を用いると、Rubyの`refine`のように、ほかで定義されているクラスに対して動作するメソッドを後から付け足して、特定のスコープだけで有効にできます。たとえば次の例では、組込みの`String`型に`quote`メソッドを付け足すための`MyString`クラスを定義しています。

```
class MyString(val s: String) {
  def quote() = "\"" + s + "\""
}
implicit def myString(s: String) =
  new MyString(s)

"foo".quote // => "foo"
```

`implicit`にはそのほかにもさまざまな応用例があり、Scalaの自由度を上げるのに一番貢献している言語機能とも言えます。

おわりに

関数型プログラミングの特徴である、不変な値を次々と計算していくスタイルをScalaで実現する方法について紹介しました。Scalaにおける関数型プログラミングのさらなる高みを目指すなら、Scalazやshapelessといったライブラリを眺めてみると良いでしょう。

関数型プログラミングに限らないScalaの言語機能にも、まだまだ紹介しきれなかったものがたくさんあります。とくに、構造的部分型、動的束縛、動的メソッド呼び出し、マクロなど、これらすべてをサポートしている言語はなかなか珍しいので、興味があればScalaのマニアックな機能を試してみると良いでしょう。SD

第4章

数学と物理遊びで垣間見る 定義で記述するHaskellのわかりやすさ

純粋関数型プログラミング言語とも言われ、数学的な専門用語を伴って説明されることが多いHaskell。敷居の高いイメージばかりが先行してしまい、ふれることすらためらっていませんか？　まずはシンプルな計算プログラムで、Haskellの書きやすさを知ることから始めましょう。

Author 上田 隆一（うえだ りゅういち）　産業技術大学院大学／USP研究所／USP友の会　Twitter @ryuichiueda

はじめに（ちょっとテンション低め）

この話をいただいたときに「えっ？　またHaskell？」と思ってしまった、本来はシェル芸の人、上田です。冒頭であまりネガティブなことを書くのは、依頼を出した方にも責任を背負わすことになり本来よろしくありません。しかし、関数型言語界隈は論客が多いこと、筆者がHaskellでご飯を食べていないどころかコミュニティにも顔を出していないこともあり、引き受けるにはそれなりに覚悟がいります。しかし、紛れもなく日常的には使っていますので、ちょっと引いた目でHaskellについて考えていることを書かせていただきます。

難しさの切り分け

本特集は「なぜ関数型プログラミングは難しいのか？」ですが、たぶん一言で「難しい」と言ったときには、複数の難しさを含んでいます。「何が難しいか」を列挙して問題を切り分けないと、ただ漠然とした不安にしかなりません。

ということで、Haskellについて筆者なりに「個々の難しさ」を挙げたら次のようになりました。

①数学的な背景の難しさ
②試してみようと思った人が数学や物理をあまり勉強しなかったゆえに感じる難しさ
③使い道を見出す難しさ

①は、やれ圏論だとか、やれ関手だとか、そういう話です。とくにHaskellの場合、ネット上では常に数学の概念の話が盛り上がっている「ように見えてしまう」のですが、初心者の人がHaskellの情報をネットで調べると、そういった盛り上がりに気をとられてしまいます。ブログなどを書いているほうは楽しく自分の関心を書けばいいのでまったく罪はありませんが、調べるほうは、まずそのようなバイアスにさらされていることは留意する必要があります。筆者も研究者ではありますが、そこらへんの話は正直に申し上げるとサッパリわかりません。いくら数学ができても、当該分野の数学をちゃんと勉強しなければ普通の人とそんなに違いません。

おそらく、筆者が考えるにもっと厄介な問題は②のほうです。Haskellを書くのに必要なのは圏論の知識ではなくて、数列とか関数とか、中学や高校で習ったり使ったりした数学や物理の知識かなあと思っています。類書を読んでも、ある程度の数学の知識が前提になっています。中にはもしかしたら「わからなくてもいいよー」的なノリの本もあるかもしれません。しかし、この特集が「なぜ難しいか」というテーマなので便乗して断言すると、「わからなくてもいいよー」はウソです。そんなムシのいい話はありません。数学における関数というものが一体何なのか、体に染み付いて理解していないとスラスラ書けませんし、それ以前にメリットを感じられることもないでしょう。

光明があるとすれば、逆にHaskellを勉強し出したら数学も勉強できて、数学も好きになる

かもしれないということでしょうか。だいたい、数学が嫌いになった人に聞きたいのですが、嫌いになった原因というのは、ほかの人についていけなくなったとか、先生がつまらなかったとか、外因だったのではないでしょうか？　数学とか物理とかは本来、「世の中の雑事を紙と鉛筆(とコンピュータ)でシンプルに扱うにはどうするか？」ということを考えるためのもので、物事がこんがらがってよくわからんというストレス地獄から解放されるためのものです。のんびり楽しく勉強していけば、万物がシンプルに見えるようになります。そしてHaskellも数学の恩恵を受けており、シンプルに書けます。

ですので本稿では、Haskellでちょっと数学遊びと物理遊びをしてみようかと思います。

コードを試す環境

Haskellのコードのコンパイルには、GHC(略さない名称は下の出力に)を使います。

```
$ ghc --version ↵
The Glorious Glasgow Haskell Compilation ➡
System, version 7.8.3
```

このバージョンのGHCならどの環境でも同じように動作するはずですが、念のために執筆で使った環境を書いておくと、計算機とOSはMacBook Pro と OS X Yosemite で、GHC は brew(1)を使って次のようにインストールしました[注1]。

```
$ brew install ghc cabal-install ↵
```

また、本稿執筆にあたり『関数プログラミング実践入門——簡潔で、正しいコードを書くために』(技術評論社刊)を読みました。筆者の薄い解説でぼんやりHaskellを理解した後は、ぜひ硬派な本書をお読みください。以上、宣伝でした。

数式を並べるようにプログラミング

さて本題です。筆者の場合、電卓でもシェル芸[注2]でも済まないような計算をしなければならないとき、まずHaskellを使おうかという気になります。なぜかと言うと、考えて紙にメモした数式を素直にコーディングしてすぐに試せる場合が多いからです。ということで、Haskellで数式を表現して遊んでみます。

まず、**リスト1**のようなコードから話を進めたいと思います。まだコンパイルは通りません。

setAとsetBは、整数が5個並んだリストです。リストは、他の多くの言語で言うところの配列だと思っても本稿では差し支えありません。数学に興味がない場合は、「ああ配列が2つあるなあ」で構いません。一応書いておくと、数学というのは同じ性質を持つものをグループ化して、別のグループのものに作用させたらどうなるかということを考えていく学問なので、数式を書き始めると、たいていこのように、何かグループ(集合)を定義することから始まります。

んな難しい話は置いておきましょう。setAとsetBからリスト順に1つずつ整数を取り出して足し合わせ、新たにsetCというリストを作ってみます。**リスト2**のように2行追加します。

次ページのようにex1.hsをコンパイルして実行すると、setCが画面に表示されます。

▼リスト1　AとBをリストで実装(ex0.hs)

```
setA = [1,2,3,4,5]
setB = [6,7,8,9,10]
```

▼リスト2　setAとsetBの各要素を足して作ったsetCを表示するコード(ex1.hs)

```
1: setA = [1,2,3,4,5]
2: setB = [6,7,8,9,10]
3: setC = zipWith (+) setA setB
4: main = print setC
```

注1) インストールにはXcodeがインストールされている必要があります。
注2) 当たり前のように使っていますが、シェル上でのワンライナーのことを筆者がこう呼んでいるだけです。詳しくは『シェルプログラミング実用テクニック』(技術評論社刊)にて。以上、宣伝でした。

```
コンパイルはこのように行う
$ ghc ex1.hs ↵
出力省略。バグがあればエラーが出る
できた実行形式(ex1)を実行
$ ./ex1 ↵
[7,9,11,13,15]
```

リスト2の3行目だけをもう少し補足しておくと、zipWithというのは最初から準備されている関数で、3個引数をとります。引数の後ろの2つはzipWithで処理したいリスト(つまりsetA、B)で、最初の(+)は、setAとsetBの要素のペア1つずつに行いたい演算(この場合は足し算)です。

コードを翻訳したらわかること

さて、ここでちょっと変なことをしてみます。**リスト3**のように、**リスト2**の各行を「人間語(ここでは日本語)」に翻訳してみます。

リスト3で面白いのは、どの行も「～は……です。」というように、何か言葉の定義をしているように見えることです(最後の4行目だけは「～は……します。」とも言えますが)。登場する記号が何であるか、逐一しっかりと定義することは、論文や数学・物理のテストの答案を書くときには大切なことですが、Haskellのコードもこのように定義だけで書いていくことができます。

答案と違ってHaskellのコードはプログラムとして動きます。やりたいことであるmain関数の処理(4行目)からトップダウンで出発し、必要な定義を持ちだしては計算を進めていきます。ex1.hsでは、mainを完了するために必要なsetCが確定していないか調べられ、setCを確定するために必要なsetA、Bと関数が調べられ……というふうに動いていきます。

ところで、定義は順番を変えても変わらないので、Haskellのコードは**リスト4**のように順番をめちゃくちゃにしても動きます。一応、動くことを確認しておきましょう。

```
$ ghc ex1-2.hs ↵
...(略)...
$ ./ex1-2 ↵
[7,9,11,13,15]
```

さて、**リスト3**の「日本語訳」は、**リスト5**のように翻訳してもよいのではないでしょうか?**リスト3**の翻訳は恣意的(しいてき)ではないでしょうか?

しかし、**リスト5**のように各行を手続きのように解釈してしまうと、**リスト4**のような並び替えを行ったときに変な感じになります。やはりHaskellのコードは定義であって、手続きではないのです。ほかの言語でも関数を書いた順番が動作に無関係な場合はありますが、Haskellの場合、手続きが書けないのでこのような性質が目立ってきます。

もう1つ、**リスト3**の翻訳の例で言っている大事なことは、Haskellのコードでは、

```
n = 1
n = 2
```

というコードがありえないということです。翻訳すると、

```
nは1です。
nは2です。
```

となってしまい、「nは結局1と2どっちなんや」ということになってしまいます。ありえません。

▼リスト3　ex1.hsの「日本語訳」

```
1: setAはリスト[1,2,3,4,5]です。
2: setBはリスト[6,7,8,9,10]です。
3: setCはsetAとsetBの各要素1つ1つ順に足して作ったリストです。
4: mainはsetCをプリントする関数です。
```

▼リスト4　ex1.hsの順序を入れ替えたコード(ex1-2.hs)

```
1: setA = [1,2,3,4,5]
2: setC = zipWith (+) setA setB
3: main = print setC
4: setB = [6,7,8,9,10]
```

▼リスト5　ex1.hsの「誤訳」

```
1: setAにリスト[1,2,3,4,5]を代入する。
2: setBにリスト[6,7,8,9,10]を代入する。
3: setCにsetAとsetBの各要素1つ1つ順に足して作ったリストを代入する。
4: mainでsetCをプリントする。
```

無限に続く数列を扱う

さて、Haskellは「無限に続くリスト」を扱うことができます。**リスト6**のex2.hsはex1.hsをちょっと書きなおしたものです。出力は、次のようにex1と同じです。

```
$ ./ex2 ↵
[7,9,11,13,15]
```

リスト6中の[1..]は[1,2,3,...]と無限に続くリストのことです。こんなものを実際に作ったらメモリがパンクするはずですが、**リスト6**の1行目は、「setAは[1..]である」とただ言っているに過ぎないので、パンクはしません。2行目の[6..]はお察しのとおり[6,7,8,...]と無限に続くリストですが、これもメモリに展開されるわけではありません。setCも[1..]と[6..]の要素を順番に足した無限のリストになります。

一方、ex2で出力される整数は5個だけですが、これは4行目のtake 5 setCというのが「setCの先頭から5個の要素」という意味になるからです。プログラムは、mainから動き始め、mainを遂行するにはsetCの先頭5個だけ必要で、またそのためにはsetAとsetBの先頭5個だけが必要で……と動きます。ですので、setAとsetBが無限のリストでも大丈夫です。

これは結局、setAとsetBで使う要素数を、setAやsetBを定義するときではなく、実際に使うときに決めることができるという例です。この性質も地味と言えば地味なのですが、少なくとも筆者は数式を書くときに、たとえば「離散時刻t = 1, 2, 3, ...があって……」というように無限に続く数列を定義することはまったく珍しくないので、それをそのまま書けて嬉しいということになります。

さらに、『関数プログラミング実践入門』にも書いてありますが、ex2.hsの1、2行目を、

```
setA = 1 : map (+1) setA
setB = 6 : map (+1) setB
```

とひねくれた書き方をしても同じ出力が得られます。ただ、これは変態向けのお題ということで、初心者の方は華麗にスルーしていただければと。

次も初心者はスルーで結構ですが、もうちょっと数学をこじらすと、**リスト7**のようなコードも書けます。複利の計算です。xが元本、yが利子(ここでは年利のつもり)です。

なぜそうなるかを理解する必要はありませんが、setF x yは[x,x*(1.0+y),x(1.0+y)^2,x(1.0+y)^3,...]という、無限に数式の入ったリストになります。2行目で元本千円、年利1%と指定し、最初の5要素(0年目、1年目、...、4年目)を出力するという指示をしているので、出力は、

```
$ ./ex3 ↵
[1000.0,1010.0,1020.1,1030.301,1040.60401]
```

となります。

▼リスト6　無限に続くリストを使う(ex2.hs)

```
1: setA = [1..]
2: setB = [6..]
3: setC = zipWith (+) setA setB
4: main = print ( take 5 setC )
```

▼リスト7　数式のリストを作る(ex3.hs)

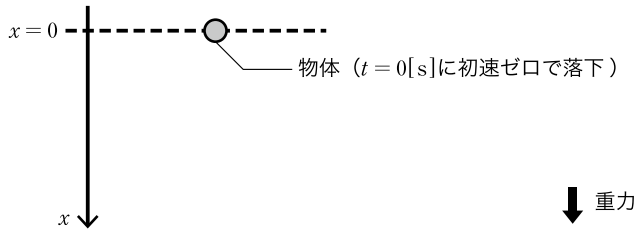
```
setF x y = x : map (* (1.0+y)) (setF x y)
main = print $ take 5 (setF 1000 0.01)
```

物理の計算

今度は物理をやってみます。と言ってもそんなに難しい話ではなく、物を落としたら何秒後にどれだけ下に落ちているか、数値計算してみるだけです。

図1に想定する実験環境を示します。求めたいのはt秒後の位置xです。この問題を紙と鉛筆で解くと、$x = \frac{1}{2}gt^2$と答えが出てしまいますが、もっと難しい問題の場合は、このように素直に答えが出てこないことがあります。こう

▼図1　実験環境



▼図2　数式表現

$$x(t)=\begin{cases}x(t-\Delta t)+v(t)\,\Delta t & (t>0)\\ 0 & (\text{otherwise}),\end{cases}$$
$$\text{where } g=9.8, \Delta t=0.000001, \text{ and } v(t)=gt.$$

いう場合は、少しずつ時間を進めてコマ送りのように物体を動かして計算する方法がとられます。これからやるのは、こういう方法のうちで一番単純な「オイラー法」というものです。

で、Haskellでどう解いていくかというところですが、先ほどの集合の演算と同様、単に定義を書いていけばよいということになります。**リスト8**は、実際に動くコードです。1行目が重力加速度gの定義、2行目がコマ送りの時間間隔です。この例では1マイクロ秒とかなり細かく動かしています。3行目は、「速度＝重力加速度×時間」という事実を表しています。数式で書くと$v(t) = gt$といったところですが、Haskellだとシェルのコマンドと同様、関数(この場合はv)の引数(この場合はt)に括弧をつける必要がないので、`v t = g * t`という書き方になります。

次の5〜7行目は、速度$v(t)$を使って$x(t)$を定義した関数です。この式には場合分けがあって、6、7行目は`| <条件> = <条件に対応する式>`という書き方になっています。6行目はtがゼロより進んだときの式で、右辺は"dt秒前の位置から速度$v(t)$でdt秒だけ位置を動かす"という式になっており、7行目は"ゼロ秒以前の物体の位置はx=0"という意味になっています。

ちなみに、たとえば7行目までの式を論文用に真面目に数式で書くと、**図2**のような書き方になります。

whereはHaskellにもあるので、実は**リスト8**の式は**リスト9**のようにも書けます。条件を先に書かなければならなかったり、括弧が数式と違ったり、Δが書けなかったりしますが、**図2**と**リスト9**の類似性は筆者にとってはただただ単純に嬉しいものです。Haskellを初めていじったときの感動が今、よみがえりました(大げさ)。

しかし逆に言えば、数式を書かない人にはピンと来ないわけで、「なんでこんな書き方するんだろう？」ということになります。ただ、そういう人もHaskellを勉強してから数学の教科書を開けば感動できますので、とくに数学をやってこなかったからといって、自分で自分を門前

▼リスト8　図1の問題を数値計算で解くコード(ex4.hs)

```
1: g = 9.8
2: dt = 0.000001
3: v t = g * t
4:
5: x t
6:  | t > 0.0   = x (t - dt) + (v t) * dt
7:  | otherwise = 0.0
8:
9: main = print (x 5.0)
```

▼リスト9　リスト8の別の書き方(ex4-2.hs)

```
1: x t
2:  | t > 0.0   = x (t - dt) + (v t) * dt
3:  | otherwise = 0.0
4:      where g = 9.8 ; dt = 0.000001 ; v t
 = g * t
5:
6: main = print (x 5.0)
```

払いする必要はありません。

さて本題に戻ります。これまでのコードの説明で**図1**の世界は説明し尽くしました。あとはやりたいことを書くだけで、それが**リスト8**の9行目や**リスト9**の6行目の「5秒後に物体がどこにあるか出力する」という`main`関数です。これを指定しておくと、あとはHaskellの処理系が1〜7行目の定義を駆使して答えを求めてくれます。

動かしてみましょう。ちゃんと$x=\frac{1}{2}gt^2$から得られる解(122.5)に近い値が得られています。`dt`の値を大きくしていくと少しずつ122.5から誤差が大きくなっていくので、試してみていただければと。

```
$ ./ex4 ↵
122.50002448674475
```

終わりに

さて、6ページでHaskellについてお伝えしてきました。筆者からは、「Haskellの背景にある数学は必ずしも理解しなくてもよいけど、Haskellで書いて楽だと思えるようになるには、ある程度数学も一緒に勉強しないといけない」とお伝えしておきます。

また、面白いことに本稿では型どころかモジュールのimportの話すら一切出てこず、ほぼミニマムなコードで済みましたが、Haskellが持っているミニマムな機能というのが数字の計算であることを暗に示しているのかもしれません。文法の説明を求めて読まれた方には懺悔(ざんげ)しておきます。

ところで、冒頭に挙げた問題③について何も述べていませんでした。とりあえず本稿のように数学や物理、金融や統計の問題を解くのがまず1つです(ただし連立方程式を書いたら勝手に解いてくれるわけではありませんが)。また、Yaccのような補助ツールなしでパーサが簡潔に書けてしまえるというのもHaskellの超強力な機能です。簡潔になるのは「数字というのは○○である。」「配列というのは○○である。」というような文法の記述方法が、定義を並べて記述するHaskellのような言語と相性が良いからです。

Haskellの用途についてはこんなところにしておきます。Haskellの学習のためには、本稿のように計算用途から入っていくのはよい入り口かと考えています。

最後に、「なぜ関数型プログラミングは難しいのか？」を思惟中、余計なことを考えてしまったのでここに書くことをお許しください。考えたことというのは、もし言語を2つ以上覚えようとすれば、プログラミング対象になる雑多な物事の知識もバランス良く知っていないと使い分けができず無意味、ということです。物事の知識というのは、このWebサービスは裏でこんな業務フローを持っていそうとか、この物理の問題はこう解くとか、化学反応はこう進むとか、そういうことです。何か作るときは言語以前に製作物の構造を把握することが大切です。

一方、残念なことに、やたら特定の言語や「関数型」などの概念に固執した人たちがネット上での不毛な喧嘩を起こしています。起点となる人たちはたいてい、何を作るかという話抜きに、ライブラリなどの貢献者への敬意抜きに、実世界で抜いたことのない刀の自慢をしています。

我々がプログラムする対象は常に実世界、もっと言えば日常にあると、筆者は考えます。日常のロジックに興味を向けることもおそらく、関数型に限らずプログラミングや言語というものを難しく考え過ぎず、やたら神話化、神格化しない訓練になるのかなと、考えています。SD

第5章

Erlang/OTPから生まれたWeb開発指向言語

Elixir入門 その現在と未来

Elixir[注1]は、Erlang/OTPとその仮想マシンBEAMの上で動く、読みやすさを考慮してデータや処理の流れに注力できることを目指しているプログラミング言語です。この記事ではErlang/OTPと比較しながらElixirの現在の状況を紹介し、今後について考察します。

Author 力武 健次(りきたけ けんじ)　力武健次技術士事務所 所長　URL http://rikitake.jp/

Elixirの歴史

Elixirは、ブラジルのWebコンサルティング企業Platformatecのリード開発者かつ共同創設者のJose Valim氏が2012年に開発を始めたプログラミング言語です。彼は2010年にRuby on Railsのコアチームに加わり、Railsの書籍[1]も出版しています。そのせいかElixirのコミュニティでアクティブに活動している人達には、Railsの経験者が多いように筆者には感じられます。

Valim氏がElixirを作ったのは、2010年に並行処理の問題を解決するのに苦労して、ほかのプログラミング・パラダイムを探していたときに、Erlangを調べ始めたことにさかのぼります[2]。彼はBEAMのしくみを追求しつつ、同時に学んでいた他のパラダイムなども参考にして、2012年1月にPlatformatec社内でBEAM上で動く言語としてElixirの最初の原案を説明するに至りました。さっそく会社の同意を得て、Elixirの開発に社内で100%の時間を割くようになりました。

Elixir開発当初は言語がどうなっていくかについては不安な側面もありましたが、その後各出版社から参考書(後述)が出るに至り、Platformatecの人達もElixirへの(創設者の時間と技術力を注ぎ込んだ)投資が正しかったという確信を得たようです。Elixirはその後2014年9月18日にv1.0をリリースし[3]、2015年6月の本稿執筆時点ではv1.0.5が最新の安定版(Stable)となっています。

Elixirのプログラミング言語としての特徴

Elixirの文法はErlangのそれとは大きく異なり、一見すると他のスクリプト言語と同じように見えます。**リスト1**にサンプルコードを示します。このサンプルコードでは`Rgb2hsv`というElixirのモジュールに`Rgb2hsv.tohsv/1`という関数を定義して、RGBとHSVの色コードの変換を行うものです[注2]。

Elixirの利点

筆者はErlangに比べ、Elixirは次の点でプログラミングの自由度をより高めた言語だと感じています。

省略記法

各種の省略記法を使えるようにすることで、文法の厳格さよりも書きやすさを重視しています。たとえば関数を複数作用させる際のパイプ演算子`|>`は、Elixirならではのものといえるでしょう。

再代入可能

Erlangの並行プログラミングを目的とした、パターンマッチング、軽量プロセスへの分割と

注1) Elixirは英語では「エリクサー」と読みます。他の欧州言語では「エリクシア」あるいは「エリクシル」に近い発音になります。もともと飲み薬、その中でも「秘薬」を指す言葉ですので、混同を避けるためにハッシュタグなどでは「#elixirlang」と書かれることが一般的です。

注2) このサンプルコードは、筆者の本誌2015年5月号での連載記事「Erlangで学ぶ並行プログラミング」第2回に示したものを書き換えています。興味のある方は比較してみてください。

▼リスト1　rgb2hsv.ex

```
RGBからHSVへの色コードの変換を行います
参考URL: https://en.wikipedia.org/wiki/HSL_and_HSV
Elixirでは複数のモジュールを1つのファイルで定義できます
モジュール名をファイル名と一致させる必要もありません
structは名前のついたmapの一種で，モジュールごとに定義します
struct HSVを定義します
defmodule HSV do
各メンバーの名前を定義します
  defstruct hue: 0.0, saturation: 0.0, value: 0.0
end
struct RGBを定義します
defmodule RGB do
  defstruct red: 0, green: 0, blue: 0
end
Rgb2hsvモジュールを定義します
defmodule Rgb2hsv do
ビット演算のためのBitwiseモジュールをインポートします
  import Bitwise
キーワードdefpはモジュール内だけで使える関数を定義します
do:のあとの式が関数の定義となります
ElixirにはErlangと違い定数値のマクロがないため関数としています
  defp maxvalue, do: 255.0
Erlang同様のパターンマッチングが使えます
  defp zerodiv(_, 0), do: 0.0
  defp zerodiv(x, y), do: x / y
  defp diffdiv(x1, x2, y), do: zerodiv((x1 - x2), y)
キーワードdefはモジュール外から見える関数を定義します
%「struct名」{}でstructそのもの
struct を代入した 変数.「メンバー名」で各メンバーの値が使えます
複数行の関数はdo - endのブロックで定義します
  def tohsv(rgb = %RGB{}) do
|>はパイプ演算子といい
複数の関数を作用させるときの表記を容易にします
v1 |> func(v2, ...) は func(v1, v2...)と同値です
同様に v1 |> f1(v11, v12) |> f2(v21, v22, v23)は
f2(f1(v1, v11, v12), v21, v22, v23)と同値です
    tohsv((rgb.red |> band(0xff) |> bsl(16))
          |> bor(rgb.green |> band(0xff) |>
bsl(8))
          |> bor(rgb.blue |> band(0xff)))
  end
Erlang同様に引数の型によるパターンマッチも使えます
  def tohsv(c) when is_integer(c) do
Erlangのビットストリングと同様です
    << r::unsigned-integer-size(8),
       g::unsigned-integer-size(8),
       b::unsigned-integer-size(8) >> =
      << c::unsigned-integer-size(24) >>
次の行のコメントを外すと上記の式の内容を出力できます
IO.inspect ["r:", r, "g:", g, "b:", b, "c:", c]
    maxval = r |> max(g)|> max(b)
    minval = r |> min(g)|> min(b)
    d = maxval - minval
case式もErlangと同様ですが
ElixirではErlangと違い再代入を既定動作として許すため
以下の場合は比較対象の変数の前に
「^」を付けて再代入を許さないことが必要です
    h = 60.0 *
        case maxval do
          ^r -> diffdiv(g, b, d) + 6.0
          ^g -> diffdiv(b, r, d) + 2.0
          ^b -> diffdiv(r, g, d) + 4.0
        end
if式もErlangと同様ですがガードの制限がありません
    hueval = if h >= 360.0, do: h - 360.0, else: h
ここではstruct HSVで返しています
    %HSV{hue: hueval,
         saturation: zerodiv(d, maxval),
         value: maxval / maxvalue()}
  end
end
```

それぞれの状態の分離、データの共有を極力最小限にする各種のしくみ、メッセージパッシング、関数による非破壊操作などの特徴はすべて維持しています。ただし、変数はErlangと違い、既定動作としては再代入可能になっています。これによって起き得る問題を防ぐために、変数参照の際再代入を禁止するためのピン演算子^が用意されています[4]。サンプルコードではcase式でピン演算子を使っています。

Erlangを越える新機能

Erlangでは実現が容易でなかった各種機能を言語ライブラリとして備えています。例として遅延評価のためのStream、キーバリュー型データ構造のmaps(Erlang/OTP 17以降にも同様の機能があります)、mapsを応用した名前の付けられるデータ構造であるStructなどがあります。

メタプログラミング

ErlangにはないLispスタイルのマクロによるメタプログラミングを使えるようにすることで、言語の拡張が容易です。

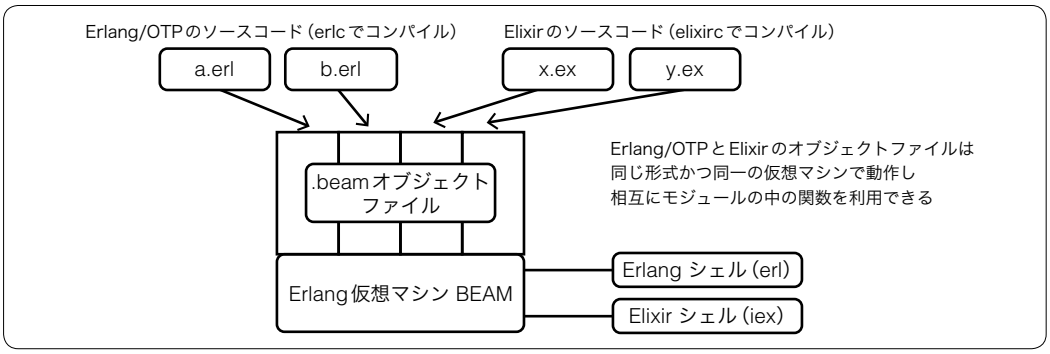
Elixirの実行方法

Elixirで書かれたプログラムはコンパイル用のelixircコマンドでコンパイルするか、シェルであるiexコマンドのスクリプトとして実行することができます。サンプルコードの実行例を**図1**

▼図1　Elixirの実行

```
Elixirのシェルでサンプルソースをコンパイルしよう
ソースコードのあるディレクトリからiexコマンドで起動する
Erlang/OTP 17 [erts-6.4] [source] [64-bit] [smp:8:8] [async-threads:10] [hipe] [kernel-poll:false] [dtrace]

Interactive Elixir (1.0.4) - press Ctrl+C to exit (type h() ENTER for help)
ソースコード(拡張子は.ex)をコンパイルする
iex(1)> c("rgb2hsv.ex")
これらの警告はすでにある.beamファイルが上書きされるという以上の意味はない
rgb2hsv.ex:6: warning: redefining module HSV
rgb2hsv.ex:11: warning: redefining module RGB
rgb2hsv.ex:15: warning: redefining module Rgb2hsv
[Rgb2hsv, RGB, HSV] ←これら3つのモジュールが定義された
さっそくRgb2hsv内の関数を実行してみる
iex(2)> Rgb2hsv.tohsv(0x775534)
%HSV{hue: 29.55223880597015, saturation: 0.5630252100840336, value: 0.4666666666666667}
struct RGBを渡しても問題なく実行できる
各メンバーはキーワードで指定するので順番は自由
iex(3)> Rgb2hsv.tohsv(%RGB{red: 100, blue: 100, green: 200})
%HSV{hue: 120.0, saturation: 0.5, value: 0.7843137254901961}
RGBの16進値で再度計算
iex(4)> Rgb2hsv.tohsv(0x64C864)
%HSV{hue: 120.0, saturation: 0.5, value: 0.7843137254901961}
iex(5)> ここでCtrl-Cを押すとErlangシェルからブレークする
BREAK: (a)bort (c)ontinue (p)roc info (i)nfo (l)oaded
       (v)ersion (k)ill (D)b-tables (d)istribution
ここで a を押すとabortして停止させる
```

▼図2　Elixir実行環境の構造



に示します。Erlangと同様にシェルから関数を評価することで各種の作業を行うことができます。

Elixirのソースコードからコンパイルされた結果はErlang/OTPと互換の.beamファイルとなるため、Erlang/OTPとElixirを組み合わせてプログラムを書いたりライブラリを相互に使うこともできます(**図2**)[注3]。

Elixirのインストール

Elixirを動かすにはErlang/OTP 17以降のインストールが必要です[注4]。Elixir自身のインストールについてはElixirのWebページ[注5]に一通り説明があります。

注3) Elixirでは関数は「モジュール名.モジュール内関数名/引数の数」と表記します。サンプルコードにあるRgb2hsv.tohsv/1は実際にはElixir.Rgb2hsv.beamというオブジェクトファイルでコンパイルされ、Erlangとの名前の衝突を防ぐようになっています。

注4) Erlang/OTPのインストールの詳細については、本誌2015年4月号の連載記事「Erlangで学ぶ並行プログラミング」第1回を参照してください。

注5) http://elixir-lang.org/install.html

最新版のリリースキットはGitHub内のレポジトリ注6から得ることができます。ここからPrecompiled.zipというErlang/OTPでコンパイル済みのアーカイブをダウンロードして展開すれば、既存のErlang/OTPの環境上でElixirを試すことができます。ソースコードを展開し、makeでビルドすることもできます。

もっとも、主要なOSでは定番のElixirインストールの方法がすでに確立されているので、そちらを使うのが賢明でしょう。以下、最新安定版について、インストール方法を各種OS別に説明します。

FreeBSD

Portsあるいはパッケージのlang/elixirをインストールするのが早道です。具体的な方法は次のとおりです。この際Erlang/OTP(lang/erlang)は必要であれば依存関係に基づく必須パッケージとしてインストールされます注7。

・portmasterを使ったPortsからのビルドとインストール

```
# cd /usr/ports && umask 0022 && portmaster lang/elixir
```

・pkgコマンドによるパッケージインストール

```
# pkg install lang/elixir
```

Mac OS X

HomeBrew(http://brew.sh/)のElixirとErlang/OTPの両バイナリパッケージ(bottle)を使うのが早道でしょう。Mac OS X 10.10.x(Yosemite)であれば、

```
# brew install elixir
```

だけでインストールができます。

Linux各種ディストリビューション

UbuntuやDebianの場合、Erlang Solutionsの配布するErlang/OTPのバイナリパッケージとの組み合わせが推奨されています注8。具体的な方法は次のとおりです。

```
パッケージを加える
# wget http://packages.erlang-solutions.com/erlang-solutions_1.0_all.deb
# sudo dpkg -i erlang-solutions_1.0_all.deb
パッケージライブラリを更新してインストール
# sudo apt-get update
# sudo apt-get install elixir
```

そのほかのディストリビューションについては、前述のelixir-lang.orgにあるインストール説明ページに紹介されています。

Windows

Windowsには専用のインストーラが提供されており、ダウンロードして実行するだけでErlang/OTPのインストールも含めて作業を完了することができます注9。この際、WindowsのErlang/OTPのバイナリには32ビット／64ビットの2つのバージョンがあるため、インストール中に選択が必要です。また、Elixirの各種実行プログラムに対して、環境変数PATHを設定し実行できるようにする必要があります注10。

Elixirの開発環境

Elixirのプログラムをエディタで編集するにはvimであればvim-elixir[5]、Emacsであればemacs-elixir[6]というプラグインが有用です。

注6) https://github.com/elixir-lang/elixir/releases/

注7) Elixirは実行環境としてUTF-8のロケールを前提としています。そのためFreeBSDのlocale設定によっては"Please ensure your locale is set to UTF-8"というエラーメッセージが出ることがあります。この場合は起動時の環境変数LANGをen_US.UTF-8に設定するなどの方法で、UTF-8を使うロケールの選択を行ってください。

注8) Linuxの各種ディストリビューションにあるErlang/OTPのパッケージでは、一通り必要なライブラリやモジュールを揃えるのが容易ではないという歴史的事情があります。

注9) http://s3.hex.pm/elixir-websetup.exe

注10) 具体的にはPATHに"C:¥Program Files¥erl6.4¥bin;C:¥Program Files(x86)¥Elixir¥bin"などの内容を追加する必要があります。Windows各種エディションの環境変数PATHの編集については(https://github.com/uzulla/how_to_setup_path_on_windows)に一通り日本語でまとめられています。

これらプラグインでは予約語などのハイライトが可能です。emacs-elixirのほうはEmacsバッファ内でのiexの実行やコードのコンパイルの機能も備えています。

Elixirはビルド環境としてMixというツールを標準で備えており、開発プロジェクトごとのディレクトリ管理などを行うことができます。また、Hex(http://hex.pm/)というパッケージマネージャもMixから使うことができます。

HexはElixirのみならずErlangまでサポートしており、Erlang/OTPの環境からも使えます。Hexはユーザ登録を行うことで、各ユーザが作成したElixirやErlang/OTPの各種ライブラリを登録できるため、最近ではElixirだけでなくErlang/OTPの開発環境としても注目されています。

Elixirにはそのほかにもユニットテストのフレームワークの ExUnitや各種デバッグログの管理を行うためのLogger、また外部のデータベースとのやり取りを行うためのライブラリEcto[7]など、一通りの開発やテストのための環境はそろっていると言えるでしょう。

Elixirはどこで使われているのか

ElixirはErlang/OTP同様のオブジェクトファイルを出力する言語処理系ですから、Erlang/OTPに比べ遜色のない性能が得られることは想像に難くありません。一例として、Webアプリケーションの世界では、並行処理に強いという特徴を活かしたWebフレームワークPhoenix[注11]が注目されています。

すでにErlang/OTPは組み込みの世界にも進出しており、Elixirも同様に応用されつつあります。一例としてLego Mindstorms EV33の制御事例[8]や、Rose PointNavigation Systemsによる航法システムへの応用[9]などがあります。

Elixirの参考情報

Elixirの入門ドキュメントとして、Getting Started[注12]は通読しておくと後々役立ちます[注13]。また、公式のドキュメントはWebページ(http://elixir-lang.org/docs.html)の下に一通りそろっています[注14]。

書籍では次の2冊が役立つかと思います。

・Dave Thomas, "Programming Elixir: Functional", Pragmatic Bookshelf, 2014, ISBN-13:978-1-93778-558-1

Pragmatic Bookshelfの創設者の1人Dave ThomasがElixirに惚れ込んだ結果が1冊の書籍になったという感じの本です。Erlang/OTP同様の並行プログラミングを含めて、プログラミング例も豊富で、Elixirでできることが一とおり網羅されています。

・Simon St. Laurent, J. David Eisenberg, "Introducing Elixir", O'Reilly Media, 2014, ISBN-13:978-1-4493-6999-6

こちらは比較的薄め(210ページ)の入門書ですが、ElixirやErlangの未経験者を前提に書かれていますので、Elixirがどんなものなのかを知るには役立つでしょう。著者のうちSt. Laurent氏は、"Introducing Erlang"というErlang/OTPの入門書も書いていて、安心して読めます。

Elixirのコミュニティ

Elixirの関連情報はhttp://elixir-lang.org/に一通りまとめられています。アクティブなメーリングリストとしては、一般的な質疑応答のた

注11) http://www.phoenixframework.org
注12) http://elixir-lang.org/getting-started/introduction.html
注13) OSC北海道2015のniku氏による発表「プログラミング言語Elixir」のスライド(http://slide.rabbit-shocker.org/authors/niku/osc15do-elixir/)で日本語による概要の解説がなされています。
注14) 日本語への非公式な翻訳もあります(http://ns.maqcsa.org/elixir/docs/v1.1.0-dev/)。

めのelixir-talkとElixir言語自身の開発者のためのelixir-coreの2つが運営されています注15。

Elixirのコミュニティは独自性を保ちつつ、Erlang/OTPのコミュニティと相互に連携して活動しています。現在のElixirがErlang/OTPのライブラリや言語のしくみをフル活用していることから考えれば、これは当然の帰結といえるかもしれません。Erlang/OTPのイベントでもElixir関連の発表は増えていますし、Elixirに特化したElixirConf注16というカンファレンスも開かれています。

日本では札幌で毎週木曜日に開かれているSapporo.beam(サッポロビーム　http://sapporo-beam.github.io)が息の長いコミュニティとして知られています(オンラインでの参加もできます)。その他の日本語によるElixir関連のコミュニティ情報としては、高橋 敬祐氏による『パーフェクト"Elixir情報収集"』[10]が大変よくまとめられています。

Erlang/OTPのコミュニティでもElixirの功績は高く評価されており、Valim氏は先日6月11、12日の両日にスウェーデンのストックホルムで開かれたErlang User's Conference 2015で、毎年Erlang関連コミュニティに最も貢献した人に与えられるErlang User of the Yearの受賞者となりました[11]。

今後のElixirの方向性

本記事では割愛しましたが、Elixirでは関数の引数の型によるポリモーフィズムを実現したProtocols[12]、関数型言語のmapやfilterなど定型的な逐次処理を実装したEnumモジュールなど、Erlang/OTPにはない抽象化の試みが取り入れられています。

Valim氏自身はv1.1以降について、パイプ演算子とStreamによる非同期実行を組み合わせてパイプライン型の並列処理を目指すStream.async()や、負荷分散用のプロセス間ルーティング機能を持つGenRouterなど、より並行／並列処理を重視した方向性の開発を目指すと今年4月のErlangConf EUのキーノートにて発表しています[13]。

おわりに

この記事ではElixirの歴史、プログラミング言語としての特徴、インストールの方法、参考情報、そして今後の方向性について紹介しました。Elixirは今後もErlang/OTPと良い関係を保ちながら独自の方向性を持って発展していくであろうと個人的には確信しており、今後が楽しみです。SD

参考文献

[1] Jose Valim, "Crafting Rails 4 Applications: Expert Practices for Everyday Rails Development", Pragmatic Bookshelf, 2013, ISBN-13: 978-1-93778-555-0
[2] Hugo Barauna, "A little bit of Elixir's history", "Introducing Elixir Radar: the weekly email newsletter about Elixir", Platformatec Blog, January 28, 2015, http://blog.plataformatec.com.br/2015/01/introducing-elixir-radar-the-weekly-email-newsletter-about-elixir/
[3] http://elixir-lang.org/blog/2014/09/18/elixir-v1-0-0-released/
[4] http://stackoverflow.com/questions/27971357/what-is-the-pin-operator-for-and-are-elixir-variables-mutable
[5] https://github.com/elixir-lang/vim-elixir
[6] https://github.com/elixir-lang/emacs-elixir
[7] https://github.com/elixir-lang/ecto
[8] http://www.elixirconf.eu/elixirconf2015/torben-hoffmann
[9] http://blog.plataformatec.com.br/2015/06/elixir-in-production-interview-garth-hitches/
[10] http://www.slideshare.net/keithseahus/elixir-48878894
[11] https://twitter.com/FrancescoC/status/609401964200378368
[12] http://www.erlang-factory.com/static/upload/media/1434458082784816joseelixireuc.pdf
[13] http://www.elixirconf.eu/static/upload/media/1438466794841215423elixirelixirconf.pdf

注15) どちらもGoogle Groups上でelixir-lang-talk/elixir-lang-coreという名前のフォーラムとして運営されています。
注16) ElixirConfは2014年7月は米国テキサス州オースティンで、そして2015年は4月にポーランドのクラクフ(Krakow)にて開催されました。

第6章

バグを生みにくい、メンテンス性の良いプログラムへ

Pythonで見る関数型言語の本質

マルチパラダイム言語で肩慣らし

Pythonは、関数型の機能も持ち合わせたマルチパラダイム言語です。本章ではそのPythonを使って、関数型的な書き方とはどんなものか、その利点はどこにあるのかを解説していきます。注目されているとは言ってもなかなかハードルが高い関数型言語。慣れている言語を使って、まずは関数型の勘所を学びましょう。

Author 辻 真吾(つじ しんご) mail shingo.tsuji@gmail.com

はじめに

関数型言語について調べると、コードのメンテナンス性が高い、バグが入り込む余地が少ない、などといった特徴が挙げられているのをよく目にします。本稿では、関数型言語のメリットとして語られるこうした特徴を、Pythonを使って実際に体験してみようと思います。

Pythonの作者Guido氏は、それほど関数型から影響を受けているとは思っていない模様ですが、Pythonには関数型言語の特徴がいくつも内蔵されています[注1]。前半では、Pythonが持っている関数の機能を紹介し、純粋な関数型言語との違いについても考えます。後半は、プログラムを関数型の思想に従って書くとどうなるか、またPythonの持つ高度な関数の機能が、そうした場面でどのように役立つかを見ていこうと思います。環境は、Python3.4を利用します。

Pythonにおける関数の機能

まずは、Pythonの関数の書き方と呼び出し方を念のため復習しておきます。引数を2乗して返す関数は次のように定義します。

```
def f(x):
  return x**2

>>> f(2)
4
```

Pythonでは、関数はファーストクラスオブジェクトですので、関数の引数にできます。たとえば、リストの要素すべてにこの関数を適用したいときは、組込み関数のmapが便利です。次のように書けば、リストのすべての要素が2乗された新しいリストが得られます。

```
>>> list(map(f, [2,3,4]))
[4, 9, 16]
```

mapは1つめの引数に関数をとり、2つめの引数にとったリストの各要素にこの関数を適用した結果を返します。Python2までは結果がリストで返ってきましたが、Python3からiterableな(要素を1つずつ返せる)map型が戻り値になりました。上の例ではわかりやすいように、リスト表示にしています。

無名関数

lambdaを使えば、無名関数を作ることができます。引数を1つとって、それを2乗して返す無名関数は次のようにして作れます。

```
>>> lambda x:x**2
```

コロンの左側が引数、右側が戻り値になります。これをこのままmap関数の引数にできるので、短い関数に名前を付けるのが面倒なときには便利です。

```
>>> list(map(lambda x:x**2, [2,3,4]))
[4, 9, 16]
```

注1) Guido van Rossum氏のブログ http://python-history.blogspot.jp/2009/04/origins-of-pythons-functional-features.html

filter

filter関数を使うと、リストの中から条件に合うものだけを選んで取り出すときなどに便利です。0から9までの整数から、偶数だけを取り出すには次のような1行を書きます。

```
>>> list(filter(lambda x:x%2==0, range(10)))
[0, 2, 4, 6, 8]
```

filter関数の1つめの引数には関数を渡します。この関数がTrueを返せばその要素は残り、Falseなら取り除かれるというしくみです。

reduce

map、filterとセットで語られることが多いreduce関数は、Python3からfunctoolsモジュールに移動しました。reduceは、引数にとったリストの要素を左から順番に2つ1組で処理していきます。リストの合計を返してくれる組込み関数sumと同じ動きをするコードを、reduceを使って書くと次のようになります。

```
>>> sum([1,2,3])
6
>>> import functools
>>> functools.reduce(lambda x,y:x+y, [1,2,3])
6
```

lambdaで作った無名関数を1つめの引数にしています。まずxに1が、yに2が代入されて、3が返ります。次に、返ってきた3がxに、残ったほうの3がyになって、最後は6になるわけです。

演算子を関数として提供してくれているoperatorモジュールがあります。足し算をしてくれるoperator.addをreduceの第一引数にしても先の例と同じ結果が得られます。

```
>>> import operator
>>> operator.add(1,2)
3
>>> functools.reduce(operator.add, [1,2,3])
6
```

リスト内包表記

Pythonのリスト内包表記を使うと、コードを簡潔に書くことができますし、関数型言語の考え方に慣れるための良いきっかけにもなるでしょう。

```
>>> nums = [2,3,4]
>>> [x**2 for x in nums]
[4, 9, 16]
```

リストの各要素を2乗した新しいリストを、リスト内包表記ではこのように書けます。最初は少しわかりにくいかもしれませんが、慣れてくるとコードを短く書けるので便利です。

先ほどfilterを使って、0から9までの整数から偶数を選び出しました。これもリスト内包表記を使って書くことができます。

```
>>> [x for x in range(10) if x%2==0]
[0, 2, 4, 6, 8]
```

2で割った余りが0になる数、つまり偶数だけが新しいリストの要素として返されます。

自然数の3つの組が、直角三角形の三辺になっているとき、それらをピタゴラス数と呼びます。1から10までの自然数で、ピタゴラス数になるものを、リスト内包表記で探してみましょう。組み合わせの羅列を得るのに、intertools.combinationsを使います。例として、1、2、3から2つを選ぶすべての組み合わせを探すには次のように書きます。

```
>>> import itertools
>>> list(itertools.combinations([1,2,3],2))
[(1, 2), (1, 3), (2, 3)]
```

結果は3通りで、それぞれの要素はタプル型になっています。

本題に戻りましょう。ピタゴラス数は3つの数の組み合わせですので、**リスト1**のようなコードで表現できます。まず、range(1,11)で定義された1から10までの整数のリストから、3つの数を選ぶすべての組み合わせを計算します。リスト内包表記のif文で、これらが三平方の定

▼リスト1　ピタゴラス数を表現

```
>>> [v for v in itertools.combinations(range(1,11),3) if v[0]**2 + v[1]**2 == v[2]**2]
[(3, 4, 5), (6, 8, 10)]
```

理を満たすという条件を書いてやれば良いわけです。

関数型言語の本質を探る

関数を引数にとれる関数があることや、無名関数などを利用して簡潔なコードが書けるということだけが、関数型言語の本質ではありません。これらの特徴を活かしてコーディングのスタイルを変えることで、メンテナンス性が良く、バグを生みにくいプログラムを作れるようになるはずです。そこでここからは、Pythonを使って関数型言語の本質を探ってみようと思います。

ユーザ登録処理

イメージをつかみやすいように、例としてWebブラウザからメールアドレスとパスワードを受け取って、ユーザ登録処理をするサーバサイドのプログラムを想定します。次のような処理の流れで、ユーザの新規登録をすることにしましょう。

❶受け取ったデータのチェック
❷ユーザがすでにデータベースに存在するかの確認
❸データベースへの書き込み

途中でエラーが発生した場合は、そのあとの処理をスキップするために、例外を送出して終了します。

受け取ったデータがdataという辞書型に格納されているとすると、**リスト2**のようなコードが書けそうです。 細かいことはさておき、**リスト2**の一連の流れを今度は関数型言語っぽく書いてみることにしましょう。

関数型言語の考え方

関数型言語の基本的な考え方は、処理をいくつかの関数に分けて、これをデータに対して順々に適用するというものです。多くのプログラミング言語では、普通に関数が使われていますの

コラム　再帰にみる関数型言語

関数を定義する際に再帰を使えるプログラミング言語は多く、もちろんPythonでもできます。再帰の例でよく引き合いに出される自然数の階乗(その数から順に1までを掛け合わせた数)を求める関数factorialは、次のように書くことができます。

```
def factorial(n):
  if n == 0: return 1
  return n * factorial(n-1)
```

引数が0のときは1を返し、それ以外のときは引数から1を引いて、再び自分自身を呼び出すという書き方です。最初にこうした再帰の定義を見ると、少しわかりにくいと感じる方もいるかもしれません。

関数型言語であるHaskellでは、パターンマッチという機能を使って次のように書けます[注A]。

```
factorial :: Int -> Int
factorial 0 = 1
factorial n = n * factorial (n-1)
```

引数が0で呼ばれた場合とそれ以外の場合を、同じ書き方で並べられるため、わかりやすさが増していると思います。

しかし、どちらも関数としての完成度を上げるためには、負の数が与えられときの対応を考える必要があります。再帰は便利な反面、バグを簡単に潜ませてしまう怖さもあります。

注A)　参考：大川 徳之,『関数プログラミング実践入門』, 技術評論社, 2014

で、これ自体は新しい考え方ではありません。ここで重要なことは、“関数を互いに接続できるようにしておく”ということです。つまり、いくつかの関数の引数と戻り値を、同じ形にするのです。そうすることで、ある関数の戻り値を引数に、別の関数を呼び出せます。これによって、処理の流れを、関数の連続的な呼び出しとして記述できます。

処理の途中でエラーが発生する可能性がありますので、これをデータとして保持するように、関数の引数になるデータを設計しましょう。先ほどのメールアドレスとパスワードの辞書に、エラー情報保持のためのリストを加え、これらをタプルにしたものを用意します。**リスト3**のコードは、関数型言語っぽく書いたPythonのコードにほかなりませんが、関数型言語の本質が少し見えてくると思います。

それぞれの関数は、データを受け取って、同じくデータを返します。ですから、一連の処理は関数の連続的な呼び出しで書くことができます。データの形は要素が2つのタプルです。途中でエラーが起きたとき、そのエラーメッセージをタプルの2つめの要素であるリストに格納するしくみになっています[注2]。関数は、もし引数のデータにエラーメッセージが入っていれば、それ以上処理ができないと判断し、引数をそのまま返します。例外が送出され、そのあとの処理をスキップするというしくみを、関数の組み合わせで実現しているのです。

▼リスト2　ユーザ登録処理のプログラム

```
import dbm
data = {'e-mail':'xxx@gmail.com', 'passwd':'pekepon3'}

# 受け取ったデータのチェック
if data['e-mail'].find('@') < 0:
  raise ValueError('E-mailアドレスが不正です。')
if len(data['passwd']) < 8:
  raise ValueError('パスワードは8文字以上にしてください。')

# すでにユーザが存在するかのチェック
with dbm.open('user_data','c') as db:
  if data['e-mail'] in db:
    raise LookupError('すでに存在するユーザです。')

# データベースへの書き込み処理
with dbm.open('user_data','c') as db:
  db[data['e-mail']] = data['passwd']
```

▼リスト3　“関数型的に書いた”ユーザ登録処理のプログラム

```
import dbm
data_t = ({'e-mail':'xxx@gmail.com', 'passwd':'pekepon3'}, [])

def validate(data):
  if data[1]: return data
  # 受け取ったデータのチェック
  if data[0]['e-mail'].find('@') < 0:
    data[1].append('E-mailアドレスが不正です。')
  if len(data[0]['passwd']) < 8:
    data[1].append('パスワードは8文字以上にしてください。')
  return data

def check_user(data):
  if data[1]: return data
  # すでにユーザが存在するかのチェック
  with dbm.open('user_data','c') as db:
    if data[0]['e-mail'] in db:
      data[1].append('すでに存在するユーザです。')
  return data

def register(data):
  if data[1]: return data
  # データベースへの書き込み処理
  with dbm.open('user_data','c') as db:
    db[data[0]['e-mail']] = data[0]['passwd']
  return data

register(check_user(validate(data_t)))
print(data_t[1]) #エラーの表示
```

仕様変更への対応

さてここで、特定のドメインのE-mailアドレスを持っているユーザ(例ではGmail)を新規登録できないようにしてほしいという仕様変更

注2）このように引数のデータを書き換える処理は、このコードが関数型的ではない点です。

の要請があったとします。関数型言語の考え方に準拠すると、1つ関数を追加して、途中の処理にこれを組み込めば、この変更が完了します(**リスト4**)。

ほかの関数を変更することなく、途中にvalidate_domainを追加できました。これは、引数と戻り値の形が同じになっているために得られる大きな恩恵です。

関数のデコレータ

関数型言語っぽくコードを書いていると、Pythonの関数型言語としての機能を、より利用できることに気がつきます。一連のユーザ登録処理に使われている関数は、引数として受け取ったデータにエラーメッセージが入っていれば、何も処理をせずそのまま引数を返すという動作を必ず行います。同じ処理をいくつもの違う場所に書いているので、これはあまり良くありません。そこで、この共通の処理を切り出して、1ヵ所にまとめることにしましょう。

これを実現するのが、「デコレータ」という機能です。デコレータは、関数を引数にとって、何らかの処理を追加した新しい関数を返します。関数の入れ子や、クロージャといった関数型言語に見られる特徴が利用されています。実際には、**リスト5**のような関数を作ります。

check_dataが引数にとる関数が、実際にデータを処理する関数です。内部で定義されているinnerの引数dataは、check_dataが受け取る関数funcの引数です。ここにエラーメッセージが入っていればそのままdataを返し、エラーメッセージが入っていなければ引数にとった関数で処理を続けるコードを書きます。デコレータには、シンタックスシュガー(構文糖衣)が用意されているので、次の2つの方法で記述できます。

▼リスト4　リスト3を"関数型的に"仕様変更

```
# リスト3に追加
def validate_domain(data):
  if data[1]: return data
  if data[0]['e-mail'].endswith('@gmail.com'):
    data[1].append('Gmailは登録できません。')
  return data

# リスト3の処理の実行部分を変更
register(check_user(validate_domain(validate(data_t))))
```

▼リスト5　デコレータの例

```
def check_data(func):
  # エラーがあれば処理をスキップするデコレータ
  def inner(data):
    if data[1]: return data
    return func(data)
  return inner
```

・定義済みの関数をデコレートする

```
validate = check_data(validate)
```

・関数を定義するときにデコレートする

```
@check_data
def validate(data):
  ...
```

合成関数

Haskellでは、ドット(.)演算子を使って関数を合成できるので便利ですが、Pythonのコードで関数を連続的に呼ぶ場合、括弧が増えてしまって不格好です(**リスト4**の最終行など)。これは、reduceを使って見栄えを少し変えられます(**リスト6**)。

reduce関数は、3つめの引数に初期値をとることができます。最初の引数としてdata_tを与え、以降はリストになった関数が連続的に適用されていきます。もちろん、リストの先頭の要素をdata_tにしてもかまいませんが、関数だけからなるリストのほうがスッキリしてい

▼リスト6　reduceを使って関数を連続的に呼ぶ

```
functools.reduce(lambda x,f:f(x), [validate, validate_domain, check_user, register], data_t)
```

てきれいです。

なぜ関数型言語はわかりにくいのか?

関数型言語は、現在主流の考え方になっているオブジェクト指向とは思考が完全にひっくり返っています。このことが関数型言語をわかりにくいものにしていることは否めません。データと関数が一体になったオブジェクトを設計し、その状態を次々に変化させていくのがオブジェクト指向だとすると、データと関数を分離して、関数の組み合わせで世界を記述するのが関数型言語の特徴だと言えそうです。この考え方は、大規模なデータにさまざまな方法論でアプローチして、結果を導き出そうとする、昨今のデータサイエンスやデータマイニングには向いているかもしれません。また、リリースしたあともニーズの変化にすばやく対応する必要があるWebサービスなどでも、関数型言語の特徴が活きてくるでしょう。

一方で、Haskellのような純粋な関数型言語にはいろいろと制約が多いのも事実です。この制約が、メンテナンス性の良さを生んでいますが、パラダイムを突然変えるのは難しいのものです。Pythonはマルチパラダイム言語ですので、Pythonを使って、これまでのスタイルでプログラミングをしつつ、来たるべき関数型言語時代に備えるというのが良いかもしれません。SD

コラム for文じゃダメなんですか?

Haskellなどの関数型言語では、変数への値の再代入ができません。これは最初に聞くと、何を言っているかわからないほど強烈な制約です。この制約があると、次のようなfor文を書くことができません。

```
v = [1,2,3]
for i in range(len(v))
  # iに毎回代入している
  v[i] += 2
  # リストの要素を変更している
```

このfor文は次のようにmap関数で書き換えられますし、mapを使うと必然的に新しいリストを返すことになるので、引数のリストを変更する処理もなくなります。

```
map(lambda x:x+2, v)
```

たとえば、インタラクティブシェルであれこれ試しながらデータを処理するコードを書いているときなど、引数のデータが変更されなければ、1つ前の処理に簡単に戻ることができるので便利です。その一方、新しく作ったリストの分だけメモリが必要です。ただこれは、最近のコンピュータの性能を考えると、あまり気にしなくても良いのかもしれません。

for文を使わず、map関数で書くようになると、Pythonのコードを書いているときにも恩恵があります。Pythonには、サブプロセスを生成して手軽に並列処理を実行できる、multiprocessingモジュールがあります(**リストA**)。ここでは、Poolの引数に生成するサブプロセスの数を指定しています。ここで出てくるmap関数は、組込みのmap関数の並列処理版ですので、使い方はほとんど同じです。最近のPCはいくつかのCPUコアを持っていることが多いので、func関数の中である程度重い処理をするときは、並列化の効果が出ます[注B]。

変数への再代入ができないという制約が、map関数という発想につながり、この考え方が並列処理と相性が良いことがわかりました。関数型言語の考え方自体は、何十年も前からあるのに、なぜ最近注目されているのか?という疑問があります。これに対しては「これまで貴重だったCPUやメモリが余り始めた」というのが、答えになると思います。最近はメモリの容量を考えながらコードを書くことはほとんどありません。また、CPUのクロック周波数は限界が見えつつあるので、コア数を増やす方向にシフトしています。こうした時代背景と関数型言語の特性が、ぴったり合ったと言えるかもしれません。

▼リストA multiprocessing**モジュール**

```
dimport multiprocessing

def func(x):
  return x+2

if __name__ == '__main__':
  with multiprocessing.Pool(4) as pool:
    pool.map(func, range(1000))
```

注B) この例のような簡単なコードでは、並列化をするためのオーバーヘッドのほうが大きいので、普通にfor文を書いたほうが高速です。

第7章

関数型が好きになる

Clojure入門 短期集中トレーニング

JavaVM上で動くLisp環境であるClojureは、ScalaやHaskellと同様にとても強力です。本稿で、ライブコーディングしながらClojureの魅力を存分に体験してみてはどうでしょうか。
筆者が惚れ込んでいる、さまざまな機能をやさしく解説してみました。ぜひトライしてください。

Author ニコラ・モドリック(Nicolas Modrzyk)　mail hellonico@gmail.com

関数型プログラミングの威力

関数型プログラミングとの出会い

> *History is the version of past events that people have decided to agree upon.*——*Napoleon Bonaparte*

もう15年前の話です。大学時代に筆者が受けていたWebセキュリティの講義の中でもっとも記憶に残っているのは、FEAL-4[注1]というアルゴリズムの説明でした。授業の目的はその暗号をクラックすること。もちろんインターネットはほとんどない時代ですので、stackoverflow.comはありません。学生は皆、Javaでちょっと汚いコードを書いて頑張ってました。筆者は、友達2人でブルートフォースするプログラムを書きました。同級生全員からデスクトップPCを借り、まるで手作りApache Sparkみたいに処理を分割して、CPUを100%フル稼働します。その結果1週間でクラックできました。先生に意気揚々と報告すると、「1週間もかかったの？　遅いわ！　Haskellでやり直して！」

——ん？　Haskell？　こういうものですね。

```
import Data.Char (digitToInt)
luhn = (0 ==) . (`mod` 10) . sum . map ↵
(uncurry (+) . (`divMod` 10)) .
       zipWith (*) (cycle [1,2]) . map ↵
digitToInt . reverse
```

今までJavaかCしか知らなかったので、読めるわけも書けるわけもありません。しかし、1日かけて書き直しました。そして実行してみると、たった1台のデスクトップPCでクラックは1時間でできました。「嘘！」ではありません。この処理速度を体験した筆者が、皆さんに伝えたいのは次の3点です。

- やはりHaskellはスゴイ！(第4章を参照)
- 正しいツールを使えば、作業が速くなる
- 作業が速くなると残業しなくても大丈夫！

ところ変わって2010年東京

日本に来た筆者は、某大手企業のプロジェクトに参加していました。まもなくカットオーバー(go live)するITシステムが、計算能力が貧弱でレポート出力できないということがわかりました。そこでワークアラウンドを探し始めたのですが——計算システムなのに、そもそも計算レポートができないのは、どうなのかなと思いながら——「外で作りましょう！」と提案してみました。その計算システムから1MBのXMLドキュメントをとりあえずメッセージキューへ飛ばして、筆者の開発した小さいアプリケーションで読み込みます。threadをたくさんspawnして、XMLをパースして計算するしくみです。その結果、チャート機能付き、おまけにリアルタイムアップデートするシステムができました。そのプロジェクトにかかったのは3週間です。

注1) FEAL(the Fast Data Encipherment Algorithm)　https://ja.wikipedia.org/wiki/FEAL

Clojureの習得に2週間、実物を作るのに1週間。コードは2,000行以下でした。

ご存じのとおりJavaでxpathを使うとしたら**リスト1**のようになります。

本章で紹介するClojureならば、

```
(prn  (map
      #(-> % :attrs :title)
      ($x "//book" "file.xml")))
```

そう。あえて3行で書いてみましたが、それでも短いですね。筆者が関数型言語で気に入っている機能はおもに次のようになります。

- コーディング量が少なくて、慣れれば基本的に何がどこにあるか、すぐわかる
- 短いうえに一度動いたら壊れない。5年前書いたコードはまだキレイに動いている
- 関数型プログラミング版のレゴのように小さいファンクションを書いて、どんどんブロックを集めて作っていく。同じファンクションをずっと使えるので、コア部分がわかりやすくて楽
- 関数型プログラミングだからこそ、オブジェクトプログラミングよりも、コアなファンクションにコンピューティングリソースを簡単にかつ効率的に使うことができる

▼リスト1　Javaで書いた場合の例

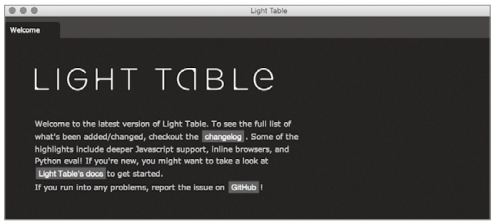
```
DocumentBuilder builder =
DocumentBuilderFactory
    .newInstance()
    .newDocumentBuilder();
Document doc =
builder
    .parse( new File("file.xml") );
XPathFactory factory =
XPathFactory.newInstance();
XPath xpath = factory.newXPath();

String location = "/test/t1/text/text()";
System.out.println( xpath.evaluate(location, doc) );

location = "//t1/t2[2]/text()";
NodeList entries = (NodeList) xpath.evaluate(
                    location, doc, XPathConstants.NODESET );
for( int i = 0; i &lt; entries.getLength(); i++ ) {
 System.out.println( entries.item(i).getNodeValue() );
}
```

Clojureの魅力

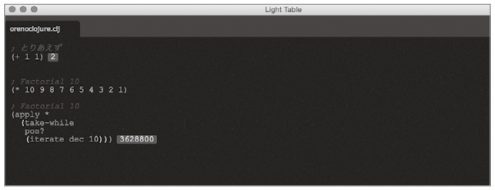
5分間でClojure環境を作りあげる

いろいろ設定するのを楽しんでる人も世の中には多いと思いますが、ここでは最小限にします。本記事ではLight Table[注2]というエディタを紹介します。まずはLight Tableをダウンロードして実行します(**図1**)。新しいファイル(Ctrl + N、Command + N)を作成します(**図2**)。

▼図1　Light Tableをダウンロードし、実行してみる



▼図2　新規ファイルを作成しコードを書く



エディタ部分に書くと、その場でライブコーディングできます。個人的にはIntellijのCursive[注3]を常用していますが、何かコードを試したいときにはLight Tableです。この章ではできる限りプラグインとほかのライブラリを使わないことを前提に執筆していますので、Light Tableで楽しんでください。いつどこででも、すぐに何かを作れることがClojureの魅力の1つですね？

注2）http://lighttable.com/
注3）https://cursiveclojure.com/userguide/

Clojureができること

コードを書く準備できたところで「Clojureで何ができるの」と思いませんか？　筆者にとってのClojureとは、次のようなものです。

・ライブ開発
・関数合成
・Mocking
・lazyness
・reducers
・コードはデータ、データはコード
・簡単なデータストラクチャ
・destructuring(分配束縛)
・Mapのキーで多型
・ウォッチャーズ(コードのスパイ)
・core.async

このリストは少し多いかもしれませんが、それだけClojureが使いたくなる、戻りたくなくなる、語りたくなる理由が多いのです。

ライブ開発(ライブコーディング)

昨年、Apple社がSwift言語とPlaygroundをアナウンスしたときに、「やっとか！」と思いました。Clojure使いならば、すでにライブ開発は当たり前です。

Light Tableに次のコード書いて、

```
(+ 1 1)
```

Ctrl + Enter(もしくはCommand + Enter)を入力すると、その行のコードが実行され、結果が表示されます。たとえばFactorial関数を書くとしましょう。ご存じのとおり、Factorial関数は引数-1の階乗である整数を返します。Pseudoコードにすると、

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ (n-1)! \times n & \text{if } n > 0 \end{cases}$$

ここをClojureの関数に変えてみましょう。

```
(defn factorial  [ n ]
  (if (= n 0)
    1
    (* n (factorial (dec n)))))
```

そしてその関数を実行すると(Ctrl + Enter)、

```
(factorial 10)
; リザルト
3628800
```

この場ですぐ結果が出るのは感動します。11、12……いろいろ試して、20を超えたら……、

```
java.lang.ArithmeticException: integer overflow
```

確かに叔母風呂(オーバーフロー)しました。

```
(defn factorial  [ n ]
  (if (= n 1)
    1
     (*' n (factorial (dec n)))))
```

'(ちょん)を付けると、「ちょん」と動きます。この'は必要なとき自動的にBigIntegerしてくれる魔法です。

```
(factorial 21)
; リザルト
51090942171709440000N
```

もうここまでくれば、読者の方も感覚的にわかってきたと思いますが、ライブ開発はこんな感じです。ライブで開発、修正もそのままでできます。「ん？　ちょっと待って、ただの関数を更新しただけでしょう？　何が特別？」

これは楽しい旅のスタート地点です。Factorial関数ではあまり驚かないかもしれませんが、たとえばその関数で音楽のループを出せば、リアルタイムで更新できます。音源の変更も可能です。グラフィック関数の場合は、drawメソッド更新すると画面がグラフィカルオブジェクトに変わります(**図3**)。

ちょっと試してみたいと思ったら、Clojureプロジェクトのサイトをご覧ください。

・Quil(http://quil.info/)
・Overtone(http://overtone.github.io/)

デザインに合わせるのもいいですが、音楽ライブのようにClojureとOvertone経由でまさにライブコーディングできます。

関数合成

Design Patternをしつこく使いたいJavaやCのプログラマは大勢いると思います。先日も、ある会社のコードを修正しようしたのですが、Visitorパターンだらけで疲れていましました。「コードを追加するために既存のコードを壊さないといけない」のは非効率です。筆者の意見としてDesign Patternだらけの言語は、やはりその言語自体があまり使いやすくないってことです。コードの目的とDesign Patternが混ざってしまい、読めなくなります。FactoryでFactoryを書いたことがある方、いらっしゃいますか？　やめましょう。

ClojureはLisp一族なので関数合成ができます。たとえばArrayの変数、それぞれに2をかけ算したい場合、倍にする関数を作れば1つの変数でArrayでも使えるようになります。

```
(defn bai-ni-suru [ n ]
  (* 2 n))

(bai-ni-suru 2)
; 4
(bai-ni-suru 4)
; 8
```

同じメソッドをそのままシーケンスにも使えます。

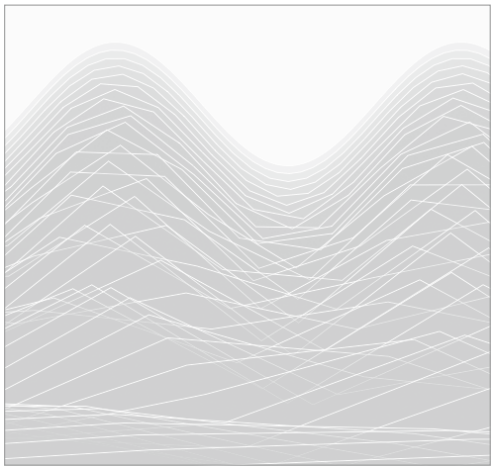
```
(map bai-ni-suru [1 2 3 4])
; (2 4 6 8)
```

mapは1つのスレッドだけ実行しますが、pmapを使うとマルチスレッドにできます。

```
(pmap bai-ni-suru [1 2 3 4 5])
; 結果は同じですが、速いはず
```

さらに、[1 2 3 4 5]を自動的に作成もできます。1から6以下までのシーケンスを作るにはrangeを使います。

▼図3　QuilとClojureを組み合わせて、グラフィックカルなデザインやアニメーションを作ることもできる



```
(range 1 6)
; (1 2 3 4 5)
```

さらに似た関数を使うとなれば、Currryingが使えます。Currryingは複数のパラメータを指定して、新しい関数を作ることができます。たとえばrangeという関数で、常に1からのスタートを指定するには、次のように書きます。

```
(def one-up-to
  (partial range 1))

(one-up-to 10)
; (1 2 3 4 5 6 7 8 9)
```

Mocking（モッキング）

テストの際など、直接テストに関係する関数の結果を指定したいときに、モッキングというやり方もよく使います[注4]。他言語だとフレームワークが必要になり、場合によっては、そのフレームワークとほかのフレームワークとが相性が悪く、時間の無駄！——と思うときもありま

注4）http://www.codeproject.com/Articles/30381/Introduction-to-Mocking、http://spock-framework-reference-documentation-ja.readthedocs.org/ja/latest/interaction_based_testing.html

す(うん、よくある！　今まさにクライアント先で奮闘中)。それは置いておいて、ここでClojureの優秀なポイントを1つ紹介します。モッキングフレームワークは使わず、関数の合成の影響でモッキングする方法です。

たとえば、作ったone-up-toの関数を期間限定！として、違う関数にしたい場合です。そこでwith-redefsを使います。そうすると、そのwith-redefsのシーケンスの中だけで違う関数になるように指定できます。

```
(with-redefs
  [one-up-to (partial range -10)]
  (one-up-to 10))
; (-10 -9 -8 -7 -6 -5 -4 -3 -2 -1 0 1 2 3 ↵
4 5 6 7 8 9)
```

先ほどの結果と違います。でもまあ、この使い方はいつもはletと使わないって言われるかも。

```
(let
  [one-up-to (partial range -10)]
  (one-up-to 10))
; (-10 -9 -8 -7 -6 -5 -4 -3 -2 -1 0 1 2 3 ↵
4 5 6 7 8 9)
```

確かにそう。ですが、こうやって書くとわかりやすくなります。

```
(defn compute [ n ]
  (apply +
    (one-up-to n)))

(compute 10)
; 45
```

compute関数はone-up-toのシーケンスの変数を、それぞれ足して結果を出します。with-redefsをもう一度使ってみようとすると、

```
(with-redefs
  [one-up-to (partial range -10)]
  (compute 10))
;  -10
(compute 10)
; 45
```

そうそう。それだよ！　compute関数の結果が変わりますが、with-redefsのシーケンスの中でだけです。なので簡単にモッキングできるし、細かく指定されたコードだけをテストできます。こんなことできたらLazyになりそうです！

I'm Lazy.「Lazyは良いことです!」

rangeもpartialもLazy関数と呼ばれています。これはなぜか？　Light Tableで開発するとすぐに結果が表示されますね。なぜなら、その結果が表示しようとしてるprint関数を読んで、Light Tableが表示してくれます。だから、その結果を表示しない限り、計算しません。それがLazynessです。

iterate関数で説明するのが一番わかりやすいでしょう。iterateは変数パラメータが2つあります。

```
(iterate f x)
; (f (f (f x))) ... 永遠
```

と永遠に続きますが待ちたくないので、少しずつ表示したいものだけを、表示させたい場合はtake関数へ。先ほど作ったone-up-to関数を利用してみましょう。

```
(take 3 (one-up-to 10))
; (1 2 3)
```

下記のコードは期間期限！なので実行する前に考えてみましょう。

```
(iterate inc 1)
```

このコードに新しいシーケンスを作成します。ただ、このシーケンスに終わりはありません。永遠に続きます。表示させたくても永遠に止まらないので、最初の5つの変数だけを表示させましょう。

```
(take 5 (iterate inc 1))
; (1 2 3 4 5)
```

先ほどのbai-ni-suru関数も、もちろん使えます。

```
(take 10 (iterate bai-ni-suru 1))
; (1 2 4 8 16 32 64 128 256 512)
```

もうお気づきかと思いますが、基本的に関数合成を使った場合、制限はありません。次のClojureコードを必ずエディタに入力して、結果を確認してください。

```
(take 10
 (filter #(< 1000 %)
   (iterate bai-ni-suru 1)))
; ...
```

関数が変数と同じ扱いになります。データが流れて、いくつかの更新にいくつかアプライして、最後のデータを返すという流れになります。すべてデータからの観点です。この記事の1つのセクションでコードisデータについて説明をしますが、その前に語っておきたい僕の友達「Reduce」ちゃんです！

レデュース（Reduce、Reducersなど）

レデュースは少し古い技法です。3年前の古い記事[注5]があります。コレクションの関数は基本的に変化するものです。しかし、今まで使っていた関数はすべてシングルスレッドです。map、filterなどの関数には、簡単に実行できるようにこのreducersのネームスペース（名前空間）があります。ネームスペースは宇宙とは関係ないものですが、単に関係するClojureコードと合体させるものです。一般的な使い方は、1ファイル＝1ネームスペースです。reducersネームスペースにはreducersに関連する関数が集まっています。ネームスペースを現在使われているところで用いるとき、こんな呼び出し方をします。

```
(require '[clojure.core.reducers :as r])
; nil
```

requireを使うとprefixをつけられるので、ここで「r」にしてみましょう。最後のサンプルのシーケンスの変数を1つ1つ足していきたい場合、「reduce +」を使います。

```
(reduce + (take 10
 (filter #(< 1000 %)
   (iterate bai-ni-suru 1))))
; 1047552
```

この呼び出し方であれば、スレッド1つだけで並列処理はまったくありません。今はもう2015年です。CPUコアもたくさんありますので楽しみましょう！

```
(r/reduce +'
 (take 10
  (filter #(< 1000 %)
   (iterate bai-ni-suru 1)))))
; 1047552
```

integer overflowが出ているので、'の処理を使いましょう。

```
(defn bai-ni-suru [ n ]
  (*' 2 n))
```

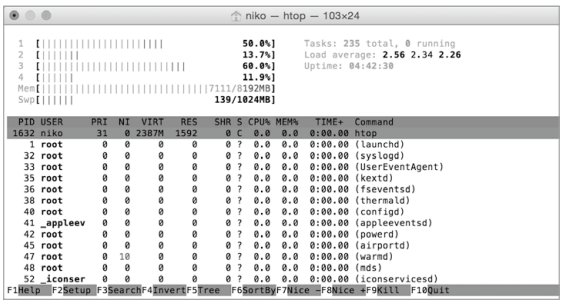
そして、あらためてreduceにすると、

```
(r/reduce +' (take 1e5 (filter #(< 1000 %)
(iterate bai-ni-suru 1))))
```

htopで見ると、コアが一生懸命頑張ってくれるのが見てとれます（**図4**）。

どんなデータ処理でもClojureの単純なコードで並列実行できます。MapReduce、Hadoopのセットアップをしなくても済む人生は素晴らしい！（笑）

▼図4　htopコマンドでCPUの状態を見る



注5）http://clojure.com/blog/2012/05/15/anatomy-of-reducer.html

このreduceの処理を次のステップに持っていくTesser[注6]という素晴らしいライブラリがあります。Tesserとparkour[注7]を使うと、簡単で素晴らしいHadoopのジョブが書けてデプロイもできます。特別なコードでもなく定型として、今回紹介したコードをベースにすぐ書けます。

コードはデータ、データがコード

Clojureのコードはデータです――ん？　どういう意味？　頭もデータですか？　頭はデータですよ！……ただこの記事は哲学論文ではないので、Clojure観点で進めます。前の節で書いたコードは、

```
(+ 1 1)
```

でしたよね？　何かがコードをパースして処理してますね。read-string関数を使ってみましょう。

```
(read-string "(+ 1 1)")
; (+ 1 1)
```

あら、楽しくない結果です。ちょっと待って！そのまま戻されたっていうのは……そう、Abstract Syntax Treeは、そのまま表示されるし、そのまま編集できます。ちなみにコードを実行する関数はevalです。

```
(eval
 (read-string "(+ 1 1)"))
; 2
```

wow。Evaluation[注8]は最高ですね！　ところでREPL、Read、Eval、Print、Loopはそういう意味では単純なことしかやっていません。Quoteのスペシャルフォーム使うとevalの実行を停止できます。

```
'(+ 1 1)
; (+ 1 1)
(quote (+ 1 1))
; (+ 1 1)
```

Macro

Macroはコードの順番と形を管理できるLispの機能で、それはClojureにもあります。Macroを使ってコードの処理順番を変えることができます。Macroを書いて、evalを呼ぶ前に、コードをデータとして編集できます。自然界の言葉に言い換えると「木の枝」を整理できます。

たとえば、f1とf2という2つの関数を時間差で呼びます(f2を先に呼んでからf1を呼ぶ)この場合のMacroの書き方は、

```
(defmacro junban-kaeru [ f1 f2 ]
  `(do
    ~f2
    ~f1))
```

`の部分は印刷の関係で読みづらいかもしれませんが、これは「コード書きます！」という意味です。~はここでコピペしてください。~f1の部分はこの位置にf1のコードをそのまま入れます。

Macroexpandを使えば、これらがどういうコードになるのか確認できます。今書いたjunban-kaeru Macroのコードを確認できます。

```
(macroexpand-1
 '(junban-kaeru
   (println "ichi")
   (println "ni")))
; (do (println "ni") (println "ichi"))
```

実行すると、

```
(junban-kaeru
   (println "ichi")
   (println "ni"))

; consoleをチェックするところですね?
```

いつも使われてるとしたら短いのも書けますし、そのままClojureのコードの形も変えられるよう

注6) https://github.com/aphyr/tesser
注7) https://github.com/damballa/parkour
注8) http://clojure.org/evaluation

になります。

```
(defmacro $ [ f1 f2 ]
   `(do
    ~f2
    ~f1))

($
 (println "ni")
 (println "ichi"))
```

こういうふうに簡単にDSL(Domain Specific Language)を書けるので参考になるかと思います。Clojureが出たときにtestフレームワークがそのまま含まれていて、clojure.testのネームスペースを読むとよいでしょう。

Destructuring（デストラクチャリング）

次に、デストラクチャリングを紹介します。現在でも使う人は少ないかもしれませんが、筆者としては、この機能がまれにとても便利なときもあります。

aisatsuという関数を書いてみましょう。

```
(defn aisatsu [ namae ]
 (println "おい、" namae))
```

```
(aisatsu "竹田さん")
; おい、竹田さん
```

defnはそのネームスペースの中で関数の登録のようなことができます。ネームスペースより小さいコードの部分に登録したい場合はletが使えます。

```
(let [aisatsu-suru (fn [ namae ]
(println"おい、" namae))]
    (aisatsu-suru "竹田さん"))
```

letのシーケンスの中でだけaisatsu-suruが使えます。なぜ、あらためてletの話をするのかというと、destructuringの話をするうえで一番わかりやすいのです。

先ほど書いたrangeとtakeのサンプルコードでは、

```
(take 2 (range 5)) ; (0 1)
```

その結果を簡単に変数にするのがdestructuringです。

```
(let [[a b] (take 2 (range 5))]
  (println a)
  (println b))
; コンソールには...
; 0
; 1
```

その上に、新しいシーケンスも作れます。

```
(let [[a b & c] (take 5 (range 10))]
  (println c))
; (2 3 4)
```

筆者は、これがかわいいと思うので、ここに書きます。次のコードをぱっと見ると、永遠に続きそうな処理に見えますが、我々はLazyなので必要なことしかしません。

```
(let [[x & xs] (range)]
[x (take 10 xs)])
```

これで、ちゃんと処理が終わりますし、結果はこうです。

```
[0 (1 2 3 4 5 6 7 8 9 10)]
```

さらに、mapにも使えます(どんどん面白くなってるって思わない？)。

```
(let [ {a :a} {:a 1 :b 2 :c 3}]
  (println a))
; コンソールには1
```

どういうことでしょうか？　{a :a}はインプットされたマップのキー:aのバリューを指定できます。インプットされたMapは{:a 1 :b 2 :c 3}なので、そのMapのキー:aのバリューは1になります。

こういうMapのdestructuringはClojurescriptの世界でもけっこう使われていますので、覚えておきましょう。

Mapのキーでディスパッチ

Mapの処理を毎回やらないといけないですね。Mapのキーで処理できてしまうこともClojureの1つの機能です。defmultiを用いて、処理を

準備します。

```
; メソッド名を定義する
(defmulti nomu :kuni)

; キーによってインプリメントする
(defmethod nomu ::french [p]
  (str (p :name) "はワインを飲みます"))
(defmethod nomu ::japanese [p]
  (str (p :name) "は日本酒を飲みます"))

; キーのバリューが存在しない場合
(defmethod nomu :default [p]
  (str (p :name) "は飲めません"))
```

そして、multimethodを試してみましょう。

```
(nomu {:kuni ::french :name "Nico"})
; "Nicoはワインを飲みます"

(nomu {:kuni ::japanese :name "Abe-san"})
; "Abe-sanは日本酒を飲みます"

(nomu {:kuni ::american :name "Chris"})
; "Chrisは飲めません"
```

よく見ると、defmultiのkuniは関数です。そう、キーワードは関数としても使われます。

Immutable（イミュータブル）

Clojureの大事な機能があります。Immutableです。使っている変数を変えることは簡単にはできません。bai-ni-suruをもう一度使いたいので再掲します。

```
; ちょん付きのバージョン2
(defn bai-ni-suru [ n ]
  (*' 2 n))

; 呼ぶと
(map bai-ni-suru [1 2 3])
; (2 4 6)
```

(2 4 6)は完全に新しいシーケンスです。Immutableなので、普通のlet,def...で作った変数は変えることはできません。新しいものになります。

```
(def bai-ni-shitai [1 2 3])
(map bai-ni-suru bai-ni-shitai)
; (2 4 6)
```

倍にしたいものは倍にできますが、それはそれで残ります。変えたい場合は、atomかrefを使います。指定されたもののバリューを変えたければ、トランザクションを使わなければなりません。

```
(def aratamete-bai-ni-shitai
  (atom [1 2 3]))

@aratamete-bai-ni-shitai
; atomを読みたいときに、deref、もしくは@を使う
; [1 2 3])
```

変数を更新(swap!)したいか、再設定(reset!)したい場合、トランザクションを実行して、アトミックな結果を出せます。

```
(swap! aratamete-bai-ni-shitai
 (partial map bai-ni-suru))
; (2 4 6)

; atomのバリューがちゃんと変えたのは確認
@aratamete-bai-ni-shitai
; (2 4 6)

; 再設定
; (reset! aratamete-bai-ni-shitai [1 2 3])
; [1 2 3]
```

同時に呼ぼうとしても、swap!もreset!もトランザクションロックを裏側に使っていますので、Multithreadな処理でも壊れません。

トランザクションにWatchers

Watchersは怪しいものではありません。Watchersは指定されたatomにトランザクションが起きるときに呼ぶものです。callbackに近いかもしれません。

Watcherを作成したい場合、add-watchで進みます。

```
(add-watch aratamete-bai-ni-shitai :watcher
  (fn [key atom old-state new-state]
    (prn "-- Atom Changed --")
    (prn "key" key)
    (prn "atom" atom)
    (prn "old-state" old-state)
    (prn "new-state" new-state)))
```

作ったら、aratamete-bai-ni-shitaiを更新す

る直前、このWatcherが呼び出されています。

Light Tableはその場ですぐに実行できるので、もう1回コードを呼びたい場合、書かなくても再現できます。swap!のところを戻して、何回も実行してみましょう。

```
(swap! aratamete-bai-ni-shitai (partial ↵
map bai-ni-suru))
```

1回目に呼ぶとコンソールには、

```
orenoclojure.clj:
"-- Atom Changed --"
orenoclojure.clj:
"key" :watcher
orenoclojure.clj:
"atom" #<Atom@334d4108: (16 32 48)>
orenoclojure.clj:
"old-state" (8 16 24)
orenoclojure.clj:
"new-state" (16 32 48)
orenoclojure.clj:
```

連続で2回目には、

```
"-- Atom Changed --"
orenoclojure.clj:
"key" :watcher
orenoclojure.clj:
"atom" #<Atom@334d4108: (32 64 96)>
orenoclojure.clj:
"old-state" (16 32 48)
orenoclojure.clj:
"new-state" (32 64 96)
```

3回目はおまけに自分でやってください。せっかくMultithread機能をたくさん見たので、最後にcore.asyncを少し紹介します。

Leiningen＆Pomegranate

あれ？　core.asyncの話をするって言ったのに、ここにはLeiningen[注9]という動物が……確かに、Leiningenを話すつもりは、今日はありません。Leiningenはビルドツールなので、話が始まるとどこで話をやめるのかたいへん面倒です。が、Light Tableの裏ではLeiningenが動いていますので新しいDependencyを追加したいときにLeiningenの設定ファイルをいじるのが一番楽。とりあえずね。USER_HOMEのフォルダの下に、こんなファイルをつくりましょう。

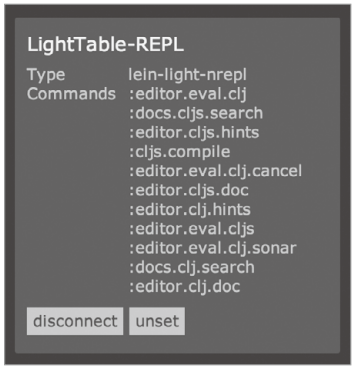
```
; ~/.lein/profiles.clj
; %USER%/.lein/profiles.clj
; 中身は
{:user
    {:dependencies
[[com.cemerick/pomegranate "0.3.0"]]}}
```

この設定ファイルにdependencyとプラグインも管理できるし、ほかのJava Machineの設定も可能です。とりあえず今日はPomegranateを使えるようにします。Pomegranateとはランタイムのときに新しいdependencyのダウンロードを管理してくれるライブラリです。

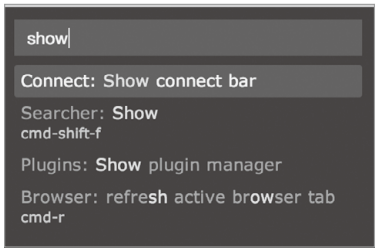
Light Tableのコマンドパネルを呼び出して、今裏で動いてるREPLを切断しましょう(**図5**、**図6**)。

設定ファイルを直してからLight Tableエディタのところで Ctrl + Enter を入力し、新しいライブ環境を作りましょう。すべてうまくいくと、こんなコードが実行できます。

▼図5　Light Tableの設定パネル



▼図6　Light TableでREPLを切断



注9）http://leiningen.org/

```
(use
    '[cemerick.pomegranate :only (add-
dependencies)])
```

単にadd-dependenciesというpomegranateのメソッドをインポートしています。

core.async

先の設定変更のおかげで単純にcore.asyncのダウンロードができます。

```
(add-dependencies
 :coordinates
 '[[org.clojure/core.async
"0.1.346.0-17112a-alpha"]])
```

さっそくcore.asyncをrequireしましょう。

```
(require '[clojure.core.async :refer :all])
```

もう、我慢できません。core.asyncの紹介ができるまでにこんな長い説明になりましたが、もういいよ。使おう！　core.asyncは並列実行できるコードを書けるだけではなく、コードはいつもの関数のような使い方になります。core.asyncの基本はチャネルで始まります。チャネルを作るためにはchanを呼びます。

```
(def a-channel (chan))
```

chanが完全に元気そうですので、ちょっと挨拶してみましょう。>!!を使うとチャネルにメッセージを送れます。

```
(>!! a-channel "Hello, chan!")
; true
```

a-channelniにメッセージ"hello,chan!"を入れました。処理したいときに<!!を使います。

```
(<!! a-channel)
; "Hello, chan!"
```

処理が終わったときにchanを無効にするには、

```
close!
; これはコードではない
(close! a-channel) ; nil
; これはコードです。
```

close!を呼んだらchanが使えなくなりますので注意してください。core.asyncのthreadの関数使うと、単純なthread、そしてバックグラウンド処理ができます。

```
(thread
 (dotimes [i 10]
  (println (str "Hello" i))))
; コンソールにはいろいろなHelloがプリントされたでしょう
```

thread間の通信能力はchanからきています。次のコードはバックグラウンドスレッドをspawnして、メインスレッドに届いてくるいくつかのchanからのメッセージをprintします。

```
(let [c (chan)]
  (thread (dotimes [i 10] (>!! c "hello2¥ン
n")))
  (print (<!! c))
  (print (<!! c))
  (close! c))
```

Voila！　synchronize、とか、リソースのロックとかいろいろ書かなくても済みます。きれいな開発ができます、コードも読みやすくなります(よっしゃー！たくさん使いましょう)。

明るい未来

core.asyncを含めて、本章で勉強したパターンのほとんどすべてをClojurescriptでも使えます。Clojureコードを JavaScriptのVM上で実行できるのがClojurescriptです。フロントエンドもバックエンドもその間も、EDN[注10]ですべて同じ形になります。より統合的に、楽に開発できるので、ビジネス価値があるところだけに集中できるようになります。Clojurescriptをもっと紹介したいのですが紙幅が尽きました。またの機会をお楽しみに！　アビエント！SD

注10) https://github.com/edn-format/edn

第2特集

安全な通信を確保するSSL/TLSの教科書

インターネットの通信セキュリティを確保するしくみをマスターしよう!

スノーデン事件、POODLE事件など、インターネット上のセキュリティ意識が高まりを見せている昨今、本特集ではその基礎技術であるSSL/TLSについて解説を行います。

第1章では、現在のインターネットの安全性に関し、通信内容の盗聴、データの改ざん、なりすましなどがないように、どのような暗号化や認証などが行われているのかを解説します。

第2章では、SSL/TLSによる暗号通信の流れと、その暗号化技術(暗号スイート)のアルゴリズム、SSL/TLSのバージョンによる違いについて詳しく解説します。

第3章では、過去に起こった実際の事件やTLSの脆弱性を例にとり、安全な通信を確保するためのTLSサーバの設定方法について解説します。

Appendixでは、TLSの今後の話としてTLS 1.3とHTTP/2の概要を紹介します。

第1章 インターネットの安全性と暗号技術

Author セコム㈱IS研究所　島岡 政基（しまおか まさき）

脅かされるインターネットの安全

暗号技術は情報通信基盤の中に溶け込んで広く普及してきました。しかしここ数年でその暗号技術に対する攻撃や脅威が盛んに話題に上るようになりました。インターネットで安全な通信を実現するには暗号技術は欠かせない存在であり、その典型例の1つにSSL/TLSがあります。筆者の所属するセコム（株）IS研究所では、SSL/TLSを始めとする認証技術と、その要素技術である暗号技術について研究しており、サーバ証明書を発行する認証局の最上位であるルート認証局の構築運用にも長く携わってきました。このため本章や第3章で述べる状況はとても残念に感じるとともに、この状況を打破すべく少しでも仲間を増やせればと思い、今回、筆を執らせていただきました。

本章では、暗号技術が安全な通信をどのように実現するのか俯瞰し、続く第2章でその具体的な実装であるSSL/TLSについて解説したあとで、第3章で我々が直面している問題と、その対策について整理していきたいと思います。

広域監視問題と暗号解読

エドワード・スノーデン氏が暴露した米国NSA（国家安全保障局）によるインターネットや電話網に対する大規模かつ高度な盗聴事件、いわゆるPRISM事件は、広域監視問題として世界に大きな衝撃を与えました。いくつか理由はありますが、その1つには、「今まで安全とされていた暗号通信ですら、一部解読可能な状態で収集されていた」という点があります。

もともとインターネットでは、そこに流れるデータの多くは平文で、通信データを傍受できるものであれば盗聴は容易とされてきました。このためクレジットカード番号やパスワードのような機微な情報をインターネット上でやりとりする場合には、盗聴困難な暗号通信を用いるべきだと言われてきたわけですが、スノーデン氏の暴露によって「暗号通信であっても、国家規模の情報収集に対しては秘匿できない」という脅威が明らかになったのです。

こうした状況で着目を浴び始めたのがPFS（Perfect Forward Secrecy）という概念です。詳しいしくみは第2章で解説しますが、暗号通信に用いる鍵を使い捨てにすることで、仮に暗号鍵を盗まれても盗聴できる期間をごくわずかに限定するものです。

PFSの概念は昔からあったものの実装が普及していませんでした。しかしPRISM事件を契機に実装の普及が進み、2015年6月の時点で世界の約2/3のサイトが何らかの形でPFSに対応しています。しかし一方で、十分な形でPFSに対応できているサイトはたかだか1/3程度にとどまっており、期待するとおりにPFSが使われるにはまだ時間がかかりそうな状況です（コラム「SSL定点観測サイト SSL Pulse」参照）。

SSL/TLSプロトコルに対する一連の脆弱性や攻撃

2011年後半から、SSL/TLSに対する脆弱性報告や攻撃手法の発見が盛んになってきましたが、その中でも典型的かつ大きな影響を及ぼし

た一例としてBEASTとPOODLEが挙げられます。これらの攻撃については第3章で触れますが、いずれもバッファオーバーフローやコードインジェクションのような一般的なソフトウェア実装の脆弱性とは大分異なり、暗号技術の実装上の脆弱性を突く、言わばSSL/TLSプロトコルや暗号スイートに対する攻撃と言えます。

したがってパッチによる対処だけでは不十分で、SSL/TLSプロトコルのバージョンアップや暗号スイートの変更などによる対処が必要となります。

しかしながら、BEASTおよびPOODLEでは、発見当時広く利用されていた暗号スイートやプロトコルバージョンが対象だったため、これらの利用を避けようとすると選択肢がかなり限定的になってしまうという課題がありました。さらには、どのクライアント環境でどのような暗号スイートを利用可能か、対応可能なプロトコルバージョンは何か、といった情報がベンダから十分に開示されておらず、対策した場合の影響が十分に見積もれない、といった課題も浮き彫りになりました。

暗号スイートを理解しよう

広域監視問題についてはPFSを導入すればよく、またその実装も普及が進んでいます。しかし、正しく設定できていないサイトも多く、混乱が見受けられます。またSSL/TLSに利用されている暗号技術に対する本格的な攻撃は、暗号スイートの正しい理解なしには対策が難しいこともわかってきました。実はPFSの設定は、暗号スイートに大きく依存しており、したがっていずれの問題も暗号スイートに帰結します。

暗号スイートを、暗号技術の理解なしに適切に設定することは難しく、このため今まであまり注目されることがありませんでした。しかし昨今のこうした事情により、これからの時代に安全な通信を実現するには、暗号スイートの適切な設定が必要になってきました。

そこで、今回の記事では安全な通信の典型的な実装技術であるTLSと、そこで利用される暗号スイートについて正しい理解を深めていきましょう。正しい知識を持って運用すれば安全な通信は十分に実現できます。少しでも安全な通信をインターネット上に増やしていきましょう。

安全な通信の3大要件

暗号通信という言葉がよく用いられますが、インターネット上での(とくにWebサイトとの)暗号通信は、単に通信を暗号化するだけでは実現できないケースの方が多いでしょう。これは、通信相手が必ずしも事前に特定できないことと、インターネットのしくみ上、途中で第三者が通信を盗聴・改ざんできるリスクを伴っていることに起因します。

本来私たちがインターネット上で実現したいと思っている暗号通信は、単なる通信の暗号化と区別するために「安全な通信(Secure Communication)」と呼ばれ、これを実現する通信プロトコルはセキュリティプロトコルと呼ばれます。この安全な通信は、以下の3要件を満たすことで実現されます。

- 通信内容が盗聴されないこと(秘匿)
- 通信内容が不正利用されていないこと(改ざん検知)
- 通信相手が偽物でないこと(認証)

これらはそれぞれ通信のCIA(Confidentiality:機密性、Integrity:完全性、Authenticity:真正性)に対応します。

本節では、これら3種類の要件について簡単に解説していきます。

秘匿

オンラインのショッピングサイトで買い物をするためにクレジットカード番号を入力したり、SNSなどのWebサービスにログインするためのパスワードを入力する場面はみなさんも経験があると思いますが、ここで送信する内容がも

し第三者に盗聴されたらどうなるでしょうか。自分になり代わって勝手に買い物をされたり、記事を投稿されたりしてしまうかもしれません。そこで盗聴されても内容が読めないように通信内容を暗号化することが必要になってきます。

一般には、受信者だけが復号できる何らかの方法を用いて、送信者が送信内容を暗号化して送信することによって暗号通信が実現されます。この「何らかの方法」を実現するのが暗号技術で、アルファベット表からn文字分ずらすような換字式暗号は、古くローマ時代から使われていました。

もちろん情報処理技術の発達した現代では、換字式暗号のような単純な方式ではすぐに解読されてしまいますから、もっと数学的に高度で複雑な暗号技術が使われるようになってきています。この暗号方式の種類については第2章で解説します。

認証

暗号通信によって盗聴対策ができても、自分が通信している、クレジットカードやパスワードを送信しようとしている相手が偽物だったらたいへんです。そこで、相手が本当に自分の通信しようとしている相手かどうかを確認する必要があります。この確認する行為のことを認証と言います。

通信相手が知っている相手であれば、あらかじめ秘密の合言葉(ヒラケゴマなど)をお互いに決めておくことも可能かもしれません。しかしインターネットにおける通信相手は、初めて訪れるWebサイトであることも少なくありませんから、こうした秘密の合言葉を事前に決めておくことも容易ではありません。

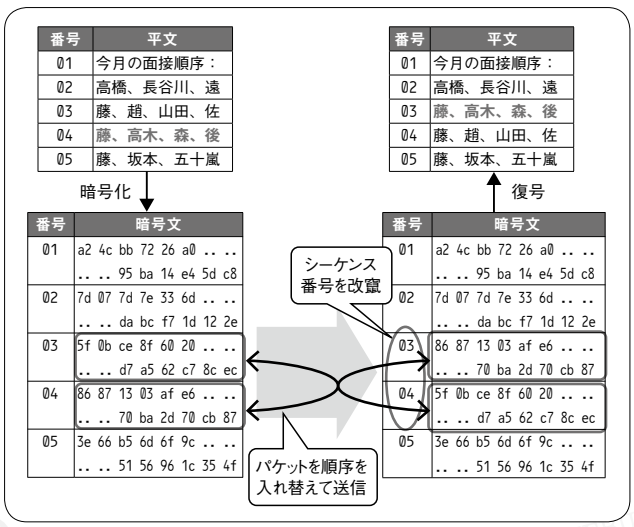
そこで、秘密の合言葉とは別の方法を使って相手を認証する必要があります。暗号方式の一種である公開鍵暗号方式によって、お互いに秘密の合言葉を事前に決めておかなくてもこれが可能になりました。今回取り上げるSSL/TLSでも公開鍵暗号方式に基づいた電子証明書(公開鍵証明書)を用いて認証を行います。公開鍵暗号を使った認証のしくみは少しややこしいので第2章で詳しく解説します。興味のある方はそちらも合わせてご覧ください。

改ざん検知

「通信内容を秘匿できていれば、そもそも改ざんできないだろう！　少なくとも復号できる形で改ざんすることはできないはずだ！」と思っている方、残念、80点です。送信したい情報を丸ごと暗号化していれば確かに改ざんできないのですが、実際の通信では情報は一定のサイズのパケットに分割されて送信されます。SSL/TLSでは、暗号化はパケットを対象として行うので、たとえ個々のパケットが暗号化されていても、場合によっては**図1**のようにパケットの並べ替えをするだけで復号したときに異なる情報に変わってしまう場合があります。

また、通信においては再送攻撃というものがあります。TCP/IPの特性上、通信中は定型的な情報が何度もやりとりされることになりますから、たとえその内容が暗号化されていても、再送

▼図1　暗号パケットの並べ替え攻撃のイメージ



攻撃など第三者に不正に再利用されてしまう可能性があります。そこで受信者は、今受信したパケットがたしかに今さっき送信者が送信しようとしたパケットなのか、あるいは送信者が過去に誰かに送信したパケットを第三者が再送しているだけなのか、区別できる必要があります。

このように、安全な通信における改ざん検知では、上で述べた並べ替え攻撃や再送攻撃など、パケットの不正な利用についても対策する必要があります。この改ざん検知技術として、MAC（メッセージ認証符号；Message Authentication Code）というものがあります。正確にはMACはそれ自身で並べ替え攻撃や再送攻撃を防ぐものではありませんが、ハッシュアルゴリズムという暗号技術を使用して、これを実現しています（詳しくは後述します）。いずれにしても、送信者がパケットにMACをつけて送信し、また受信者は受信したMACを検証することによって、パケットが改変されたり再送されたものではないことを確認することができます。

安全な通信を実現する暗号技術

安全な通信を実現するには、認証と暗号化、改ざん検知（MAC）が必要であることは前節で述べました。本節では、これを実現する暗号技術について説明します。

暗号方式：共通鍵暗号と公開鍵暗号

この3つの要件を実現する要素技術が暗号技術です。暗号技術は、暗号鍵の特徴によって共通鍵暗号、公開鍵暗号に大別されます。

共通鍵暗号は、事前にお互いに暗号・復号に必要な秘密の情報（つまり暗号鍵）を共有してお

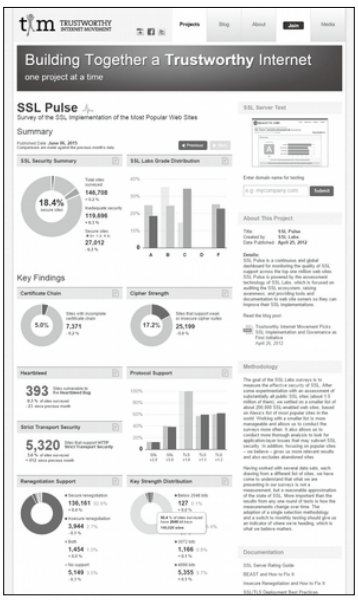
Column SSL定点観測サイト SSL Pulse

SSL Pulse[注1]（**図2**）は、SSL/TLSに対する攻撃が激化したことを受けて、2012年に有志が立ち上げたサイトで、世界の主要なWebサイト約20万件のSSL実装状況について毎月調査を行い、そのレポートを公開しています。SSL Pulseが観測している項目として、たとえば次のようなものがあります。

- SSL/TLSプロトコルバージョンのサポート状況
- HeartbleedやBEASTなど主要な脆弱性対応状況
- PFSやSPDYなど主要な技術のサポート状況

きちんとツボを押さえた項目（しかも新しく影響の大きな脆弱性が発見されるとすばやく調査項目に追加してくれます！）を調査していることに加え、レポートページも最近流行りのいわゆるダッシュボード風で一覧性も高いので、ぜひ一度アクセスしてご覧になってみてください。サイト管理者の方は、PFS導入などにあたって経営層への説明資料としても役立つかもしれません。

▼図2　SSL Pulseのレポートページ

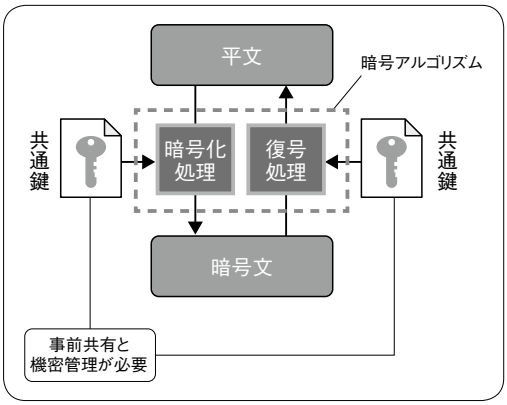


注1）https://www.trustworthyinternet.org/ssl-pulse/

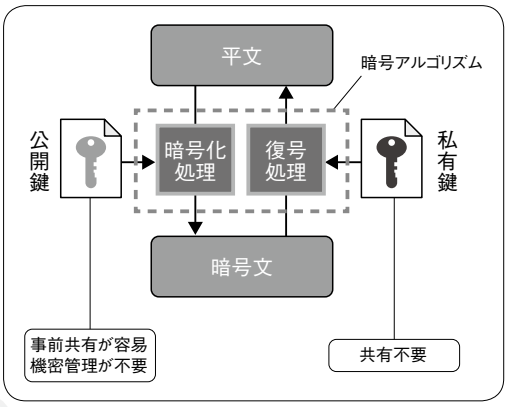
いて、これを用いて暗号化・復号を行うものです(**図3**)。秘密の情報ですので、事前に共有する際に共有する相手を間違えてはいけません。また共有した両者は暗号鍵が不要になるまでは大事に保管しておかなければいけません。これは少数で利用するうちはよいですが、多数の相手と同時ないし並行して利用することになると煩雑になってくることは容易に想像できるかと思います。いろいろ面倒そうにも聞こえますが、次に述べる公開鍵暗号と比べるとしくみが複雑でない分、処理速度がかなり速いです。

一方の公開鍵暗号は、送信者が暗号化を行うための公開鍵と、受信者が復号を行う私有鍵を組み合わせて利用します(**図4**)。暗号化を行う公開鍵は文字どおり秘密にする必要がないので、共通鍵と比べて不特定の送信者と事前に共有することも容易です。秘密にする必要があるのは復号に必要な私有鍵だけで、これは送信者など他人と共有する必要がありません。つまり、暗号鍵の管理という点では公開鍵暗号に利があります。その代わり共通鍵暗号と比較すると暗号化(アルゴリズムによっては復号)処理に時間がかかります。Webサイトの認証については、事前に秘密情報を共有することなく実現できる公開鍵暗号が向いていることは先に述べました。

▼図3　共通鍵暗号の概念



▼図4　公開鍵暗号の概念



鍵交換による公開鍵暗号から共通鍵暗号へのスイッチ

暗号技術には、公開鍵暗号と共通鍵暗号の2種類があることがわかりました。一方で、安全な通信に必要なのは、認証と秘匿(暗号化)とMACです。

Webサイトの認証には、公開鍵暗号が向いています。これは初めて訪問するサイトと事前に暗号鍵を共有することは難しいですが、公開鍵を公開することはそれほど難しくないからです。厳密に言うと、公開している公開鍵の真正性を確認する必要がありますが、これは信頼する第三者機関である認証局からの証明書発行によって解決します。詳しくは第2章で解説します。

次に秘匿と改ざん検知ですが、これは認証できたWebサイトとの通信において、通信パケットを暗号化するとともにMACを生成・付与することによって実現します。パケットごとに暗号化やMACの生成・付与を行うには、十分な処理速度が求められるため、ここは共通鍵暗号を使いたいところです。そこで登場するのが鍵交換です。鍵交換は、共通鍵暗号を事前共有なしに必要時にその場で共有する方法で、公開鍵暗号のしくみを使って実現されます。つまり公開鍵暗号があれば、共通鍵暗号をその短所なしに活用できるわけです。

MACとハッシュアルゴリズム

メッセージ認証符号(MAC)は、ハッシュアルゴリズムという暗号技術を用いて生成されます。

このハッシュアルゴリズムとは、任意の長さのデータを入力として固定長のデータを出力するもので、出力されたデータはハッシュ値と呼ばれます。ハッシュアルゴリズムには、2つの性質が求められます。1点目は一方向性で、ハッシュ値から元のデータを復元できない性質です。2点目は耐衝突性で、どのデータから得られたハッシュ値も重複しない性質です。どちらも完全に満たすことは論理的に不可能ですが、十分な高い確度でこれらの性質を満たすものは暗号学的ハッシュアルゴリズム(以下、単にハッシュアルゴリズム)と呼ばれます。ハッシュアルゴリズムは共通鍵暗号よりもさらに軽量に処理できます。

これらを整理すると、**図5**に示すように、

a)公開鍵暗号を用いた認証と鍵交換
b)共通鍵による暗号化とハッシュアルゴリズムによるMACの生成・検証

という2段階で処理されることになります。

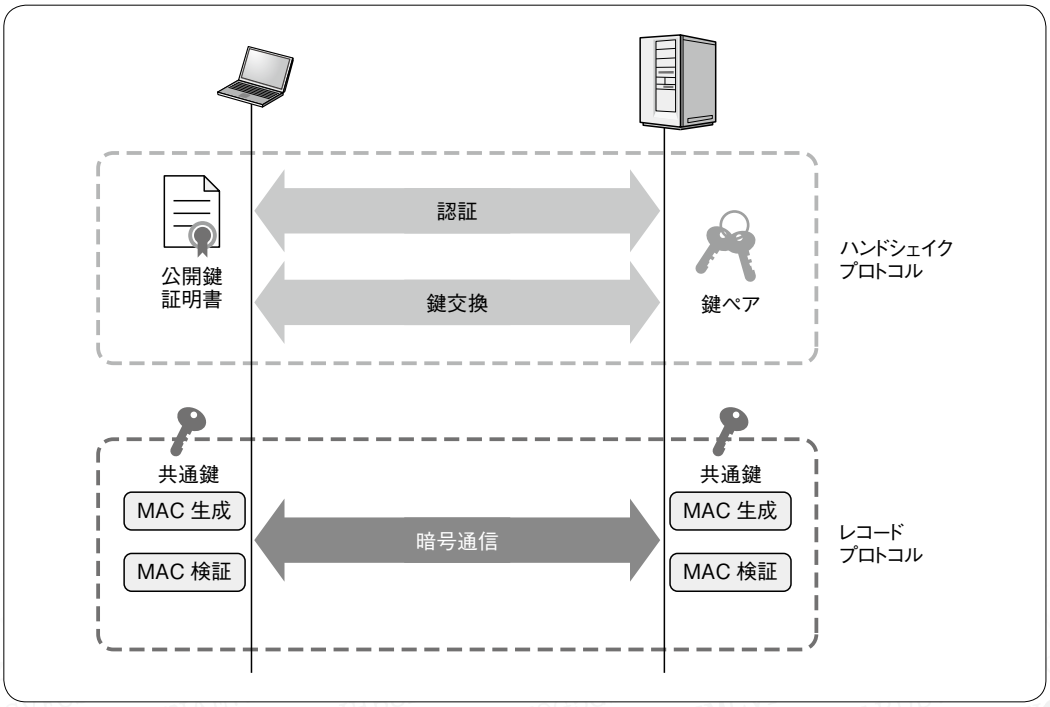
a)は、処理機会が通信開始時などに限られるため、公開鍵暗号で処理に若干時間がかかるとしてもトータルでの性能には影響しにくいです。

b)は、通信確立後は常時処理されるため、処理速度の速い共通鍵暗号やハッシュアルゴリズムを用いて処理します。

このように、認証から改ざん検知までの安全な通信のための一連の要件が公開鍵暗号と共通鍵暗号(とハッシュアルゴリズム)を組み合わせてとても合理的に設計されており、SSL/TLSではそれぞれハンドシェイクプロトコル、レコードプロトコルとして実装されています(詳しくは2章で解説します)。

とはいえ何事も準備は必要で、Webサイトは事前に公開鍵と私有鍵を用意しておく必要があります。一般にはこれは認証局からのサーバ証明書発行という形で実現されます。SD

▼図5　公開鍵暗号と共通鍵暗号の役割分担



第2章 SSL/TLSと暗号スイートを理解しよう

Author セコム㈱IS研究所　島岡 政基（しまおか まさき）、伊藤 忠彦（いとう ただひこ）、国井 裕樹（くにい ひろき）

SSLとTLS

SSL（Secure Socket Layer）は、1995年ごろにNetscape Communications社（以下、Netscape社）によって開発されました。1995年ごろとしたのは、最初の仕様のはずのSSL 1.0は非公開で、SSL 2.0が1995年に公開されたからです。しかしすぐに脆弱性が発見され、翌1996年にはSSL 3.0が公開されました。その後、このSSL 3.0はIETF（コラム「IETF」参照）で1999年にTLS（Transport Layer Security）1.0として標準化されました。このあたりの歴史的な話は、本章の後半で詳しく解説します。

なお、本章以降ではTLS 1.2をベースに解説をしていきますので、SSL/TLSという表記は両者を包括して言及するときのみとし、基本的には単にTLSと表記することにします。

BEASTやPOODLEなどSSL/TLSプロトコルに対する一連の攻撃は、単に脆弱性に対してパッチを当てればよいという話ではなく、暗号スイート（暗号の組み合わせ、後述）を適切に選択する必要性があることをあらためて認識させる事象となりました。しかし暗号スイートの適切な選択、とくにこのような高度な暗号技術への攻撃に対して適宜対応する形で選択することは、暗号スイートを適切に理解していないとできません。また昨今の多様な攻撃手法の登場に対してどの暗号スイートを選択すべきか、というのは暗号技術者の間でさえ議論が分かれるところでもあります。これについては第3章であらためて述べます。

それに先立って本章では、この暗号スイートについて少しでも理解を深めていただけるように、暗号スイートの扱いにフォーカスを絞る形でTLS 1.2を解説していきます。

したがって、これからTLSプロトコルを実装してみようという方向けの逐次解説になっていないこと、また典型的なサーバ認証のシーケンスに限った解説になることを、ご了承ください。

クライアント認証のシーケンスも含め逐次解説については、章末に参考文献を示しておきます。残念ながら発行時期が古いままのものも一部ありますが、まとまった資料としてはとても参考になります。実装してみたい方やクライアント認証での暗号スイートの扱いに関心のある方は、本記事と併せて目を通して（そしてぜひ手を動かして）みてください。

Column IETF

IETF（Internet Engineering Task Force；インターネット技術タスクフォース）はインターネット技術の標準化を推進する団体です。公式ドキュメントとしてRFC（Request For Comments）を無償で公開し、TLSをはじめ多くの技術を標準化してきました。参加者は実装者に近い立場の人が多いのが特徴で、PRISM事件などタイムリーな話題を扱うことでも有名です。

プロトコルの概要と通信の流れ

プロトコルの特徴

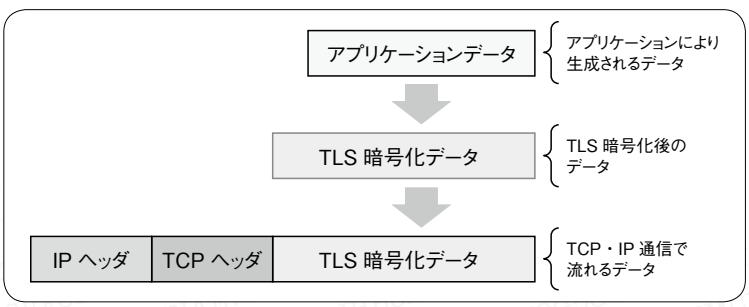
TLSなどのセキュリティプロトコルを用いない一般的なTCP通信では、**図1**のペイロード部分であるアプリケーションデータは平文でやりとりされます。そのため通信情報を閲覧できる人であれば、誰でもアプリケーションデータの盗聴ができてしまいます。また、通信経路上の中継者は、容易にヘッダ情報やアプリケーションデータの書き換えが可能であり、通信相手の変更や通信内容の改ざんができます。インターネットの通信では、中継者は必ずしも善人とは限りません。しかし、TLSを使うことで、盗聴・なりすまし・改ざん(不正利用)を防ぐことができます。

TLSは、OSI参照モデル7階層のうち、セッション層(第5層)に位置するセキュリティプロトコルとなり、第1章で述べた3つの要件であるデータの秘匿・データの改ざん検知・サーバ(場合によりクライアント)認証を、TCP通信において提供します。また、UDP通信に対し前述3要件を実現するプロトコルとしては、DTLS(Datagram TLS)がRFC 6347として公開されています。

TLSの利点としてアプリケーションプロトコル独立である、という点が挙げられます。典型的なアプリケーションプロトコルはもちろんHTTPですが、メールの通信プロトコルであるPOP3やIMAP、またディレクトリプロトコルのLDAPなどさまざまなアプリケーションプロトコルと組み合わせることができます。このため、アプリケーション開発者はTLSを利用することで、アプリケーションごとに前述3要件を実装する必要がなくなり、アプリケーション機能の開発に専念できるようになります。

次節で詳しく解説しますが、TLSは共通鍵暗号アルゴリズム、公開鍵暗号アルゴリズム、鍵交換アルゴリズム、ハッシュアルゴリズムなど実に多様な暗号技術によって構成されており、それぞれにおいて各種アルゴリズムに対応できる拡張性の高い設計になっています。

▼図1　TLSによる暗号化



プロトコルの概要

TLSのプロトコルは、レコードプロトコルとハンドシェイクプロトコルに大別されます。ハンドシェイクプロトコルでセキュア通信に必要な準備を行い、セキュア通信確立後はレコードプロトコルを用いてアプリケーションデータを送受信します。

ハンドシェイクプロトコル

ここでは、ハンドシェイクの中で暗号スイートがどのように使われるのか、**図2**に基づいて解説します。

・ClientHello：

TLS通信は、クライアントからサーバに対する通信要求によって開始されます。このときにクライアントは、自分が利用可能なTLSのプロトコルバージョンや暗号スイートの優先度つきリストを同時に送信します。

・ServerHello：

`ClientHello`を受信したサーバは、クライアントから提示されたプロトコルバージョンや暗号ス

イートのリストに基づいて、サーバが利用可能な範囲で最も最新のプロトコルバージョン、最も優先度の高い暗号スイートを返信します。

・Certificate：

続いてサーバは、自身のサーバ証明書をクライアントに送信します[注1]。必要に応じてサーバ証明書を発行した上位認証局の証明書なども一緒に送信できます。このサーバ証明書は、クライアントが接続先サーバの真正性を確認したり、そのあとの鍵交換に必要なパラメータを参照するために利用します。送信する証明書や、そこに含まれる鍵交換パラメータなどは、当然先にServerHelloで送信した暗号スイートに対応している必要があります。

・ServerKeyExchange：

続いてサーバは、鍵交換に必要なパラメータを送信できます。前述のCertificateで送信した証明書によって事足りる場合は、このフローを省略できます。送信するパラメータの内容は、ServerHelloで選択した暗号スイートに含まれる鍵交換アルゴリズムによって異なります。

・ServerHelloDone：

CertificateまたはServerKeyExchangeの送信に成功すると[注2]、サーバ側からハンドシェイクに必要な情報はすべて送信できたことになるので、その意思表示としてServerHelloDoneを返します。

・ClientKeyExchange：

ServerHelloDoneの受信に成功すると、クライアントは鍵交換に必要な情報をサーバ側からすべて受け取ったことになります。クライアントはServerHelloで返された鍵交換アルゴリズムとCertificateまたはServerKeyExchangeの内容に基づいて、クライアント側の鍵交換パラメータを生成し、サーバに送信します。

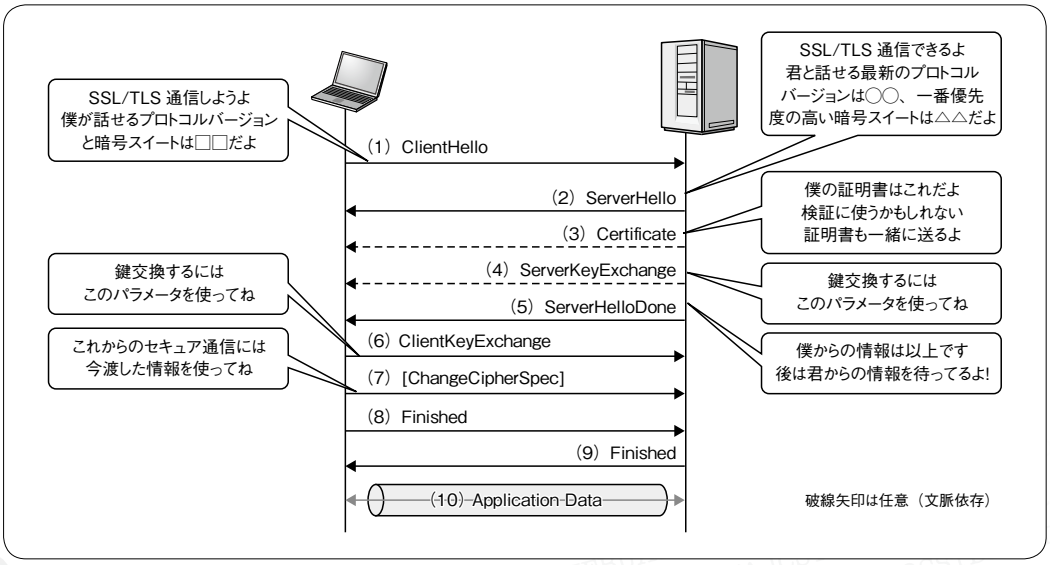
・ChangeCipherSpec：

レコードプロトコルに用いる暗号スイートを

注1）ServerHelloDoneで指定する暗号スイートによってはCertificateを省略できる（つまりサーバ認証を行わない）場合があります。ただし、およそ一般的な事例ではないので本記事では省略します。

注2）これはあくまで典型的なサーバ認証の話で、クライアント認証や再認証など、ほかのシーケンスではServerHelloDoneを送信するための条件は異なります。詳しくはRFC 5246などでご確認ください。

▼図2　TLSの典型的なシーケンス



変更する通知です[注3]。この通知は、ここまでのハンドシェイクプロトコルで決定した暗号スイートとその鍵交換パラメータが、以降のレコードプロトコルに適用されることを意味します。

・**Finished**：

`ChangeCipherSpec`まで送信すれば[注4]、クライアント側から送信すべきハンドシェイクに必要な情報はすべて送信できたことになるので、その意思表示として`Finished`を返します。

レコードプロトコル

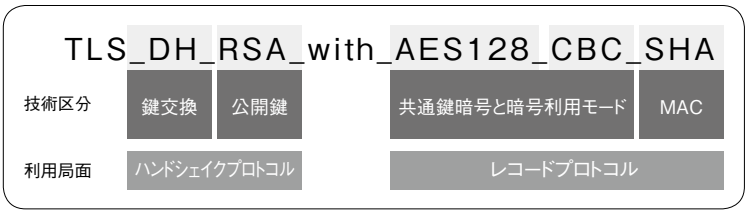
ハンドシェイクが終わると、レコードプロトコルを用いて、アプリケーションデータを暗号化し、送受信できるようになります。アプリケーションデータは再送攻撃や並べ替え攻撃を受けないよう、後述する**図7**に示すようにシーケンス番号を含むTCPヘッダとともにMACタグを計算します[注5]。その後アプリケーションデータとMACタグ、必要に応じてパディングがハンドシェイクから得られた共通鍵をもとに暗号化されます。

暗号スイートとその構成

図3はTLSハンドシェイクで提示される「暗号スイート」の一例です。TLSではこのように、鍵交換、公開鍵暗号技術、共通鍵暗号、暗号利用モード、MACを組み合わせることで、安全な通信を実現しています。また、**図3**のように、各技術区分から1つずつ選んでひとまとめにしたものは暗号スイートと呼ばれます。IETFでは多くの暗号スイートを定義しており、その数は300件を超えます（これらについては後述します）。

ハンドシェイクプロトコルでは、サーバとクライアントは、お互いが利用可能な暗号スイートのリストを提示、参照し、利用するスイートを決定します。とくにサーバのデフォルト設定では、多様な実装環境に対応するため、多くの

▼図3　暗号スイートとその構成



注3）`ChangeCipherSpec`は、厳密にはハンドシェイクプロトコルではなくレコードプロトコルなのですが、ハンドシェイクにおいても流用されます。
注4）これも同様にサーバ認証に限った話で、ほかのシーケンスにおける終了条件はそれぞれ異なります。詳しくはRFC 5246などでご確認ください。
注5）後述するGCMモードなど一部の暗号利用モードでは、これ以外の実装方法もあり得ます。

Column 暗号技術は軍事技術？

暗号技術は古くから軍事技術として利用されており、現代においても軍事技術の一種としてみなされています。米国においても、古くはCOCOM[注6]、現在はワッセナーアレンジメント[注7]で暗号技術の輸出が規制されています 。インターネットの普及とともに、暗号技術も民間に行き渡り、規制も大幅に緩和され、形骸化していますが、規制がなくなったわけではありません[注8]。

こうした歴史的背景から、一部の暗号アプリケーションには、規制対象国への輸出用の、弱い暗号を利用させるしくみが残っています。TLSにおいても、export-gradeと呼ばれる「弱い」暗号アルゴリズムを利用するしくみが存在します。しかし、近年export-gradeを狙った攻撃が頻発していることもあり、IETFでもexport-gradeを排除する方向に進んでいます。

注6）Coordinating Committee for Export Control（対共産圏輸出統制委員会）の略。
注7）通常兵器の輸出管理に関する、国際的な申し合わせ。
注8）実際に2014年には、Wind River Systems社が、政府向けの高度な暗号ソフトウェアを特定の国に許可なく輸出したとして、米国商務省から制裁金を言い渡されています。

暗号スイートに対応する傾向にあります。たとえば、OpenSSL 1.0.2c + Apache 2.4.12では、初期設定で97件の暗号スイートを提示します。

その中には、古い暗号スイートや、十分に安全でない暗号スイートもあります。攻撃者はより「弱い」暗号スイートを狙うことが予想されるため、安全でない暗号スイートは提示しないようにすべきでしょう。ところが、300種類以上の中から適切なスイートのリストを選ぶことは、容易ではありません。

本章では、各技術区分の概観を解説したあと、各技術区分の主要なアルゴリズムを紹介します。そして、第3章で、適切な暗号スイートの選び方について解説します。

利用される暗号技術の解説

TLSの通信は多くの暗号技術を組み合わせて行いますが、本節では、認証アルゴリズム、鍵交換アルゴリズム、共通鍵暗号アルゴリズムと利用モード、MACアルゴリズム、の4つの技術区分に分けて解説します。

認証アルゴリズムと公開鍵暗号

クライアントがサーバと安全な通信を実現するには、通信相手が意図するサーバかどうか、つまり第三者によるなりすましや意図しないサーバでないことを確認すること(認証)が必要となります。TLSにおいては、認証方法の典型例として公開鍵暗号に基づいたサーバ認証が挙げられます。これは公開鍵暗号に基づく電子署名のしくみを応用したものですので、まず電子署名のしくみについて解説します。

電子署名は、**図4**に示すようにちょうど公開鍵暗号の公開鍵と私有鍵の役割を入れ替える形で実現されるもので、RSA、DSSやECDSAなどがあります。あるデータに署名できるのは私有鍵を持つ署名者のみで、一方でその署名が正しい(改ざんされていない)ことは公開鍵があれば誰でも検証できます。

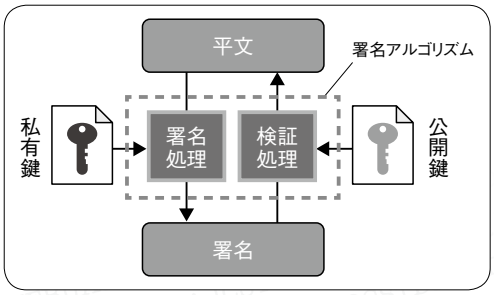
認証では、クライアントがnonceと呼ばれる乱数を生成してサーバに送信し、サーバがこれに私有鍵で署名して返します。クライアントはサーバ証明書から取得した公開鍵でこの署名を検証し、検証に成功すれば、今通信している相手がサーバ証明書に記載されたサーバと同一だと判断します(すなわちサーバの認証に成功したとみなします)。

ところでサーバが送付する証明書が偽物でないとの保証はありません。通信のしくみ上、中間者攻撃(**図5**下部)の可能性は排除できません。中間者攻撃を防ぐには、**図5**に示すように、今通信している相手が差し出す証明書を無闇に信用するのではなく、偽物に対して証明書を発行することのないような認証局をクライアントがあらかじめ信頼しておくことによって対策できます。具体的には、ハンドシェイクにおけるCertificateで送られてきたサーバ証明書が信頼している認証局の公開鍵で検証できれば、そのサーバは、なりすましなどではない、意図したサーバであるとみなすことができます。

このクライアントからあらかじめ信頼された認証局はパブリック認証局とも呼ばれ、みなさんが使っているOSやブラウザなどに数百近いパブリック認証局があらかじめインストールされています。暗号技術の危殆化、私有鍵の漏えい、攻撃者による認証局の不正利用などが起きない限り、これらのサーバ証明書を安心して使うことができます[注9]。

注9) ちなみに最近ではsuperfishやWERDLODなどのように認証局ではなくブラウザそのものを改ざんして、不正な認証局を勝手に信頼させる手法も出てきました。

▼図4 電子署名の概念



● 鍵交換アルゴリズム

鍵交換アルゴリズムは大きく、鍵配送型のアルゴリズムと、DH(Diffie-Hellman)型のアルゴリズムに分けられます。

鍵配送型は、送信者が共通鍵を作り、暗号化して受信者に送る方式です。

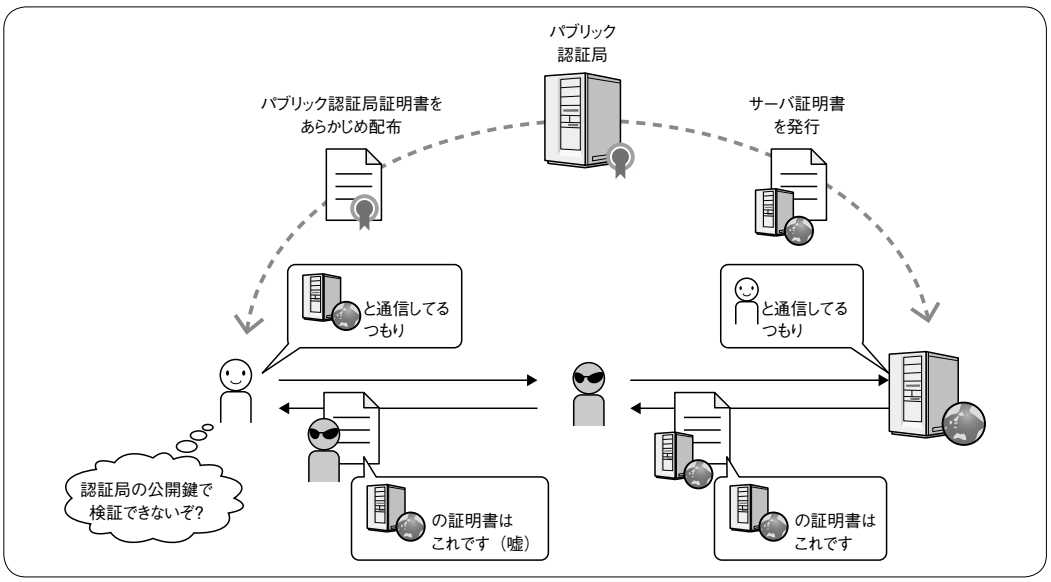
DH型鍵交換は、サーバとクライアントがお互いの秘密を持ち寄り、共同して鍵を計算する方式です。DH型鍵交換でクライアントとサーバは、**図6**のように、秘密の情報(私有鍵のXc又はXs)と公開鍵(Yc又はYs)を持ち、自分の私有鍵と相手の公開鍵と事前に共有した公開情報を合わせることで、同じ鍵を共有できます。本章では、DH鍵交換における秘密情報を「DH私有鍵」、事前共有した公開情報と公開鍵Yc又はYsを合わせて「DH公開鍵」と呼びます。

DH型の鍵共有には、DH公開鍵を電子証明書に記載するDH鍵交換と、都度違うDH公開鍵を使うDHE鍵交換があります。また、DH(E)鍵交換を、楕円曲線上の性質を利用して行う方式はECDH(E)鍵交換と呼ばれます。各方式の詳細については、後述します。

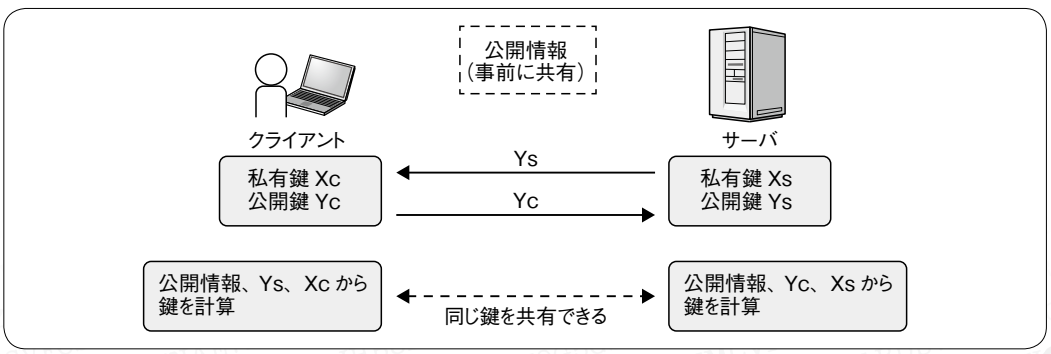
● 共通鍵暗号

共通鍵暗号の利用では、第1章の**図3**で示したように送信者と受信者が同一の鍵(共通鍵)を利用します。送信者は、共通鍵を利用し、送信メッセージを暗号化することで暗号文を作成・

▼図5　中間者攻撃対策としての認証局



▼図6　DH鍵型交換



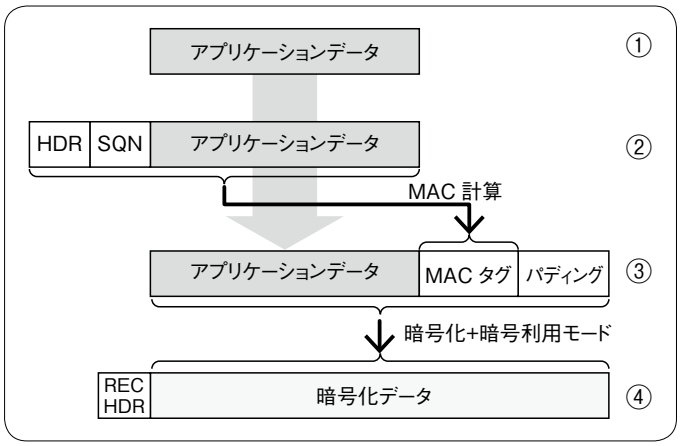
送信します。受信者は共通鍵を利用し、受け取った暗号文を元のメッセージに復号します。

共通鍵暗号は、ブロック暗号とストリーム暗号に分けることができます。ブロック暗号では、暗号化を行ううえでパディングや暗号利用モードが必要なため、ここではそのしくみについて解説します。

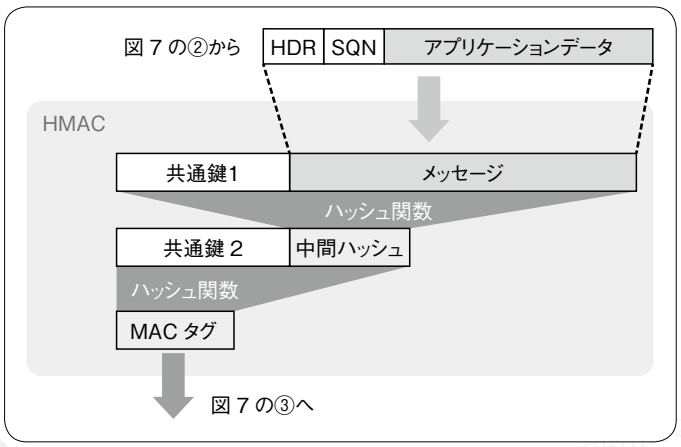
ブロック暗号では決まったbit数(ブロック長)のメッセージを入力として受け取り、決まったbit数の暗号文を出力します。そのため、ブロック暗号で暗号化するときは、入力(**図7**③)をブロック長の倍数にしなければいけません。アプリケーションデータは必ずしもブロック長の倍数とならないため、パディング(詰め物)を行います(**図7**)。具体的には、次節で述べるMACを付与したあと、ブロック長の倍数になるまで、決まった手順で埋めます。

なお、ストリーム暗号は1bitごとに暗号化するため、パディングや暗号利用モードの指定が必要ありません。

▼図7　ブロック暗号による暗号化とMAC



▼図8　HMAC



(共通鍵暗号の)暗号利用モード

ブロック暗号では、何も工夫せずに各ブロックを同一の鍵で暗号化すると、同じ入力に対応する暗号文が同じものになってしまい、復号まではできなくても鍵や元の平文などの一部の情報を予測しやすくなってしまう問題があります。アプリケーションデータに含まれるHTTPヘッダなどはこうした攻撃に悪用される可能性があります。

そこで、同一の入力・同一の鍵であっても、異なる暗号文になるようないくつかの暗号利用モードというものが作られました。その種類については後述します。

MACアルゴリズム

MACアルゴリズムは、メッセージの改ざんを検知するために使われます。ここでは、HMACに絞り解説します。HMACは内部的に**図8**のようなしくみで、ハッシュアルゴリズムと共通鍵を使い、MACタグを計算する方式です。利用する共通鍵1および2は、ハンドシェイク時に共有するパラメータから決まった手順で計算されます。そのため、サーバもクライアントも、これらの鍵は管理する必要がありません。

TLSでのMACタグは、**図7**の②のように、ヘッダ(HDR)、

シーケンス番号(SQN)、アプリケーションデータを合わせてHMACの入力メッセージとすることで計算されます。

暗号スイートにおけるHMACアルゴリズムの指定は、ハッシュアルゴリズムの名前を指定することにより行うことができます。各種ハッシュアルゴリズムの詳細については、後述します。

主要なアルゴリズムの特徴

デフォルトの暗号スイート(のリスト)は相互接続性を重視するあまり、安全でない暗号アルゴリズムも含まれていることもあると「暗号スイートとその構成」冒頭で指摘しました。これを安全性にも配慮したリストにするにも、暗号スイートが多過ぎて、適切なスイートを選択するのが難しいという課題があります。TLSでは**表1**に挙げられているアルゴリズムが記載されていますが、各暗号スイートの特徴を理解するため、本節では、これらのアルゴリズムのうち主要なアルゴリズムとその特徴を解説します。

認証アルゴリズム

認証アルゴリズムに、RSA、DSS、ECDSAなどと書かれていた場合は、それらの公開鍵暗号アルゴリズムを利用し、認証を行います。サーバの証明書に記述されている公開鍵(またはDH公開鍵)の種類により、使える認証アルゴリズムが異なる点は注意が必要です。

鍵交換アルゴリズム

表1に示す鍵交換アルゴリズムの記載なしにTLS_RSA_*と暗号スイートに書いてあった場合、クライアントは乱数を作り、サーバの公開鍵で暗号化し、送信します。両者はその乱数から共通鍵を作成します。ここで、公開鍵暗号は暗号化の用途で利用されます。

DH_*と書いてあった場合、クライアントとサーバはDiffie-Hellman鍵交換アルゴリズムを利用し、共通鍵を生成します。ここで、サーバの電子証明書に記載されているDH公開鍵が鍵交換に利用されます(DH公開鍵が電子証明書に記述されていなければ、利用することはできません)。サーバが証明書を変更しない限り、利用されるDH公開鍵は常に同じものとなります。

DHE_*と書いてあった場合、クライアントとサーバはEphemeral Diffie-Hellman(DHE)鍵交換アルゴリズムを利用し、共通鍵を生成します。単なるDHアルゴリズムと異なり、サーバはDH公開鍵とDH私有鍵を(使い捨てにし)鍵交換のたびに新たに作りなおします。そして、サーバの電子証明書に記載されている公開鍵に対応する私有鍵を利用し、DH公開鍵に署名を行います。サーバの証明書に記載されている公開鍵がRSAの場合はDHE_RSA、DSSの場合はDHE_DSSとなります。DHEでは私有鍵と公開鍵を使い捨てにすることで、Perfect Forward Secrecyという性質(「PFS」の節で後述)を持たせることができ、サーバ証明書の私有鍵が漏えいしたときの脅威を低減できます。

▼表1　TLS 1.2の暗号スイートがサポートするアルゴリズム

鍵交換	認証	共通鍵暗号と暗号利用モード	MAC
ECDH	RSA{|-PSK}	AES_{128|256}_{CBC|GCM|CCM}	SHA
ECDHE	DSS	3DES_EDE{ |_CBC}	SHA-256
DH	ECDSA	DES_CBC	SHA-384
DHE	KEB5	RC4_{40|128}	MD5
SRP	PSK	CAMELLIA_{128|256}_CBC	NULL
NULL	anon	IDEA_CBC	
DH_anon_export	NULL	ARIA_256_{CBC|GCM}	
	RSA_EXPORT	SEED{CBC}	
	DSS_EXPORT	RC2_CBC_40	
	KBR5_EXPORT	NULL	

ECDH_RSAは、DH_RSAと似ていますが、サーバの証明書に、DH公開鍵の代わりにECDH公開鍵が記述されています。

ECDH_ECDSAはDH_DSSと似ていますが、DH鍵交換のみでなく、電子証明書の検証にも楕円曲線を使うアルゴリズムです。

ECDHE_*は、ECDH_*に似ていますが、DHE_*と同じように、EphemeralなDiffie-Hellman型鍵交換を行い、ECDH公開鍵を使い捨てにするアルゴリズムです。

*DHE_*鍵交換において、鍵交換に利用するDH公開鍵が短い(弱い)ために、他の暗号鍵に比べ、DH公開鍵だけが大幅に弱くなることが問題になることがあります。多くのアルゴリズム(*DH_*)では、認証局が責任を持って発行した電子証明書に記載されているDH公開鍵が使われるため、「弱い」鍵を使ってしまう問題は起こりにくいです。しかしながら、DHE、ECDHEでは、使う鍵にとくに制約がなく、実装により鍵の長さが異なります。つまり、実装によっては、弱い(鍵の長さが短い)DH鍵が生成されてしまうため、注意が必要となります[注10]。

共通鍵暗号

AESは、現在最も広く使われている共通鍵暗号アルゴリズムです。ブロック長(入出力)は128bit、鍵は128bitと256bitから選ぶことができます。ブロック暗号アルゴリズムですので、暗号利用モードを指定して使う必要があります。

DESは、1970年代から利用されているブロック暗号アルゴリズムです。ブロック長は64bit、鍵は56bitとなります。現在においてDESで暗号化された文章は容易に解読できるためTLS 1.0以降では利用できません。

3DESは、異なる鍵を利用したDESの処理を3回行います。3DESのブロック長は64bit、鍵は168bit(3key-3DES)または112bit(2key-3DES)となります。2keyと3keyの暗号の強さの比較については、コラム「ビットセキュリティ」をご参照ください。現在において3DESの安全性は随分低下していますが、TLSで選択できます。暗号利用モードを指定して使う必要があります。

RC4は、ストリーム暗号アルゴリズムの一種です。ストリーム暗号では、ブロック長の制約なく暗号化できます。RC4では128bitの暗号鍵を使用します。3DES同様、十分な安全性は確保できませんが、最新のTLS 1.2でも選択できます[注11]。

暗号利用モード

暗号利用モードには多くの種類がありますが、本節では広く使われているCBCモードと、TLS 1.2で新たに実装されたGCM、CCMモードを解説します。近年、暗号利用モードのしくみを利用した攻撃が多数発見されましたが、これらの攻撃の詳細については第3章で解説します。

CBCモードは、前のブロックを暗号化した出力を、次に暗号化するブロックの文字列とXORし、その値を暗号関数の入力とする暗号アルゴリズムです。CBCモードでは暗号化処理の並列化ができない反面、復号処理の並列化は容易です。

GCMモード、CCMモードは、暗号化機能と改ざん防止機能を統合する暗号利用モードです[注12]。GCM、CCMモードを利用すると、MACアルゴリズムを使わずに改ざんを防げます。GCM、CCMモードはTLS 1.2で実装されたアルゴリズムですので、TLS 1.1以下では利用できません。また、GCMは暗号化・復号ともに並列化が可能です。

MACアルゴリズム

本節では、HMAC内部で呼び出されるハッシュアルゴリズムについて解説します。

注10) たとえばDHE_RSAでは、DH公開鍵、DH私有鍵の生成方法は実装依存です。そのため、セキュリティ強度(コラム「ビットセキュリティ」参照)を確認するためには、実装まで調べる必要があります。

注11) ただし、これはRFC 5246での仕様であり、その後TLS 1.0からRC4の使用を禁止するRFC 7465が発行されています。

注12) これらの暗号利用モードを利用する共通鍵暗号アルゴリズムは、とくにAEAD (Authenticated Encryption with Associated Data)と呼ばれます。

ハッシュアルゴリズムは、電子署名やMAC計算に利用され、どんな長さの文字列を入力しても、決まった長さの文字列(ハッシュ値)を出力する関数です。また、少しでも入力データが異なればハッシュ値は大きく変わり、特定のハッシュ値を出力するように入力データを調整することは難しいです。このような性質を利用して、MACアルゴリズムでは暗号化データの改ざんを検知しています。

代表的なハッシュアルゴリズムには、MD5、SHA-1、SHA-2(SHA-256など)などがあります。

MACアルゴリズムにMD5を指定すると、HMACでハッシュアルゴリズムのMD5を呼び出します。MD5は128bitの出力を持つMACアルゴリズムです。古く弱いアルゴリズムですので、利用するべきではありません。

SHAと指定すると、SHA-1をHMACで呼び出すことを示しており、160bitの出力を持つMACアルゴリズムです。SHA-2に比べ強度の面で劣ります。

SHA-256、SHA-384と指定すると、各ハッシュアルゴリズムをHMACで呼び出すことを示しています。各数字は、出力bit数を表しています。

PFS

多くの暗号方式は、私有鍵が漏れないように運用することが求められます。しかしながら、管理ミスや、NSAなどによる諜報活動、内外からの攻撃などにより、私有鍵が実際に漏えいした例もあり、そのリスクは無視できません。DHEまたはECDHE以外のTLS通信では、私有鍵が漏れた場合は、(いくつかの前提は必要ですがたとえばPRISM事件のように通信パケットがキャプチャされていると)原理的に過去の暗号通信が復号可能となります。

私有鍵が漏えいする状況においても、以前の暗号通信を保護できるしくみを持つことをPFS(Perfect Forward Secrecy)と呼びます。

PFSの大まかなアイデアは「DH私有鍵(とDH公開鍵)を使い捨てにしてしまおう」というものです。秘密もメモリ上から破棄してしまえば漏えいすることはなく、証明書の私有鍵から暗号化用の共通鍵を復元することもできなくなります。つまり、通信パケットをすべてキャプチャされている状況で私有鍵が漏えいしても、その暗号通信の内容が解読されることはありません。

DHEおよびECDHEでは、**図9**のようにDH私有鍵を使い捨てにすることで、仮に電子証明書の私有鍵が漏えいしても、暗号化用の共通鍵が計算できないようなしくみを実現しています。

注意が必要なのは、本章の「認証アルゴリズムと公開鍵暗号」のところで解説したように、サーバ証明書の形式によっては、使えないアルゴリズムがある点です。

SSL/TLSのバージョン

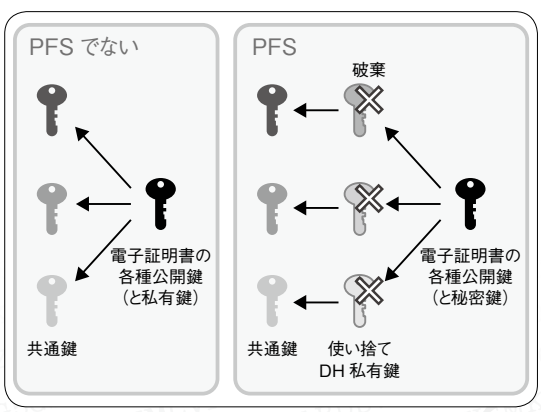
各バージョンの差分や策定経緯

ここまで多くのアルゴリズムや、それを組み合わせた暗号スイートについて紹介してきましたが、SSL/TLSがどのような歴史的経緯をふまえてアップデートされていったかを見ていきたいと思います。

・SSL 1.0

冒頭でも触れましたが、SSLはNetscape社が

▼図9 PFS



Secure Socket Layerとして1994年に設計しました。初期バージョンの1.0では秘匿用にRC4での暗号化を用いましたが、メッセージの整合性をCRC(Cyclic Redundancy Check)でしか担保していないため、メッセージの改ざんが可能なものでした。社内でのレビュー時にこのような脆弱性がいくつか発見されたため、SSL 1.0が実装されることはありませんでした。

・SSL 2.0

SSL 1.0での脆弱性を修正し、Netscape社はSSL 2.0を1994年後期に発表しました。メッセージの改ざん防止にハッシュアルゴリズム(MD5)が導入され、1995年には「The SSL Protocol」としてIETFに投稿されました[注13]。Netscape社はこの時SSLの特許を米国で取得していますが、これを無償で解放しています。

残念ながらSSL 2.0にも、いくつかの脆弱性が存在していました。ハンドシェイクの改ざん検知を行っていないことが原因となり、中間者がハンドシェイクを改ざんし、輸出グレードの暗号スイートを強制的に選択させる脆弱性などがありました[注14]。

また当初は安全だと思われていたMD5もその後危殆化し、安全性が低下してしまいました。

このような事実がありながら、つい最近まで、

注13) SSL2.0はその後RFCにはならず、草稿であるInternet Draftのまま終わりました。
注14) この結果、ビットセキュリティの低い暗号が使われ攻撃者の暗号解読が容易になってしまいます。

Column ビットセキュリティ

本章で多くの共通鍵暗号、公開鍵暗号、MAC、鍵交換アルゴリズムを紹介してきました。基本的に同じ暗号アルゴリズムであれば、鍵が長いほど安全になります。しかしながら、異なるアルゴリズムでは、単純に鍵の長さだけで安全性を比較できません[注15]。たとえば、3DESを168bitの鍵で利用するより、AESを128bitの鍵で利用した方が安全ですし、AESを128bitの鍵で利用するのと同じ程度の安全性を、RSAで実現するためには、3,072bitの鍵が必要となります。

そこで、各種アルゴリズムがどの程度の「強度」を持っているかを判断する指標として、ビットセキュリティという概念があります。**表2**は代表的な暗号アルゴリズムと、それらのビットセキュリティの一覧です。暗号通信を攻撃する人間は、一番強度が低い場所を狙うことが予想されます。そのため、すべての区分で一定(たとえば128)以上のビットセキュリティを持つことが推奨されます。

注15) ハッシュアルゴリズムは、HMACを利用した場合の強度であり、電子証明書で利用する場合は、強度が大幅に下がります(具体的には、SHA-1は電子証明書で利用する場合はビットセキュリティが80まで下がるため、利用は推奨されません)。

▼表2 各セキュリティ強度に対応した暗号アルゴリズム(NIST SP800-57_part1_ rev3より)

セキュリティ強度		80	112	128	それ以上の強度
ハンドシェイク	認証	RSA1024 ECDSA160	RSA2048 ECDSA224	RSA3072 ECDSA256	RSA4096など ECDSA384など
	署名検証時のハッシュ(参考情報)	SHA-1	SHA-224	SHA-256	SHA-384、 SHA-512など
	鍵交換	1024bitの DH公開鍵	2048bitの DH公開鍵	3072bitの DH公開鍵	より長い DH公開鍵
		160bitの DH私有鍵	224bitの DH私有鍵	256bitの DH私有鍵	より長い DH私有鍵
レコード	秘匿	3DES(2key)	3DES(3key)	AES128	AES256
	改ざん検知			SHA-1(HMAC)	SHA-1(HMAC) SHA-256(HMAC)
使用期限について		すでに非推奨	2030年まで	2031年以降の利用	

暗に安全ではないとわかりつつもSSL 2.0は使用されてきましたが、2011年にはついにIETFからRFC 6176「Prohibiting Secure Sockets Layer (SSL) Version 2.0」が公開され、SSL 2.0の使用は禁止されました。

・SSL 3.0

1995年にSSL 2.0の脆弱性を修正した、SSL 3.0がNetscape社からリリースされました。機能も拡張され、証明書については中間認証局にも対応するなど、さまざまな改良が施されました。また著名な暗号研究者であるTaher Elgamalらを迎えセキュリティをより堅牢なものとしました。

すでに実装され広まっていたSSL 2.0と混在することとなるため互換性を持たせ、ハンドシェイクの際にSSL 3.0で通信するかSSL 2.0で通信するかを決定するしくみが導入されました。しかし、この互換性を悪用してクライアントがSSL 3.0で通信を開始するためのハンドシェイクClientHelloを改ざんできる脆弱性が存在しました。中間者によってClientHelloがSSL 2.0にダウングレードさせて開始するように改ざんされると、サーバはSSL 2.0で通信を開始します。クライアントもサーバの応答がSSL 2.0を指定しているため、これを受容し、低いセキュリティで通信を行ってしまう問題がありました。

これを回避するため各ブラウザはSSL 3.0が普及したタイミングでSSL 2.0をデフォルトで無効化する対策を施しました。それにともなって、これまでSSL 2.0しかサポートしていなかったサーバもSSL 3.0で接続できるようにアップデートされていきました。

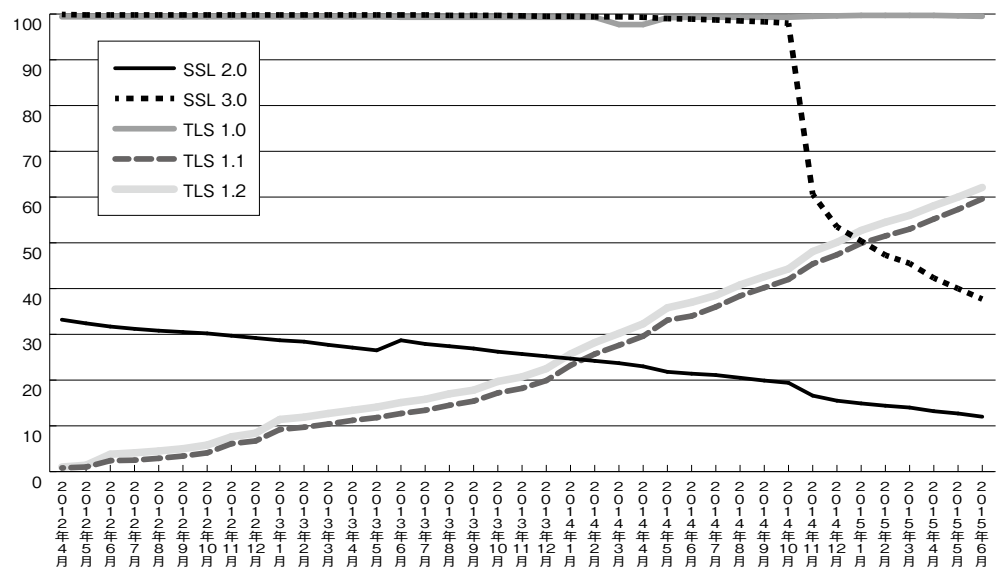
SSL 3.0は公表されてから20年近く多くのブラウザ／サーバでサポートされていましたが、2014年に脆弱性（POODLE）が報告され、SSL 3.0をサポート外とする動きが急増しています[注16]（**図10**）。

・TLS 1.0

これまでNetscape社から公開されていたSSLでしたが、世界中で広く利用されていることから、IETFでの標準化も開始されました。そうして1999年にIETFから公開されたのが

注16） 2015年6月にSSL 3.0を廃止するRFC 7568「Deprecating Secure Sockets Layer Version 3.0」が発行されました。

▼図10　サーバ側のSSLサポートシェア[注17]



注17） SSL Pulseが公開しているデータに基づいています。

RFC 2246 Transport Layer Security（TLS）のバージョン1.0です。

SSLから名称が変更されましたが、SSL 3.0をもとに策定されているため両者に大きな差はありません。差分としてはセキュリティ向上のための更新が施されていたり、一部鍵交換アルゴリズムや暗号アルゴリズムの実装がSSL 3.0では任意だったものがTLS 1.0では必須になったことがあげられます。

TLS 1.0以降では前述のPOODLEの脆弱性に対応できることもあり、現在ではSSL 3.0からTLS 1.0（さらにそれ以降のバージョン）への移行が大きく進められています。ただしTLS 1.0はあくまで仕様上安全であるというもので、実装の問題でPOODLEの脆弱性を含むものも存在することに注意が必要です[注18]。

・TLS 1.1

TLS 1.0のセキュリティを向上させるアップデートとして2006年にIETFからRFC 4346 TLS 1.1が公開されました。

大きな変更としては以前のバージョンで問題が指摘されていたCBCモードの改善や、エラー処理・警告処理を解読のヒントにする攻撃者への対策などです[注19]。

また現在、さまざまな暗号製品で広く使われているAESはこのTLS 1.1からRFCに組み込まれました（TLS 1.0には2002年にAESが追加されています）。

・TLS 1.2

本原稿執筆時点で最新のバージョンであるTLS 1.2は2008年にIETFからRFC 5246として公開されました。当然ながら全バージョンでセキュリティは一番高く、ハッシュアルゴリズムのSHA-256や暗号利用モードGCM、CCMといった新しい技術や、さまざまなセキュリティ対策が取り入れられています。

・TLS 1.3

TLSの次期標準として1.3がIETFで提案・検討されています。この規格については本特集のAppendixを参照してください。

注18）たとえばプロトコルバージョンとしてはTLS 1.0を返すものの、ほかの実装はSSL 3.0のまま、というものがありました。

注19）BEASTやPOODLEの脆弱性ではこういった情報が暗号解読の手がかりとされています。

互換性のメリット・デメリット

ここまでSSL/TLSの各バージョンについて解説してきましたが、SSL/TLSは相互接続性を確保するため後方互換性を持つように設計されています。仮に互換性がない場合、TLS 1.2に対応しているサーバとSSL 3.0に対応しているクライアントではバージョンが異なるため通信ができません。しかしここでサーバの実装がSSL 3.0にも対応していれば、ハンドシェイクのClientHelloを受けてSSL 3.0で通信を開始できます。これによりユーザはSSL/TLSのバージョンを気にせずに通信ができます。

とはいえSSL 3.0の解説にもあったように、古いバージョン、とくにSSL 2.0、SSL 3.0での通信にはリスクも存在します。本来ならば最新バージョンであるTLS 1.2ですべての通信が行われるのが望ましいのですが、2000年前後にSSL 3.0までが実装され、アップデート機能を持たないクライアント製品が未だに使われています。これらの製品を無視してサーバ側がSSL 3.0での通信を一方的に禁止した場合、ユーザは突然接続ができなくなってしまいます。この問題を解決するためには製品をすべて新しいものに交換するほかなく、SSL/TLSの大きな課題となっています。SD

【参考文献】

・『マスタリングTCP/IP SSL/TLS編』
Eric Rescorla著、オーム社、2003年
・『PKI関連技術情報』、http://www.ipa.go.jp/security/pki/index.html、IPA
・『自堕落な技術者の日記』
http://blog.livedoor.jp/k_urushima/

第3章 脆弱性の分析から見えてくる安全なTLSサーバ設定

Author NTT セキュアプラットフォーム研究所　神田 雅透（かんだ まさゆき）
Author ㈱レピダム　林 達也（はやし たつや）注1

はじめに

TLSは、インターネット上のセキュリティプロトコルの1つというだけでなく、もはやビジネスには必要不可欠なツールです。それほど重要なTLSですが、ここ数年、「TLSに対する致命的な脆弱性」と報じられる攻撃が頻繁に発生しています。

そこで、本章前半では、致命的な脆弱性と報じられた攻撃がどのような脆弱性を使って実際の攻撃につながったのかを解説します。また、後半では、こういった攻撃に対する耐性を高め、TLSを安全に利用するためにサーバ設定で対処できることについて紹介します。

事件を振り返ろう——TLSに潜む脆弱性

脆弱性にはどのようなものがあるか

一言に脆弱性と言っても、その内容や発生場所はさまざまです。たとえば、TLSはセキュリティプロトコルの1つですので、ほかのセキュリティプロトコルと同様、次の3つの脆弱性はすぐに思い当たるでしょう。

・暗号解読
・秘密鍵（私有鍵やセッション鍵注2）の漏えい
・実装脆弱性

次は、TLSに特化した脆弱性です。TLSの特徴は相互接続を重視していることであり、事前に面識がないサーバとブラウザであっても、両者がその場でハンドシェイクを行うことで暗号通信できるようなしくみにしています。このことは、逆の見方をすれば、次のような脆弱性を想定する必要があります。

・中間者攻撃
・ブラウザのマルウェア感染
・フィッシングサイト誘導
・サーバ証明書不正利用

そもそも何が問題だったのか

ここでは、TLSの致命的な脆弱性と呼ばれるものの正体を見てみましょう。先の脆弱性の分類と照らし合わせると、**表1**のようになります。

▼表1　最近のTLSの脆弱性／事件に対する分類

脆弱性／事件	分類
OpenSSL Heartbleed	実装脆弱性
BEAST攻撃、POODLE攻撃など	マルウェア感染＋中間者攻撃
CCS Injection	実装脆弱性＋中間者攻撃
FREAK攻撃、Logjam攻撃など	暗号解読＋中間者攻撃
スノーデン事件、台湾国民電子証明書事件	秘密鍵の漏えい
MD5証明書偽造攻撃	暗号解読＋実装脆弱性
Diginotar事件	サーバ証明書不正利用 ＋フィッシングサイト誘導

OpenSSL Heartbleed

2014年4月に発覚したHeartbleedは、RFC 6520注3

注1）本章の執筆は、「OpenSSL CCS Injection」の項が林達也氏、それ以外の節・項は神田雅透氏によるものです（編注）。
注2）鍵交換でサーバとブラウザが共有する共通鍵のこと。
注3）https://tools.ietf.org/html/rfc6520

で規定されたTLS 1.2の拡張機能Heartbeatを実装したOpenSSLの実装脆弱性を利用した攻撃[注4]です。TLSそのものに対する攻撃ではないにもかかわらず、TLSサーバを構築する際によく使われるOpenSSLに問題があったため、TLSの脆弱性とか暗号ソフトウェアの脆弱性との見出しがついたニュースが流れたりもしました。

この攻撃を引き起こした脆弱性は極めて単純なものです。具体的には、RFC 6520では

・サーバはHeartbeatMessageのペイロードに書かれたデータをそのままフルコピーしてブラウザに送り返す
・ブラウザとサーバが送り合うデータ（HeartbeatMessage）は最大でも16.4KBを超えてはいけない
・ペイロードの長さがあまりにも大きい場合には、サーバはリクエストを破棄しなければならない

と決められていたにもかかわらず、OpenSSLでのHeartbeat実装においてメモリサイズのチェック文がまったく入っていませんでした。

このため、ペイロードの長さが64KBと書かれており、かつペイロードには短いデータXしか書かれていないブラウザからのHeartbeat Messageに対しても、サーバはデータXを含む64KB分のメモリデータをペイロードにフルコピーしてブラウザに送り返してしまいました。

結果として、返信すべきデータXに隣接するメモリデータにパスワードやクレジットカード番号などがたまたま含まれていると、それらが漏えいしたことになります。

Heartbleed攻撃に対する影響を見るうえで問題となったのが、

・HeartbeatMessageの長さがRFC仕様違反であることを除けば、通常のHeartbeat通信と変わらないため、不正が行われていることを

注4） http://heartbleed.com/

Column

続きがある、Heartbleedの余震

TLSサーバ運用上、もっとも重要なデータとしてサーバの私有鍵があります。私有鍵が漏えいすれば、もはやTLSの安全性を確保することはできません。そのため、最悪ケースとして「私有鍵自体が漏えいした可能性」を考慮した対策が求められました。具体的には、

①運用中のサーバ証明書の失効
②新しい私有鍵の生成
③②の私有鍵に対する新しいサーバ証明書の取得・設定

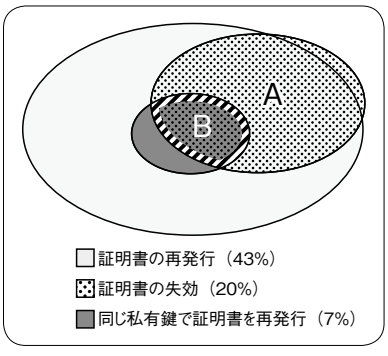
を行う必要がありました。

ところが、英Netcraft社の調査結果[注5]によれば、Heartbleedの公表後1ヵ月間で43%のサーバ証明書が更新されましたが、①～③の3つの対策すべてを正しく実施した（**図1**のAの部分）のは14%に過ぎませんでした。

一方、サーバ証明書を更新しているにもかかわらず、運用中のサーバ証明書を失効させなかった（①を実施しなかった）ケースが23%（**図1**のAとBを除いた部分）ありました。もっと悪いことに、運用中のサーバ証明書を失効させたにもかかわらず、漏えいした可能性がある同じ私有鍵をそのまま再利用してサーバ証明書を取得したケース（②を実施しなかった）が全体の5%（**図1**のBの部分）もあったそうです。

これらの対策ミスは、「Heartbleedの事後対策としてサーバ証明書を更新」するという指示が、文字どおり「サーバ証明書を更新」するという作業指示として受け取られ、本来の「漏えいした可能性がある私有鍵を失効」させるという指示が伝わらなかったケースと言えましょう。

▼図1　英Netcraft社の調査結果より



注5） http://news.netcraft.com/archives/2014/05/09/keys-left-unchanged-in-many-heartbleed-replacement-certificates.html

サーバは検知しづらく、ログに攻撃の痕跡が残っていなかった
・2年以上も未発見だった
・意図せず偶然持っていかれたデータなので漏えいした内容の特定が困難

といった点にあります。そのため、被害の全容を把握できず、重要なデータが漏えいしたとの前提で対策することが必要となりました。

BEAST攻撃／POODLE攻撃

BEAST攻撃[注6]やPOODLE攻撃[注7]は、共通鍵暗号であるブロック暗号で長いメッセージの暗号化を行うCBCモードを利用したときの脆弱性をついた攻撃です。

攻撃方法の原理は、次の条件

・攻撃対象のブラウザにマルウェアを感染させるなどにより、攻撃者がブラウザからサイトAに任意のメッセージを同じセッション鍵で暗号化させたうえで大量に送信できる状態にある
・攻撃者は、ブラウザからサイトAに送るメッセージの中身を自由に加工できる
・攻撃者は、ブラウザからサイトAに送る暗号化されたパケットをすべて傍受できる

がそろったときに、約256回のトライ＆エラーで、攻撃者は暗号化された情報の中身を1バイトずつ知ることができ、それを繰り返すことで情報全体がわかるようになるというものです。ここでのポイントは、「攻撃者がセッション鍵を知らなくても暗号化された情報の中身を知ることはできる」ということにあります。

これらの攻撃では、約256回のトライ＆エラーで1バイトの情報を求めることを繰り返していくことから、サイズは小さいが効果が大きい情報が攻撃対象としてふさわしいと言えます。たとえば、効果的な攻撃対象としてまず考えられるのは、ログイン状態などを記録してサイトとブラウザとの間で一定時間共有する暗号化されたCookie（セッションID）の中身です。

これらの攻撃でユーザUのCookieを攻撃者が得ることに成功すると、Cookieのしくみからユーザ UのID／パスワードを知らなくても、ユーザUになりすましてそのサイトAに不正アクセスできます。

SSL 3.0には致命的なPOODLE攻撃

POODLE攻撃は、SSL 3.0でブロック暗号をCBCモードで利用する場合のパディングチェックの仕様上の脆弱性をついた攻撃です。

具体的には、SSL 3.0ではパディングの最終1バイト分だけをチェックして正しければメッセージ全体が正しいと判断する仕様であるため、攻撃者が作った偽メッセージであっても1/256

注6）Thai Duong, Juliano Rizzo, "BEAST - Here Come The ⊕ Ninjas"
注7）https://www.openssl.org/~bodo/ssl-poodle.pdf

Column POODLE again

POODLE攻撃の反響が落ち着き始めたころ、POODLE againということで「TLS 1.xでもPOODLE攻撃が可能」との情報が公開されました。

しかし、注意しなければならないのは、SSL 3.0へのPOODLE攻撃は仕様上の脆弱性であるのに対し、TLS 1.xへのPOODLE againは実装上の脆弱性に起因していることです。具体的には、TLS 1.xでのパディングチェックのしくみが、仕様上はパディングの全データをチェックすべきところを、SSL 3.0と同じ最終1バイト分しか行っていない製品が数多く見つかり、TLS 1.xを使っていてもPOODLE攻撃と同じ手法が使えてしまったことが原因でした。

本来の仕様どおりに実装されていれば、SSL 3.0の場合とは違って、攻撃者が作った偽メッセージをサーバが受理する確率は極めて小さく（具体的には2^a分の1。aはパディング長を表す）、POODLE攻撃は成功しなかったでしょう。

の確率で正しいものとしてサーバが受理してしまう脆弱性を利用しています。

簡単に攻撃方法の概要を示すと、**図2**のように、暗号文全体の最後のCnを攻撃者が求めたい1バイトの情報（P_3の最下位1バイト）を含む場所にある暗号文C_3に置き換えて送信させます。

このような改変を行った暗号文は、ほとんどの場合、復号するとパディング値が正規のものと異なるためサーバで棄却されます。しかし、たまたま最終1バイトの復号結果が正規のパディング値と同じ値になった場合には、いったんサーバが暗号文を受理しますので、攻撃者には「暗号文が受理された＝CnをC_3に置き換えた復号結果の最終1バイトが正規のパディング値と同じ」ことがわかってしまいます。これらの情報から、攻撃者は暗号解読することなく、P_3の最下位1バイトの値を計算で求めることができます。

OpenSSL CCS Injection

ChangeCipherSpec（以下CCS）Injection脆弱性は、2014年6月に発見されたOpenSSL実装固有の脆弱性です（"Early CCS Attack"などと呼ばれることもあります）[注8]。

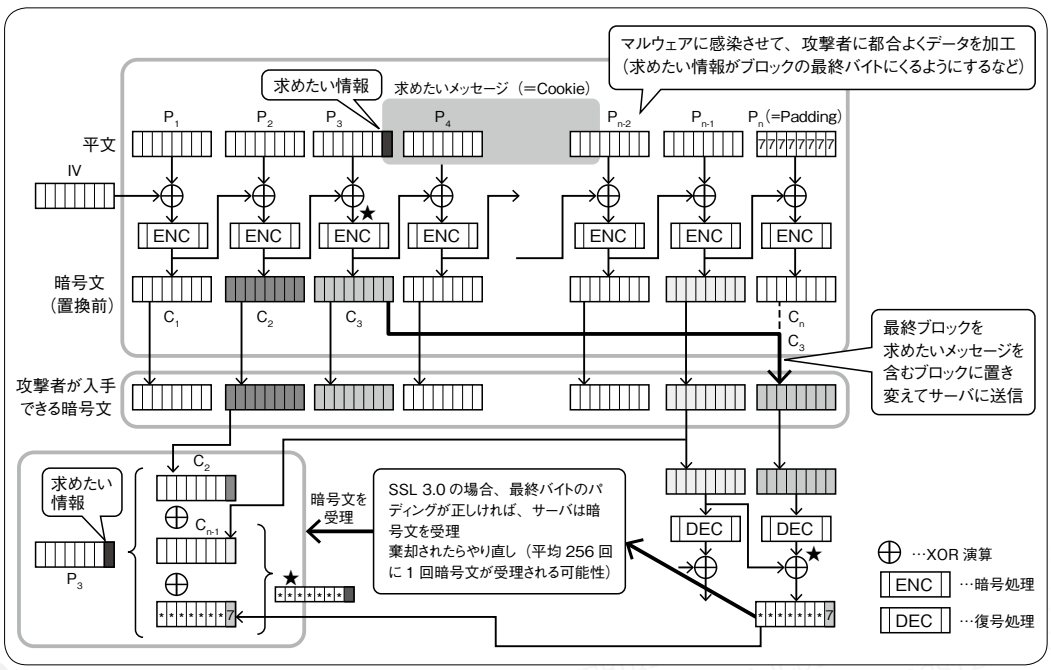
これは、OpenSSLのChangeCipherSpecメッセージの処理に欠陥があり、中間者攻撃と組み合わせることで暗号通信の情報が漏えいする可能性があるというものです。

ChangeCipherSpecメッセージは名前のとおり暗号（Cipher）スイートを変えることを指すメッセージです。中間者（攻撃者）が通信に割り込んで不適切なCCSを挿入すると、OpenSSLは適切な検証をせずそのCCSを受理してしまいハンドシェイクが不適切に行われて、弱い暗号鍵を使った暗号通信を開始させることができます。その結果、中間者が通信を完全に解読できる状態になってしまいます。

結果的に、修正前のバージョンのOpenSSLを使用していた環境でのWebの閲覧、電子メールの送受信、VPNといったソフトウェアでは保護されていた通信内容や認証情報などを詐取・

注8） http://ccsinjection.lepidum.co.jp/ja.html

▼図2　POODLE攻撃の概要



改ざんされる危険性がありました。中間者攻撃が必須であるためHeartbleedほどの危険性はないものの、標的型攻撃での危険性が高いと思われました(中間者攻撃の実際の危険性に関しては議論がありますが、公衆無線LAN／Wi-Fiが現実的な攻撃手段になり得ると思われます)。

この脆弱性は、Heartbleed脆弱性の発見を受けて、TLSプロトコルの仕様から、状態が複雑な個所を推測して発見されたもので、筆者の所属する㈱レピダムで発見し、OpenSSL ProjectやCERT/CC、JPCERT/CCなどの協力を得て公開に至りました。この脆弱性は、OpenSSLの最初のリリースから存在し、16年間発見されることはありませんでした。細かい発見の経緯や背景に関しては、弊社のブログやスライド[注9]を参照してください。

FREAK攻撃／Logjam攻撃

FREAK攻撃[注10]は、中間者攻撃に分類される中でもとくにダウングレード攻撃と呼ばれる攻撃手法の一種です。攻撃者はハンドシェイクの処理に割り込み「RSAを利用する輸出規制対象の暗号スイート(RSA_EXPORT)」に強制的にダウングレードさせます。

RSA_EXPORTは、2000年前後まで続いていた輸出規制に対応するためのもので、あえて暗号強度を弱める処理を行います。具体的には、たとえサーバ証明書で鍵長2,048bitのRSAを使ってセッション鍵を交換するように記載されていても、強制的に暗号強度を大きく弱めた鍵長512bitのRSAを利用してセッション鍵を交換するように制御します。もしRSAが解読できればセッション鍵を取り出すことができるため、当該TLS通信を復号することが可能です。

発見者によれば、鍵長512bitのRSAはAmazon EC2で100ドル出せば12時間以内に解読できると主張しています。実際、鍵長768bitのRSAの解読事例が2010年に発表されていることを考慮すれば、鍵長512bitのRSAが簡単に解読されたとしてもおかしくはありません。

Logjam攻撃[注11]もほぼ同様の攻撃手法であり、RSA_EXPORTの代わりに「DHE_EXPORT」に強制的にダウングレードさせます。DHE_EXPORTも輸出規制に対応するためのものであり、鍵長512bitのDHを使ってセッション鍵を交換します。RSAとDHは同じ鍵長であればほぼ同じ安全性であると考えられていますので、鍵長512bitのDHEも簡単に解読されると考えて良いでしょう。

秘密鍵の漏えい

秘密鍵(私有鍵やセッション鍵)が漏えいする原因としては、プログラムなどの実装ミスや秘密鍵の運用・管理ミス、サイバー攻撃やウィルス感染によるものなど、暗号解読以外が原因となっている場合のほうが圧倒的に多いです。

たとえば、2013年6月、エドワード・スノーデン氏が英文紙Guardianに、米国政府が世界中の数万の標的を対象に電話記録やインターネット利用を極秘裏に監視していたことを暴露しました。実際にどのようなことが行われていたかは不明な点が多いですが、仮にサーバの私有鍵やセッション鍵が米国政府に提供されていれば、暗号化している意味がありません。

また、台湾国民電子証明書のように実装上の問題がある場合にも私有鍵の漏えいが起こりえます。実際に、200万人以上の台湾国民の電子証明書を調べたところ、私有鍵の生成方法に問題があることが見つかりました。その結果、184個の私有鍵が算出できることがわかりました[注12]。

MD5証明書偽造攻撃

安全性が低下したハッシュ関数MD5の脆弱性

注9) http://ccsinjection.lepidum.co.jp/blog/2014-06-05/CCS-Injection/index.html
https://speakerdeck.com/lef/purotokorufalsexing-shi-jian-zheng-tocui-ruo-xing-fa-jian-falsexian-shi-case-of-ccs-injection
注10) https://freakattack.com/
注11) https://weakdh.org/
注12) http://crypto.2013.rump.cr.yp.to/55e2988c4ed3c9f635c9a4c3f52fa0b1.pdf

を悪用して、実際に偽認証局のCA証明書を偽造する攻撃で、2008年末のChaos Communication Congressで発表されました[注13]。

この偽認証局のCA証明書は、ブラウザでの証明書検証ロジックのわずかな脆弱性をついて、本物の認証局が発行した正規のCA証明書であるとブラウザが判断するように作られていました(**図3**)。このため、偽認証局が発行した偽サーバ証明書でも、ブラウザは最終的に正しいサーバ証明書であると誤認証してしまうことを実際に示しました。

幸運だったのは、この発表が脆弱なハッシュ関数MD5を使った偽CA証明書が実際に偽造でき、PKIの中で有効に機能してしまうことを証明するために行われた実験であったため、実害が生じることはなかったことです。実際、この発表を契機に多くの認証局がMD5を使うCA証明書やサーバ証明書の発行を取りやめました。

DigiNotar事件

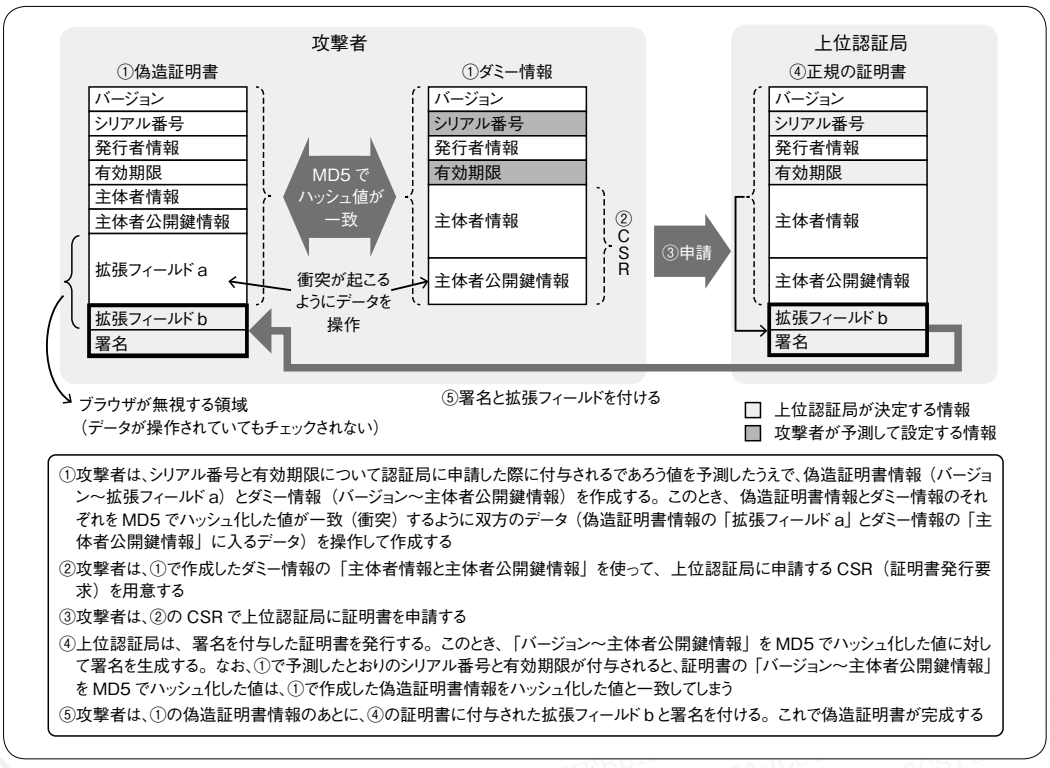
DigiNotar社はパブリックルート認証局としておもにオランダ国内を市場として証明書を発行していましたが、2011年6月に同社の認証局システムがComodo hackerと名乗る人物に不正侵入され、1ヵ月以上に渡る遠隔操作により少なくとも531枚ものサーバ証明書が不正発行されたことが判明しました。そのうち、少なくとも344枚には、Google、Microsoft、Mozilla、Skypeなど有名なドメイン名が使われ、またイスラエル諜報特務局、英国MI6、米国CIAといった諜報機関のドメインも含まれていました。

この事件が単なる不正侵入事件にとどまらなかったのは、

・Comodo hackerは約半年前に、別の認証局Comodo社でも偽サーバ証明書を不正発行させる攻撃に成功していた(この事件が発端で

注13) http://www.win.tue.nl/hashclash/rogue-ca/

▼図3　偽造証明書の作成方法



①攻撃者は、シリアル番号と有効期限について認証局に申請した際に付与されるであろう値を予測したうえで、偽造証明書情報（バージョン～拡張フィールドa）とダミー情報（バージョン～主体者公開鍵情報）を作成する。このとき、偽造証明書情報とダミー情報のそれぞれをMD5でハッシュ化した値が一致（衝突）するように双方のデータ（偽造証明書情報の「拡張フィールドa」とダミー情報の「主体者公開鍵情報」に入るデータ）を操作して作成する

②攻撃者は、①で作成したダミー情報の「主体者情報と主体者公開鍵情報」を使って、上位認証局に申請するCSR（証明書発行要求）を用意する

③攻撃者は、②のCSRで上位認証局に証明書を申請する

④上位認証局は、署名を付与した証明書を発行する。このとき、「バージョン～主体者公開鍵情報」をMD5でハッシュ化した値に対して署名を生成する。なお、①で予測したとおりのシリアル番号と有効期限が付与されると、証明書の「バージョン～主体者公開鍵情報」をMD5でハッシュ化した値は、①で作成した偽造証明書情報をハッシュ化した値と一致してしまう

⑤攻撃者は、①の偽造証明書情報のあとに、④の証明書に付与された拡張フィールドbと署名を付ける。これで偽造証明書が完成する

Comodo hackerとの名がついた）
・Comodo hackerはイラン在住の人物であることを名乗っており、実際にDigiNotarから不正発行されたサーバ証明書がイラン政府機関（体制側）などによる盗聴行為に利用された可能性があった
・Microsoftなどの主要ブラウザベンダによって、初めてこのような事件を理由に（セキュリティパッチの緊急発行という形で）ルート証明書の失効が行われた
・DigiNotarが信用を失って破産した

など具体的かつ深刻な実害が発生したことです。

サーバを守るために考えるべきこと

これまでに7つの事例を紹介しましたが、一言に脆弱性と言っても種類がまったく違うことがおわかりいただけたのではないかと思います。紹介した以外のものを含めた最近のTLSへの攻撃事例をまとめたものがRFC 7457[注14]として公開されています。

それでは、ここからは自らがTLSサーバを構築し、運用する場合に何に気をつけるべきかを考えましょう。

まず、TLSの利用有無にかかわらず、サーバを運用する以上、実装脆弱性に対する最新の対策（とりわけセキュリティ対策）を講じることは必要不可欠です。そのうえで、TLSサーバ特有の観点として、サーバの私有鍵の正しい運用・管理も必要となります。その際、意図しない私有鍵の漏えいが発生したときの対処方法や影響範囲の局所化を考慮した事前対策（PFSなど）を併せて検討しておくことが望まれます。

さらに、脆弱性の中には、TLSのしくみ上、サーバ構築者側では対策が取りにくいものがあります。とくにTLSは事前に面識がないブラウザからの接続要求でも受け付け、その場でハンドシェイクを行うことで暗号通信できるしくみであることから、「ブラウザが信頼できる状態かどうかをサーバ側では正確に判断できない」ということを前提にしておく必要があります。

このことから、サーバ構築時には「中間者攻撃を受けるリスク」と「ブラウザがマルウェアに感染しているリスク」についてはあらかじめ織り込んでおき、これらの事象が発生したとしても、それに伴う悪影響をサーバ側でできる限りブロックする設定をしておくことが大切です。

また、フィッシングサイトへ誘導する偽サイトが構築されないようにするには、自らのサイトの身分証明をはっきりさせ、ブラウザ利用者に誤認させないことが重要です。そのため、サーバ証明書は正しく生成しなければなりません。

SSL/TLS暗号設定ガイドライン

サーバ設定に関するもう1つの重要な視点は、TLSは相互接続性の確保を優先しながら20年間使われ続けてきたプロトコルであり、安全性のレベルがまったく異なる設定が数多く含まれているという点です。その理由は、プロトコルの脆弱性に対応するため、何度かバージョンアップが行われている影響で、製品の違いによってサポートしているプロトコルバージョンや暗号スイートなどが異なることにあります。さらに、基本的には後方互換を持つように実装・設定されています。

しかし、その結果、それらの設定の中には、安全性上の要求条件同士、あるいは実際の実装環境と矛盾するケースすら出てくるようになりました。たとえば、

・RSAとDHEのどちらを使うべきか
実装率や利用可能な鍵長を考えればRSAのほうが良いが、秘密鍵漏えい時の事前対策としてはDHEのほうが良い
・ブロック暗号とストリーム暗号（RC4）のどちらを使うべきか

注14）https://tools.ietf.org/html/rfc7457

暗号アルゴリズムとしては、RC4は脆弱であり、Triple DESのほうがはるかに安全である。しかし、POODLE攻撃に対しては、マルウェア感染のリスクを考慮すると、RC4よりTriple DESのほうがリスクが高いとも言える

といったことがあります。

そこで、TLSでの設定を考える際には、安全性と相互接続性の確保のバランス論を抜きに語ることはできません。もっとも、どの程度の安全性と相互接続性の確保が必要かは、最終的にサーバ構築者が判断することになります。

その際、参考になるのが、IPAから公開された「**SSL/TLS暗号設定ガイドライン**」[注15]です。本ガイドラインには、サーバ構築者が、安全性と相互接続性の確保についてどのような考え方をしたら良いか、またそれを実現するためにどのようなサーバ設定をすべきかについてまとめています。CRYPTREC[注16]の運用ガイドラインWGによって取りまとめられました。

以降では、本ガイドラインの概要を簡単に紹介します。詳細については、ガイドライン本体をご覧ください。

どんな設定を考えるべきか

本ガイドラインでは、安全性と相互接続性のバランスを**表2**の3種類の設定基準に分けたうえで、各々の設定基準に応じて「プロトコルバージョン」「サーバ証明書」「暗号スイート」の3つの要求設定が取りまとめられています。

このうち、現状では、高セキュリティ型は「とりわけ高い安全性を必要とするケースであって、一般的な利用形態で使うことは想定していない」としていることから、一般的な利用形態としての「推奨セキュリティ型」、もしくは安全性よりも相互接続性の確保を優先させた「セキュリティ例外型」のどちらかを選ぶことになると思われますので、ここでは推奨セキュリティ型とセキュリティ例外型のみを取り上げることにします。

● 推奨セキュリティ型

推奨セキュリティ型は、現時点での安全性と相互接続性の確保をバランスさせてTLS通信を行うための標準的な設定基準としており、次の利用形態などが代表例とされています。

- 金融サービスや電子商取引サービス、多様な個人情報の入力を必須とするサービスなどを提供する場合
- 既存システムとの相互接続を考慮することなく、新規に社内システムを構築する場合

この設定が正しく行われていれば、実装脆弱性以外の脆弱性が見つかったとしてもサーバ運用上の影響を受けるリスクはかなり低減できるでしょう。たとえばPOODLE攻撃、CCS Injection攻撃、FREAK攻撃、Logjam攻撃などのいずれからも影響を受けることはありません。

● セキュリティ例外型

セキュリティ例外型は、脆弱なプロトコルバージョンや暗号が使われるリスクを受容したうえで、安全性よりも相互接続性に対する要求をやむなく優先させてTLS通信を行う場合に、許容し得る最低限度の設定基準です。とくに、シス

注15) http://www.ipa.go.jp/security/vuln/ssl_crypt_config.html
注16) Cryptography Research and Evaluation Committeesの略。電子政府推奨暗号の安全性を評価・監視し、暗号技術の適切な実装法・運用法を調査・検討するプロジェクト。http://www.cryptrec.go.jp/

▼表2 設定基準の種類

設定基準	安全性	相互接続性
高セキュリティ型	標準的な水準を大きく上回る高い安全性水準を達成	最新のPC・ブラウザなどでなければ接続できない可能性が高い
推奨セキュリティ型	標準的な安全性水準を実現	一部の古い機器を除き、ほぼ接続できる
セキュリティ例外型	短期的な利用を前提に、許容可能な最低限の安全性水準を満たす	ほぼすべての機器が接続できる

テムなどの制約上、SSL 3.0の利用を全面禁止することの影響は無視できず、安全性上のリスクを受容してでもSSL 3.0を継続利用せざるを得ないと判断される場合にのみ採用するのが望ましいとしています。

なお、推奨セキュリティ型への早期移行を前提として暫定的に利用継続するケースを想定していることから、近いうちにSSL 3.0を利用不可に設定するように変更される可能性があります。IETFでRC4の利用を禁止するRFC[注17]が発行されたり、ブラウザベンダがSSL 3.0を利用不可にする設定変更を行ったりするなどの対策が進められています。

具体的な要求設定

プロトコルバージョン

基本的に、プロトコルバージョンがあとになるほど、以前の攻撃に対する対策が盛り込まれるため、より安全性が高くなります。一方、相互接続性も確保する観点から、多くの場合、古いプロトコルバージョンも利用できるように実装・設定されています。

したがって、プロトコルバージョンの選択順位を正しく設定しておかないと、予想外のプロトコルバージョンでTLS通信を始めてしまう恐れがあります。とくに中間者攻撃でのダウングレード攻撃で悪用される恐れがあります。

そこで本ガイドラインでは、**表3**のようなプロトコルバージョンの要求設定になっています。

本来的にはTLS 1.2が使える状態にするのが一番いいわけですが、利用している製品によってはTLS 1.2やTLS 1.1をサポートしていない場合もあります。パターン1だけではTLS 1.2をサポートしていない製品を利用している場合、TLS 1.2をサポートしている製品に交換する必要性が生じますので、推奨セキュリティ型に準拠させること自体を断念する恐れがあります。一方、脆弱なSSL 3.0を利用不可にすることは設定変更だけで実現可能ですので、製品を交換するよりははるかに簡単に対応できるはずです。そのため、TLS 1.2やTLS 1.1をサポートしていない製品を利用している場合の推奨セキュリティ型の要求設定として、パターン2、パターン3が用意されています。

なお、推奨セキュリティ型とセキュリティ例外型の差はSSL 3.0の利用可否の部分のみです。

サーバ証明書

サーバ証明書は、「①ブラウザに対して正しいサーバであることを確認する手段を提供すること」と「②TLS通信を行うために必要なサーバの公開鍵情報をブラウザに正しく伝えること」の2つの役割を持っています。その目的を実現するため、本ガイドラインでは、**表4**のようなサーバ証明書の要求設定になっています。

なお、推奨セキュリティ型では、認証局の署名アルゴリズムでSHA-256の利用を必須としているため、SHA-256が扱えないブラウザではサーバ証明書の検証ができず、警告表示が出るか当該サーバとの接続が不能となります。一方、セキュリティ例外型では、SHA-1の利用も許容しているため、過去のシステムとの相互接続性は高くなっています。ただ、最新のブラウザではSHA-1を使うサーバ証明書に対して警告表示を出すように変

注17) https://tools.ietf.org/html/rfc7465

▼表3　プロトコルバージョンの要求設定

推奨セキュリティ型

パターン	TLS 1.2	TLS 1.1	TLS 1.0	SSL 3.0	SSL 2.0
1	◎	○	○	×	×
2	−	◎	○	×	×
3	−	−	◎	×	×

セキュリティ例外型

パターン	TLS 1.2	TLS 1.1	TLS 1.0	SSL 3.0	SSL 2.0
1	◎	○	○	○	×
2	−	◎	○	○	×
3	−	−	◎	○	×

○：設定有効(◎：優先するのが望ましい)　×：設定無効化　　−：実装なし

わってきていることに注意が必要です。

このほかにガイドラインには、両者に共通の要求設定として、「サーバ証明書の発行・更新時の鍵情報の生成」に関する項目と「ブラウザでの警告表示の回避」に関する項目があります。

● 暗号スイート

TLS通信においては、ハンドシェイク時に暗号スイートが選択され、その選択された暗号スイートに記載の鍵交換、署名、暗号化、ハッシュ関数によりTLSにおける各種処理が行われます。つまり、TLSにおける安全性にとって、暗号スイートをどのように設定するかが最も重要なファクタになります。とくに、暗号スイートの優先順位の上位から順にサーバとブラウザの両者が扱える暗号スイートを見つけていくのが一般的であるため、暗号スイートの選択のみならず、優先順位の設定も重要です。

そこで、本ガイドラインでは、暗号スイートの選択および優先順位の付け方の基準として、次の方針を採用しています。

推奨セキュリティ型

次の条件を満たす暗号スイートを選定する

・CRYPTREC暗号リスト[注18]に掲載されているアルゴリズムのみで構成
・暗号化として128bit安全性以上を有する
・DSAを含まない
・ECDHE、ECDH、ECDSAを利用するか否かは十分な検討のうえで決めることが望ましい

次の条件に従い、暗号スイートの優先順位を付ける

・通常の利用形態において、128bit安全性があれば十分な安全性を確保できることから128bit安全性を優先する
・鍵交換に関しては、PFSの特性の有無と実装状況に鑑み、DHE/ECDHE、次いでRSA、ECDHの順番での優先順位とする

セキュリティ例外型

基本的には、推奨セキュリティ型の方針を踏襲する。そのうえで、セキュリティ例外型ではSSL 3.0を容認することから、利用可能な暗号スイートとしてRC4とTriple DESを含む暗号スイートを最後に追加する。なお、本来的にはRC4はSSL 3.0に限定して利用すべきであるが、TLS 1.0以上のプロトコルバージョンでRC4の利用を不可にする設定を行うことが難しいため、TLS 1.0以上であってもRC4が使われる可能性が排除できないことにも注意がいる

これらの基準によって決められた要求設定が**表5**になります。**表5**では、グループ内の暗号スイートの優先順位は任意とする一方、グループ間の優先順位は、A、B、C、……の順番で設

注18）http://www.cryptrec.go.jp/images/cryptrec_ciphers_list_2013.pdf

▼表4　サーバ証明書の要求設定

設定基準	サーバ証明書の暗号アルゴリズムと鍵長
推奨セキュリティ型	サーバ証明書の公開鍵情報（Subject Public Key Info）のアルゴリズムと鍵長は、次のいずれかを必須とする ・RSAで鍵長は2,048bit以上 ・楕円曲線暗号で鍵長256bit以上 認証局の署名アルゴリズム（Certificate Signature Algorithm）と鍵長は、次のいずれかを必須とする ・RSA署名とSHA-256の組み合わせで鍵長2,048bit以上 ・ECDSAとSHA-256の組み合わせで鍵長256bit以上
セキュリティ例外型	サーバ証明書の公開鍵情報（Subject Public Key Info）のアルゴリズムと鍵長は、次を必須とする ・RSAで鍵長は2,048bit以上 認証局の署名アルゴリズム（Certificate Signature Algorithm）と鍵長は、次のいずれかを必須とする。組み合わせとしてはSHA-256のほうが望ましいが、状況によってはSHA-1を選んでも良い ・RSA署名とSHA-256の組み合わせで鍵長2,048bit以上 ・RSA署名とSHA-1の組み合わせで鍵長2,048bit以上

定することを求めています。たとえば、

・鍵交換DHE、署名RSA、モードGCM、ハッシュ関数SHA-256での鍵長128bitのAESとCamelliaなら、どちらもグループAの暗号スイートになるので、AESとCamelliaのどちらを優先するかは任意
・署名RSA、暗号化(鍵長128bit)AES、モードCBC、ハッシュ関数SHA-1で、鍵交換がDHEとRSAである場合、グループAの暗号スイートである鍵交換DHEのほうを、グループBの鍵交換RSAより優先させる

ということを意味しています。

とにかく設定してみよう

本ガイドラインは「暗号技術以外のさまざまな利用上の判断材料も加味した合理的な根拠」を重視して現実的な利用方法を目指す観点で有識者の知見を集めて作られています。そのため、要求設定が机上の空論ということはありません。実際、本ガイドラインのAppendixにはApacheなどの設定方法例も記載されています。

実際の製品では、とくに暗号スイートの設定に関して、本ガイドラインの記載どおりに細かく設定できない可能性はあります。しかし、その場合でも、本ガイドラインの暗号スイートの基準にもっとも近い設定を行うことで、安全性を高めることができます。

ここで解説した事項以外にも、TLSを安全に使うために考慮すべきポイントなどの記載もあります。TLSサーバを構築する際には、デフォルト設定のままにしておくのではなく、本ガイドラインを参考にした安全なサーバ設定に変えていってほしいと思います。SD

▼表5　暗号スイートの要求設定

推奨セキュリティ型

グループA	鍵交換DHE、署名RSA、暗号化(鍵長128bit)AESまたはCamellia、モードGCMまたはCBC、ハッシュ関数SHA-2またはSHA-1
	鍵交換ECDHE、署名ECDSAまたはRSA、暗号化AESまたはCamellia、モードGCMまたはCBC、ハッシュ関数SHA-2またはSHA-1
グループB	鍵交換RSA、署名RSA、暗号化(鍵長128bit)AESまたはCamellia、モードGCMまたはCBC、ハッシュ関数SHA-2またはSHA-1
グループC	鍵交換ECDH、署名ECDSAまたはRSA、暗号化(鍵長128bit)AESまたはCamellia、モードGCMまたはCBC、ハッシュ関数SHA-2またはSHA-1
グループD	鍵交換DHE、署名RSA、暗号化(鍵長256bit)AESまたはCamellia、モードGCMまたはCBC、ハッシュ関数SHA-2またはSHA-1
	鍵交換ECDHE、署名ECDSAまたはRSA、暗号化(鍵長256bit)AESまたはCamellia、モードGCMまたはCBC、ハッシュ関数SHA-2またはSHA-1
グループE	鍵交換RSA、署名RSA、暗号化(鍵長256bit)AESまたはCamellia、モードGCMまたはCBC、ハッシュ関数SHA-2またはSHA-1
グループF	鍵交換ECDH、署名ECDSAまたはRSA、暗号化(鍵長256bit)AESまたはCamellia、モードGCMまたはCBC、ハッシュ関数SHA-2またはSHA-1
設定すべき鍵長	鍵交換でDHEを利用する場合には鍵長1,024bit以上、RSAを利用する場合には鍵長2,048bit以上、ECDHEまたはECDHを利用する場合には鍵長256bit以上の設定を必須とする。なお、DHEの鍵長を明示的に設定できない製品を利用する場合には、DHEを含む暗号スイートは選定すべきではない

セキュリティ例外型

グループA～F	推奨セキュリティ型と同じ
グループG	鍵交換RSA、署名RSA、暗号化(鍵長128bit)RC4、ハッシュ関数SHA-1
グループH	鍵交換DHEまたはRSA、署名RSA、暗号化Triple DES、モードCBC、ハッシュ関数SHA-1
設定すべき鍵長	推奨セキュリティ型と同じ

Appendix

TLSを取り巻く環境、そしてTLSの今後について (TLS 1.3、HTTP/2)

Author ㈱レピダム　林 達也（はやし たつや）

TLS 1.3

スノーデン事件、PRISMの発覚以来、通信の暗号化は非常に注目を浴びるトピックとなり、TLS、そしてHTTPSはそれを担う最も大事な通信プロトコルとなりました。

TLSはIETF（第2章参照）で標準化が行われており、プロトコル仕様に根ざす脆弱性などが発覚した場合には、IETFでも即座に対策について話し合われるようになってきています。仕様そのものを変更するのはなかなか難しいため、多くの対策は追加の形で仕様が作られています（例：POODLE AttackにおけるTLS_FALLBACK_SCSVの導入など[注1]）。今後に向けてプロトコルの仕様そのものを更新する動きも始まっており、TLS 1.3として現在IETF TLS WGで作業中です。

TLS 1.3は2015年6月現在、まさに策定中であり、今後どうなるかはわかりませんが、現時点で挙げられている大きな課題としては次のようなものがあります。

- renegotiation（通信中の再ネゴシエーション）の禁止
- compression（通信の圧縮機能）の禁止
- Perfect Forward Secrecy（PFS）の必須化
- Authenticated Encryption with Associated Data（AEAD）の必須化
- 脆弱な暗号アルゴリズムの整理
- ネゴシエーションの高速化
- TLS 1.2との互換性
- 楕円曲線暗号採用の是非

このようにTLS 1.0や1.1に比べ、1.3では今までよりも大きな変更が行われることが予測されます。これは、もちろん現実世界におけるTLSの重要性や現状の変化をふまえたものです。

仕様の変更、とくにセキュリティに関わる仕様の変更は、その仕様を実装する組織や人々に大きな影響を与えるため、慎重に行われます。また、仕様を策定してから、実装が行われ、一般的になる（デプロイが終わる）までにはかなりの時間がかかります。TLSは今までそういったことに配慮した結果、より安全で好ましい選択肢があるにもかかわらず、なかなかそれを仕様として条件付けできずに、何度も危機的な状況に陥ってしまっている状況にありました。

そういった背景をふまえ、TLS 1.3ではTLSを近代的な仕様に刷新し、基準となるベースラインを底上げする方向に向かっていると言えるでしょう。

TLS 1.3を先取りするHTTP/2

このTLSの取り組みの一部は、最近RFCとなったHTTP/2[注2]でも配慮されており、たとえば暗号アルゴリズムの要求などはTLS 1.3の方向性を先取りする形でHTTP/2に反映されている[注3]など、少しずつ実社会に影響を与えてきています。

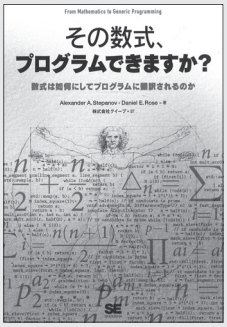
通信の暗号化という意味では、HTTP/2の策定段階でも「常時暗号通信を行うべきなのではないか」といった激しい議論があり、「HTTPの平文通信はなくすべき」といった話もありましたが、最終的には一応暗号化されていない平文通信も仕様上は残ることになりました。一方で、現時点でも平文のHTTP/2を実装しているブラウザはほとんどなく、率先してHTTP/2のブラウザ実装を公開してきたGoogle（Chrome）もMozilla（Firefox）もHTTP/2の実装はHTTPSのみの対応となっています。

今後のWebは暗号化が必須!?

また、TLS通信で大きな障壁ととらえられがちな証明書に関しても、DV（Domain Validation）証明書[注4]に関しては無料で発行してしまい、HTTPSを普及させようというLet's Encrypt Project[注5]のような、ある意味過激とも言える取り組みまで始まっています。

HTTPもTLSが事実上主流になるなど、さまざまな状況をふまえると、今後のインターネット上の通信はほとんど暗号化されることが明白です。今後Webおよびインターネットに関わる人にとって、TLS、HTTPS、そして暗号通信は避けて通れない重要な基礎技術となっていくと思われます。SD

注1）https://tools.ietf.org/html/rfc7507
注2）https://tools.ietf.org/html/rfc7540
　　http://summerwind.jp/docs/rfc7540/ （Moto Ishizawa氏による日本語の参考訳）
注3）RFC 7540 Appendix A. TLS 1.2 Cipher Suite Black List
注4）ドメインが存在するか（レジストラへ登録されているか）のみを確認して発行される証明書。組織が実在するかを確認し基準を満たしているかを確認するEV（Extended Validation）、OV（Organization Validation）証明書といった、より厳格な証明書も存在する。
注5）https://letsencrypt.org/



その数式、プログラムできますか?

アレクサンダー・A・ステパノフ、
ダニエル・E・ローズ 著、
株式会社クイープ 訳
A5判／348ページ
2,600円＋税
翔泳社
ISBN＝978-4-7981-4110-7

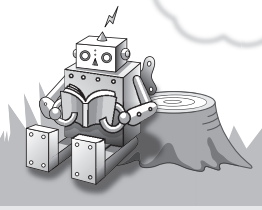
Web全盛の昨今、プログラムする対象はWebブラウザとサーバ間のやり取りが中心になり、科学技術計算的な問題を解決するためのプログラミングは減少したかのように見える。しかし、現在はコンピュータシステムが極めて大規模になり、たとえばHadoopのような抽象度の高い分散処理を使いこなすにはアルゴリズム、ひいては数学的な知識が必要になってきている。本書のスタンスは、アルゴリズムとデータ構造を設計することに焦点を合わせた「ジェネリックプログラミング」である。さらに、これを学ぶうえで必要なものは抽象代数学、そして数論である。理解のために丁寧な解説と付録がついているが、やはりハードルは高い。目の前の仕事にすぐに役立つものではないが、実力を涵養（かんよう）するために、じっくり読んでほしい本だ。

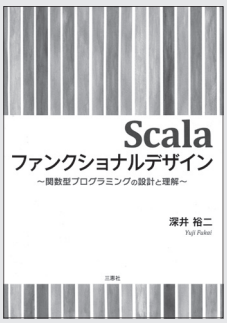
ＩＴプロジェクトの英語

塚本 俊、小坂 貴志 著
A5判／224ページ
1,800円＋税
ジャパンタイムズ
ISBN＝978-4-7890-1599-8

本書は企画・設計・開発・保守・管理・評価といったITプロジェクトの大まかな流れに沿って、各場面で使われる英単語・フレーズを紹介している。send an internal decision request（稟議を上げる）、avoid rework（手戻りを防止する）といったほかの業界でも使えるフレーズのほか、Define non-functional requirements（非機能要件を定義する）、implement functions（機能を実装する）、Cope with incidents（インシデントに対応する）など普通の英語書には載っていないフレーズも学ぶことができる。本の中に出てくる201の英語例文については、MP3形式の音声ファイルが用意されており、出版社のサイトから無料でダウンロードすることで、発音やイントネーションを確認しながらの勉強ができる。

SD BOOK REVIEW





Scala ファンクショナルデザイン

深井 裕二 著
A5判／300ページ
2,500円＋税
三恵社
ISBN＝978-4-86487-379-6

Scalaについて、その関数型機能を中心に解説した1冊（対象とするバージョンはScala2.1.1）。「コレクション」「高階関数」「クロージャ」「部分適用とカリー化」「パターンマッチング」などの関数型の特徴となる機能を、簡潔で短いコードで説明している。プログラミングの基礎については簡単な説明しかなく、Scalaの文法を網羅しているわけではないので、対象読者は中級者と思われる。しかし、第4章「関数」では、関数の定義と実行のしくみ、引数と戻り値の関係などが実例を挙げて詳しく解説されており、以後の「高階関数」などの発展的な章の理解の助けになる。Scalaは学習コストの高い言語と言われているが、本書では「文法」ではなく「機能」に解説の重点を置くことで、読者の早期理解を助けている。

ゲームプログラマのためのコーディング技術

大圖 衛玄 著
A5判／256ページ
2,480円＋税
技術評論社
ISBN＝978-4-7741-7413-6

職業プログラマは、保守しやすく可読性の高いコードを書くことが求められる。しかし、コーディングパターンやオブジェクト指向など、知識として知っていても実際の開発に反映させるのはなかなか難しい。本書は、すぐに実践できる順番にコーディング技術が紹介されている。前半では、わかりやすいコードを書くために複雑なコードを小さく分割する方法を紹介している。“とりあえず動く”C++のサンプルコードをもとに解説しているので、コードを改善する流れが理解しやすい。後半では、オブジェクト指向設計の原則をもとに、シンプルなクラス設計とは何かを解説している。全体を通して平易な文章で解説しており、高度に感じられるテクニックも楽に読み進められる。C++プログラマであれば、手元に置いておきたい1冊と言える。

エンタープライズJavaの進化

AWSで始めよう！ モダンなJavaアプリケーション開発

しなやかで強いソフトウェアの作り方

Author 永瀬 泰一郎(ながせ たいいちろう) Twitter nagaseyasuhito
Mail nagase@nagaseyasuhito.net

JavaとDockerで始める 実践Elastic Beanstalk入門

Amazon Web Services(AWS)やGoogle Cloud Platformなどのクラウドプラットフォームの出現によって、少ない費用、短い期間でITインフラの調達ができるようになり、クラウドプラットフォームは個人やスタートアップ企業などにとって欠かせないものになっています。しかし、Webサービスの公開などにあたり、アプリケーションサーバの構築、ロードバランサの設定、監視やスケールアウトの設計などインフラ層の設定は多岐にわたるため、運用コストも少なくなくWebサービスの開発だけに注力するには、まだまだ難しいのが現状です。そのような問題を解決するのがAWSが提供するElastic Beanstalkというサービスです。今回は一番ホットな仮想化ソフトウェアのDockerとJavaを例にElastic Beanstalkを使ったWebサービスを公開する方法を紹介します。

Amazon Web Servicesの提供するPaaS、Elastic Beanstalkとは?

Elastic Beanstalkとは何か

AWSが提供するElastic Compute Cloud(EC2)やSimple Storage Service(S3)のことをご存じの方は多いと思いますが、Elastic Beanstalkというサービスを知っている方はあまり多くないかもしれません。

Elastic Beanstalkは2011年にAWSが公開したサービスで、EC2やS3、Elastic Load Balancing(ELB)といったAWSのサービスを組み合わせて提供されているPlatform as a Service(PaaS)の1つです。Elastic Beanstalk自体に費用は発生せず、構築したWebサービスのEC2やS3、ELBなどの利用分だけに費用が発生します。

先述したアプリケーションサーバの構築やロードバランサの設定などはすべてElastic Beanstalkがよしなに面倒をみてくれるので、ユーザはWebサービスを開発し、Elastic Beanstalkにデプロイを行うだけで、負荷にあわせて自動的にスケールするWebサービスを簡単に公開できます。

もちろんアプリケーションサーバはEC2のインスタンスなので緊急時や不具合の調査のためにSSHで接続して各種操作を行うことも可能です。

公開当初にサポートされている言語はJavaだけでしたが、現在は次のようにさまざまな言語とプラットフォームに対応[注1]しています。

- PHP
- Python
- Node.js
- Ruby
- .NET
- Go

さらにDockerというコンテナ型の仮想化ソフトウェアにも対応したため、Dockerのコンテナとして提供すれば事実上どのようなプラットフォームでもElastic Beanstalkの恩恵を受けることができます。

JavaとElastic Beanstalk組み合わせの利点

コンシューマ向けWebサービス界隈では敬遠される傾向にあったJavaですが、2014年に

注1) http://docs.aws.amazon.com/elasticbeanstalk/latest/dg/concepts.platforms.

リリースされたJava 8はラムダ式やStream APIなど関数型言語のエッセンスを取り入れたことで、過去の資産を生かしつつモダンなアプリケーション開発ができるようになりました。

そして企業システムを始めとしたエンタープライズ向けと位置づけられるJava EEですが、本質は大規模で信頼性の高いサーバアプリケーションを開発するためのプラットフォームです。

いまやデータベースとフロントエンドだけではWebサービスは成り立ちません。メール送信やバッチ処理、メッセージキューといった開発や運用に欠かせないバックエンドの機能もJava EEには盛り込まれています。いくつものミドルウェアを組み合わせずに複雑なWebサービスをJava EEだけで開発できる点はコンシューマー向けWebサービスでも非常に魅力的ではないでしょうか？

またJavaのビルドツールのApache Maven[注2]にはElastic Beanstalkに対応したプラグインがあり、通常のビルドライフサイクルにElastic Beanstalkへのデプロイをシームレスに統合できる点も見逃せません。

シンプルなしくみでありながらスケールするWebサービスを提供できるElastic Beanstalkと、さまざまな機能がアプリケーションサーバに盛り込まれたJava EEを組み合わせることで、パワフルなWebサービスを短期間でリリースできるようになります。

アプリケーションと環境

Elastic Beanstalkのアーキテクチャ

図1のとおりElastic BeanstalkはほかのAWSのサービスを組み合わせて成り立っています。Elastic Beanstalkの**アプリケーション**は1つのWebサービスを表すもっとも大きな単位で**環境**と**アプリケーションバージョン**を持ちます。

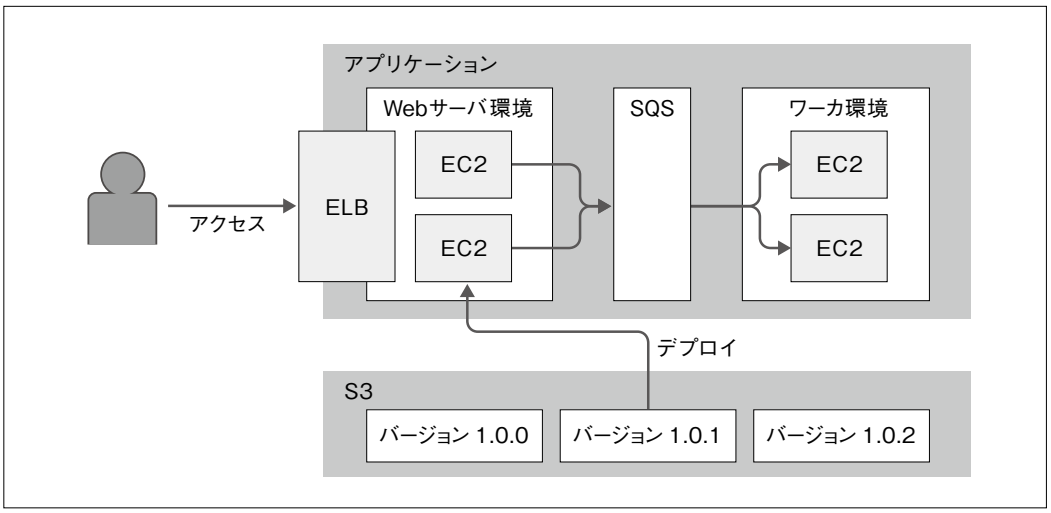
環境はアプリケーションの機能の単位です。目的に応じて複数の**環境**を持つことができるので、例えばWebサービスのユーザ向けの環境と、管理者用のアプリケーションを論理的に分割して管理する使い方ができます。

環境はWebサーバ環境枠・ワーカ環境枠の2種類の**環境枠**から1つを選択して構築します。

Webサーバ環境枠はユーザからのHTTPリクエストを受け付けてレスポンスを返す、いわゆるアプリケーションサーバ用の環境です。

注2）https://maven.apache.org/

▼図1　Elastic Beanstalkのアーキテクチャ



[任意の名前].elasticbeanstalk.comというドメイン名を割り当てられます。

ワーカ環境枠というのはAWSのサービスのひとつSimple Queue Service(SQS)というメッセージキューのサービスを使ったバックグラウンド処理用の環境枠です。この環境枠はSQSのキューに配信されたメッセージをHTTPのPOSTリクエストに変換します。つまりHTTPリクエストとしてキューのリクエストを受け取り、レスポンスを返すバックエンドWebサービスを開発するだけで簡単にバックグラウンド処理を記述できます。

環境枠はTomcatやGlassFishといった言語やアプリケーションサーバなどが定義された**プラットフォーム**を選択できます。事前に設定済みの環境からDockerを用いた独自の環境まで、さまざまなプラットフォームが用意されています。

環境はロードバランシングでオートスケーリングするものと、ひとつのインスタンスのみを使うシングルインスタンスの2つの**環境タイプ**から選べます。本番環境などはロードバランシングでオートスケーリングする環境を選び、トラフィックが少ないことが想定される場合や、検証に使う場合はシングルインスタンスの環境タイプを選択するといいでしょう。

Javaのウェブアーカイブ(war)などアプリケーションサーバにデプロイするアーカイブは、アプリケーションバージョンと呼ばれるラベル付けがおこなわれた状態でS3に保管されていて、いつでもデプロイすることができます。

Elastic BeanstalkとJavaの関係

TomcatとGlassFish

Elastic Beanstalkで選択できるJavaのプラットフォームは、リリース当初からサポートされているTomcatと、Dockerコンテナとして提供されるGlassFishの2種類です。

新たにWebサービスを開発するのであればGlassFishを選択しましょう。その理由は3つあります。

1つめは、GlassFishはJava EEのすべての機能を提供しているのに対し、Tomcatはサーブレットをはじめとした Java EEの一部の機能だけの提供にとどまっていることです。つまりGlassFishは、Tomcatの機能をほぼ包含しているのです。

2つめに、パフォーマンスの面でもGlassFishは優れていることが挙げられます。筆者が行ったGETリクエストを送るだけの単純な負荷テストだけでもGlassFishはTomcatのおよそ4倍ものスループットを記録しました。

3つめは、GlassFishはDockerコンテナ上で動いているので、Dockerのしくみを使ってカスタマイズを行えば、ローカルマシンでもElastic Beanstalkと同じ環境を再現して動作確認ができます。

これらの理由からJavaのプラットフォームは既にあるTomcatに依存したアプリケーションを移行するのでなければGlassFishを選択するのが無難でしょう。

もちろん独自にDockerイメージを作ればPlay Framework[注3]やSpring Framework[注4]などほかのアプリケーションサーバでもWebサービスを公開できます。

Dockerとは?

さて、GlassFishはDockerコンテナとして提供されていると書きましたが、Dockerとはどのようなものか簡単に説明しておきましょう。

Dockerはコンテナ型の仮想化ソフトウェアでDocker社によりオープンソースソフトウェア(OSS)として提供されています。

仮想化ソフトウェアにはいくつかの種類があり、たとえばEC2はXenというハイパーバイザ型の仮想化ソフトウェアを使っていて、物理

注3) https://www.playframework.com/
注4) http://projects.spring.io/spring-framework/

サーバを複数の仮想サーバとして動作させています。それぞれの仮想サーバは物理サーバと同じような振る舞いをするので、ユーザは使い勝手を損なうことなくリソースの有効活用ができるという利点があります。

それに対しDockerは**図2**のように**Dockerエンジン**と呼ばれる仮想化ソフトウェアがホストOS上の1つのプロセスとして動作し、**Dockerコンテナ**と呼ばれる仮想環境がDockerエンジン上で動作する構成のため、ほかの仮想化技術と比較して起動やリソースアクセスのオーバーヘッドが少ないことが特徴です。

Dockerは**Dockerイメージ**と呼ばれるアプリケーションや設定ファイルなどが含まれたパッケージをDockerエンジンにデプロイしてDockerコンテナとして動作させます。

Elastic BeanstalkはWebサービスとアプリケーションサーバをパッケージングしたDockerイメージを生成しDockerコンテナとしてデプロイしています。

開発に便利なMavenプラグイン

Elastic BeanstalkにWebサービスをデプロイする方法として、マネジメントコンソールから手動でデプロイする方法やebコマンドを使う方法などいくつかありますが、Javaのビルドツール Mavenにはbeanstalk-maven-plugin[注5]というプラグインがあります。

このプラグインはElastic Beanstalk APIのラッパーです。アプリケーションや環境の作成、アプリケーションアーカイブのアップロード、環境のスワップなどが行えます。

環境のスワップとは、稼働中の環境の他に新たにデプロイするアプリケーション用としてもう1つ環境を準備し、それぞれの環境に割り当てられたCNAMEをスワップすることで、ダウンタイムなしでアプリケーションをバージョンアップする方法です。

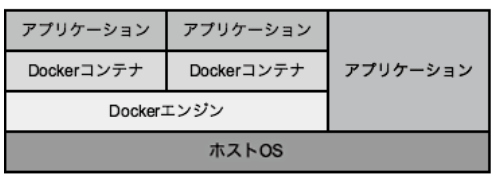
Mavenのビルドライフサイクルで Elastic Beanstalkの操作が行えるため、テストが通ったwarをアップロードし、作成した環境にデプロイしてから稼働中の環境とスワップするといった使い方ができます。

またDocker上で動いているGlassFish用のWebサービスの場合、docker-maven-pluginが役立ちます。

このプラグインもその名のとおりDocker用のプラグインです。Mavenのビルドライフサイクルで Dockerイメージの作成が可能になるので、より本番環境に近い形でテストを行うことができるので活用しましょう。

注5）http://beanstalker.ingenieux.com.br/beanstalk-maven-plugin/

▼図2　Dockerのアーキテクチャ



Elastic Beanstalkを使ってみる

サンプルアプリケーションを動かす

Elastic Beanstalkにはそれぞれのプラットフォームにサンプルアプリケーションが用意されています。まずはこのアプリケーションを動かしてみましょう。

AWS Management Consoleにログインしたあとにメニューから Elastic Beanstalkを選択すると**図3**のようなホーム画面[注6]が表示されます。ページ右上の[Create New Application]をクリックして、次のApplication Information画面でApplication nameに任意のアプリケーション名を入力します。

次のNew Environment画面では環境の選択を行

注6）https://console.aws.amazon.com/elasticbeanstalk/home

います。Web Server EnvironmentがWebサーバ環境枠、Worker Environmentはワーカ環境枠です。ここではWeb Server EnvironmentのCreate web serverボタンをクリックします。

Permissionダイアログ画面が開くのでIdentity and Access Management(IAM)プロファイルを作成、もしくは選択します。作成した場合はaws-elasticbeanstalk-ec2-roleというロール名になります。

Environment Type画面では起動するプラットフォームと環境タイプの選択をします。Predefine configurationはGlassFishを選択し、Environment typeはひとまずLoad balancing, auto scalingを選択しましょう。

Application Version画面では起動するアプリケーションアーカイブを選択します。すでにwarがある場合はUpload your ownを選択してwarをアップロードしましょう。ここではまずAWSが用意しているサンプルアプリケーションを起動するためSourceにSample applicationを選択します。Deployment Limitsはデフォルト値のままで大丈夫です。

Environment Information画面では環境名を指定します。Environment nameには任意の環境名を入力し、Environment URLはブラウザからのアクセスに使うURLのサブドメイン名を入力します。

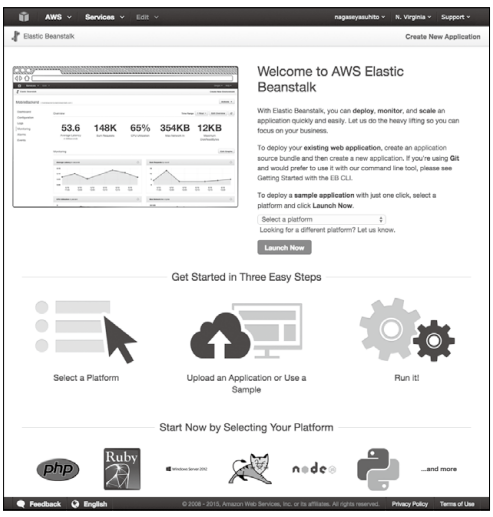
Additional Resourcesはデフォルトのままでよいでしょう。MySQLなどのリレーショナルデータベースを扱うAmazon Relational Database Service(RDS)や、仮想プライベートネットワークを扱うAmazon Virtual Private Cloud(VPC)などを利用する場合はチェックを入れましょう。

Configuration Detailsの画面ではEC2インスタンスの設定などを行います。Environment Tags画面ではとくに何も設定せずでよいでしょう。

Review Information画面でこれまで設定した内容を確認します。問題がなければLaunchボタンをクリックすれば環境の構築が始まります。環境の構築が始まるとEC2やELBに対する課金が始まります。5〜10分ほどでEnvironment URLで設定したURLにアクセスすると**図4**のようなページが表示されるはずです。

アプリケーションを終了するには、Elastic Beanstalkのホーム画面よりアプリケーション名の右側にあるActionsボタンからDelete Applicationをクリックすると終了できます。

▼図3　Elastic Beanstalkのホーム画面



Webサービスを開発する

Elastic Beanstalkに適したプロジェクトとは?

Elastic Beanstalkのしくみがわかったところで実際にWebサービスを開発してみましょう。今回は公開前のWebサービスにありがちなメールアドレスを事前登録するだけの簡単なWebサービスを例にします。ソースコードは

▼図4　サンプルアプリケーションの画面

https://github.com/nagaseyasuhito/gymnasterに公開しているのでぜひ手元でビルドして試してみてください。

非常に単純なWebサービスですが、JSF、JPA、CDIなどJava EEの基本的なコンポーネントを網羅しています。

基本的には通常のWebサービス開発と変わりませんが注意点がいくつかあります。

まずEC2のインスタンスはオートスケールにより自動的に増減します。そのためインスタンスのストレージデバイスにデータを保存するような仕様だと、インスタンスが減った際にデータも失われてしまうので気をつけてください。ユーザデータなどはRDSやDynamoDBへ、画像データなどはS3などに保存しましょう。

またオートスケールにより複数のインスタンスが起動しても、アプリケーションサーバがクラスタリングされるわけではないので、ステートレスな設計にするかインスタンスの増減時に適切にクラスタリングを行うようにアプリケーションサーバの設定を修正する必要があります。

データベースへの接続情報などはwarには含めず環境変数など外部から受け取れるようにしましょう。新規アプリケーション作成時にAdditional ResourcesでRDSをデータ環境枠として選択した場合はRDS_HOSTNAME、RDS_PORT、RDS_DB_NAME、RDS_USERNAME、RDS_PASSWORDといった環境変数にデータベースへの接続情報が格納されるので便利です。

GlassFishのDockerfileを覗く

Elastic BeanstalkのGlassFishはDockerコンテナ上で動作していると書きましたが、ここではDockerfileと呼ばれるDockerイメージを生成するためのスクリプトを覗いてみましょう。

Elastic Beanstalkで使われているGlassFishのDockerイメージはDocker Hub[注7]という公式のDockerレジストリで公開されていて[注8]、ソースコードはGitHub[注9]で管理されています。

このリポジトリにはJDK7で動くGlassFish4.0と、JDK8で動くGlassFish4.1のDockerイメージがあります。ここではGlassFish4.1のものを例に解説します。

リスト1は4.1-jdk8ディレクトリにある

注7) https://registry.hub.docker.com/
注8) https://registry.hub.docker.com/u/amazon/aws-eb-glassfish/
注9) https://github.com/aws/aws-eb-glassfish-dockerfiles

▼リスト1　GlassFish4.1用のDockerfile

```
FROM        java:8-jdk

ENV         JAVA_HOME         /usr/lib/jvm/java-8-openjdk-amd64
ENV         GLASSFISH_HOME    /usr/local/glassfish4
ENV         PATH              $PATH:$JAVA_HOME/bin:$GLASSFISH_HOME/bin

RUN         apt-get update && ¥
            apt-get install -y curl unzip zip inotify-tools && ¥
            rm -rf /var/lib/apt/lists/*

RUN         curl -L -o /tmp/glassfish-4.1.zip http://download.java.net/glassfish/4.1/release/ラ
glassfish-4.1.zip && ¥
            unzip /tmp/glassfish-4.1.zip -d /usr/local && ¥
            rm -f /tmp/glassfish-4.1.zip

EXPOSE      8080 4848 8181

WORKDIR     /usr/local/glassfish4

# verbose causes the process to remain in the foreground so that docker can track it
CMD         asadmin start-domain --verbose
```

GlassFish4.1用のDockerfileです。

まずFROMですが、ここにDockerイメージ名を指定して別のDockerイメージを継承できます。java:8-jdkはその名のとおりJDK8がインストールされたDockerイメージなのでGlassFishが必要な今回のケースの継承元としてはぴったりのDockerイメージです。

ENVは環境変数を定義します。JAVA_HOMEやGlassFishのホームディレクトリ、PATHにGlassFishの管理コマンドasadminの存在するパスの追加などを行っています。

RUNは環境構築のためのコマンドを実行します。ここではapt-getで必要なパッケージのインストールと、GlassFishのアーカイブのダウンロードと展開をしています。

EXPOSEはDockerの特色の1つでDockerコンテナの外に公開するポート番号です。GlassFishはHTTP用に8080、HTTPS用に8181、管理コンソール用に4848を使うのでそれぞれ列挙しています。

WORKDIRはカレントディレクトリを指定します。

CMDはアプリケーションを起動するコマンドを指定します。GlassFishは、通常asadmin start-domainで起動させますが、Dockerはアプリケーションがフォアグラウンドで起動する必要があるため--verboseオプションを付けてフォアグラウンドで起動させています。

デプロイのしくみ

先述したDockerイメージはGlassFishを起動するだけでwarのデプロイなどは行っていません。Elastic Beanstalkはアプリケーションのデプロイ時にアプリケーションアーカイブをGlassFishにデプロイしたDockerイメージを生成してからDockerコンテナを起動します。**リスト2**はその際に使われる4.1-jdk8-aws-eb-onbuildのDockerfileです。

先ほどのGlassFishがインストールされたDockerイメージを継承し、CMDではなくENTRYPOINTでglassfish-start.sh(**リスト3**)というシェルスクリプトを使ってGlassFishの起動とwarのデプロイを行っています。

Elastic BeanstalkはデプロイするアプリケーションアーカイブをVar/appに展開します。glassfish-start.shは/var/app以下をzipに圧縮したwar相当のアーカイブをasadminコマンドを使ってGlassFishにデプロイします。

最後にinotifywaitコマンドでPIDファイルを監視し、削除されるまで処理をブロックします。こうすることで管理コマンドなどでアプリケーションサーバが終了するまでDockerコンテナを起動させ続けます。

GlassFishのカスタマイズ

このようにElastic BeanstalkのGlassFishはDockerと密接に関わっていて簡単にカスタマイ

▼リスト2　4.1-jdk8-aws-eb-onbuildのDockerfile

```
FROM        glassfish:4.1-jdk8

WORKDIR     /var/app

ADD         glassfish-start.sh /

ONBUILD     ADD . /var/app/

CMD         []
ENTRYPOINT  ["/glassfish-start.sh"]
```

▼リスト3　glassfish-start.sh

```
#!/bin/sh

PID_FILE=$GLASSFISH_HOME/glassfish/➥
domains/domain1/config/pid

# when deploying a directory, Glassfish ➥
expect all submodules to be extracted
# which is usually not the case for EARs
# zip app back into a bundle and let ➥
Glassfish handle it
rm -f /var/app/Dockerfile
rm -f /var/app/Dockerrun.aws.json
zip /var/app.zip -r .

asadmin start-domain
asadmin deploy --contextroot / --name ➥
current-app /var/app.zip

inotifywait -qq -e delete_self $PID_FILE
```

ズできるしくみになっています。Dockerfileもシンプルな構文なので容易に習得できるでしょう。

Elastic Beanstalkで用意されているDockerベースのプラットフォームは、アプリケーションアーカイブにDockerfileが含まれていた場合、そのDockerfileを使ってDockerイメージの生成を行います。つまりGlassFishの場合はwarにDockerfileを含めるだけで簡単にカスタマイズができるのです。

たとえば起動前にJDBCドライバをダウンロードしたり、asadminコマンドでGlassFishの設定を変更するなど柔軟に対応できます。Dockerfileで記述するのでローカルマシン上のDockerエンジンで動作すれば、理論上Elastic Beanstalkでも同じように動作するので「本番環境ではなぜか動かない」といったことが大幅に少なくなります。このようにアプリケーションサーバの設定も含めてパッケージングできることがDockerの大きな魅力の1つですね。

ここではデータベースと連携するための設定方法を例として取り上げます。GlassFishはデータベースと連携するために、

●JDBCドライバのダウンロード
●コネクションプールの設定

が必要です。今回はH2 Database Engine[注10]というインメモリデータベースを例に実現してみましょう。

まずDockerfileをMavenビルド時にwarファイルに含める設定をします。src/main/dockerというディレクトリを作り、そこにDockerfileを作成します。このディレクトリをwarファイルに含めるため**リスト4**のようにmaven-war-pluginをproject/build/plugins以下にプラグインを定義します。

リスト5がカスタマイズするためのDockerfileです。先ほどのamazon/aws-eb-glassfish:4.1-jdk8-onbuild-3.5.1を継承します。

glassfish-start.shも修正するので改めてADDでリソースの追加を行います。またmaven-war-pluginでは個々のファイルのパーミッションは変更できないため、RUNでパーミッションを変更しておきます。

GlassFishは連携するデータベース用のJDBCドライバを$GLASSFISH_HOME/glassfish/domains/domain1/libに配置する必要があるため、wgetコマンドでMavenのセントラルリポジトリからH2 Database EngineのJDBCドライバをダウンロードします。

コネクションプールの設定はGlassFishが起動したあとに行う必要があるので、**リスト6**のようにglassfish-start.shに記述します。

asadmin start-domainでGlassFishを起動し、asadmin create-jdbc-connection-poolでコネクションプールを作成します。

asadmin create-jdbc-resourceでは作成したコネクションプールをJDBCリソースとしてJNDIで参照できるようにしましょう。

▼リスト4　DockerFileを含める例

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-war-plugin</artifactId>
  <version>2.6</version>
  <configuration>
    <webResources>
      <resource>
        <directory>src/main/docker</directory>
        <targetPath></targetPath>
        <filtering>true</filtering>
      </resource>
    </webResources>
  </configuration>
</plugin>
```

▼リスト5　H2 Database EngineのJDBCドライバをダウンロード

```
FROM amazon/aws-eb-glassfish:4.1-jdk8-➥
onbuild-3.5.1

ADD glassfish-start.sh /

RUN chmod 755 /glassfish-start.sh
RUN wget -P $GLASSFISH_HOME/glassfish/➥
domains/domain1/lib http://central.➥
maven.org/maven2/com/h2database/h2/➥
1.4.187/h2-1.4.187.jar
```

注10）http://www.h2database.com/html/main.html

このようにjarのダウンロードなどGlassFish起動前に必要な処理はDockerfileに、asadminコマンドなどGlassFish起動後に必要な処理はglassfish-start.shに記述すると良いでしょう。

ローカルマシンで動作確認するには

Elastic Beanstalkへのデプロイは早くても数分かかるため開発時の動作確認には不向きです。効率よく開発を進めるためにローカルマシンにDockerをインストールして動作確認をする環境を整えましょう。

最近のLinuxディストリビューションであればDockerパッケージが用意されているのでAPTやyumなどの管理コマンドでインストールするだけですぐ使えるようになります。

Mac OS XやWindowsであればBoot2docker[注11]というVirtualBox[注12]上で動作するLinuxディストリビューションをインストールしましょう。

Mac OS Xの場合はHomebrew[注13]を使うと、DockerをはじめVirtualBox、Boot2dockerなど必要なものはコマンドラインからインストールできます(**図5**)。インストールしたら**図6**のようにコンテナを初期化して起動します。

Boot2dockerを起動すると環境変数の設定を促すメッセージが表示されるので.bash_profileに記述するなど適宜設定してください。docker infoを実行してDockerの情報が正常に出力されればDockerの準備は完了です。

Dockerイメージを作る

それではカスタマイズしたGlassFishをローカルマシンのDockerで動かしてみましょう。Mavenにはビルド時にDockerイメージを生成するdocker-maven-pluginというプラグインがあるのでproject/build/plugins以下に定義します(**リスト7**)。

Dockerイメージにはビルドされたソースコー

注11) http://boot2docker.io/
注12) https://www.virtualbox.org/
注13) http://brew.sh/

▼リスト6　JDBCドライバの設定を追加したglassfish-start.sh

```
#!/bin/sh

PID_FILE=$GLASSFISH_HOME/glassfish/domains/domain1/config/pid

# when deploying a directory, Glassfish expect all submodules to be extracted
# which is usually not the case for EARs
# zip app back into a bundle and let Glassfish handle it
rm -f /var/app/Dockerfile
rm -f /var/app/Dockerrun.aws.json
zip /var/app.zip -r .

asadmin start-domain

asadmin create-jdbc-connection-pool ¥
  --datasourceclassname org.h2.jdbcx.JdbcDataSource ¥
  --restype javax.sql.XADataSource ¥
  --property url=jdbc¥¥:h2¥¥:mem¥¥: ¥
  ${project.artifactId}

asadmin create-jdbc-resource ¥
  --connectionpoolid ${project.artifactId} ¥
  jdbc/${project.artifactId}

asadmin deploy --contextroot / --name current-app /var/app.zip

inotifywait -qq -e delete_self $PID_FILE
```

ドなどを含める必要があります。そのためmvn clean package docker:buildのようにpackageゴールのあとにdocker:buildゴールを指定してMavenを実行してください。Mavenの実行に成功するとDockerイメージが生成され、docker imagesコマンドでビルドしたDockerイメージの情報が表示されます(**図7**)。

Dockerコンテナの起動はdocker runコマンドで行います。-pはポートフォワーディングのオプションでDockerコンテナ内のポートとDockerエンジンのホストのポートをマッピングしています。上記の例だとDockerイメージのIDは0b216a8c1113なので引数の最後に指定して起動します(**図8**)。

Boot2dockerを使っている場合は**リスト8**のようにVirtualBoxのVMとホストのマッピングをする必要があります。

ブラウザでhttp://localhost:18080/にアクセスすると**図9**のような画面が表示されます。テキストフィールドにメールアドレスを入力しPre-Registerボタンをクリックすると画面が遷移して、データベースに新たに行が追加されます。

Dockerコンテナを終了する場合はdocker psコマンドでコンテナのIDを調べてdocker killコマンドで終了させます。

▼図5　インストール各種

```
# Homebrewのインストール
$ ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"

# VirtualBoxをインストールするためHomebrew-Caskをインストール
$ brew install caskroom/cask/brew-cask

# VirtualBoxのインストール
$ brew cask install virtualbox

# DockerとBoot2dockerのインストール
$ brew install docker boot2docker
```

▼図6　コンテナの起動

```
# Boot2dockerの初期化
$ boot2docker init

# Boot2dockerの起動
$ boot2docker up
```

▼リスト7　docker-maven-pluginの設定

```
<plugin>
  <groupId>com.spotify</groupId>
  <artifactId>docker-maven-plugin</artifactId>
  <version>0.2.6</version>
  <configuration>
    <!-- Dockerイメージの名前 -->
    <imageName>${project.groupId}/${project.artifactId}:${project.version}</imageName>

    <!-- Dockerイメージ化するディレクトリ -->
    <dockerDirectory>${project.build.directory}/${project.build.finalName}</dockerDirectory>
  </configuration>
</plugin>
```

▼図7　Dockerイメージの情報

```
$ docker images
REPOSITORY                          TAG                 IMAGE ID            CREATED             VIRTUAL SIZE
com.github.nagaseyasuhito/gymnaster 0.0.1-SNAPSHOT      0b216a8c1113        About an hour ago   768.5 MB
amazon/aws-eb-glassfish             4.1-jdk8-onbuild-3.5.1 772b9da45423     5 months ago        767.5 MB
```

Elastic Beanstalkにデプロイする

手動でデプロイする

まずは手動でWebサービスをデプロイしてみましょう。mvn clean packageコマンドでwarを生成したら、Elastic Beanstalkの環境のダッシュボード画面の中央にUpload and Deployというボタンがあるのでクリックしましょう(**図10**)。

開いたダイアログボックスのUpload applicationはwarを選択し、Version labelにはアプリケーションバージョンを識別する文字列を入力してください。Deployボタンをクリックするとデプロイが始まり、Healthのステータスが Updatingになります。

デプロイが完了するとHealthがふたたびGreenになります。ブラウザからアクセスしてみるとローカルマシンで動かしたときと同じ画面が表示されます。

デプロイ時のログは画面左側のメニューのLogsからダウンロードして確認できます。Request LogsボタンからFull Logsをクリックすると一覧にダウンロードできるログのzipファイルが追加されます。Downloadをクリックしてログをダウンロードしてみましょう。zipファイルを展開すると/var/log以下のログが格納されているのがわかります。

GlassFishのログは/var/log/eb-docker/containers/eb-current-appに出力されるので、不具合が起こった場合などはインスタンスにSSHで接続しなくてもログを確認できます。

Mavenプラグインでデプロイする

それではElastic BeanstalkへのデプロイをMavenのビルドライフサイクルに含めるためbeanstalk-maven-pluginを使ってみましょう。まずpom.xmlのproject/build/plugins以下にプラグインを定義します(**リスト9**)。

そしてデプロイに関する設定をpropertiesに定義します。

beanstalk.versionLabelはElastic Beanstalkのアプリケーションバージョンです。MavenのPOMのバージョンと揃えておくとわかりやすいので${project.version}のようにMavenプロパティを指定しましょう。

beanstalk.environmentNameは作成した環境名を指定してください。ここでは${project.artifactId}を指定しています。

beanstalk.s3Bucketには${project.artifactId}を定義していますが、S3のバケット名は**S3全体でユニークでなくてはならない**ので必要に応じて書き換えてください。

▼図8　Dockerイメージの IDで起動

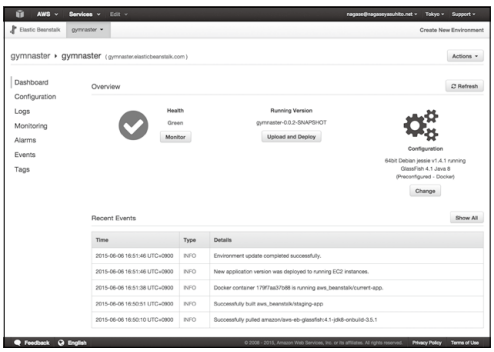
```
docker run -p 8080:8080 0b216a8c1113
```

▼リスト8　ホスト(ポート18080)とVirtualBox(ポート8080)間のポートフォワーディングの設定

```
VBoxManage controlvm "boot2docker-vm" natpf1 ↵
"glassfish,tcp,127.0.0.1,18080,,8080"
```

▼図9　デプロイしたアプリケーションの画面

▼図10　環境のダッシュボード



デプロイにはAWSのアクセスキーとシークレットキーが必要です。これらのキーをまだ生成していない場合はIAMで生成しておいてください。

キーは環境変数でMavenに渡すのが簡単ですが、シェルの履歴に残ってしまうので注意しましょう。安全に取り扱いたい場合はMavenのsettings.xmlに暗号化して記述しておく方法[注14]もあります。

リスト10のようにMavenを実行してみましょう。

まとめ

いかがだったでしょうか？　そのしくみ自体はシンプルなElastic Beanstalkですが、監視やデプロイ、スケールアウトの機能などが準備されているため運用の負荷が大幅に削減できることを感じられたのではないでしょうか。

充実したMavenのプラグインのおかげでJavaベースのWebサービスをすぐにデプロイできるうえ、Dockerでシステム構成をイミュータブルに保てるので開発環境でも本番と同じ構成の再現が簡単にできます。一見制約にもみえるこのしくみのおかげで、Webサービスを堅牢に育てていく大きなメリットとなるでしょう。

ぜひ、みなさんもElastic Beanstalkを実際に試し、スピーディにWebサービスを公開する方法を体験してください。SD

▼リスト10　Maven設定例

```
AWS_ACCESS_KEY_ID=[アクセスキー] ¥
AWS_SECRET_KEY=[シークレットキー] ¥
mvn clean package ¥
  beanstalk:upload-source-bundle ¥
  beanstalk:create-application-version ¥
  beanstalk:update-environment
```

▼リスト9　beanstalk-maven-pluginの設定

```
<build>
  <plugins>
    <plugin>
      <groupId>br.com.ingenieux</groupId>
      <artifactId>beanstalk-maven-plugin</artifactId>
      <version>1.4.0</version>
    </plugin>
  </plugins>
</build>

<properties>
  <!-- アプリケーションバージョン -->
  <beanstalk.versionLabel>${project.version}</beanstalk.versionLabel>

  <!-- 環境名 -->
  <beanstalk.environmentName>${project.artifactId}</beanstalk.environmentName>

  <!-- サブドメイン名 -->
  <beanstalk.cnamePrefix>${project.artifactId}</beanstalk.cnamePrefix>

  <!-- 利用するリージョン -->
  <beanstalker.region>ap-northeast-1</beanstalker.region>

  <!-- アプリケーションアーカイブをアップロードするS3のバケット名 -->
  <beanstalk.s3Bucket>${project.artifactId}</beanstalk.s3Bucket>
</properties>
```

注14) http://beanstalker.ingenieux.com.br/beanstalk-maven-plugin/security.html

短期集中連載

プログラマに優しい現実指向JVM言語

Kotlin入門

第5回 null安全

前回はKotlinにおけるクラスとその周辺の機能、文法を紹介しました。
今回はKotlinのユニークな機能であるnull安全について解説します。

Author 長澤 太郎(ながさわ たろう) Twitter @ngsw_taro Mail taro.nagasawa@gmail.com

背景

java.lang.NullPointerExceptionは、Javaプログラマがよく出会う例外でお馴染みです。オブジェクトが必要な場面でnullを使用してしまうことでスローされる例外です。具体的にはString型の変数にnullを代入しておき、その変数に対してlengthメソッドを呼び出した場合にNullPointerException、「ぬるぽ」の愛称で親しまれる例外(以下、NPEと表記)が投げられます。

nullは、値が存在しないときに使用されます。たとえば、指定したIDを持ったユーザが存在しないときにfindUserByIdのようなメソッドがUserクラスのインスタンスを返す代わりにnullを返すと言った具合です。

このような観点で、nullは便利に働きます。しかしこのnullのおかげで筆者たちは見たくもない例外、NPEと遭遇するはめになるのです。適切にnullチェック、つまりif文などでnullでないことを確認すれば回避できるのですが、なぜそれができないのでしょうか。

nullを返さないことがわかっているメソッドの戻り値に対してnullチェックはしないのが普通だと思います。ここが重要なのですが、仕様的にnullを返し得ないメソッドがあっても、Javaのコードでそのことを保証することはできません。Javaにおいて、変数やメソッドの戻り値はいつでもどこでもnullになり得ます。つまりnullチェックをすべきものと、nullチェックが不要なものがごちゃまぜになっているので、うっかりNPEを招いてしまうのです。

nullと上手に付き合う方法

nullかもしれないものと、nullではないものを区別するための方法が世の中にはいくつかあります。

メソッドシグネチャの工夫

原始的な方法です。nullを返す可能性があるメソッドのシグネチャを工夫して、プログラマに注意喚起します。たとえばgetNameOrNullのような名前のメソッドです。名前を見ればnullが返されるかもしれないことに気づくわけです。

静的解析ツール

メソッドにアノテーションを付けて、静的解析ツールに指摘してもらう方法です。getNameメソッドがnullを返すかもしれない場合には@Nullable String getName() {...}と記述し、nullを返し得ない場合には@Nonnull String getName() {...}と記述します。

型で表現

存在しない可能性のある値を表現するためにnullの代わりに新しく定義した型を使う方法です。具体的にはJava SE 8で導入されたjava.util.Optionalクラスです。値が存在し

▼リスト1　誰にもnullは止められない!

```
// Javaコードです
@Nonnull
Optional<String> getName() {
  return null;
}
```

▼リスト2　NotNullの変数

```
var a: String = "Hello"
a = "Goodbye"
a = null // ここでコンパイルエラー
```

ないときにはOptional#emptyで返されるオブジェクトを使用し、値が存在するときはその値をOptional#ofの引数に渡してラップします。Optional型とそれ以外の型で、存在しないかもしれない値と絶対に存在する値の区別が容易になるだけでなく、Optionalにはさまざまな便利なメソッドが提供されています。

Kotlinのnull安全

静的解析ツールやOptionalを使うことはとても良いことです。しかし繰り返しになりますが、Javaにおいて変数やメソッドの戻り値はいつでもどこでもnullになり得ます。すべてを台無しにするコードをご覧ください(**リスト1**)。

そこでKotlinのnull安全機構の登場です。Kotlinではnullの可能性のある値(以下Nullable)とnullではない値(以下NotNull)の区別を言語組込みの機能としてサポートします。

Optionalを使う方法とは異なり、新しいインスタンスの生成(とGC)が不要なのでその分のオーバーヘッドがなく、Androidなどのリソースが限られた環境で有利のようです。

基本的な使い方

前回、前々回と使用してきたごく普通の変数初期化と再代入を**リスト2**に示します。変数aはString型です。ここでは型を明示していますが省略しても問題ありません。varキーワードにより変更可能な変数として宣言しているので"Goodbye"を代入できます。しかしその次の行のnullを代入する部分でコンパイルエラーが起こります。変数aはNotNullとして宣言されているのでnullの代入をコンパイラが許しません！　逆を言えば、変数aは常にnullではないと安心して使用できます。

では、nullを代入できるNullableな変数はどのように宣言すれば良いのでしょうか。簡単です。通常の型アノテーションのあとに?を置くだけです。**リスト3**を見てください。変数bの型アノテーションがString?になっています。これは「nullが代入可能なString型」と読めます。2行目でnullを代入していますが、コンパイルに成功します。今回の場合、変数bの型アノテーションを省略できないことに注意してください。なぜなら、"Hello"はString?ではなくStringとして推論されるからです。

KotlinではNullableとNotNullを明確に区別することがわかりました。NotNullの変数にはnullが入ってこないので、これを扱ううえでNPEは起こらないので安全です。Nullableの変数にはnullが入る可能性があるのでNPEが起こりそうです。ということでNPEを起こしてみましょう(**リスト4**)。

String?な変数sにnullを代入して初期化しています。このsにlengthメソッドを呼び出してNPEを起こそうとしています。が、実際にはコンパイルエラーとなります。KotlinはNPEを起こさせたくないので、NPEの可能性のある操作をコンパイルエラーとするのです。Nullableに対するメソッド、プロパティアクセスは禁止されています。

しかし現実問題、Nullableのメンバにアクセ

▼リスト3　Nullableの変数

```
var b: String? = "Hello"
b = null
```

▼リスト4　NPEを起こしたい

```
val s: String? = null
s.length() // ここでコンパイルエラー
```

▼リスト5　nullチェックするとNotNullになる

```
if(s != null) {
  println(s.length())
}
```

スできないのは不便どころか使い物になりません。もちろんアクセスする方法が用意されています。その方法とは、nullチェックすることです！

リスト5のように変数sがnullでないことを確認すると、それが保証される範囲内(ifのブロック内)でsをNotNullとして扱えるようになります。sに"Hello"が代入されていれば**リスト5**を実行すると「5」と出力されます。

Nullableの便利な機能

KotlinのNullableを使ううえで必要な知識は、前節までの内容でまかなえます。ですがnullチェックをするコードを書くのは退屈で面倒な作業ですので、KotlinはNullableを便利に使える機能を提供しています。

安全呼び出し

リスト5ではNullableのメソッドを呼び出すためにnullチェックを行いました。これを簡潔に記述できるように、安全呼び出し(Safe Call)と呼ばれるしくみがあります。

リスト6の最初の行ではString?型の変数sのlengthメソッドを安全に呼び出しています。通常のメソッド呼び出しと異なるのは、ドットの前に?を置くことです。これにより安全呼び出しとなり、Nullableのメソッドを安全に、すなわちNPEの心配なく呼び出すことができます。

仮にsがnullだった場合には、メソッドを呼び出さずnullを返します。1行目の安全呼び出し方式と、2行目のnullチェック方式は等価です。

安全呼び出しはメソッドチェーンを形成したい場合などではとくに効果を発揮します(**リスト7**)。foo()?.bar()?.baz()のように記述した場合、途中でnullが返されても安全呼び出しがチェーンして最終的にnullが返されるだけです。

▼リスト6　安全呼び出し

```
// 安全呼び出し方式
val length1: Int? = s?.length()
// nullチェック方式
val length2: Int? = if(s != null) s.length() else null
```

デフォルト値

デフォルト値、nullだった場合に使用する値、を簡単に指定できます。Nullableのあとに続けてエルビス演算子(?:)とデフォルト値を記述します。**リスト8**を見てください。s?.length()は安全呼び出しにより、文字列の長さかnullが返されます。nullが返された場合、デフォルト値として0を使用するように指定しています。nullでない場合、デフォルト値は評価されません。

禁断の!!演算子

最後に紹介するのは禁断の演算子です。

!!演算子は、Nullableの直後に置き、強制的にNotNullへの変換を試みます。**リスト9**ではString?である変数sを!!演算子により強制的にStringに変換しています。

この例はたまたまうまく行きました。しかし**リスト10**は実行時に例外を投げてクラッシュします。nullであるものに対して!!演算子を使うとKotlinNullPointerExceptionを投げます。

nullの場合に例外が投げられる。これって結局今までと同じです。!!演算子を使用したくなったらnullチェックや安全呼び出し、デフォルト値の使用を検討してください。どうし

▼リスト7　安全呼び出しでメソッドチェーン

```
// 安全呼び出し方式
val result1 = foo()?.bar()?.baz()
// nullチェック方式
val foo = foo()
val result2 = if(foo != null) {
  val bar = foo.bar()
  if(bar != null) bar.baz() else null
} else {
  null
}
```

▼リスト8　デフォルト値

```
// エルビス演算子
val length1: Int = s?.length() ?: 0
// nullチェック
val len: Int? = s?.length()
val length2: Int = if(len) len else 0
```

▼リスト9　強制的にNotNull化

```
val s: String? = "Hello"
println(s!!.length()) // => 5
```

▼リスト10　実行時例外が起こる

```
val s: String? = null
println(s!!.length())
```

▼リスト11　自作filterNotNull

```
fun <T> filterNotNull(list: List<T?>): List<T> =
  list.filter { it != null }
      .map    { it!! }
val a: List<String?> = listOf("foo", null, "bar")
val b: List<String> = filterNotNull(a)
println(b) // => ["foo", "bar"]
```

▼リスト12　標準拡張関数let

```
fun <T, R> T.let(f: (T) -> R): R = f(this)
```

▼リスト13　letの使用例

```
5.let {
  println(it * 3) // => 15
}
```

てもやむを得ない場合に限り!!演算子を使いましょう。その際にはコメントとして使った理由や経緯を記しておくと良いでしょう。

1つ、!!演算子を使いたくなるような例を示します。要素としてnullを許容するリスト(List<T?>)から、nullを排除してNotNullな要素だけの新しいリスト(List<T>)を得るための関数がほしいとします。その場合の実装は**リスト11**のようになります。

list.filter { it != null }により、要素がnull以外のものに絞り込みます。しかしリストの型は依然List<T?>のままです。そこで次のmap { it!! }で強制的に要素の型をT?からTの変換しています。

実際には!!演算子を使用せずに実装できますし、filterNotNullはコレクションの標準メソッドとして提供されています。

標準拡張関数 let

Kotlinの標準ライブラリとして、任意の型に対してletという拡張関数[注1]が提供されています(**リスト12**)。

letは、レシーバとなるオブジェクト(this)に、引数に取る関数(f)を適用しているだけです。使用例を**リスト13**に示します。

関数リテラルに渡る唯一の引数(暗黙の変数it)はletのレシーバと同一オブジェクトですので、この例ではitは5です。

さて、この単純な関数は何の役に立つのでしょうか。ずばり、NullableにNotNullな引数を取る関数を適用するときに便利です。具体的に見て行きましょう。

リスト14で、NotNullなIntを引数に取る関数succを定義しました。Nullableである変数aを、この関数の引数として渡したいです。しかしNullableとNotNullの違いがあるので、素直には渡せません。succは関数であり、Intのメソッド(あるいは拡張関数)ではないので安全呼び出しも使えません。となるとifでnullチェックしてNullableを安全にNotNullとして扱えるようにするしかありません。

ここでletの登場です。まず、letは任意の型の拡張関数ですからa?.let {...}のような安全呼び出しができます。そしてletの引数となる関数(**リスト12**におけるf)が受け取る引数は、レシーバと同じ型のNotNullです。レシーバがnullのときはlet拡張関数は呼び出せないので理に適っていますね。

ということでletを使うことで**リスト14**のsucc適用の部分は**リスト15**のように書き換えられます。ちなみに**リスト15**はもっと簡潔に

注1) 正確にはpublic inline指定されていますが便宜上省略しています。

▼リスト14　NotNullを受け取る関数を適用したい

```
// NotNullを受け取る関数
fun succ(n: Int): Int = n + 1
// Nullableな変数
val a: Int? = 3
val b: Int? =
  if(a != null) succ(a)
  else null

println(b) // => 4
```

記述できます。第3回で紹介した定義済み関数の参照を得るスタイルで記述するとリスト16のようになります。

Javaですでに Optional に親しんでいる人には、let は Optional における map、flatMap、ifPresent の3役こなす存在だと思っていただけると理解しやすいでしょう。

Javaからコードを呼び出す

Kotlin から Java コードを呼び出す場合、NotNull/Nullable の扱いが特殊になります。リスト17のような hello メソッドを Kotlin から呼び出して結果を変数に代入する際、`val msg = Sample.hello("World")` のように型アノテーションを省略した場合、変数 msg は NotNull としても、Nullable としても扱える型となります。そのため `msg.length()` も、`msg?.length()` もコンパイラは許可します。もし msg が null であれば `msg.length()` は NPE を起こします。

型アノテーションを明示することで NotNull/Nullable を表明できます。`val msg: String = Sample.hello("World")` とすれば、msg は NotNull として扱えます。もし `String.hello("World")` が null を返すようなことがあれば、NotNull を表明している msg に代入する時点で java.lang.IllegalStateException を投げます。これは例外を投げるタイミングが早いという観点で、msg の型アノテーションを省略したうえで NotNull として扱うより幾分マシです。安全側に倒すなら `val msg: String?` と

▼リスト15　letを使うとすっきりする

```
val b: Int? = a?.let { succ(it) }
```

▼リスト16　関数参照でよりすっきり

```
val b: Int? = a?.let(::succ)
```

▼リスト17　Javaからコードを呼び出す

```
// Javaコードです
public class Sample {
  public static String hello(String name) {
    return "Hello, " + name + "!";
  }
}
// Kotlinコードです
val msg = Sample.hello("World")
println(msg.length()) // => 13
```

Nullable を表明するとよいでしょう。

ただし、Java のコンストラクタ、int や boolean などのプリミティブ型を返すメソッドが返す値は、Kotlin では常に NotNull となります。

まとめ

今回は Kotlin のユニークな機能である null 安全に関して、そのモチベーションから文法、活用方法まで紹介しました。

少なくとも現時点の Java では NotNull と Nullable を厳格に区別することはできません。Kotlin ではこの区別を厳格にすること、さらに Nullable の扱い方をとことん注意深くすることで NPE との決別を図っています。Nullable はそのままでは扱えないことが多く不便に思われがちですが、null チェック後に NotNull として使えることや、安全呼び出し、デフォルト値など言語組込みのサポートにより簡単に扱えるようになっています。標準ライブラリとして提供されている let 拡張関数を組み合わせればさらに null 安全ライフが楽しいものとなります。

いよいよ次回は Kotlin による Android プログラミングについて解説します。SD

バックナンバーのお知らせ　BACKNUMBER

バックナンバー好評発売中！常備取り扱い店もしくはWebより購入いただけます

本誌バックナンバー（紙版）はお近くの書店に注文されるか、下記のバックナンバー常備取り扱い店にてお求めいただけます。また、「Fujisan.co.jp」（http://www.fujisan.co.jp/sd）や、e-hon（http://www.e-hon.ne.jp）にて、Webから注文し、ご自宅やお近くの書店へお届けするサービスもございます。

2015年7月号

第1特集
あなたにもできる!
ログを読む技術［セキュリティ編］

第2特集
黒い画面（tmux）の使い方
プロになるためのターミナル活用術

第3特集
6人の先駆者に訊く
スペシャリストになる方法

定価（本体1,220円+税）

2015年6月号

第1特集
新入さん歓迎特集
Git&GitHub入門

第2特集
OpenLDAPの教科書
ユーザ／ネットワーク管理の基本と活用例

一般記事
・Sambaによる Active Directoryの機能性と移行性を検証する

定価（本体1,220円+税）

2015年5月号

第1特集
テキスト処理ベーシックレッスン
手を動かしてデータを操ろう!

第2特集
ファイル共有自由在在
［徹底入門］最新・Sambaの教科書

特別付録
・3分間 HTTP& メールプロトコル基礎講座［特別篇］

定価（本体1,300円+税）

2015年4月号

第1特集
トラブルシューティングの極意
達人に訊く問題解決のヒント

第2特集
［最新］DNSの教科書
ネットワークを支える本物のインフラを学ぶ

特別付録
・3分間ネットワーク基礎講座［特別篇］

定価（本体1,300円+税）

2015年3月号

第1特集
カンファレンスネットワークの作り方

第2特集
いまからでも遅くない!
Hadoop超²入門

一般記事
・Cisco VIRLでネットワークシミュレーション［前編］
・Snappy Ubuntu Core

定価（本体1,220円+税）

2015年2月号

第1特集
Linux systemd入門
あなたの知らない実践技

第2特集
そろそろ、やめませんか?
なぜ「運用でカバー」がダメなのか?

一般記事
・Intel DPDK技術詳解
・これはなんて読む？ UNIX用語読み方指南
・Googleベンチャーズが提唱するデザインスプリントとはほか

定価（本体1,220円+税）

Software Design　バックナンバー常備取り扱い店							
北海道	札幌市中央区	MARUZEN & ジュンク堂書店　札幌店	011-223-1911	神奈川県	川崎市高津区	文教堂書店　溝の口本店	044-812-0063
	札幌市中央区	紀伊國屋書店　札幌本店	011-231-2131	静岡県	静岡市葵区	戸田書店　静岡本店	054-205-6111
東京都	豊島区	ジュンク堂書店　池袋本店	03-5956-6111	愛知県	名古屋市中区	三洋堂書店　上前津店	052-251-8334
	新宿区	紀伊國屋書店　新宿本店	03-3354-0131	大阪府	大阪市北区	ジュンク堂書店　大阪本店	06-4799-1090
	渋谷区	紀伊國屋書店　新宿南店	03-5361-3315	兵庫県	神戸市中央区	ジュンク堂書店　三宮店	078-392-1001
	千代田区	書泉ブックタワー	03-5296-0051	広島県	広島市南区	ジュンク堂書店　広島駅前店	082-568-3000
	千代田区	丸善　丸の内本店	03-5288-8881		広島市中区	丸善　広島店	082-504-6210
	中央区	八重洲ブックセンター本店	03-3281-1811	福岡県	福岡市中央区	ジュンク堂書店　福岡店	092-738-3322
	渋谷区	MARUZEN & ジュンク堂書店　渋谷店	03-5456-2111				

※店舗によってバックナンバー取り扱い期間などが異なります。在庫などは各書店にご確認ください。

デジタル版のお知らせ　DIGITAL

デジタル版Software Design 好評発売中！紙版で在庫切れのバックナンバーも購入できます

本誌デジタル版は「Fujisan.co.jp」（http://www.fujisan.co.jp/sd）と、「雑誌オンライン.com」（http://www.zasshi-online.com/）で購入できます。最新号、バックナンバーとも1冊のみの購入はもちろん、定期購読も対応。1年間定期購読なら5%強の割引になります。デジタル版はPCのほかにiPad／iPhoneにも対応し、購入するとどちらでも追加料金を払うことなく、読むことができます。

Erlangで学ぶ並行プログラミング

Author 力武健次技術士事務所 所長 力武 健次(りきたけ けんじ) http://rikitake.jp/

第5回 OTPのビヘイビアとgen_server

この連載ではプログラミング言語Erlangとその並行プログラミングについて紹介していきます。今回は、Erlang/OTPのプログラミング用フレームワークであるビヘイビア(behaviour)と、その典型例であるgen_server[注1]について紹介します。

OTPの今後のロードマップ

本題に入る前にErlang/OTPの今後のロードマップについて紹介します。17.x系の安定版は17.5.6.1[1]が6月25日にリリースされました。GitHubレポジトリでタグを指定することでソースコードからビルドできます(詳細は本連載第3回の注8で説明)。

6月24日にOTP 18.0がリリースされました。インストールの方法は本連載第1回の説明を参考にしてください。リリース番号は「18.0」となり、ダウンロードディレクトリ[注2]に一式用意されています[注3]。OTP TeamのErlang User Conference 2015の報告[2]では、今後は次の予定となるようです。

- 18.1……2015年9月～10月ごろ
- 18.2……2015年12月～2016年1月ごろ
- 18.3……2016年4月
- 19 ……2016年第2四半期

注1) "behaviour"はイギリス(および米国以外)での英語の綴りで、米国の綴りではbehavior(uがありません)ですが、どちらもErlangのコンパイラでは同様に使えます。筆者は普段米国の英語の綴り方をおもに使っているので、あえて専門用語としてuの入ったほうをErlangのコーディングでは使っています。また、gen_serverは筆者は「ジェンサーバ」と読んでいます。

注2) http://erlang.org/download/

注3) http://www.erlang.org/

OTPでの実行プログラム作成

Erlangは複数のモジュールをどう組み合わせるかについて高い自由度を持っています。しかし、現実に大規模な実行プログラムを作成するには、細部を共通化しておかないと、ささいな実装上の相違でやっかいなバグが入り込んだり、デバッグが難しくなったりします。そこでOTPでは実行プログラムをどう書くかについての指針[3]を定めており、開発者にもこれに準拠することを要請しています。この指針では、プログラムを次の構成で組むことを想定しています。

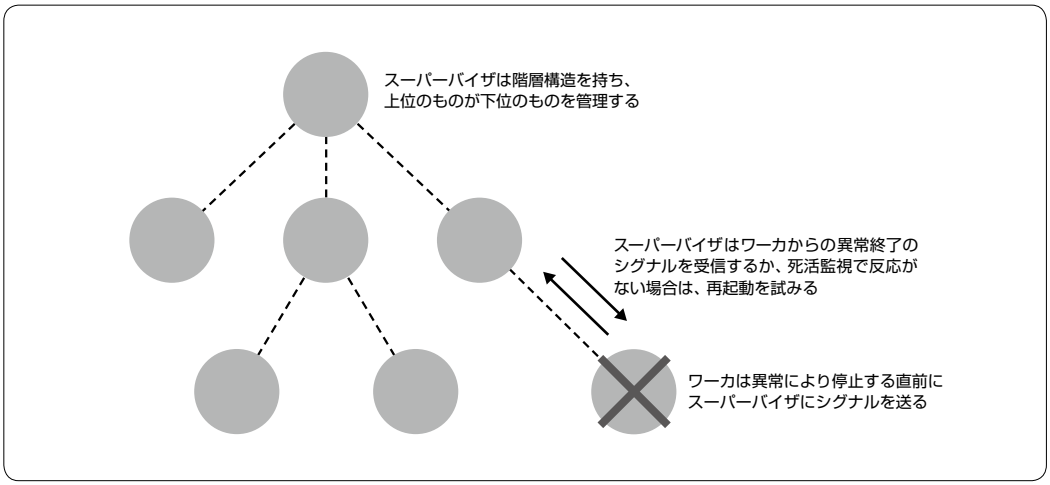
プロセス管理ツリー(図1)

プログラムを構成するプロセスについて、実際に処理をするワーカ(worker)と、ワーカを監視するスーパーバイザ(supervisor)に分けます。ワーカに異常が発生した場合は、担当するスーパーバイザが再起動します。スーパーバイザはさらに上位のスーパーバイザで監視され階層構造を作っており、これによって耐障害性を持つプログラムを実現できます。

ビヘイビア

管理ツリーの配下にある各種プロセスの多く

▼図1　プロセス管理ツリー



は類似した役割と構造を持っており、共通部分を定式化して切り分けることで、プログラマは個別処理に注力でき、かつ共通部分の見通しとデバッグが容易になります。

アプリケーション[注4]

OTPのアプリケーションは、ある特定の目的を持った複数のモジュールの集まりに名前をつけたものです。Erlang/OTPの最小限のシステムはkernelとstdlibより成ります(前回連載を参照)。アプリケーションは個別に起動や終了したり、設定したりすることが可能です。OTPでの大規模な実行プログラムは、アプリケーションの集合体と考えることができます。

リリース

OTPでの「リリース」という単語は、実行プログラムに必要な複数のアプリケーションと、Erlang/OTPの必要な部分(BEAMを含む実行時システム(ERTS)とOTPの中で関連するアプリケーション群)をまとめたものです。このリリースをインストールすることで、実行プログラムを動かすことができます。リリースはリリースハンドリングによって、新しいバージョンに更新したり、あるいは古いバーションに戻したりすることができます。

ビヘイビアによるプログラミングパターンの抽象化

OTPのビヘイビアは、その「ふるまい」という英語の原義のとおり、各種の定型的なプログラミング上のパターンを定式化したものです。具体的には、プログラムを次の2つの部分に分けて、その片側をビヘイビアとしてOTPが提供していると考えることができます。

- それぞれのプログラムで個別に処理する部分(コールバックモジュール)
- 各種パターンで共通に使う部分(ここをビヘイビアで提供)

ビヘイビアを使って書かれたコードは、コールバックモジュールとビヘイビアのモジュールが互いに呼び合う構造を持ちます。これによって記述するモジュールを分離しながら1つのプログラムとして協調動作をさせることができます。

コールバックモジュールにて、コンパイラに"-behaviour(gen_server)."として指示することでビヘイビアが使えるようになります。こ

注4) この単語は、一般に「何かを行うためのプログラム」として使われる「アプリケーション」ではなく、OTP特有の定義かつ意味を持っていることに注意が必要です。

れで指示したビヘイビアに則ったプログラムを書くことができます。

OTPでは次のビヘイビアを標準として提供しています[注5]。

- gen_server：一般的なクライアント＝サーバ構成でのサーバ
- gen_fsm：有限状態機械（通信プロトコルの状態管理などで多用）
- gen_event：イベントハンドリング（ログやアラートなどのイベント処理用）
- supervisor：プロセス管理ツリー中のスーパバイザを書くためのビヘイビア

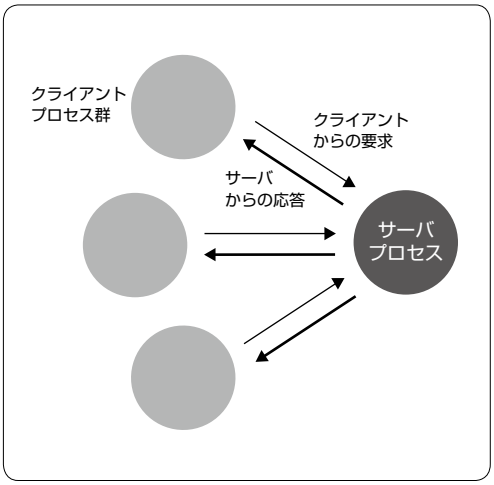
gen_serverビヘイビアの使い方

gen_server[注6]は、サーバがクライアントから処理要求を受け取り、それぞれの処理要求に対して返答を返す、という典型的なクライアント＝サーバ間処理をするためのビヘイビアです（**図2**）。クライアントとサーバの間のやり取りはErlangのメッセージとして行われます[注7]。

本連載第3回、第4回で例としているカウンタサーバを今回も例として使うことにします。サンプルコードを**リスト1**に示します。

リスト1のサンプルコードでは、最初にサーバにしてほしい作業をgen_server:call/2経由でメッセージとして送る関数を列挙し、その後にサーバからのコールバック関数（handle_call/3など）として、受け取ったメッセージをどのように処理するかについて記述してい

▼図2　gen_serverビヘイビアが想定するクライアント＝サーバ間通信



ます。このように抽象化することで、サーバでの処理を増やしたいときはメッセージの送信関数と対応するコールバック処理（一般的にはhandle_callの中の節）を増やしていけばよいようになっています。

図3にサンプルコードを動かした結果の例を示します。所定の動作をしていることがわかります。この際、erlang:process_info/{1,2}という関数を使うと、プロセスが存在しているか、また存在している場合は内部状態を知ることができます。

図4ではサンプルコードを連載第4回と同様の分散したBEAMノード間で実行した結果を示します。ここで注目すべき点は、コードそのものがローカルノードの場合とまったく変わっておらず、またBEAMノード間の通信はgen_serverの中で処理されるためプログラマは個別にメッセージングのためのコードを書く必要がないことです。

sysモジュールによるサーバの状態監視と操作

OTPのビヘイビアを使ってサーバを書くことの利点の1つとして、OTPのデバッグライブラリを活用することができます。今回は**図5**

注5） 実際にはユーザが独自にビヘイビアを書くことも可能です。またOTPにもSSHを使う際のssh_channelモジュールなど、特定のプロトコル処理を個別のビヘイビアに切り分けている実装が多数あります。

注6） gen_serverの設計と動作の詳細については（http://www.erlang.org/doc/design_principles/gen_server_concepts.html）を、使う関数については（http://www.erlang.org/doc/man/gen_server.html）を参照してください。

注7） サンプルコードでは割愛しましたが、実際には登録済みプロセスとしてサーバに名前を付けたり、さらにそれらを分散したBEAMノード間でglobalモジュールなどを介して統一した名前として管理するようにすることもできます。

▼リスト1　gen_serverで実装したカウンタサーバ（msgcounter_gen_server.erl）

左段下から続く↙

```
カウンタを状態に持つサーバをgen_serverで書いた
詳細は第3回のmsgcounter.erlを参照してほしい
-module(msgcounter_gen_server).

gen_serverビヘイビアを使う旨を宣言
-behaviour(gen_server).
gen_serverのためのコールバック関数もexportする
-export([start_link/0, init/1,
         inc/1, dec/1, zero/1, val/1, stop/1,
         handle_call/3, terminate/2,
         handle_cast/2, handle_info/2, ↵
code_change/3]).

レコードの中にカウンタの内部状態を入れる
-record(state, {counter = 0}).
リンク付きでサーバを起動する
成功すると {ok, Pid} で、pidがタプルの中に返る
-spec start_link() -> {ok, pid()}.
start_link() ->
    ?MODULEはこのモジュール自身の名前のアトム
    gen_server:start_link(?MODULE, [], []).

ここからはgen_server:call/2を通じて
サーバにどんなメッセージを送るかを書くための関数
内部状態のカウンタの値を1つ増やしてその後の値を返す
-spec inc(pid()) -> integer().
inc(Pid) ->
    gen_server:call(Pid, inc).
内部状態のカウンタの値を1つ減らしてその後の値を返す
-spec dec(pid()) -> integer().
dec(Pid) ->
    gen_server:call(Pid, dec).
内部状態のカウンタの値を1つゼロにして成功したらokを返す
-spec zero(pid()) -> ok.
zero(Pid) ->
    gen_server:call(Pid, zero).
内部状態のカウンタの値を返す
-spec val(pid()) -> integer().
val(Pid) ->
    gen_server:call(Pid, val).
サーバを止める
-spec stop(pid()) -> ok.
stop(Pid) ->
    gen_server:call(Pid, terminate).

ここから先はビヘイビアからのコールバック関数の定義
gen_serverからのサーバ初期化作業のコールバック関数
-spec init([]) -> {ok, #state{}}.
init([]) ->
    カウンタの値をゼロに初期化する
    {ok, #state{counter = 0}}.
gen_server:call/2で同期メッセージを送った際に
返事を返すためのコールバック関数
この関数単独ですべてのメッセージをさばくため
各種要求はセミコロンで区切られた
パターンマッチングの節として実装している
-spec handle_call(term(), pid(), #state{}) ↵
-> term().
第1引数にはgen_server:call/2からのメッセージの内容
第2引数には呼出元のPidと識別用のタグの組み合わせ
第3引数には呼ばれる前の内部状態が与えられる
handle_call(inc, _From, #state{counter = ↵
Count}) ->
    返り値としてタプルを設定し次の動作を決める
    第1要素は動作の指定(replyなら返答を返す)
    第2要素は返答の内容、
    第3要素は返答を返した後の内部状態を示す
    {reply, Count + 1, #state{counter = ↵
Count + 1}};
前の節ではinc/1の処理、ここからはdec/1の処理
handle_call(dec, _From, #state{counter = ↵
Count}) ->
    {reply, Count - 1, #state{counter = ↵
Count - 1}};
zero/1の処理
handle_call(zero, _From, _S) ->
    {reply, ok, #state{counter = 0}};
var/1の処理
handle_call(val, _From, S = #state{counter = ↵
Count}) ->
    {reply, Count, S};
stop/1の処理
handle_call(terminate, _From, S) ->
    タプルの第1要素がstopだとサーバのプロセスは停止して消滅する
    第2要素は停止理由（terminate/2で判断する）
    第3要素は返答内容、第4要素は内部状態を示す
    {stop, normal, ok, S}.
これでhandle_call/3の処理は終了なので最後はピリオドで終わっている
terminate/2はgen_serverから
プロセス終了時に呼ばれるコールバック関数
-spec terminate(normal, #state{}) -> ok.
第1引数は停止理由（問題がなければokを返しておく）
terminate(normal, _S) -> ok.
ここから先の関数は本サンプルコードでは明示的には使わない
handle_cast/2はcast/2で送られた非同期メッセージを処理する
-spec handle_cast(term(), #state{}) -> term().
handle_cast(_Msg, S) -> {noreply, S}.
handle_info/2はcallやcast以外のメッセージを受けとった時の処理をする
-spec handle_info(term(), #state{}) -> term().
handle_info(_Info, S) -> {noreply, S}.
code_change/3ではモジュールの動的なアップデートの際に
サーバの内部状態を更新するかどうかを指定する
-spec code_change(term(), #state{}, term()) ↵
-> {ok, #state{}}.
code_change(_OldVsn, S, _Extra) -> {ok, S}.
```

右段へ続く↗

にOTP標準のsysモジュールによるサーバの内部でのBEAMの処理（リダクション）回数やメッセージの入出力回数といった統計情報の取得や、内部の状態を操作する例を示しました。sysモジュールを使ってこのように動作しているプロセスの状態を外部から確認できるのは、各種ビヘイビアを使って開発する大きな利点であると筆者は考えます[注8]。

注8） ビヘイビアを使ったプロセス動作のデバッグについては（http://www.erlang.org/doc/design_principles/spec_proc.html）を参照してください。

▼図3　サンプルコードの実行例

```
Eshell V6.4.1  (abort with ^G)
モジュールをロードする
1> l(msgcounter_gen_server).
{module,msgcounter_gen_server}
1つ目のカウンタP1を起動する
2> {ok, P1} = msgcounter_gen_server:start_link().
{ok,<0.35.0>} % {ok, Pid}という値が戻る
この時P1はパターンマッチングでPidの値になっている
2つ目のカウンタP2を起動する
3> {ok, P2} = msgcounter_gen_server:start_link().
{ok,<0.37.0>}
P1を2回増やしてみる
4> msgcounter_gen_server:inc(P1).
1
5> msgcounter_gen_server:inc(P1).
2
P2を1回減らしてみる
6> msgcounter_gen_server:dec(P2).
-1
カウンタの値を確認
7> [msgcounter_gen_server:val(P) || P <- [P1, P2]].
[2,-1]
P2のカウンタの値をゼロにする
8> msgcounter_gen_server:zero(P2).
ok
再度カウンタの値を確認
9> [msgcounter_gen_server:val(P) || P <- [P1, P2]].
[2,0]
P1を止めてみる
10> msgcounter_gen_server:stop(P1).
ok
P1の状態をerlang:process_info/2で確認する
11> process_info(P1, status).
undefined 存在していない状態
P2の状態をerlang:process_info/2で確認する
12> process_info(P2, status).
{status,waiting} メッセージを待っている状態
P2を止めてみる
13> msgcounter_gen_server:stop(P2).
ok
P2の状態をerlang:process_info/2で確認する
14> process_info(P2, status).
undefined 存在していない状態
```

まとめ

今回はErlangのビヘイビアとgen_serverについて紹介しました。次回はErlang/OTPでのテストとデバッグ用各種ツールについて紹介する予定です。

▼図4　BEAMノード間での実行結果

```
ホストalphaとbravoは同じネットワーク上にあり
同じサブドメインに属するものとする
ホストalphaでは
erl -sname node1 -setcookie cookie_string
というコマンドを実行しておく
ホストbravoでは
erl -sname node2 -setcookie cookie_string
というコマンドを実行しておく
以下はalpha上で動くBEAMノード「node1@alpha」の
シェルでの実行結果
操作を楽にするためノード名を変数に定義しておく
(node1@alpha)1> Remotenode = 'node2@bravo'.
node2@bravo

 死活監視用の関数を実行する
(node1@alpha)2> net_adm:ping(Remotenode).
pong % ノードは生きている

相手のノードでmsgcounter_gen_server:start_link/0を使って
カウンタサーバを起動してみる
(node1@alpha)3> {ok, P} = rpc:call(Remotenode, msgcounter_gen_server, start_link, []).
{ok,<6243.44.0>} % 問題なく起動できた

以下相手のノードのカウンタは同じプログラムでまったく問題なく
動いている
(node1@alpha)4> msgcounter_gen_server:inc(P).
1
(node1@alpha)5> msgcounter_gen_server:inc(P).
2
(node1@alpha)6> msgcounter_gen_server:val(P).
2
```

ソースコードとサポートページ

連載の記事で紹介したソースコードなどGitHubのレポジトリに置いています[注9]。どうぞご活用ください。SD

注9） https://github.com/jj1bdx/sd-erlang-public/

参考文献

[1] https://github.com/erlang/otp/commits/OTP-17.5.6.1

[2] News From the OTP Team, Erlang User Conference, Stockholm 2015, http://www.erlang-factory.com/static/upload/media/1434558750592554otpnewseuc2015.pdf

[3] OTP Design Principles, http://www.erlang.org/doc/design_principles/des_princ.html

▼図5　sysモジュールによるサーバ状態取得と変更の例

左段下から続く↙

```
Eshell V6.4.1  (abort with ^G)
1> {ok, P} = msgcounter_gen_server:start_link().
{ok,<0.34.0>}

サーバの動作統計情報を取り始めている
2> sys:statistics(P, true).
ok

何もしていない状態の情報を取る
3> sys:statistics(P, get).
{ok,[{start_time,{{2015,6,19},{18,42,41}}},
起動時刻
     {current_time,{{2015,6,19},{18,42,50}}},
取得した時点の時刻
     {reductions,18},
これは内部演算であるリダクションの回数を示す
     {messages_in,0},  いくつメッセージを受信したかを示す
     {messages_out,0}]}  いくつメッセージを送信したかを示す

4> msgcounter_gen_server:inc(P).
1

5> msgcounter_gen_server:inc(P).
2
2回カウンタを増やした後の状態
6> sys:statistics(P, get).
{ok,[{start_time,{{2015,6,19},{18,42,41}}},
     {current_time,{{2015,6,19},{18,43,15}}},
     {reductions,66},  リダクションの回数が増えている
     {messages_in,2},
メッセージを2つ受信していることを示している
     {messages_out,0}]}

この関数ではgen_serverで保持している内部状態を示す
7> sys:get_state(P).
これはレコードの #state{} の最初の要素（counter）が2であることを示している
{state,2}

カウンタをゼロにしてみる
8> msgcounter_gen_server:zero(P).
ok

内部状態も変わった
9> sys:get_state(P).
{state,0}

内部状態を変更する関数をサーバプロセスに適用してみる
（注意：デバッグ用途以外で使うのは推奨しない）
10> sys:replace_state(P, fun(_) -> {state,100}
end).
{state,100}

カウンタの値も相応して変化している
11> msgcounter_gen_server:val(P).
```

右段へ続く↗

Sphinxで始めるドキュメント作成術

川本 安武 KAWAMOTO Yasutake Twitter @togakushi

第5回 目次、用語集、索引を付けよう ——大きめのドキュメントを読みやすくするために

Sphinx拡張

今回はSphinxの特徴の1つである「Sphinx拡張」の基本的な使い方、少し大きめのドキュメントを作成する際に便利に使えるディレクティブを紹介します。

Sphinx拡張には、あらかじめSphinxに機能として実装されている「組み込み拡張機能」と、インストールして機能を追加するサードパーティ製の拡張機能があります。さまざまな拡張機能がありますが、おもな拡張はディレクティブを追加し、Sphinxで表現できる記法を増やします。ほかの外部コマンドの実行結果を取り込むような拡張もあります。

今回は組み込み拡張の1つ、「todo拡張」について紹介します。どの拡張機能も有効にする方法は同じですので、公式ドキュメントを参考に他の拡張機能も使用してみてください。

todo以外の拡張機能については公式ドキュメント[注1]を参照してください。

拡張機能を有効にする

拡張機能を使用するにはプロジェクトディレクトリのルートにあるconf.pyに設定を追加します。todo拡張を有効にするには次のように設定します。

```
変更前
extensions = []
変更後
extensions = ['sphinx.ext.todo']
```

複数の拡張機能を使用する場合はカンマ(,)で区切って並べます。1行で記述しても問題ありませんが、次の例では可読性を高めるために改行を入れています。

```
複数の拡張機能を使用する例
extensions = [
  'sphinx.ext.todo',
  'ほかの拡張機能',
]
```

追加されるディレクティブとオプション

todo拡張を有効にすると「todo」と「todolist」というディレクティブが追加されます。

todoディレクティブは、ToDoとしてメモしておきたい内容[注2]をreSTに埋め込みます(**図1**)。ディレクティブの要素にToDoの内容を記述します(**リスト1-①**)。なお、ToDoが1行の場合には、ディレクティブの引数に内容を記述することもできます(**リスト1-②**)。

todolistディレクティブは、プロジェクト全体のtodoの内容を列挙します。どのreSTファイルの何行目にあるtodoなのかも合わせて表示します(**図2、リスト2**)。

しかし、これらのToDoはこのままではHT

注1) http://sphinx-doc.org/extensions.html
http://docs.sphinx-users.jp/extensions.html

注2) OSSのドキュメントであれば、将来実装したい機能やあとで追記する予定の部分をToDoで書いておくなど。

MLには表示されません。todo拡張を有効にすると、「todo_include_todos」というオプションが追加されます。conf.pyに「todo_include_todos = True」と追加で設定すると、todo、todolistが表示されるようになります。

目次を作る

以前、本連載の中で紹介したtoctreeディレクティブは、ドキュメント全体の目次を作成するものでした。ページ内の目次を作るにはcontentsディレクティブを使います。contentsディレクティブは、記述した位置にページ内のセクションを目次として埋め込むディレクティブです(**図3、リスト3**)。

図3ではcontentsディレクティブ部分の見出しに「目次」と表示されていますが、これはcontentsディレクティブの引数で「目次」と見出し名を指定しているからです。引数が省略された場合は「contents」という見出しが付きます。

▼リスト1　todoの使用例

```
ToDoサンプル
============

Webサーバ要件
-------------

.. todo:: Webサーバの要件はまだ決まっていないのであとで書く

DBサーバ要件
------------

.. todo::
   DBサーバの要件はまだ決まっていないのであとで書く

   (クラスタにすることだけ決まっている)
```

② ディレクティブの引数に記述

① ディレクティブの要素に記述

▼リスト2　todolistの使用例

```
ToDo一覧
========

.. todolist::
```

▼リスト3　contentsの使用例

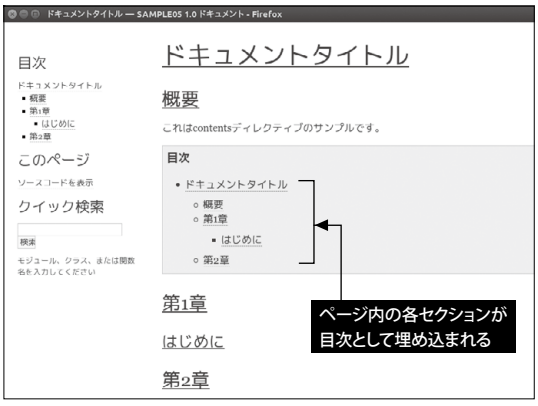
```
====================
ドキュメントタイトル
====================

概要
====

これはcontentsディレクティブのサンプルです。

.. contents:: 目次

第1章
=====

はじめに
--------

第2章
=====
```

引数に見出しの文言を記述する

▼図1　todoの適用例

▼図2　todolistの適用例

▼図3　contentsの適用例



contentsディレクティブは**表1**のオプションを指定できます。

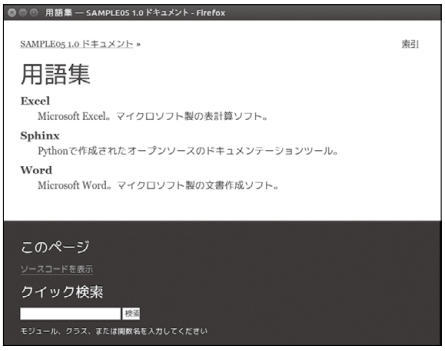
contentsについてはDocutilsの公式ドキュメント[注3]も参照してください。

用語集を作る

Sphinxには用語集を作成するための「glossary」というディレクティブがあります(**図4、リスト4**)。glossaryディレクティブに記述したキーワード(用語)は、「term」というロール(後述)で参照できます(**図5、リスト5**)。また、キーワードとして定義した単語は、索引(後述)のページ

注3) http://docutils.sphinx-users.jp/docutils/docs/ref/rst/directives.html#table-of-contents
http://docutils.sourceforge.net/docs/ref/rst/directives.html#table-of-contents

▼図4　glossaryの適用例



▼リスト4　glossaryの使用例

```
.. glossary::
   :sorted:

   Sphinx
      Pythonで作成されたオープンソースのドキュメンテーションツール。

   Excel
      Microsoft Excel。マイクロソフト製の表計算ソフト。

   Word
      Microsoft Word。マイクロソフト製の文書作成ソフト。
```

▼表1　contentsディレクティブのオプション

オプション	意味
depth	目次に載せるセクションの深さを数値で指定する。省略時はすべてのセクションが目次になる
local	contentsディレクティブが記述されているセクション内だけの目次を作成する。省略時はページ全体のセクションが目次になる
backlinks	セクションから目次へ戻るためのバックリンクを生成する。指定できる引数は、entry(目次のエントリーに対して作成)、top(目次の見出しに対して作成)、none(バックリンクを作成しない)。省略時はentryとなる

COLUMN

更新されないドキュメント

todoとtodolistをそれぞれ別のファイルに記述している場合、todoを追加したあとに「make html」を実行しても、todoが記述されているドキュメントは更新されますが、todolistが記述されているドキュメントは更新されません。これは、makeが更新のあったreSTファイルのみを対象にドキュメントの変換を行うためです。todolistも変換の対象にするには、todolistが記述されているreSTファイルのタイムスタンプをtouchコマンドなどを使用して更新します。

別の方法として、「make clean」を実行する方法もあります。make cleanはmake htmlによって変換されたすべてのファイルを削除します。reSTのファイル名を変更した場合、新しいファイル名でドキュメントが生成されますが、古いファイル名のドキュメントは削除されません。このような場合もmake cleanを使用します。

なお、make clean後はすべてのドキュメントを再生成することになるので、ドキュメントの量によっては変換に時間がかかります。

にも載り、そこからも参照できます。

glossaryはディレクティブの要素にキーワードを指定し、さらにインデントを付けてキーワードの説明を記述します。複数のキーワードを指定する場合は空行で区切って並べます(**リスト4**)。これは「定義リスト」と呼ばれるreSTの記法の1つです。定義リストの詳しい内容については、Docutilsのドキュメント[注4]を参照してください。

glossaryは1個所にまとめて記述する必要はなく、複数ページで使用しても問題ありません。

また、**リスト4**ではsortedオプションを指定しています。このオプションは、HTML変換時にキーワードをアルファベット順に並べ替えます。

glossaryの詳しい使用方法は公式ドキュメント[注5]を参照してください。

ロールとは

ロールは記述されているテキストが特別な意味を持つことをSphinxに伝える役目をする記法です。

意味を持たせるテキストをバッククォート(`)で囲み、その前後のどちらかにロールマーカと呼ばれる「ロールである」ことを示す記法を追加します。ロールマーカはコロン(:)で挟んで記述します。

具体的には次のような記述になります。

```
:ロール名:`意味を持たすテキスト`
`意味を持たすテキスト`:ロール名:
```

前回までに紹介した、ドキュメント間のリンクを表現する「`:ref:`」や「`:doc:`」もロールの一種です。このほかにも多数のロールが存在します。詳細は公式ドキュメント[注6]を参照してください。

索引を作る

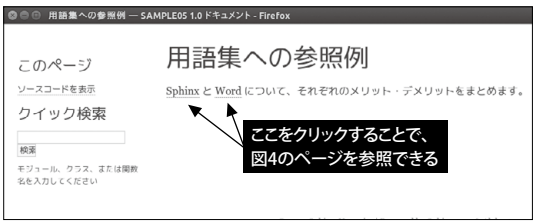
前述したように、索引にはglossaryディレクティブで指定したキーワードが載ります。さらにindexディレクティブ、またはindexロールを

注4) http://docutils.sourceforge.net/docs/ref/rst/restructuredtext.html#definition-lists
http://docutils.sphinx-users.jp/docutils/docs/ref/rst/restructuredtext.html#definition-lists

注5) http://sphinx-doc.org/markup/para.html#glossary
http://docs.sphinx-users.jp/markup/para.html#glossary

注6) http://sphinx-doc.org/markup/inline.html#cross-referencing-syntax
http://docs.sphinx-users.jp/markup/inline.html#cross-referencing-syntax

▼図5　termロールの適用例



▼リスト5　glossaryを参照する例

```
:term:`Sphinx` と :term:`Word` について、それぞれのメリット・デメリットをまとめます。
```

COLUMN

索引へのリンクについて

Sphinx1.3のデフォルトテーマのAlabasterでは、ブラウザのウィンドウの横幅が860px未満の場合、ページの右上に索引へのリンクが表示されます(**図4**)。しかし、横幅が860px以上あるとリンクは消えてしまいます。

リンクを表示するには、ブラウザの横幅をサイドバーが消えるまで縮める(横幅860px未満)か、ページ内に「`:ref:`genindex`」としてリンクを作成してください。テーマによっては、ページの右上、または右下に索引のリンクが常に存在するものもあります。テーマの変更については次回紹介します。

使用すれば、索引に載せる単語を指定できます。本節ではその方法を紹介します。

indexディレクティブ

indexディレクティブの引数に単語を指定することで、索引にその単語が載るとともに、索引からindexディレクティブを記述した場所へ自動的にリンクが作成されます(**リスト6**)。索引からのリンクはindexディレクティブを記述した位置に作成されます。リンクが作成される位置にはテキストが存在しないため、索引に載せる単語が示すものの直前(セクションや段落など)に記述します。オプションによって、キーワードに階層を持たせたり、複数のキーワードで相互にリンクを作成することもできます。オプションについては、公式ドキュメント[注7]を参照してください。

indexロール

また、本文中の単語にロールを付けることでも索引に載せられます(**リスト7**)。indexディレクティブと同様のオプションも使用できますが、1行で記述する必要があるため複雑なオプションはreSTの可読性を下げてしまいます。

indexとglossaryを使用している索引は**図6**のようになります。**リスト4、6、7**で指定した単語が索引に載っていることがわかります。また、現在のバージョンのSphinxでは日本語は記号に分類されてしまいます。

▼リスト6　indexディレクティブの使用例

```
.. index:: 索引に載せたい単語

索引に載せたい単語とは
----------------------
```

▼リスト7　indexロールの使用例

```
索引に :index:`単語を載せる` にはこのようにします。
```

▼図6　indexとglossaryを使用している例

注7) http://sphinx-doc.org/markup/misc.html#index-generating-markup
http://docs.sphinx-users.jp/markup/misc.html#index-generating-markup

COLUMN

ロールを使って取り消し線を引く

reSTには取り消し線(HTMLのsタグ)を表現する記法がありません。

Sphinxでは新しくロールを定義することができ、それにスタイルシートを適用させられます。この機能を利用して取り消し線を引いてみます。

新しいロールの定義は「role」というディレクティブで行います。追加したロールが使用される前に、同じreSTファイル内で定義しておきます。ほかのロールと同様の使い方で、テキストに色を付けられます。

追加するロール

```
.. role:: strike
```

追加するスタイルシート

```
.strike {
    text-decoration: line-through;
}
```

スタイルシートを追加する方法は前回のコラム「HTMLのテーブルに罫線を表示する」で紹介した方法で行います。適用結果は**図A**のとおりです。

reSTに記述するロール

```
ここに :strike:`取り消し線` を引きます。
```

▼図A　新しいロールを適用した結果

ここに ~~取り消し線~~ を引きます。

◆ ◆ ◆

これまでにreSTの基本的な記法や、よく利用するSphinx独自のディレクティブを紹介しました。次回は、テーマの変更方法、Sphinxドキュメントを Webサーバで公開する際に知っておきたいことを取り上げます。SD

COLUMN

PyCon APAC 2015 in Taiwan

Author 清水川 貴之

本連載執筆者の1人、清水川です（直近では第4回「テーブルを使いこなそう」の執筆を担当）。

2015年6月5～7日に台湾（台北）でPyCon APAC 2015[注A]が行われました。PyCon APACはアジア太平洋地域の国で毎年行われるPythonのカンファレンスです。2010年にシンガポールで第1回が開催され、2013年は日本で、昨年と今年は台湾で開催されました。

筆者はこのイベントで、Sphinxの機能を紹介する2つの発表を行ってきました。また、カンファレンス2日目の夜のパーティー兼コミュニティーブースという構成の企画にも応募し、Sphinxブース展示をさせてもらいました。そこで本欄を利用して、発表内容の概要と、ドキュメントを書くことやSphinxに対してカンファレンス参加者がどのような様子だったかを紹介したいと思います。

■発表で紹介した機能

1つ目の発表は「Easy contributable internationalization process with Sphinx（Sphinxによる貢献しやすい翻訳プロセス）」[注B]です。本発表では、Sphinxの基本機能とドキュメント翻訳サポート機能を紹介しました。また、翻訳ボランティアがOSSプロジェクトに参加しやすいしくみを提供する方法として、システム構成や自動化手法ついても話しました。

2つ目の発表は「Sphinx autodoc: automated API documentation（Sphinx autodoc：APIドキュメントの自動生成）」[注C]です。この発表では、Sphinxの自動ドキュメント機能を使うと、Pythonプログラムの充実したドキュメントを手軽に作れることを紹介しました。

■台湾でのSphinxへの反応

カンファレンス2日目の夜に、パーティー企画の一部[注D]としてSphinxのブース展示を行いました（**写真A**）。約2時間の展示で、50人ほどの人にSphinxを紹介しました。ブースに来てくれた参加者の中でSphinxをすでに知っている、使っている、という人は3人程度しかいませんでした。Sphinxを知らなくても、Sphinxで作られているPython公式ドキュメントは全員が読んだことがあるため、Python公式ドキュメントがどのようなしくみで作られているのかを紹介しました。

ドキュメントの作成については、簡単に作りたいと思っているものの、どう書いたら良いのかわからない、という方が多かったようです。そのため、ドキュメントを手軽に生成できるSphinxには、みんな興味を持ってくれたようで、熱心に質問をしていました。

▼写真A　ブースでSphinxの紹介中

Pythonのデファクトスタンダードとなっているsphinxのことはもう少し知られているかと思ったのですが、そうでもないことがわかりました。こういった違いを知ることができるのも、遠征することの楽しみの1つですね。

より詳しいPyCon APAC 2015のレポートをgihyo.jpに掲載しております。そちらもご参照ください[注E]。

注A）PyCon APAC 2015 https://tw.pycon.org/2015apac/en/

注B）http://www.slideshare.net/shimizukawa/easy-contributable-internationalization-process-with-sphinx-pycon-apac-2015-in-taiwan-49057754

注C）http://www.slideshare.net/shimizukawa/sphinx-autodoc-automated-api-documentation-pycon-apac2015

注D）https://tw.pycon.org/2015apac/en/program/night-party/

注E）海外PyCon発表修行レポート2015
http://gihyo.jp/news/report/01/overseas-pycon-presentation-training-2015

Android Wear アプリ開発入門

～より生活に密着するスマートデバイスの世界～

第6回（最終回） Wearアプリで GPS機能を活用！

神原 健一（かんばら けんいち） Web http://blog.iplatform.org Twitter @korodroid
iplatform.orgにて情報発信するかたわら、「セカイフォン」などを開発。Droidconなどでのカンファレンス講演、MWC/CES/IFAでのプロダクト展示、執筆などの活動も実施。NTTソフトウェア株式会社テクニカルプロフェッショナル。現在はAndroid以外のモバイルOSにも取り組み、公私にわたってモバイルアプリの世界に没頭中。

はじめに

前号（本誌7月号）では、Android Wear（以降「Wear」と表記）らしいアプリを開発するために考慮すべきこと、および、Wear向けに提供されているUIライブラリについて解説しました。これらを適切に活用することで、ユーザに好まれるアプリに近づけることができます。詳細は、前回の記事をご参照ください。

今回は話題を変えて、Wear上のGPS機能の活用方法について解説します。Wearはユーザが普段身につけているという大きな特徴を持っています。そのため、ユーザがいる位置に連動した有益な情報があれば、スマホ以上にユーザに価値をもたらすことができます。たとえば、街を歩いているユーザに対して近隣のお店の情報をリアルタイムに配信したり、自分が旅行先で移動した経路を記録しておき、あとで振り返ることができるようにするといった使い方が考えられるでしょう。このように、位置情報に連動したサービスは、スマホ以上にウェアラブルのユーザに価値を提供できます。それでは、Wear上でどのように位置情報を取得するのかを解説していきます。

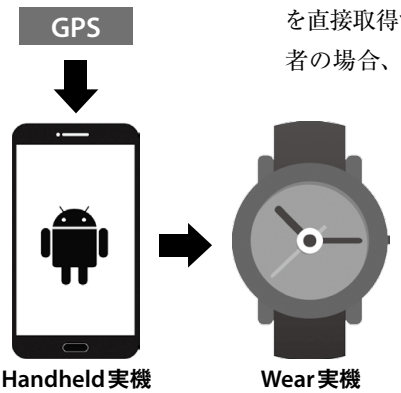
Wearアプリでの位置情報取得

Wear実機にはハードウェアとしてGPSを搭載している機種（Sony SmartWatch 3など）もあれば、内蔵していない機種（LG G Watchなど）もあります。Wearアプリで位置情報を利用したい場合、前者であれば、Wear単体で位置情報を直接取得することができます（**図1**）。一方、後者の場合、Wear単体で位置情報を取得できないため、代わりにスマホやタブレット（以降「Handheld」と表記）で取得した位置情報を用いることになります（**図2**）。

ただ、いずれの方式でも、位置情報を取得する実装に大きな違いはありません。では実装方法を解説していきます。具体的

▼図1 Wearでの位置情報取得（直接方式）

▼図2 Wearでの位置情報取得（間接方式）



▼リスト1　AndroidManifest.xmlでの宣言

```
<application>
    ...(略)...
    <meta-data android:name="com.google.android.gms.version" android:value="@integer/
google_play_services_version" />
</application>
...(略)...
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
```

▼リスト2　build.gradleの定義例

```
dependencies {
    compile fileTree(dir: 'libs', include: ['*.jar'])
    ...(略)...
    compile 'com.google.android.gms:play-services-wearable:7.5.0'
    compile 'com.google.android.gms:play-services-location:7.5.0'
}
```

▼図3　サンプルアプリ

には、Google Play Servicesに含まれるLocation APIを利用します[注1]。

位置情報を取得するプログラムの開発

今回は、位置情報を取得するサンプルアプリ(**図3**)を用いて、実装の流れを紹介します。アプリを起動すると、一定間隔で位置情報を取得し、その結果をトーストとして表示するという機能を持たせます。Wearアプリとして実装しますが、具体的な実装のステップは次のとおりです。

①AndroidManifest.xmlでの宣言
②build.gradleの設定
③GoogleApiClientに関する実装
④LocationAPIに関する実装

①AndroidManifest.xmlでの宣言

Google Play Servicesに含まれる機能を用いるため、その宣言をAndroidManifest.xmlで行います。**リスト1**のとおり、<application>～</application>内に、<meta-data />の定義を行ってください。また、位置情報の取得に必要となるパーミッション"ACCESS_FINE_LOCATION"を追加しておきます。

②build.gradleの設定

アプリで利用するライブラリをbuild.gradleのdependenciesで定義しておきます。今回のアプリでは、Google Play ServicesのAPIに含まれるAndroid WearとLocationライブラリを用いるため、**リスト2**のとおり、定義を行っておきます。

余談ですが、Google Play Servicesには、その他にもさまざまなライブラリが含まれています。提供されているAPI一覧、および、build.gradleの定義内容に関しては、公式サイト[注2]を参照ください。今後、これらのAPIは増減したり、バージョンアップなども行われると思いますので、最新情報を適宜ウォッチすることを心がけてください。

③GoogleApiClientに関する実装

Google Play Servicesを利用するには、GoogleApiClientクラスを用いる必要があります。具体的には、**リスト3**のコードを記述します。

まず、ActivityのonCreate()メソッド内で、GoogleApiClientのインスタンスを生成してい

注1）位置情報といえば、Androidフレームワークに含まれるLocation API(android.locationパッケージ)もありますが、現在は非推奨となっています。既存のアプリで利用している場合もGoogle Play Servicesへの移行が推奨されていますので留意しましょう。

注2）http://developer.android.com/google/play-services/setup.html

ます(**リスト3(1)**)。また、接続成功、中断を監視するリスナーをaddConnectionCallbacks()メソッドで**リスト3(2)**、接続失敗を監視するリスナーをaddOnConnectionFailedListener()メソッドで登録しています**リスト3(3)**。これにより、接続成功時はonConnected()メソッド、接続中断時はonConnectionSuspended()メソッド、接続失敗時はonConnectionFailed()メソッドがコールバックされます。さらに、Activityがフォアグラウンドに移行するonResume()の契機でGoogleApiClientの接続処理を**リスト3(4)**、バックグラウンドに移行するonPause()の契機で切断処理を行っています(**リスト3(5)**)。

④LocationAPIに関する実装

今回は、Google Play ServicesのLocation APIを利用するため、**リスト3**に対して次の4つの改造を行います。

ⓐLocationListenerをimplementsする
ⓑLocation APIを利用することを宣言する

▼リスト3　GoogleApiClientの接続処理

```
                                                    リスト4aの個所
public class MainActivity extends Activity implements
        GoogleApiClient.ConnectionCallbacks,
        GoogleApiClient.OnConnectionFailedListener{

    private GoogleApiClient mGoogleApiClient;
    private static String TAG = "MainActivity";
                                                              リスト4bの個所
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        mGoogleApiClient = new GoogleApiClient.Builder(this)  ―― (1)
                .addApi(Wearable.API)
                .addConnectionCallbacks(this)  ―― (2)
                .addOnConnectionFailedListener(this)  ―― (3)
                .build();
    }

    @Override
    protected void onResume() {
        super.onResume();
        mGoogleApiClient.connect();  ―― (4)
    }

    @Override
    protected void onPause() {
        super.onPause();
        mGoogleApiClient.disconnect();  ―― (5)
    }
                                                        リスト4cの個所
    @Override
    public void onConnected(Bundle bundle) {  ―― 接続成功時
        Log.i(TAG, "onConnected");
    }

    @Override
    public void onConnectionSuspended(int i) {  ―― 接続中断時
        Log.i(TAG, "onConnectionSuspended");
    }

    @Override
    public void onConnectionFailed(ConnectionResult connectionResult) {  ―― 接続失敗時
        Log.i(TAG, "onConnectionFailed");
    }
}                                                   リスト4dの個所
```

ⓒ位置情報の取得処理を開始する
ⓓLocationListenerの処理を実装する

ⓐでは、位置情報取得を監視するリスナーであるLocationListenerを実装しています(**リスト4a**)。

ⓑでは、GoogleApiClientで利用するLocation Servicesを利用するため、addApi()メソッドで宣言を行っています(**リスト4b**)。

ⓒでは、GoogleApiClientの接続完了(onConnected()が呼び出されて以降)の契機で、位置情報取得処理を開始しています(**リスト4c**)。まず位置情報取得を行うためのLocation Requestオブジェクトを生成し、3つのパラメータを設定しています。setPriority()メソッドで精度(今回はPRIORITY_HIGH_ACCURACY(高精度))、setInterval()メソッドにて位置情報の取得間隔をミリ秒単位で(今回は5,000ミリ秒)、setFastestInterval()メソッドにて正確な位置情報の取得間隔をミリ秒単位で(今回は5,000ミリ秒)指定しています。次に、位置情報のプロバイダであるLocationServices.FusedLocationApiクラスのrequestLocationUpdates()メソッドを呼び出すことで、位置情報の取得を開始します。

ⓓでは、位置情報が変化する契機で呼び出されるonLocationChanged()メソッドをオーバーライドして処理を実装しています(**リスト4d**)。呼び出される際に、位置情報が含まれるLocationオブジェクトが渡されるというしくみになっています。今回は、取得した位置情報の緯度と経度を、それぞれgetLatitude()、get

▼リスト4a　改造処理その1

```
public class MainActivity extends Activity implements
    GoogleApiClient.ConnectionCallbacks,
    GoogleApiClient.OnConnectionFailedListener,
    LocationListener {   追加行
    ...(略)...
}
```

▼リスト4b　改造処理その2

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);   追加行
    mGoogleApiClient = new GoogleApiClient.Builder(this)
        .addApi(LocationServices.API)   追加行
        .addApi(Wearable.API)
        .addConnectionCallbacks(this)
        .addOnConnectionFailedListener(this)
        .build();
}
```

▼リスト4c　改造処理その3(差し替え)

```
private static final long UPDATE_INTERVAL_MS = 5000;
private static final long FASTEST_INTERVAL_MS = 5000;

@Override
public void onConnected(Bundle bundle) {
    LocationRequest locationRequest = LocationRequest.create()
        .setPriority(LocationRequest.PRIORITY_HIGH_ACCURACY)
        .setInterval(UPDATE_INTERVAL_MS)
        .setFastestInterval(FASTEST_INTERVAL_MS);

    LocationServices.FusedLocationApi
        .requestLocationUpdates(mGoogleApiClient, locationRequest, this);
}
```

▼リスト4d　改造処理その4(全体追加)

```
@Override
public void onLocationChanged(Location location) {
    Toast.makeText(this, "Latitude=" + location.getLatitude() +
                ", Longitude=" + location.getLongitude(), Toast.LENGTH_SHORT).show();
}
```

▼リスト5　GPS機能の搭載判定

```
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    setContentView(R.layout.main_activity);
    if (!hasGps()) {
        Log.d(TAG, "GPS未搭載です。");
    } else{
        // GPSを用いる処理
    }
    ...(略)...
}

private boolean hasGps() {
    return getPackageManager().hasSystemFeature(PackageManager.FEATURE_LOCATION_GPS);
}
```

Longitude()の両メソッドにより取得し、トーストとして表示しています。

これによりWearアプリで位置情報取得が行われます。しかし、冒頭で紹介したとおり、WearにはGPS機能が搭載されている機種もあれば、そうでないものもあります。また、Wear実機とペアリング済みのHandheldが、Wear実機と接続されている場合もあれば、切断されている場合もあります。これらをすべて場合分けしなければいけないのでしょうか。Android WearのOSの振る舞いとしては、そこまで面倒なことをせずとも位置情報をよしなに取得してくれるため、どのように位置情報が取得されるかについては基本的に心配する必要はありません。

たとえば、Wearアプリで位置情報の取得を試みると、Wear実機とHandheld実機が接続されているときは、Handheldで取得した情報が利用されます。Wear実機とHandheld実機が切断されているときは、Wear実機のGPS搭載有無により振る舞いが変わります。GPSがあればその値が返却され、ない場合は値を取得できないことになります。

位置情報取得に関するTips

位置情報を扱うWearアプリを開発する場合に、あらかじめ留意しておくべき事項をいくつか紹介します。

Wear実機でのGPS機能の搭載判定

Wear実機とHandheld実機が接続されている場合は、Handheld経由で位置情報を取得できますが、その接続が解除されると、当然ながら、GPS未搭載のWear実機では位置情報を取得できなくなります。そのとき、Wear実機のGPS搭載有無により、Wearアプリの処理を変更したい場合もあるでしょう。

たとえば、**リスト5**のようなコードを記述することで実現できます。hasSystemFeature()メソッドを用いて、引数として、判定したい機能(GPSの場合は、PackageManager.FEATURE_LOCATION_GPS)を与えることで、その機能が搭載されているか否かを取得できます。

WearとHandheldが切断されたときの制御

Wear実機がHandheld実機と切断されると、Wearアプリは Handheld経由では位置情報を取得できない状態となります。その際、GPS未搭載のWear実機は位置情報を取得できないため、位置情報を必要とするサービス(歩行ルートのトラッキングなど)は利用できなくなります。そのとき、アプリの利用者にはその旨を通知したほうがよいでしょう。そのためには、WearとHandheldの切断契機を知る必要があります。

これはWearableListenerServiceを使えば実現できます。具体的には、**リスト6**に示すコー

ドを記述します。同サービスを継承し、切断時に呼び出されるonPeerDisconnected()メソッドをオーバーライド実装します。その中でユーザに通知したい内容をトーストとして表示しています。また、同ServiceとIntentFilterの宣言をAndroidManifest.xmlで行っておく必要があります。**リスト7**のとおり、<application>～</application>内に定義すればOKです。

最新の位置情報取得

GPS未搭載機種がHandheld実機と切断されると、Wearアプリで位置情報を取得できなくなりますが、そのときでも、過去に取得した最新の位置情報を用いて処理を実行したい場合もあるでしょう。**リスト8**に示すコードを記述すればOKです。getLastLocation()メソッドにGoogleApiClientのインスタンスを渡すことで、過去に取得した中で最新の位置情報が格納されたLocationオブジェクトを取得できます。あとは、その中から緯度・経度を取得するという流れです。

おわりに

今回は、位置情報取得に関する実装方法や関連するトピックを解説しました。

2015年3月号より開始した本連載ですが、この回をもち、いったん充電期間をいただくことにしました。これまでの連載記事が皆様のWearアプリ開発のヒントになっておりましたら、著者として大変うれしいかぎりです。読者の皆さんが素敵なWearアプリを世の中に提供することで、ウェアラブルの世界を一緒に盛り上げてくださることを願うばかりです。また近いうちに、皆様のお目にかかることもあるかと思います。その節はどうぞよろしくお願い致します&これまで本当にありがとうございました！

SD

▼リスト6　接続が切断されたときの実装

```
public class NodeListenerService extends WearableListenerService {

    @Override
    public void onPeerDisconnected(Node peer) {
        if(!hasGps()) {
            Toast.makeText(this, "位置情報を取得できないため、利用できる機能が制限されます",
Toast.LENGTH_SHORT).show();
        }
    }
    ...(略)...
}
```

▼リスト7　AndroidManifest.xmlでの宣言

```
<application ...>
    <service android:name=".NodeListenerService">
        <intent-filter>
            <action android:name="com.google.android.gms.wearable.BIND_LISTENER" />
        </intent-filter>
    </service>
    <...>
</application>
```

▼リスト8　最新の位置情報取得

```
Location location = LocationServices.FusedLocationApi.getLastLocation(mGoogleApiClient);
Toast.makeText(this, "Latitude=" + location.getLatitude() +
                ", Longitude=" + location.getLongitude(), Toast.LENGTH_SHORT).show();
```

Mackerelではじめるサーバ管理

Writer 坪内 佑樹(つぼうち ゆうき) ㈱はてな
Twitter @y_uuk1

第6回 Mackerel周辺の運用ツールとAWS連携ノウハウ

今回から、より実運用を想定したMackerelの使い方を特集していきます。前半では、特定の操作を一括で行える「mkr」、Chefなどの構成管理ツールをサポートする「cookbook-mackerel-agent」を紹介。後半ではAWSとMackerelの連携に関するTipsを紹介します。

前回は、メトリック関連のAPI、およびチェック監視機能とそのAPIを説明しました。今回は、Mackerelによるサーバ運用をサポートするツール、さらにAWS(Amazon Web Services)のサービスをMackerelで監視するためのノウハウを紹介します。

運用ツール

まず、Mackerel周辺の運用ツールとして、CLIツール「mkr」と、Chef cookbookの「cookbook-mackerel-agent」を紹介します。

mkr

Mackerelのような管理ツール系のサービスでは「ある操作を一括で行いたい」といった柔軟な操作性を求められます。たとえば、特定サービスに属するホストのステータスをすべて「standby」にする操作が挙げられます。

しかし、一括操作をWebUIで表現することが難しいという事情もあります。そこで、コマンドラインツールmkr[注1]を開発しました。mkrとUNIXのパイプやリダイレクトを活用することにより、一括操作はもちろん、ほかのUNIXツールと組み合わせることで運用の幅を広げることができます。

mkrはホスト情報の参照・更新やメトリックを投稿するためのコマンドラインツールです。Go言語で開発されており、高速に動作することが特徴です。まず、使い方を見てみましょう。特定ロールのホスト一覧情報を取得するには**図1**のようなコマンドを叩きます(My-Service、proxyはそれぞれ架空のサービス名とロール名です)。

mkrの出力形式は基本的にJSONになります。jq[注2]コマンドにより、自由に出力内容をカスタマイズできます。たとえば、「My-ServiceサービスにおけるproxyロールのhostIDのみの一覧」を取得したい場合は、**図2**のようなコマンドを叩きます。jqの記法の詳細については公式マニュアル[注3]を参照してください。

今度は、特定ホストのステータスとロールを更新してみましょう。「2eQGEaLxiYUホストのステータスをmaintenance、サービスをMy-Service、ロールをdb-masterに」変更します(**図3**)。

注1) URL https://github.com/mackerelio/mkr

注2) URL http://stedolan.github.io/jq
注3) URL http://stedolan.github.io/jq/manual

▼ 図1 mkrで特定ロールのホスト一覧情報を取得

```
# mkr hosts --service My-Service --role proxy
[
    {
        "id": "2eQGEaLxiYU",
        "name": "myproxy001",
        "status": "standby",
        "roleFullnames": [
            "My-Service:proxy"
        ],
        "isRetired": false,
        "createdAt": "Nov 15, 2014 at 9:41pm (JST)"
    },
… (略) …
]
```

さらに応用として、My-Serviceサービス、proxyロール配下のホストのステータスをすべてworkingに変更してみます。mkr updateコマンドの引数には複数のホストIDを指定できるため、**図4**のようにmkr hostsコマンドの結果をmkr updaeコマンドの引数としてシェルに展開させます。参照系と更新系のコマンドをシェル上で組み合わせることにより、特定サービスや特定ロール配下のホスト群に対しての一括操作ができます。

ここまで、mkrを単独で利用する方法を紹介しました。しかし、mkrの活用はこれにとどまるものではありません。たとえば、本連載の第4回にて紹介したtmux-cssh[注4]と組み合わせることもできます。My-Serviceサービスのappロール配下のすべてのホストにsshログインする場合は、**図5**のようなコマンドになります。

執筆現在、mkrは開発版のみの提供です。GitHubリポジトリ[注5]からソースコードと実行ファイルを取得できます。ぜひご利用ください。

▼図2　特定ロールの、hostIDのみの一覧を取得

```
# mkr hosts --service My-Service --role proxy |
 jq -r -M ".[].id"
2eQGEaLxiYU
2eQGDXqtoXs
```

▼図3　特定ホストのステータスとロールを更新

```
# mkr update --status maintenance --rolefullname
 My-Service:db-master 2eQGEaLxiYU
```

▼図4　ロール配下のホストのステータスを一括で変更

```
# mkr update --status maintenance --role
 My-Service:db-master $(mkr hosts --service
 My-Service --role proxy | jq -a -M -r ".[].id")
```

▼図5　tmux-csshと連携して複数のホストに一括ログイン

```
# tmux-cssh $(mkr hosts --service My-Service --role app)
```

cookbook-mackerel-agent

Mackerelを利用するためには、ホスト情報やメトリック情報を取得するために、監視対象ホストにmackerel-agentをインストールする必要があります。mackerel-agentはLinuxホストへのインストールを容易にするために、rpmおよびdebパッケージを、それぞれaptリポジトリ、yumリポジトリ上で提供しています。

しかし、実際にはパッケージをインストールするために、aptコマンドやyumコマンドを直接叩かずに、ChefやAnsibleのような構成管理ツールを利用されている方も多いでしょう。

そこで、mackerel-agentのインストールをサポートするためのcookbook-mackerel-agentを用意しました。これはmackerel-agentパッケージのインストール、mackerel-agent-pluginsのインストールおよびmackerel-agent.confを設定する機能を提供します。RHEL系とDebian系の両方のディストリビューションに対応していますので、yum・aptなどのパッケージマネージャの違いを意識することなく利用できます。

Berkshelf（Chefのcookbookを管理するためのツール。RubyGemsにおけるbundlerのようなもの）をお使いの場合は、Berksfileに次のような1行を追加します。バージョンは執筆現在では1.2.0が最新です。なるべく最新版を使用するようにしてください。

```
cookbook 'mackerel-agent', '~> 1.2.0'
```

cookbook-mackerel-agentは基本的には、適切にアトリビュートを設定し、レシピをincludeするだけで使えます。

まず、**リスト1**にアトリビュートの設定例を載せました。必須の指定項目はapikeyのみです。node['mackerel-agent']['conf']以下の名前空間はmackerel-agent.confの設定に対応しています。次に、mackerel-agentのインストールが必要なレシピ内で、次のようにmackerel-agentのレシピをincludeします。

注4）URL https://github.com/dennishafemann/tmux-cssh
注5）URL https://github.com/mackerelio/cookbook-mackerel-agent

```
include_recipe 'mackerel-agent'
```

最後に、対象サーバに対してChefを実行すると、mackerel-agentが起動します。

cookbook-mackerel-agentはGitHub上で開発していますので、不具合などがあれば、issueで報告していただくか、修正のためのPull Requestを送っていただければと思います。

MackerelとAWSとの連携

ここからは、AWS上で構築したシステムをMackerelを用いて管理する方法を紹介します。

EC2インスタンス情報の表示

執筆現在、MackerelにはAWSをサポートするための機能として、EC2のインスタンス情報をホスト詳細画面やホスト一覧画面に表示するというものがあります。

mackerel-agent 0.14.3以降をEC2インスタンスにインストールすると、**図6**のように追加のホスト情報として、インスタンスタイプ、セキュリティグループなどが表示されます。

実際の運用では、Availability Zone単位の障害に備えて、Availability Zoneをまたがってインスタンスを配置することにより、可用性を担保することがあります。そのような場合、Mackerel上で同じロール内のホストがすべて同じAvailability Zoneに追加されていないかなどを確認できます。

ほかには、たとえば、Mackerelのパフォーマンスグラフを眺めつつインスタンスタイプを調整する場合にも、Mackerel上でインスタンスタイプを閲覧できて便利です。

EC2以外のAWSサービスとの連携

基本的に、EC2以外のAWSサービスにはmackerel-agentを直接インストールする術がありません。しかし、実際にはELBやRDSのメトリックを監視したいという要求があります。ここでは、RDSを例に取り、AWSサービスをMackerelで監視するノウハウを紹介します。

RDSのメトリックを取得し、Mackerelのグラフとして表示するための方法はいくつかあります。その中でも、RDSのインスタンスをMackerelのホストとして登録する方法が最も手軽で、扱いやすいでしょう。すでに述べていますが、RDSインスタンス自体にはmackerel-agentをインストールできません。したがって、mackerel-agentによるMackerelへのホスト登録ができません。しかし、MackerelにはAPI経由で任意のホスト情報を投稿できる機能があります。先ほど紹介したmkrコマンドを用いて、**図7**のようなコマンドでRDSのエンドポイントをホスト名として、RDSホストを作成します。

今は、ホストを作成しただけの状態です。次

▼図6　Mackerelに表示されたEC2インスタンスの情報

▼リスト1　cookbook-mackerel-agentを使うためのアトリビュートの設定例

```
node.default['mackerel-agent']['conf']['apikey'] = 'Your API KEY' # 必須
node.default['mackerel-agent']['conf']['roles'] = ["My-Service:app", "My-Service:proxy"] # 複数指定可能
node.default['mackerel-agent']['plugins'] = true # mackerel-agent-pluginsパッケージをインストールするかしないか
node.default['mackerel-agent']['conf']['plugin.metrics.nginx'] = {
  'command' => '/usr/local/bin/mackerel-plugin-nginx -port=8081 -path=/server-status',
}
```

に、メトリックを取得し、投稿する必要があります。RDSのメトリックを取得し、投稿するには、**図8**のようにmackerel-plugin-aws-rdsとmkrを組み合わせます。mkr throwコマンドは、mackerel-agent-pluginのメトリック出力形式を入力として、<hostId>で指定されたホストに対して、メトリックを投稿できます。

図8のコマンドを1回実行すると、現時点のデータが送られるだけです。毎分メトリックを投稿するためには、開発ホストなどの適当なホストのcronを使うのが手軽でしょう(**図9**)。

さらに、RDSの実体はMySQLですので、mackerel-plugin-mysqlとmkrを組み合わせた、MySQLのメトリック表示もできます(**図10**)。

このようにして、ホストを作成し、メトリックを投稿すると、loadavgなどの基本メトリックグラフを省いたカスタムメトリックのみのホスト詳細ページができあがります(**図11**)。当然、メトリックに対して監視条件を指定しアラートを発生させることができます。

▼図7　mkrコマンドを用いてRDSホストを作成

```
# mkr create --roleFullname My-Service:rds-master
 examplerds01-xxxxxxxxxxx.ap-northeast-1.elb.amazonaws.com
```

▼図8　RDSのメトリックを取得し、投稿

```
# mackerel-plugin-rds -identifier=examplerds01 |
 mkr throw --host <hostId>
```

RDS以外にも、mackerel-plugin-aws-elb、mackerel-plugin-aws-elasticache、mackerel-plugin-aws-ec2-cpucredit、mackerel-plugin-aws-sesなどのAWS関連のプラグインがあります。これらを活用して、AWSサービスのメトリックをMackerelで表示し、監視できます。

◆　◆　◆

Mackerel関連の運用ツール「mkr」「cookbook-mackerel-agent」、さらにAWSとの連携のノウハウについて紹介しました。

MackerelではREST API[注6]を公開していますので、ユーザのみなさんがAPIを利用した運用ツールを自作できます。ぜひ、みなさんの運用環境に合わせたオリジナルのツールを作ってみてください。SD

注6）URL http://help-ja.mackerel.io/entry/spec/api/v0

▼図9　毎分メトリックを投稿するcron

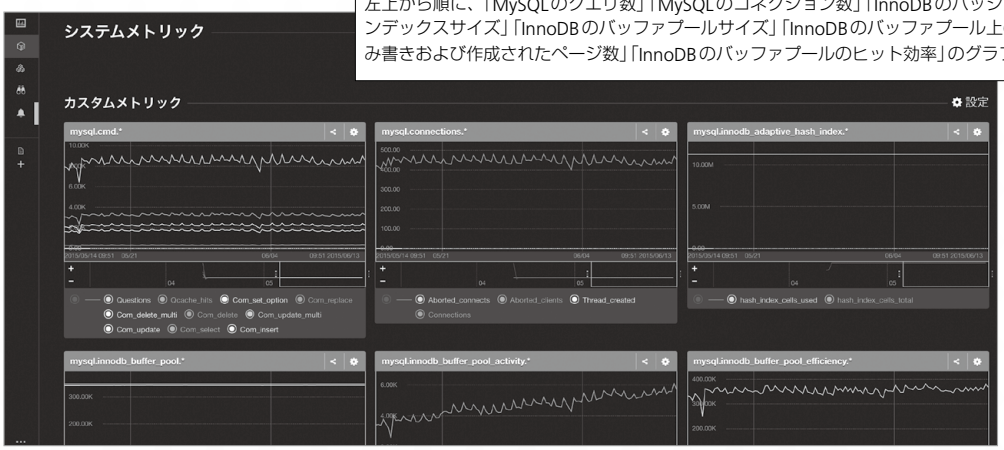
```
*/1 * * * * root /usr/local/bin/mackerel-plugin-aws-rds -identifier=examplerds01 |
 /usr/local/bin/mkr throw --host <hostId> 1>/dev/null | logger -t mackerel-plugin-aws-rds
```

▼図10　MySQLのメトリックを表示するcron

```
*/1 * * * * root /usr/local/bin/mackerel-plugin-mysql -host=examplerds01-xxxxxxxxxxx.ap-northeast-1.
elb.amazonaws.com -username=monitor -password=monitor |
 /usr/local/bin/mkr throw --host <hostId> 1>/dev/null | logger -t mackerel-plugin-mysql
```

▼図11　RDSのメトリックグラフ

左上から順に、「MySQLのクエリ数」「MySQLのコネクション数」「InnoDBのハッシュインデックスサイズ」「InnoDBのバッファプールサイズ」「InnoDBのバッファプール上の読み書きおよび作成されたページ数」「InnoDBのバッファプールのヒット効率」のグラフ



書いて覚える Swift 入門

第 7 回 Swiftが愛される理由

Writer 小飼 弾（こがい だん）　twitter @dankogai

WWDC15で得られたものは何か？

前回と今回の間に、WWDC15[注1]がありました。素人的には、「見るべきものがなかった」前回以上に「つまらない」WWDCだったかもしれません。前回同様、新ハードウェアの発表はゼロ。Mac OS XとiOSのバージョンが1つずつ上がるのはいつもどおり。しかもOS X v10.11の名前「El Captan[注2]」は、Mac OS Xの名前の由来であるYosemite[注3]の中の地名。命名的には、Mac OS X v10.5 LeopardとMac OS X v10.6 Snow Leopardより変化に乏しいと言えなくもありません。

その「素人的にはつまらなかった」前回が、「玄人的にはサイコー」だったのは、新言語Swiftの発表に尽きるでしょう。それからわずか1年で、Stack OverflowのDeveloper Surveyで「最も愛される開発言語（**図1**）」の座を射止めました[注4]。

しかし、現時点において、Swiftで開発できるソフトウェアはMac OS XとiOSという、わずか2種類のプラットフォームのみ。「Watch OSもあるではないか」という意見もありますが、Apple WatchがiPhoneを必須とするのと同様、Watch OSが現時点でiOSを必須とする以上、ひいき目に見ても対象プラットフォームは3ではなく2.5といったところでしょう。しかもすべてApple製ハードウェア上でしか動かない。あくまで現状は「Apple帝国公用語」であって、「ギークの共通言語」ではないのです。

Appleは、それが半年後に変わることを公約しました。

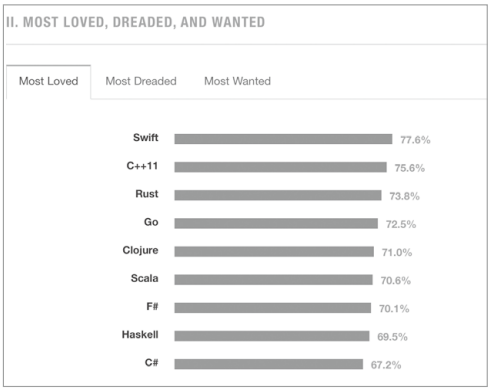
Swiftは2015年末にオープンソースソフトウェアになるのです。

Appleの公約

WWDC15でAppleが公約したのは、次のとおりです[注5]。

- Swiftのソースコードが OSI認可のpermissiveなライセンスのもとで公開されること
- Apple自身がMac OS X、iOSに加えLinuxポートを寄贈する所存であること
- ソースコードにはSwiftコンパイラおよび標準ライブラリが含まれていること

▼図1　Stack OverflowのDeveloper Surveyより



注1）https://developer.apple.com/wwdc/
注2）https://en.wikipedia.org/wiki/El_Capitan
注3）https://en.wikipedia.org/wiki/Yosemite_National_Park
注4）http://stackoverflow.com/research/developer-survey-2015
注5）https://developer.apple.com/swift/blog/

本連載第0回で、筆者はこう書きました。

> *口上だけみると、SwiftはiOS/OS Xアプリケーション開発専用に見えますが、専用にしておくにはもったいないほどよくできた言語で、学べば学ぶほど汎用向けの言語であることが明らかになってきます。Swiftをオープンソース化するつもりがあるかをAppleはつまびらかにしていませんが、同社のオープンソース戦略、とくにLLVMへのコミットメントを考えるとその可能性は低くないと筆者は考えています*

予言というより願望だったのですが、それが成就したといってもよいでしょう——このSwiftの沿革は、Objective-Cのそれと比べると実に興味深い。

Back to the Future

Swiftの登場まで「Apple帝国公用語」だったObjective-Cは、

- クロスプラットフォームな
- C言語上位互換の
- オブジェクト指向言語

として誕生しましたが、

- NextSTEPで採用された以外は、(gccに標準サポートされているにもかかわらず)他プラットフォームではあまり採用されず
- NeXT社のAppleによる買収——という形をとった、Steve Jobsによる「大政奉還」—にともなって、Mac OS Xの事実上の標準開発言語となり
- それがiPhone OS、後のiOSにも引き継がれ
- 事実上のAppleプラットフォーム専用言語

となって今にいたっているわけですが、Swiftは

- Mac OS XおよびiOS専用言語として生まれ
- 事実上どころか公式のAppleプラットフォーム第1言語(Lingua Prima)となった後
- オープンソース化とクロスプラットフォーム化がなされる

というわけでObjective-Cの歴史を逆再生しているようです。それも20倍速ぐらいで。

Appleはなぜそうしたのでしょう?

後出しジャンケンは勝ってなんぼ

21世紀において、電脳言語というのは、はじめからクロスプラットフォームかつオープンソースとして公開されるのが常識になってきています。Mozilla Foundationが公開したRust[注6]しかり、Googleが公開したGo[注7]しかり。PerlやPythonやRubyは前世紀どころか「オープンソース」という言葉が生まれる前からそうでしたし、それらの言語の成功が、「言語はオープンソースが当たり前」という現況の原動力になったのはたしかでしょう。

しかし悲しいかな、言語が普及するにあたって最も重要なのはオープンソースであることではないのです。RustとGoを比較してもそれはわかります。言語としてより先進的なのはどう見てもRustで、SwiftもRustから多くの特長を取り入れています。しかしどちらがより使われているかといえば、Goのほうでしょう。なぜか?

ライフワーク(life work)ならぬ「ライスワーク(rice work)」、つまりその言語を学ぶことで日々の糧を得ることがより容易だからです。普及した言語におよそ例外は見当たりません。COBOLとFORTRANの時代から、数多の言語が群雄割拠する現在に至るまで。思い起こせば、Java、正確にはJavaの生みの親であるSun Microsystems(以降Sun略記)による標準のJDKもオープンソース化されたのは、1995年から11年後の2006年。しかもJDKの「ほとん

注6) http://www.rust-lang.org

注7) http://golang.org

ど」であってすべてではありませんでした。

言語を普及させるのに、オープンソースであることは必須ではないのです。

それではなぜ、Appleはオープンソース化に踏み切ることにしたのでしょうか？

主導権を握り続けるためだ、と筆者は考えています。

Javaの事例は、格好の反面教師となっています。SunがJDK——の全部ではなく大部分——をオープンソース化した2006年、Sunはかつて誇っていた主導権をさまざまな分野で失っていました。サーバはLinuxに押され、もともとJavaが狙っていたシンクライアントの分野は分野そのものがテイクオフしたとは言い切れず……Javaのオープンソース化は同社の起死回生の一手(の一環)でしたが、結局同社はそれから4年後の2010年、Oracleによって買収されたのは読者のみなさんもご存じのとおりです。

そのJavaの本来の目的に最も忠実なのは、Androidでしょう。Javaの深慮遠謀はスマートフォンによって花開いたわけですが、その果実がもたらされる前に生みの親はなくなってしまいました。その「最も成功したJava」であるAndroidのJavaが、「それは本物のJavaじゃない、勝手に使うな」とばかりにSunを買収したOracleに訴えられているにいたってはもう何がなんだかです。

食えなきゃ誰も食ってくれない。

オープンでなければ、誰も食いつづけてくれない。

よって、まずは食える言語としての地位を確立しておき、まだ主導権がある内にオープンにする。

それが、Swiftに対するAppleの基本戦略だったと弾言できます。

歴史の教訓によく学んだ、悪く言えば見事な後出しジャンケンと言えるでしょう。

それだけに、負けるわけにはいかない。

言語の普及競争において、Swiftほど高いオッズを持つ言語が見当たりません。

Swift2「後方互換性？ なにそれおいしいの？」

ところで、オープンソース化されるSwiftは、Version 2以降のものになります。で、このSwift2、Swift1のコードは**ほとんどそのままでは動きません**。

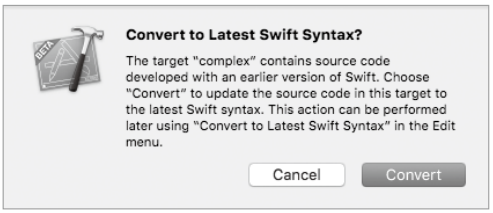
「後方互換性？　なにそれおいしいの？」とばかりの改変がもりだくさんです。

たとえば`println()`と`print()`が`print()`のみになり、改行の有無を2番目の引数`appendNewline: Bool = true`(つまりただの`print()`はSwift1の`println()`と等価)という変更だけみても、「ひょえぇぇ」です。Python 2とPython 3の違いに匹敵する違いがあります。

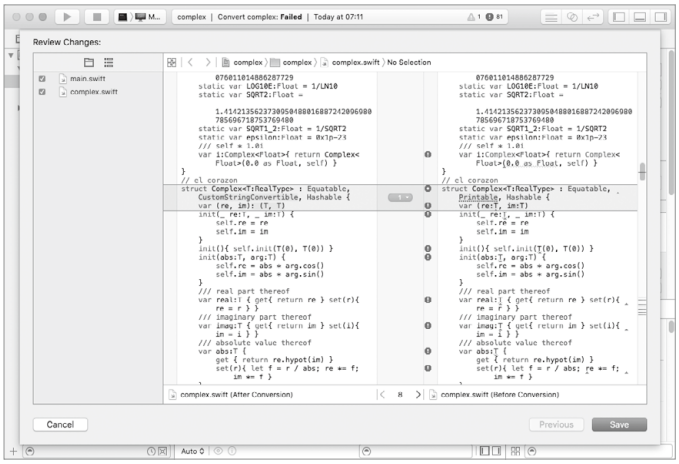
その代わり、Swift 2の現時点における唯一の実装、Xcode 7 betaにはコンバータが付いています(これもPythonっぽい)。Swift 1のプロジェクトをSwift 2に変換してくれるのですが、試しに`swift-complex`[注8]を食わせてみたところ、手による修正ゼロでコンバートできました(**図2**、**図3**、**図4**)。同プロジェクトはプロトコル、ジェネリクス、演算子定義といったSwiftの特長をめいっぱい活用したプロジェクトであることを考えると、まさに驚き桃の木もといリンゴの木。

ただし、Swift 2の言語仕様[注9]は、執筆現在に

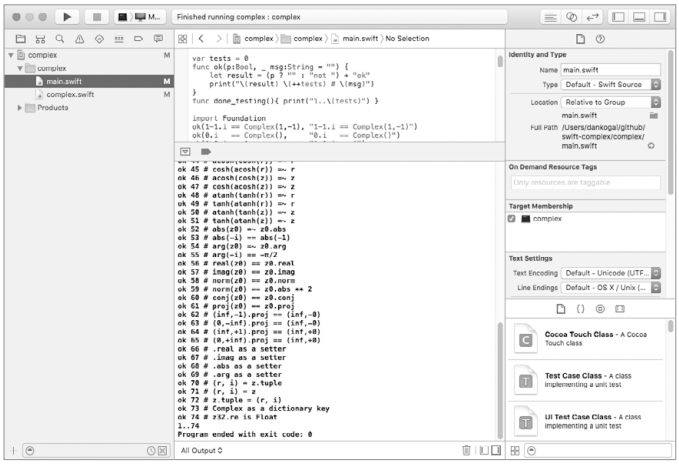
▼図2　コンバート開始



注8) https://github.com/dankogai/swift-complex
注9) http://adcdownload.apple.com/WWDC_2015/Xcode_7_beta/Xcode_7_beta_Release_Notes.pdf



◀ 図3　変更点が指摘される



◀ 図4　いっきに修正、手ずからの作業はゼロ

おいて**リリースノート**のみ。Swift 0→Swift 1のころの変更の激しさを思い起こすと、βが取れるまではかなり頻繁な変更が予想されます。Swift 2への移行は正式版が出た後で、ただしSwift 2正式化以後はSwift 1の後方互換性サポートも捨てるのがよさそうです。筆者がGitHubに上げているプロジェクトはそうするつもりです。

import POSIX // ???

今回はほとんどコードが出てきませんでした。いや、政治も立派なcode(法)ではあるのですが、Swiftのソースコードが出ていなかったのはたしかです。というわけで1つだけ。

Appleのいうところの「標準ライブラリ」とはいったい何を指すのでしょう？

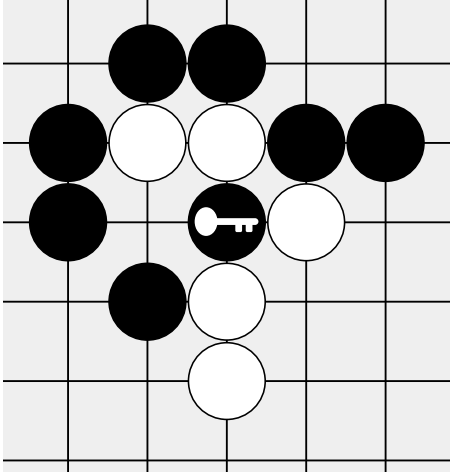
同社の文化と沿革から考えて、FoundationやIOKitまでそこに含まれることは考えがたい。現在「最低限*nix的なAPI」は、Darwinという名前でこれはオープンではあるけれど標準とは言い難い。というわけで今から予想しておくとそれはPOSIXという名前になるのでないでしょうか。

```
import POSIX // write once, run everywhere
```

この予測が当たるかどうか、半年後が今から楽しみです。SD

セキュリティ実践の基本定石

すずきひろのぶ
suzuki.hironobu@gmail.com



みんなでもう一度見つめなおそう

【第二三回】IP電話のセキュリティ

「かけた記憶のない国際電話の料金が請求され、調べてみるとIP-PBX（IP電話交換機）に何者かが接続し、そこから勝手に発信していた」という報道がつい最近ありました。突然、高額な国際電話料金を請求される問題は古くて新しい問題です。その原因となっているIP電話やIP-PBXについて考えていきます。また、家庭用ルータに付属するIP電話機能などのセキュリティも一緒に考えてみましょう。

最近多発するIP電話乗っ取り事件

まずは実際にあった事件を読売新聞の記事から引用します。

インターネット回線を利用したIP電話が乗っ取られて高額の電話料金を請求される問題で、東京都内の通信機器販売会社の顧客のうち4割が被害を受けていたことが分かった。

読売新聞（2015年6月13日）[注1]より

この報道がなされる前日の2015年6月12日には、総務省から「第三者によるIP電話等の不正利用に関する注意喚起」という文章が出されています。こちらはより具体的です。

利用者がIP電話等の電話サービスを利用する際にインターネットに接続している通信機器（PBX、IP電話対応のルータ等）におけるソフトウェアやハードウェアの設定の問題や、セキュリティ上の脆弱性を突いた「なりすまし」や「乗っ取り」による不正利用が原因であることが確認されています。

総務省サイト「第三者によるIP電話等の不正利用に関する注意喚起」[注2]より

これらの情報から、IP-PBXやIP電話対応のルータなどのアカウントの乗っ取りや不正中継が発生している様子がわかります。

IP電話とは

IP電話とは、別の言い方をすると、SIPプロトコルを利用しているTCP/IPネットワーク（以下単にネット）上の音声通信（VoIP：Voice over IP）です。

VoIPだけを取り上げると、ネット上であれば、音声をやりとりするのはTCPプロトコルで接続しても、UDPプロトコルでパケットを送っても、既存の規格を使っても、独自の規格を使っても、とにかく音声さえ届き通話さえできればVoIPです。Skypeでも、LINEの無料通話でも、Googleハングアウトでも、音声（Voice）がTCP/IPネットワークを介して（over IP）通信できているので、VoIPということになります。

一方、SIPはSession Initiation Protocolの略で、直訳すると「セッションを初期化するプロトコル」となります。これはネット上にあるSIP端末を呼び出す役目を果たします。SIP規格は1999年のRFC 2543が最初です。

筆者の記憶を思い出すと、インターネット電話という、これまでの電話にとってかわる重要な転換点ということで、そこでの主導権争いや、すでに

注1）「都内の会社 客4割被害　IP電話乗っ取り 総務省聞き取りへ」 読売新聞、2015年6月13日、朝刊、38面
注2）http://www.soumu.go.jp/menu_kyotsuu/important/kinkyu02_000191.html

ITU-T（国際電気通信連合の電気通信標準化部門。簡単にいうと電話の国際規格を決めるところ）で同様の規格が策定されていたことなど、いろいろな要素が重なりたいへんだったので、ずいぶん議論が長引いて、結局1999年まで延びに延びたという印象があります。

さて、ここでのSIP端末とは、SIPプロトコルが使えるVoIPソフトウェアを搭載した機器のことです[注3]。古典的に電話の形をしたSIP端末もありますし、インターネットに接続しているルータがSIP端末としての能力を持っており既存の電話をそのルータに接続するものもありますし、あるいはPCやスマートフォンのアプリケーションとして用意されているSIPアプリを利用してSIP端末とすることもあります。

SIP端末はSIPサーバにネット経由で接続します（**図1**）。SIPサーバは同じLAN上にあるかもしれませんし、組織内ネットワークに用意されているかもしれませんし、外部のサーバやクラウド上で動作しているかもしれません。SIPサーバは、接続してきたSIP端末が登録されているかをデータベース（数が少なければファイルの中のリストとして登録し保持している場合もあります）と照合し、受け入れます。通常はアカウント名（内線電話番号）とパスワードで接続します。

同じSIPサーバに収容されている場合、そのSIP端末は内線につながれている内線電話そのものです。それは内線電話のシステムのようなものだといっても過言ではありません。ですので純粋なSIPサーバとしてだけではなく、構内交換機（PBX）が必要としている各種機能（たとえば音声案内や留守番電話など）を持っています。そのためIP-PBX（IP - Private Branch eXchange：IPベースの構内交換機）とも呼ばれます。

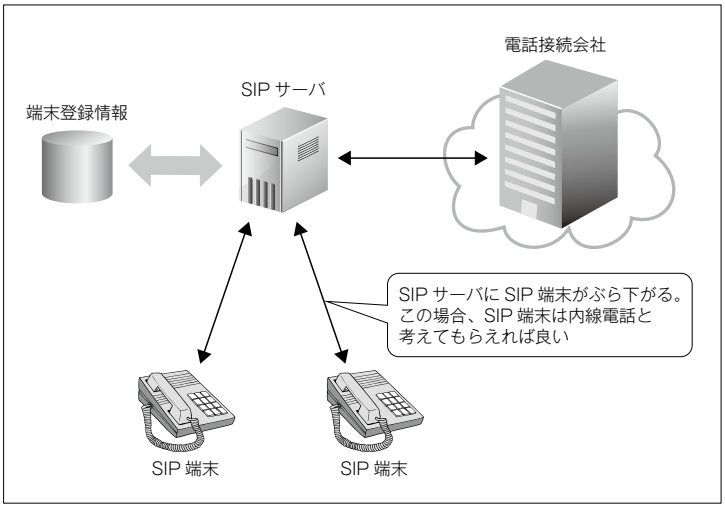
基本的にSIPサーバは、SIPサーバに接続しているSIP端末同士を呼び出し通話することが可能なだけです。もし、普通の電話と同じように電話網を経由してどこにでも電話をかけられるようにしたければ、IP-PBXから接続が可能な（外部の）電話会社と契約し、IP-PBXからその電話会社と接続できるようにします。これで内線から外部に電話をかけられるようになります。

IP-PBXのメーカーとして、海外ではCisco Systems社、Avaya社、日本国内では富士通、沖電気、日立、NECなどの会社が知られています。一方で、現在、海外でも国内でもオープンソースのIP-PBXであるAsteriskが急速に広がってきています。

Asterisk

Asteriskは、米Digium社が開発しているオープンソースのIP-PBXです。運用も安定していて機能が良いので、IP-PBXのプラットフォームとして広く使われています。IP-PBXが登場したことで、これまでたいへん高価であった内線電話システムが大幅にコストダウン可能になりました。さらに

◆**図1　SIP/IP-PBXのモデル**



注3）VoIPのコーデック（音声フォーマット）とは別な話ですので注意してください。

Asteriskの登場で、内線電話システムのコストを劇的に低く抑えられるようになりました。現在では、Asteriskを(あるいはそのコア部分を)組み込んでいる製品がいくつもあります。

Asteriskは、これまでのIP-PBXのためのサーバだけではなく、クラウドやVPS、はてはRaspberry Piの上でも動作するIP-PBXです。また、Digium社はビジネス版Asteriskを販売していますし、日本国内にもそれを扱う代理店があります。

オープンソースですから、自分で自作SIPサーバを立てて運用している人もいます。筆者自身も実験SIPサーバをクラウド上で用意していて、運用の実験をしています。

国際電話詐欺

知らないうちに国際電話を使っていて多額の請求が行われる。これは昔からある国際電話のしくみを使った詐欺です。その金額の何割かが犯罪者にフィードバックされるしくみです。

これはインターネット時代よりも古いパソコン通信の時代からありました。ポルノサイトに自動的に接続するというプログラムがあり、それを入手してモデムを使ってパソコンで接続すると大量のポルノ写真が手に入ります。時間をかけて大量にダウンロードしたはいいが、実はそれは謎でもなんでもなく海外のサイト。1ヵ月後に数十万という金額が電話会社から請求されることになります。こういう事例は1980年代にはすでにありました。

携帯電話、スマートフォン時代の今も、この詐欺は形を変えて健在のようです。これは少なくとも2013年には行われていました。知らない相手から自分に電話がかかってきます。出る前に切れるか、出てもすぐに切れます。おかしいなと思い、かけ直したら延々と音楽が流れます。電話番号を確認するとアフリカあたりの小国だったりします。

たぶん、これはその遠くの小国から直接かかってきたわけではなく、日本国内から発信者電話番号を偽装してかけてきたものと筆者は推定します。実際に国内でも携帯電話に発信者電話番号が偽装されて着信し、犯罪に利用されていたという報告があります[注4]。

これであれば、かけるほうのコストは小さく、国際電話のコールバック詐欺で1回に得られる金額はたとえ数百円とか数千円とかいう単位であっても大量にかけてくるでしょうから、最終的にはかなりの金額になるのではないでしょうか。しかも、数百円や数千円ならば、契約している電話会社に連絡し、支払いを差し止めるといったことは、面倒なのでしないでしょうから、泣き寝入り＝犯罪者の懐に入る、という図式になっているかと思います。

IP-PBXの不正利用

IP-PBXに登録しているSIP端末のアカウントとパスワードがわかれば、当たり前ですが、このIP-PBXを利用することができます。

そして、このIP-PBXが電話会社に発信をかけられるようにしており、インターネット側(外部から)からのアクセスには特段のフィルタリングなどしていないと仮定します。その場合、世界中どこからでもIP-PBXに登録しているSIP端末(SIPクライアント・アプリケーション)であれば接続可能な設定になっている可能性が高いでしょう。

ここで2つの疑問が出ると思います。1つは「どうやって現在登録されているアカウントがわかるのか」、もう1つは「どうやってパスワードがわかるのか」です。

Asteriskのログを解析してみて

筆者は実験的に外部のクラウド上にAsteriskを立ち上げていて、SIPクライアントの利用方法の運用実験をしています。そのログを分析した経験からわかったことを書いてみたいと思います。

なお、これは意図的に記録させるためにfail2ban

注4) 発信者電話番号が偽装されて着信する通話について http://www.tca.or.jp/information/camouflage.html

などの設定をしませんでした。通常はfail2banなどを導入してフィルタリングします(後述)。

パスワードの総当たり攻撃

ログにはextrensions.confのデフォルトにあるナンバーをターゲットにして攻撃をしかけられている形跡が大量にあります。

Asteriskの設定ファイルであるextrensions.confは、Asteriskのダイアルのプランを記述するファイルです。ここには、1000、1234、1236、500、600という電話番号がサンプルとして用意されています。ログには、この番号とアカウント名が同じで、かつSIP端末が割り当てられているという前提で、パスワードの総当たり攻撃をしかけてきている記録が残っています。

2014年7月22日3時55分から2014年7月24日12時19分の間に**表1**のようなパスワードの総当たり攻撃が記録されていました。

デフォルトでも筆者の設定でも、これらのダイヤル番号にはSIP端末が割り当てられていないので、いくらパスワードを試行しても接続するのは無理です。さて600番の内訳を見てみると、おもに3つのIPアドレスが発信源でした(**表2**)。

辞書攻撃(アカウントとパスワードが連動)

たとえば、1つのアカウントに対してパスワード試行は5回程度のものがこれにあたります。電話番号、アカウント、パスワードが同じ、あるいはパスワードは123456といった単純なものであるという前提で次々にアカウントを試していっていると考えられます。実際のログを見ると、狙っているアカウントは100、101、1010、1111、2020……といった単純なものになっています。

拡張番号を試す

電話がつながっているとして、直接、拡張番号を入力する方法です。直接外線にかけることを試みているようです。筆者もこの方法が成功するであろう設定(あるいは設定ミス)は、わかっていません。

総当たり攻撃対応

fail2ban[注5]は総当たり攻撃などに対応するためにログファイルを解析し、何度も繰り替えして失敗している相手のIPアドレスをiptablesでフィルタリングしてしまう定番のツールです。Asteriskだけではなく、SSHなどのパスワード総当たり攻撃にも使えます。

高い率で成功している点に着目

冒頭で紹介した新聞報道の「東京都内の通信機器販売会社の顧客のうち4割が被害を受け」という点に着目して考えてみると、これはすべてが同じ問題を抱えていたのではないかと思います。つまり、どの種類のIP-PBXを使っていたかはわかりませんが、脆弱性と呼べるような共通の問題があったと考えるほうが合理的です。

Asteriskもそうですが、IP-PBXの設定はわかりやすいものではありません。ユーザが自分でマニュアルを見ながら簡単に変更する、といったものではありません。業者が顧客ごとに毎回バラバラのものを設定するのではなく、顧客すべてが共通の設定

◆**表1　筆者のAsteriskサーバに来たパスワード総当たり攻撃**

電話番号	パスワード試行回数
1236	368,761
1000	386,474
1234	411,343
500	635,920
600	1,226,261

◆**表2　筆者のAsteriskサーバに来たパスワード総当たり攻撃(600番の内訳)**

IPアドレス	国・地域	パスワード試行回数
その1	ドイツ・ハノーバー	492,783
その2	アメリカ・ジョージア州	377,416
その3	アメリカ・ジョージア州	356,040

注5) http://www.fail2ban.org

ファイルを使用していた可能性があります。つまりアカウントもパスワードも同じであったという可能性です。

ありがちなのが、IP-PBXにテスト接続用の電話番号／アカウントが用意されており、そのパスワードが電話番号アカウントの数字の並びと同じか、あるいは12345といった定番のものになったまま出荷されている可能性です。もしかするとユーザ・マニュアルには「テスト用のパスワードはすぐに変更すること」と説明をしているのかもしれません。そのため、残りの6割は被害が及ばなかったということも考えられます。

いずれにしても共通の問題がないのに偶然、顧客の4割が被害を受けたというのは、通常では考えられません。

IP電話機能を持つ家庭用ルータ

050IP電話対応のSOHO家庭用ブロードバンドルータを使ったIP電話の場合、050の電話番号を割り当てるIP電話事業者（SIPプロバイダ）に接続するだけのSIP端末（ここにアナログの電話をつなげる）と、複数のSIP端末を扱える機能を持っているものと、いろいろなものがあります。

これらは機材に大きな脆弱性が発見されない限り安全なように作られています。単純にSIP端末の機能しか持たないものであれば、あとに述べる問題以外に関しては大きく心配する必要はないでしょう。

簡易なIP-PBX機能があっても内部のIPアドレスのみアクセスできる仕様になっているでしょうから、不正に外部からこのルータにSIP端末を接続する、というような問題が発生する可能性は小さいでしょう。

ただし、このIP電話機能を持つ家庭用ルータから発信される可能性が小さいからといって国際電話の料金が請求されない、というわけではありません。なぜならば（ユーザが設定できないタイプの）SIPプロバイダ側のアカウントとパスワードが盗まれる可能性があるからです。この場合、ユーザ側でできることはあまり多くはありません。

パスワードで防御する限界

対策を施すとしても、現在のSIPのプロトコルを使う限り、本質的にはユーザIDとパスワードの強化しかありません。電子署名の機能が入っていて、それでサーバ側でユーザを認証して高い安全性を保つといったプロトコルではないからです。

IP電話専用の機材を使うといった時代ではなくなり、現在ではスマートフォンやパソコンのアプリケーションからIP電話をかける場面も多いかと思います。その状態でマルウェアが、SIPのユーザIDとパスワードを盗んでいくことも十分視野に入れて考えなければなりません。

自分のSIPプロバイダのアカウントが盗まれてしまった場合、世界中どこからでもかけることができますし[注6]、それが盗まれて使われてしまっていても自分で気がつくチャンスはほぼないでしょう。結果として、突然、高額な国際電話料金が請求されて驚くということになります。

国際電話の禁止

筆者がこの手の被害を知ったのは2007年です。被害側となったヨーロッパのとあるSIPプロバイダのエンジニアの講演を聞きました。これもユーザになりすまし、SIPプロバイダに接続していたものでした。ですから、これらの詐欺は新しいようで、古い話ではあります。

Skypeのようにプリペイド方式であれば、どんなに使われようと、事前に買った分までしか被害はありませんが（それでも困りますが）、後払い方式だと青天井になってしまいます。これでは対応のしようがありません。デフォルトでIP電話からは国際電話はかけられないようにするのも1つの方法でしょう。ただし、すべてのIP電話の会社がこのような

注6）厳密な運用をするIP-PBXでは、接続される機材のIPアドレスやMACアドレスでアクセス制御をしている場合がありますが、少なくとも日本国内のSIPプロバイダでこのような厳密な管理をしているところを、筆者は知りません。

対応を取ってくれるわけではないようです。

たとえば、国際電話料金のレートが高額な地域にはかけることができないといったサービスを提供しているようなSIPプロバイダは、少なくとも国内では見つけることができませんでした。

これまでに日本でも、IP電話に外部から不正に接続され、多額の国際電話料金を請求されるといった事例はありました。ただそれは、雑誌やWebからの情報を見て自分でIP-PBXを立ち上げて設定し、さらに050電話を使えるようにしていて、そこに穴があったため、というケースだったようです。あるいは、トラブルがあった場合でも、SIP端末のユーザIDもパスワードもSIPプロバイダが管理していて、SIPプロバイダが被害を被るという形で対処しており、あまり表には出なかったようです。

今回は、ユーザ側はとくに知識もなくIP-PBXを導入し、業者が十分な配慮をしていなかったということが伺い知れる状況で、IP-PBXを使うユーザ側から不正な国際電話が発信され、高額な通信費が請求される[注7]という、これまでとはずいぶん違うケースだと言えるでしょう。だからこそ総務省は「第三者によるIP電話等の不正利用に関する注意喚起」という告知を出したのだと思います。

また「面倒くさいからお金を払う」という単純な話でもありません。なぜならばお金を払えば、それは犯罪者に届くからです。お金が入るのですから、それゆえに犯罪は続くことになるでしょう。

すでに古い問題に分類されるであろうSIPアカウントの乗っ取りによる国際電話詐欺ですので、日本でもとうとう表に出てきたか、という感じでとらえてもらえれば良いと思います。個人で何か対策を施すといっても限りがあります。これはきちんと業界団体や省庁で連携を取り、環境を整備すべき問題だと筆者は考えています。

金銭目的以外のアカウント狙い

筆者はAsteriskサーバのログを分析していてちょっとしたことに気づきました。

Asteriskのパスワードクラッキングの発信元は、アメリカ、ドイツ、フランスといったインターネット先進国からが多いのですが、それに混じってパレスチナからの記録がありました。数ヵ月を通してみるとコンスタントにやっているのでもしかすると同じ人、あるいは同じグループなのかもしれません。

WCLSCAN[注8]の経験からは中東方面からの脆弱性探しなどのパケットはイスラエルかトルコがほとんどでそれ以外の地域はめったに見ることができません。ですので、Asteriskへはパレスチナからコンスタントに探りを入れてくるのはたいへん興味深いところです。これは世界に散らばっている無名なSIPサーバを見つけ、それを通話に使い、連絡の中継地点にしたいのではないかと考えると合点がいきます。

まとめ

突然騒がしくなった気がするIP電話の不正利用による高額な国際電話料金の請求ですが、これはプロの詐欺師集団が昔からやっている犯罪モデルです。それを近年ではIP電話に置き換えただけのものです。外部からIP-PBXへ不正接続するといった事例はすでに7～8年前から知られています。

海外からみると日本はIP電話の普及およびIP-PBXの導入が遅れていましたが、最近は増えてきて、それに伴い今回のようなトラブルが表面化したと考えたほうが良いでしょう。

また、そのようなトラブルの経験や知識は国内では共有されておらず、右往左往している状態といって良いのかもしれません。

このような問題に対しては、セキュリティ技術の面では業界内での知識や経験を共有すること、また、不当な料金などに関しては消費者を守り、犯罪者にお金が回らない制度設計をすること、そしてこれら全体をプロモートする省庁の動きが不可欠です。SD

注7）この費用の負担は今後どう処理されるのかは、筆者は見当もつきません。
注8）筆者の研究している「インターネット早期広域攻撃警戒システム」のこと。www.wclscan.org

思考をカタチにするエディタの使い方

るびきち流 Emacs超入門

Writer | るびきち twitter@rubikitch | http://rubikitch.com/

第16回 Emacsと長く付き合っていくために 番外編コラム

今回はいつもの連載から少し視点を変え、「Emacsとの付き合い方」についてのコラムをお届けします。Emacsを自分好みにカスタマイズするのに外部パッケージは非常に便利ですが、同時にサポートの終了や設定の競合などのリスクを孕みます。そういった「利便性と代償との折り合い」を中心に、一歩引いた目線でEmacsを再考します。

Emacsは小宇宙

ども、るびきちです。本連載のここ3回は基本に立ち返り、標準コマンドに光を当ててみました。さすがEmacsは奥が深く、標準コマンドに少し触れるだけでも3回分かかってしまいました。これまで当たり前に思ってきた標準コマンドの機能性を再発見してもらえれば幸いです。

しかし、標準コマンドで終わるEmacsではありません。Emacsユーザは当然、Emacsを自分好みの色に染めたいものですよね。個々のユーザの要望にとことん応えてくれるのがEmacsの魅力です。

Emacsは単なるテキストエディタではなく、テキストエディタの顔をした「Lispマシン」です。Lispというプログラミング言語の比類なき柔軟性により、ありとあらゆるタスクがEmacsで実現できます。Lispは実行中にプログラムそのものを変更できるので、あなたの意のままの形態に変化します。変数を設定したり、関数を定義したり……、Lispファイルをロードすればその時点で“世界で1つ、あなただけのためのEmacs”なのです。elispのおかげで日常的なテキスト編集がより快適になり、突き詰めれば統合開発環境にもなりますし、ゲームソフトにすらなってしまいます。

人体は小宇宙と言われていますが、筆者はEmacsも小宇宙であると考えています。Emacsは人間によって書かれた1プログラムにしか過ぎませんが、あたかも生き物のように思えるときがあります。動物は食物を摂取しますが、Emacsにとってはelispパッケージが食物（や薬）に相当します。健康を維持するには何を食べるかが重要であるように、快適なEmacsを保つにも、どんなパッケージを使うかが重要です。薬の飲み合わせがあるように、お互い相性の悪いパッケージのせいで無用なトラブルに巻き込まれることもあります。

Emacsは標準添付のパッケージも充実していますが、Emacs24からはパッケージシステムが導入されたので、世界中の外部パッケージを簡単に導入することができます。誰もが簡単に外部パッケージを導入し、設定すればすぐにあなたのEmacsに反映されます。門は開かれました。あなたのEmacsをデザインしてください。

こういう話をしていて、いつも頭をよぎるのが筆者の少年時代に遊んだミニ四駆です。電池2本で一直線にしか走れない玩具の車ですが、当時はいかに速くコースを走るかが競われていました。ミニ四駆本体を買って組み立て、電池を入れて電源を入れればすぐに走ります。しかし、より速く走らせるためにはより強力なモーターに交換したり、軽量化などの改造を施したりし

ます。取り替えられるパーツは多くのメーカーが生産していて、その選択は多岐に渡ります。「これは」と思ったパーツに取り替え、自分オリジナルのミニ四駆を構築したものでした。そして大人になった今、このノリがGNU/Linuxマシン構築やEmacs環境構築に活かされています。

拡大と収縮

筆者はEmacsを使い始めた途端いきなり魅了され、数ヵ月しないうちにelispを書くようになりました。そして「こんなコマンドがあったらいいな」を自分で実装していました。右も左もわからず、試行錯誤の日々が何年も続きました。

当時はパッケージシステムという便利な代物は存在せず、公開されているelispは少ないものでした。お手本となるelispもないなか、自分なりに学んでいました。ネット上でelispを見つけても手動でダウンロードして手動インストールの時代でした。暗中模索のなかだったとはいえ、ミニ四駆の魔改造のようにEmacsにいろいろな機能を追加する作業はとても楽しいものでした。

一気に大量のelispコードを書いては、しばらく時間がたてばその機能の存在すら忘れてしまうことを繰り返して10年以上になりました。その間、コードを書いては捨て、書いては捨てという感じでinit.elの総行数は15,000行を突破していました。自作／外部問わず、ありとあらゆるelispを突っ込みまくり、筆者のEmacsはモンスターのように肥大化しました。

現在はパッケージシステムによって検索すれば何かしら見つかる時代になり、自分でコードを書く必要性がめっきり減りました。また、日々使うelispはほんの一部にすぎないことにも気づきました。せっかく書いたコードも今となっては役立たないものも多いです。init.elをスリムにした結果、現在は5,000行くらいに落ち着いています。さらに絞れば数千行ほどになるかもしれませんが、あいにく時間が取れていません。

このように、筆者のEmacs遍歴は拡大と収縮の繰り返しです。筆者サイト「日刊Emacs」を更新するために、毎日パッケージをインストールしていますが、実運用しているものは少ないです。膨大なelispを見てきた中で、システムに対する洞察力が身につき、elispそれぞれの性格、およびトラブルの臭いを嗅ぎ分けられるようになりました。今回は長年のEmacs歴で学んだ教訓をお伝えしていきます。

外に開かれたEmacs

今からEmacsを始める人はとても恵まれています。なぜならパッケージシステムにより、世界中の数千ものelispパッケージから簡単にインストールして試せるからです。パッケージシステムが登場する前はEmacsWikiというEmacs情報集積場にノウハウやelispが集められていましたが、パッケージシステムほど使いやすいものではないので、elisp開発者はパッケージへと移行しました。何かパッケージがほしいと思ったら`M-x list-packages`からインクリメンタルサーチやoccurでキーワード検索すればいいのです。ほかの人にも使ってもらいたいとパッケージを作成した開発者は、MELPAに登録しないわけがありません。パッケージシステムはEmacsの標準機能ですので、一度MELPAに登録したら、世界中のEmacsユーザに知れ渡るのです。とても便利な時代になりました。

RPGをプレイしたことがあるならば、船や飛空艇などの乗り物を得たときの解放感は忘れられないでしょう。これまで徒歩のみの移動で世界のごく一部しか行けなかったのが、乗り物によって新しい大陸へ自由に行けるようになったとき、ものすごくワクワクしたことでしょう。筆者も夢中になって新しい街や洞窟を見つけに世界を探険したものでした。今のEmacsは、RPGでいえば飛空艇で世界中のいたるところに行ける状態です。Emacsは、子供のころのあの楽しかった思い出を想起させてくれます。

Emacs歴が短いのならば、ぜひともいろいろ

なパッケージを試してみてください。実際に使うことによってEmacsの経験値が増えます。新たな角度から問題を眺められるようになり、より便利な方法が見えてきます。

とはいえインストールするとトラブルが怖いと思うのはよくわかります。けれどもパッケージからインストールするだけでは、すぐにコマンドが使えるようになるだけで勝手に機能が有効になったりはしません。筆者は「日刊Emacs」のために毎日新規パッケージをインストールしていますが、インストールしたことによるトラブルは経験したことがありません。インストールは無害です。

筆者が知る限り唯一の例外はbetter-defaultsパッケージで、インストールするだけで変数を変更してしまいます。しかしそれは“よりよいデフォルト設定を提供する”という動機でわざとやっています。MELPA登録はpull requestによって人力チェックが入るので、そのほかのパッケージでこのような不作法はないでしょう。

円熟してくると

これは筆者の経験ですが、Emacsに魅了され常にいじくり回し、多くのパッケージを使っていれば、次第にいろいろなことが見えてきました。ちょうど熱愛から円熟に差し掛かってきます。既存のパッケージもそうですが、実現させたい機能が見つからない場合は自分で実装していました。首尾よく実装できると、最初は作った機能に満足します。しかし、たいていの場合は次第に使わなくなり、しまいにはその機能の存在すら忘れてしまうようになりました。見方によっては時間の無駄のように思えますが、プログラミングした経験により学びが得られます。

昔は機能が欲しければ真っ先に手が動いてプログラミングを始めていましたが、今では次のように自問しています。

・この機能は本当に必要なのか？
・自分で実装する必要はあるのか？
・すでに誰かが実装していないか？

これらがNOなら自分でコードは書きません。

パッケージを実運用する際も慎重になるようになりました。「果たしてパッケージを導入してどれくらい操作性が上がるのか？」と自問するようにしています。なるべく多くの局面で操作性が上がるのならば、そのパッケージを導入すれば良いです。プログラムの最適化ではボトルネックの最適化に集中するのと同じように、Emacsの操作性のボトルネックを改善するパッケージならば導入すべきです。いわゆる「プロファイリング思考」です。すると、無闇に外部パッケージや自前の実装に頼ることが減り、標準機能に回帰するようになりました。外部パッケージは真新しい実装よりも、標準機能を拡張するものを好むようになりました。

このことはあなたに「外部パッケージに頼るな。標準機能だけを使え」と言っているわけではありません。Emacsの経験が少ないのならば多くのパッケージを経験するべきです。ちょうど海外旅行を多く経験して初めて日本のありがたみがわかるように、標準機能がそれなりによくできていると悟るようになります。

機能は適切に管理しよう

スマホを始めとする多くの電化製品は機能性をアピールしています。そういうセールストークに筆者はうんざりしています。普通の人は「こんなこともできるんだ！　すご〜い」という感じで機能性に魅入られます。しかしいざ購入したとき、アピールされた機能を使いこなせているでしょうか？　多くはNOだと思います。機能性の感動は一時のものに過ぎず、結局は基本的な機能＋αくらいしか使わないことでしょう。

筆者はEmacsについても同じように感じています。長年Emacsを使っていて、数多くのパッケージの設定を組み込んでいますが、結局常用

している機能はごくごく一部でしかないことに気づきました。せっかく設定しても長い間使っていないと設定したことすら忘れてしまいます。

このことから「できることが多いということが偉大なのではない。要は適切にコントロールすることが大事だ」という教訓を得ました。パッケージを入れまくればいいわけではありません。せっかく導入しても使い方を忘れているのでは意味がありません。機能志向は、結局は機械に使われるハメになります。

利便性には必ず代償を伴う

筆者の座右の銘を紹介します。それは「利便性には必ず代償を伴う」です。

たとえば、インターネットの普及によって人間関係をネットで済ませるような人が出てきました。常時接続されたインターネットは距離に関係なく、いくらでも相手と交信できる利便性を生みました。しかし、同時に人間関係スキルが磨かれなくなり、リアルでは希薄な関係に終わってしまう傾向になっています。

セキュリティと利便性はトレードオフであることが多々あります。sshのパスフレーズを毎回入力するならばセキュリティは強固ですが、面倒ですね。そこでkeychainを使ってパスフレーズ入力の手間を省けますが、ちょっとした不注意でスキを与えてしまいます。

これらの事例を見てわかるように、便利なものにはダークサイドがあるのです。何か便利なものを発見したときには、単に魅了されるのは極めて危険です。きちんとリスクを把握したうえで適切にコントロールしておきましょう。

普段のEmacsの使い勝手を大きく変えるパッケージにはよくよく注意が必要です。便利でもしくみが複雑なパッケージは、いざトラブルが起きたとき、自分で問題解決することが困難です。筆者は、パッケージはその利便性の代償を受け入れられる人だけが使うべきだと考えています。エラーで使えなくなった場合は、その問題を自分で解決するか、代替の手段を講じなくてはなりません。これがEmacsにおける利便性の代償です。「特定のパッケージがなきゃ生きていけない」というのは、内部構造を熟知して、すばやく問題解決できるようになって初めて言えることです。自分が理解できないパッケージは使わないほうがいいです。自分が何をやっているのかがわからない状態は、機械の奴隷に成り下がっていることと同じです。

トラブルの少ないパッケージの選び方

トラブルの少ないパッケージの選び方の指針は、なるべく通常のEmacsからかけ離れないようにすることだと考えています。“Emacs的に自然”であるということです。

まず標準パッケージで実現できることであれば標準パッケージで済ませるのが一番です。なぜなら、Emacs本体とともに継続的にメンテナンスされているからです。将来のバージョンアップでも間違いなくその機能が使えるからです。

確かに標準パッケージそのものがobsolete（サポート打ち止め）になってしまうことはありますが、その場合は新しい手段が用意されているものです。Emacs 24.4ではiswitchbがobsoleteになり、以前からある標準パッケージido（バッファやファイル名を絞り込み選択）やicomplete（ミニバッファに補完候補を表示）に取って代わられたのは記憶に新しいです。obsoleteになってもしばらくの間は削除されないので、時間のあるときにゆっくりと移行していけば良いです。

外部パッケージは、放置され、新しいEmacsでは動作しなくなるリスクが伴うことに留意してください。もっともelispは互換性を重視しているので、昔に書かれたelispもそのまま動くことが多いです。それでも放置リスクについては頭の片隅に置いておいてください。

編集コマンドのみが定義されているのは安全です。そういうコマンドはいくらあってもいいです。Emacsが新しいコマンドを学習しただけ

で、ほかへの影響がないからです。たとえばzop-to-charパッケージは`M-x zop-to-char`を実行するだけで使用できます。そしてそれを便利に感じて初めて`M-z`に割り当てればいいです。

特定のファイルの編集を快適にしてくれるメジャーモードも安全です。なぜなら、対象のファイルを開いたときに初めて有効となるからです。そのほかの局面で影響はおよびません。ファイル名とメジャーモードの関係を定義するauto-mode-alistもパッケージをインストールした時点で設定されることが多いです。たとえば、lua-modeパッケージをインストールすると、その時点で`*.lua`のファイルに対してlua-modeになるように設定されます。編集したいファイルに対してメジャーモードが標準で存在せずに外部パッケージになっている場合は、安心して導入してください。あまり知られていませんが、`(require 'generic-x)`を設定に加えれば、多くの設定ファイル用のメジャーモードが定義されます。

Emacsの挙動を変更するパッケージには注意が必要です。それらはフックやアドバイスを定義しているので、影響が広範囲におよぶことがあるからです。また、ほかのパッケージとの相性が悪いことがあります。

自動で動作する機能は、一段と注意して使う必要があります。Emacsには「タイマー」と「各コマンド実行前後に行うアクション」が定義できます。前者は、一定時間後に自動で関数を実行します。後者は、pre-command-hookとpost-command-hookです。これらの機能はEmacsを便利にしてくれる超強力な機能ですが、代償を伴います。自動実行される関数でエラーが起きた場合わかりづらいのです。タイマーでのエラーはエコーエリアに`error`と出るだけですので見落とされがちです。2つのフックでのエラーは自動的にその関数がフックから外されます。開発者側にとってもこれらの関数のデバッグは困難を極めます。よって、派手な自動実行に過度に依存しないようにするのもEmacs的処世術です。

マイナーモードは干渉の恐れがあります。とくにキーバインドを定義しているマイナーモードは、マイナーモードの有効順によっては動作しないことがあります。うまく解決できない場合はそのマイナーモードを無効にしてください。

最後に、開発がとても活発なパッケージはしばしば非互換な変更がなされます。以前のバージョンでは動いても、バージョンアップしたとき名称が変更されたり削除されたりする場合は、元の設定では動作しません。

helmは複雑さのコストを上回る利便性

Emacsの操作性を大きく改善した大人気パッケージhelmは、以前の連載(2015年3、4月号)で紹介しました。ミニバッファにクエリを入力して絞り込み検索を行い、複数の情報源(バッファ、最近開いたファイル、ブックマーク、カレントディレクトリのファイルなど)から多くのアクションが実行できます。内部はとても複雑ですが、影響範囲は各種helmコマンド(`M-x helm-mini`、`M-x helm-for-files`など)を実行している間のみですので、Emacs全体にまで及んでいません。仮にhelm使用時にエラーが起きて使えなくなっても、元の標準コマンドやidoでしのげばいいです。よって、利便性が複雑さのコストを上回っていると筆者は考えています。

ただ、helmのありがたみはEmacs初心者にはわからないものです。標準コマンドに不便さを感じるようになって、あらためてhelmの良さが実感できるものです。中級者になればhelmは手放せなくなるでしょう。

キー割り当てを変更するのは安全

新たなパッケージを模索するよりも、なるべく標準コマンドでうまくやりくりするほうが賢明と考えます。前述したように標準機能はずっとメンテナンスされるので安心して使えます。デフォルトの設定では不便に感じた場合、キー割り当てを変更することで劇的に操作性が上が

ることがしばしばあります。

たとえば別ウィンドウに切り替える(other-window)にはC-x oと2ストローク必要です。フレームを3分割以上している場合、C-x oを繰り返すかC-x zでリピートする必要があり、ストレスがたまります。頻繁にウィンドウを切り替えるのならば、次のように1ストロークのキーに割り当て直すのが無難です。

```
(global-set-key (kbd "C-t") 'other-window)
```

これはswitch-windowパッケージなどを導入するよりも手軽です。

直前のウィンドウ構成に戻すことは標準パッケージのwinnerを使えば可能です。別のバッファに切り替えたあとに元のバッファに戻したり、ウィンドウ分割状態を戻したりすることはよくやります。winner-undoを次のように1ストローク化してしまえば、それだけでpopwinと同等の操作性を享受できます。

```
(winner-mode 1)
(global-set-key (kbd "C-q") 'winner-undo)
```

連続して実行され得るコマンドはsmartrepやhydraを導入すればプレフィクスキーを省略できます。また、repeat(C-x z)を1ストロークに割り当て直すのも、お手軽かつ強力な方法です。

1日の作業の終わりにEmacsを閉じる

筆者がいつもやっている習慣をお教えします。無駄なバッファやデータが増えるとEmacsが重くなりますが、そういう場合はEmacsを再起動することで軽快さを取り戻せます。ですので1日の作業が終わったら、お疲れさまと言ってEmacsを終了しています。tempbufパッケージはしばらく使っていないバッファを自動的に削除しますが、1日ごとにEmacsをリセットすればそれも不要です。パッケージも最新版が使われるようになります。ちょうど使ったものを片付けるようにEmacsを閉じ、明日は新しい気持ちで作業が始められる心理的なメリットもあります。あなたも試してみては？

Emacsは人生

Emacsはいくらでも強くなれます。しかし強力な機能はそれなりの代償が伴うので、あくまでもあなたの理解の範囲内に留めましょう。理解を超えたパッケージは、逆に機械に使われてしまいます。ですので、便利なパッケージに惚れ込んだ場合「もっと早く出会いたかった」と後悔する必要はありません。Emacs力が未熟な段階で出会っても受け入れ態勢ができていなかったことでしょう。今、その便利なパッケージを使っている——この事実に満足しましょう。

これは、算数と数学の関係と同じです。方程式を使えばあっさり解けてしまう算数の難問奇問がありますが、限られた知識の範囲内で悪戦苦闘したからこそ、方程式という飛び道具のありがたみを感じるものです。いきなり方程式を教えられても、小学生当時のあなたは理解できたでしょうか。

Emacsは人生です。

洞察力を深めるには多くのパッケージに触れて理解に努めましょう。ほしいパッケージが存在しなければ、自分でelispを書きましょう。

終わりに

いかがだったでしょうか？　今回は総論的な話題で退屈したかもしれません(笑)。

筆者は「日刊Emacs」以外にもEmacs病院兼メルマガのサービスを運営しています。Emacsに関すること関しないこと、わかる範囲でなんでもお答えします。「こんなパッケージ知らない？」「挙動がおかしいからなんとかしてよ！」はもちろんのこと、自作elispプログラムの添削もします。集中力を上げるなどのライフハック・マインド系も得意としています。SD登録はこちら➡
http://www.mag2.com/m/0001373131.html

ShowNetが示すネットワークの近未来

第5回 ShowNetの裏側 ～ホットステージレポート～

インターネット技術とビジネスが出会う国内最大のイベント「Interop Tokyo」。ほかでは類を見ないその最大の特徴である“ShowNet”は、会場全体に構築される最先端の技術を駆使したネットワークです。このネットワークの敷設はInteropの会場である幕張メッセにて約2週間前から行われます。今回はShowNetの裏側、イベント開始直前の様子をお届けします。

取材・文 編集部
URL http://www.interop.jp/

ShowNetはどのように作られる？

本連載でこれまで紹介してきたShowNetが、去る2015年6月10～12日まで千葉・幕張メッセで行われたInterop会場でお披露目となりました。連載第5回は、この準備期間中の様子を紹介します(**写真1、2**)。まずは、次のWebサイトに掲載されているShowNetのトポロジ図(ネットワーク構成図)を見てみてください[注1]。

ShowNet 2015　今年の見どころ
http://www.interop.jp/2015/shownet/highlight.html

本連載の第2回(2015年5月号)で書かれていたとおり、この1枚に今年のShowNetで構築されたネットワークのすべてが集約されています。このトポロジ図を見ながら、ShowNet NOC(Network Operation Center)チーム、コントリビュータおよびSTM(ShowNet Team Member)が一丸となってネットワークを構築していきます[注2]。

2015年の準備期間(通称：ホットステージ)は5月28日の資材・機器搬入と管理用ネットワーク構築から始まり、6月5日の14時でいったんの完成をみます。わずか9日間足らずで、最新技術を盛り込んだ大規模イベントネットワークが構築されるわけです。その後、会場全体への配線作業が入り、次いで各ブースの機材が搬入されてShowNetへのネットワーク接続が開始

注1) PDFとしてダウンロードもできます。

注2) 本文中に記載した「今年の見どころ」ページにある動画を見ると、トポロジ図の変遷が垣間見えます。

▼写真1　ShowNetブース構築中！

▼写真2　机上で作業を行うNOCチームメンバー。こことブースにあるラックを行ったり来たりする

されます。各出展ブースへのネットワークサービスもShowNetの構築メンバーが担っているため、NOCメンバーとSTMはシフトを組んで接続検証とともに最終調整を行います。なお、今年のShowNet構築・運用に携わった関係者は400名以上にのぼりました。

注目のラック(構築中)紹介

ShowNet 2015では、合計16個のラックに機器が搭載されました。その中からこれまでの連載で取り上げてきたものを中心にいくつか見ていきましょう。

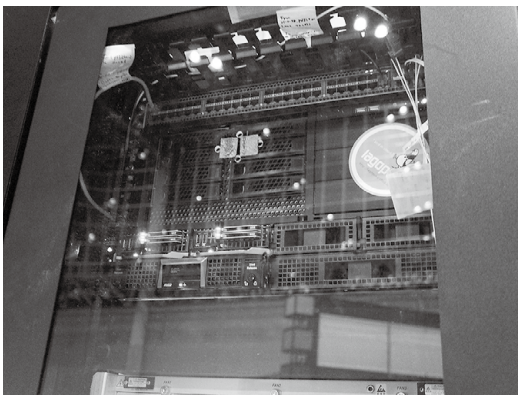
SDNを用いたIXとバックボーンネットワーク

今回のShowNetにおいて今後のインターネット技術に影響を与えそうなものの1つとして、インターネットサービスプロバイダ同士が相互接続に利用するインターネットエクスチェンジ(IX)を、SDNで構築している点が挙げられるでしょう。

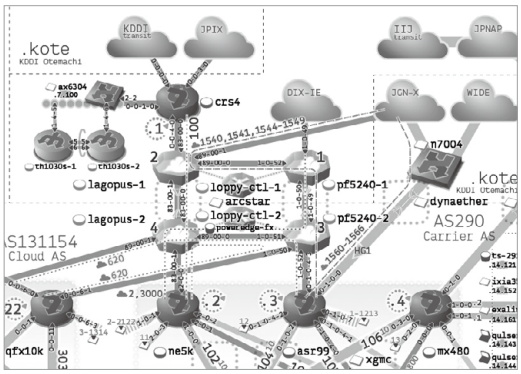
写真3のラックには、NECのPF5240(OpenFlowスイッチ)、NTTが開発、オープンソース化したOpenFlow対応ソフトウェアスイッチ「Lagopus」を搭載したサーバなどが収められ、NTT大手町に設置されたもう一対のOpenFlowスイッチと接続しています。トポロジ図では**図1**のあたりに該当します。ここでは4つのOpenFlowスイッチでDDoS攻撃を遮断する実証実験も実施しています(解析側のラックは後述)。IXにSDN技術を取り入れることで、DDoS攻撃による被害事業者の運用者自身がコントローラを介してフィルタ設定を行い、攻撃への迅速な対応が可能となります。

写真4の3つのラックでは、こちらもDDoS攻撃対策技術の1つとして注目されるBGP Flowspec[注3]や、経路情報の安全性を高めるために証明書による認証を行うRPKI(Resource Public Key Infrastructure)、NTP(Network Time Protocol)よりも高い時刻精度を持つPTP(Precision Time Protocol)といった3種類の相互接続実証が実施されていました。

▼写真3　SDNインターネットエクスチェンジが構築されたラック



▼図1　写真3のラックに該当するトポロジ図



▼写真4　ラック前でBGP Flowspec、RPKI、PTPの検証を行うNOCとコントリビュータ

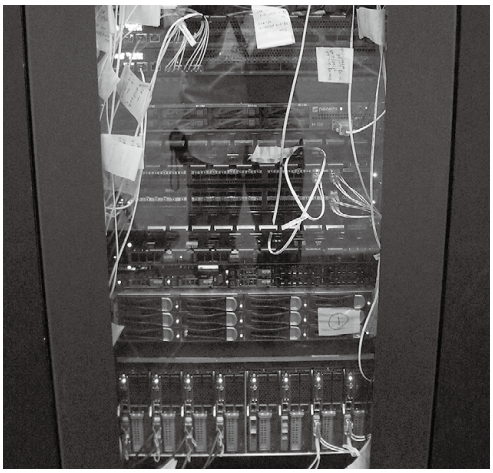
注3)　詳細は本誌2015年7月号の連載第4回を参照ください。

NFVのスケールアウト実証実験

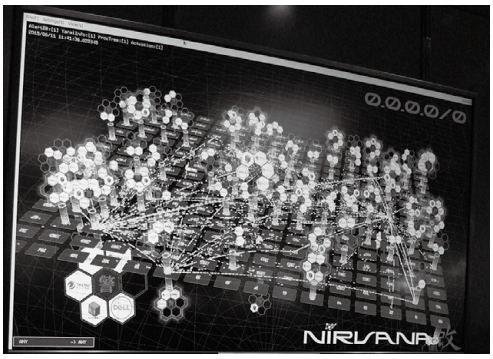
ネットワーク仮想化実験でもう1つ注目したいのは、NFV(Network Function Virtualization)によるリアルスケールアウトの実証実験です。スケールさせるには、個々のサービスを複数の仮想マシンや機器で構成します。DELLのPowerEdge R630に載せたJuniper Networksの仮想ファイアウォールvSRXで、ユーザごとにどのサービスを適用するかをパケットのTOS(Type of Service)フィールドにマークし、NECのOpenFlowスイッチPF5459/PF5248とNOCチームが自作したOpenFlowコントローラで、マークに応じた適切なサービスへパケットを転送します。適用する場合は、PowerEdge C6220に載せたCiscoの仮想ルータCSR 1000Vや、HuaweiのX6800サーバに載せたPalo Alto Networksの仮想ファイアウォールVM-300、さらにA10 NetworksのDDoS防御装置Thunder 6435 TPSといった各サービスへトラフィックを分散してスケールアウトさせるというしくみです(**写真5**)。

このラックに収まっている機器をトポロジ図で探すと、中央にある「.nfv」という囲みに該当します。

▼写真5　下からX6800、PowerEdge、PF5459と収められている



▼写真6　NIRVANA改の監視用画面
(※この写真のみ会期中のものです)

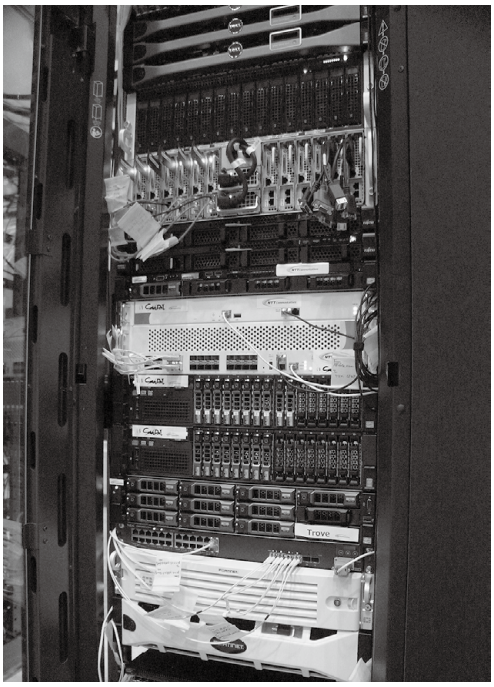


セキュリティ／監視

もう1つの大きなテーマであるセキュリティでは、NOCチームメンバーから昨年のステージで提唱された次世代型多層防御モデルを導入し、実際に運用で活用していました。

例年と異なる点は、ファイアウォールとサンドボックス(Palo Alto NetworksのPAシリーズおよびWF500、CiscoのASA/FirePower、FortinetのFortiGateおよびFortiSandbox)をインラインで設置している点です。これにより、

▼写真7　DDoS攻撃の分析機器が収められたラック



攻撃を階層的に解析でき、脅威となるトラフィックをより詳細に抽出することが可能となります。さらに今年は、SIEM(**写真6**にあるNICTのNIRVANA改)とフォレンジック(Savviusの Omnipliance と Arbor の Pravail)の本格導入と活用によって、インシデント発生時の早期通知とその他のアラートとの相関分析、被疑端末の危険度を早期に判断し対応できるようにしていました。

▼**写真8　ShowNetブースのラックを裏側から**

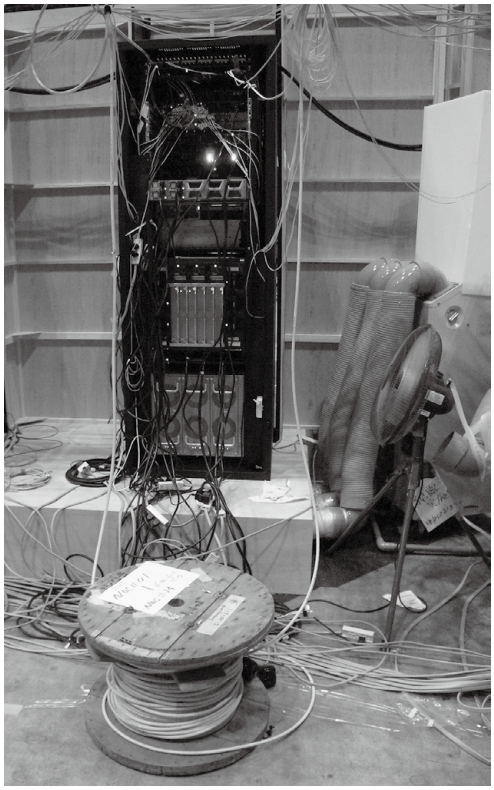
また、DDoS対策の最適分散配置の検知装置として、フロー情報を収集して攻撃を検知するNTTコミュニケーションズのSAMURAIが稼働していました(**写真7**)。たとえば、回線を埋め尽くすほどのDDoS攻撃を検知した場合には、このSAMURAIがNTTコミュニケーションズ社内のDDoS緩和装置と連動し、トランジットISP側で対処してくれるというもので、攻撃量と回線速度、攻撃経路などによって最適な場所(ISP内やSDN－IX内、ShowNet内)でDDoS攻撃を防御または緩和することができます。

ネットワークエンジニアの祭典

ラックの裏側も見せてもらいました(**写真8**)。排熱に関しては**写真9**のように冷気をラック下から送り込み、上に風が抜けるしくみですが、下からの熱気が上の機器に影響を与えないよう、ラックの途中途中で熱が逃げるような空気の流れを作っているとのことでした[注4]。

わずか3日間のイベントネットワークですが、ホットステージ期間中、NOCチーム、コントリビュータとSTMの方々は土日も昼夜も問わずネットワーク構築に尽くします。彼ら、彼女らの高い技術力と協調性、そして何より新しいことにチャレンジする熱意がShowNetを支えています。

▼**写真9　排熱処理は写真右にある送風装置からダクトを伝ってラックの下から送り、上へと逃がす構造**



次回は本連載の最終回として、ShowNet 2015での実証実験で得られた知見を披露していただきます。SD

注4) 空気の流れを計測する機器もコントリビュータからの提供です。

Red Hat Enterprise Linuxを極める・使いこなすヒント

.SPECs ドット・スペックス

第14回 xhyveでRHELを動かしてみよう

米国時間2015年6月10日にMac OS X用の仮想化ハイパーバイザであるxhyveがリリースされました。Mac OS Xで動作する仮想化ソフトウェアはすでにあるものの、xhyveは荒削りながら軽量ということやCUIだけで操作できる点が魅力です。

Writer レッドハット(株)サービス事業統括本部
プラットフォームソリューション統括部ソリューションアーキテクト部長 藤田 稜(ふじた りょう)

xhyveとは?

xhyveは浅田拓也氏による本誌連載「ハイパーバイザの作り方」で紹介されたFreeBSD用の仮想化ハイパーバイザ・bhyveをMac OS Xにポーティングしたものです。Mac OS X 10.10"Yosemite"には仮想化ハイパーバイザのフレームワークが実装されており、これを利用したDOSエミュレータであるhvdos[注1]がすでにありますが、より本格的な仮想化ハイパーバイザがhvdosと同じ作者によって実装・公開[注2]されました。

bhyveのポーティングということもあり、VGAに対応していないことを除くと基本的な機能は実装されているため、Mac OS X上でLinuxの動作を検証するといった用途であれば十分にこなせるレベルにあります。またすべてMac OS Xのユーザスペースで実装されているため、Mac OS Xのカーネルに悪影響を及ぼす懸念もありません。

xhyveの準備

xhyveを利用するにはコンパイルが必要となるため、Mac OS Xの開発環境であるXcodeをMac OS Xにインストールしておきます。Mac OS Xのターミナルでgitコマンドを実行するとXcodeのインストールが必要である旨のメッセージが表示されるので、メッセージに従って操作すればMac OS XのApp Storeが起動してインストールされます。Xcodeのインストール完了後、ターミナルで**図1**のコマンドを実行します。

とくに問題がなければ、xhyve/build/の下にxhyveコマンドがコンパイルされます(**図2**)。

執筆時点で取得したソースコードから生成されるコマンドは222KBしかなく、非常に軽量であることがわかります。また、xhyve/test/以下にはTiny Core Linuxが用意されているので、xhyve/xhyverun.shスクリプトを

▼図1 xhyveの準備

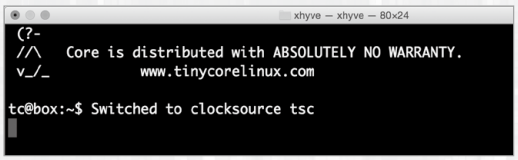
```
$ git clone https://github.com/mist64/xhyve
Cloning into 'xhyve'...
remote: Counting objects: 313, done.
remote: Compressing objects: 100% (160/160), done.
remote: Total 313 (delta 156), reused 305 (delta 148), pack-reused 0
Receiving objects: 100% (313/313), 11.16 MiB | 3.88 MiB/s, done.
Resolving deltas: 100% (156/156), done.
Checking connectivity... done.
$ cd xhyve/
$ make
```

▼図2 xhyveのファイルサイズ

```
$ ls -lh build/xhyve
-rwxr-xr-x  1 rio  staff   222K  6 12 12:01 build/xhyve
```

注1) https://github.com/mist64/hvdos 注2) https://github.com/mist64/xhyve

▼図3　Tiny Core Linux on xhyve



▼図4　アクティビティモニタ（プロセスを選び、左上の×印のボタンで停止する）

実行すれば、すぐにLinuxが動作することが確認できます（**図3**）。

```
$ sudo ./xhyverun.sh
```

なお、Tiny Core Linuxを終了するには、Tiny Core Linuxのターミナルでsudo haltコマンドを実行してください。

インストールの前に

Tiny Core Linuxが動いたところでニヤニヤしていても「仕事」にならないので、xhyveでRHEL 7をインストールしてみます。実はxhyveを紹介したブログエントリー[注3]にはUbuntuのインストール方法が紹介されているのですが、RHEL/Fedora/CentOSをインストールするにはもう少し手順が必要ですので、筆者が確認した方法を以下で紹介します。

なお何らかの理由でxhyveを強制的に停止する必要があるときは、Mac OS Xのアクティビティモニタで「すべてのプロセス」を表示し、xhyveのプロセスを停止してください（**図4**）。

▼リスト1　xhyverun.sh

```
#!/bin/sh

KERNEL="test/vmlinuz"
INITRD="test/initrd.gz"
CMDLINE="earlyprintk=serial console=ttyS0 acpi=off"

MEM="-m 1G"
#SMP="-c 2"
#NET="-s 2:0,virtio-net"
#IMG_CD="-s 3,ahci-cd,/somepath/somefile.iso"
#IMG_HDD="-s 4,virtio-blk,/somepath/somefile.img"
PCI_DEV="-s 0:0,hostbridge -s 31,lpc"
LPC_DEV="-l com1,stdio"

build/xhyve $MEM $SMP $PCI_DEV $LPC_DEV $NET $IMG_CD
$IMG_HDD -f kexec,$KERNEL,$INITRD,"$CMDLINE"
```

▼リスト2　rhel71install.sh

```
#!/bin/sh

KERNEL="rhel71/vmlinuz"
INITRD="rhel71/initrd.img"
CMDLINE="console=ttyS0 acpi=off inst.vnc inst.
vncpassword=hogehoge inst.sshd"

MEM="-m 1G"
#SMP="-c 2"
NET="-s 2:0,virtio-net"
IMG_CD="-s 3,ahci-cd,/Users/rio/Downloads/rhel-
server-7.1-x86_64-dvd.iso"
IMG_HDD="-s 4,virtio-blk,rhel71/hdd.img"
PCI_DEV="-s 0:0,hostbridge -s 31,lpc"
LPC_DEV="-l com1,stdio"

build/xhyve $MEM $SMP $PCI_DEV $LPC_DEV $NET $IMG_CD
$IMG_HDD -f kexec,$KERNEL,$INITRD,"$CMDLINE"
```

RHEL 7.1インストール用スクリプト

前述のとおり、Tiny Core Linuxを起動する際に付属のxhyverun.shを実行しました。中を見ると非常に単純でxhyveコマンドに渡す引数を列挙しているだけです（**リスト1**）。

これを“rhel71install.sh”のようにリネームして、RHEL 7.1のインストール用に変更したものが**リスト2**です。

VNCを経由したGUIインストールが必要なければ、inst.vncおよびinst.vncpasswordオプションは必要ありませんが、inst.sshdオプションはインストール完了時に作成されたinitramfsなどをMac OS X側に転送するために必要となります。

注3） http://www.pagetable.com/?p=831hyve

▼図5　インストーラDVDはMac OS Xではマウントできない

▼図6　RHEL 7.1のインストールの事前作業

```
$ mkdir -p ~/xhyve/rhel71/boot/
$ dd if=/dev/zero of=/tmp/tmp.iso bs=2k count=1
$ dd if=~/Downloads/rhel-server-7.1-x86_64-dvd.iso bs=2k skip=1 >> /tmp/tmp.iso
1899519+0 records in
1899519+0 records out
3890214912 bytes transferred in 12.717327 secs (305898785 bytes/sec)
$ hdiutil attach /tmp/tmp.iso
$ cp /Volumes/RHEL-7.1¥ Server./images/pxeboot/initrd.img ~/xhyve/rhel71/
$ cp /Volumes/RHEL-7.1¥ Server./images/pxeboot/vmlinuz ~/xhyve/rhel71/
$ hdiutil detach /Volumes/RHEL-7.1¥ Server./
$ rm /tmp/tmp.iso
```

▼図7　hdd.img(16GB)のイメージ作成

```
$ dd if=/dev/zero of=~/xhyve/rhel71/hdd.img bs=1g count=16
```

RHEL 7.1のインストール

スクリプトを実行する前にインストーラDVDからvmlinuzおよびinitrd.imgを抜き出す作業が必要です。しかしながらインストーラDVDをダブルクリックしても**図5**のようにマウントできません[注4]。そこでMac OS Xのターミナルでちょっとした作業をします(**図6**)。

次にインストール先となるhdd.imgを作成します。**図7**では16GBのイメージを作成していますが、もしLVMを用いない場合はより大きなイメージ、たとえば32GBが必要な点に注意してください。

以上でインストールの準備ができたのでインストール用スクリプトを実行します。インストーラが無事起動すると**図8**のように表示されます。

```
$ sudo ./rhel71install.sh
```

ここで注意点が2つあります。anacondaの仮想ターミナルの切り替えが可能ならばシェルを実行できるため、initramfsなどの転送ができます。しかし、Mac OS Xのターミナルではできません[注5]。このためインストール用のスクリプトでsshdが起動するように設定しました。また執筆時点で理由は判明していませんが、ネットワークを有効化するにはanacondaのメニューの"5) Network settings"での手順を繰り返さないといけないことがあります。

▼図8 インストーラ(anaconda)が起動した状態

▼図9　ネットワーク接続が成功した場合

```
Wired (eth0) connected
 IPv4 Address: 192.168.64.14 Netmask: 255.255.255.0
Gateway: 192.168.64.1
 DNS: 192.168.64.1
```

1. anacondaのメニューで"5"を入力してEnter
2. Network settingsのメニューで"2"(Configure device eth0)を入力してEnter
3. Device configurationのメニューで"8"(Apply configuration in installer)を入力してEnter

ネットワーク接続が完了すれば、**図9**のようなメッセージがコンソールに出力されます。

Mac OS Xのターミナルで別ウインドウあるいはタブを開いてsshで接続できることを確認しておきます。rootのパスワードは不要です。

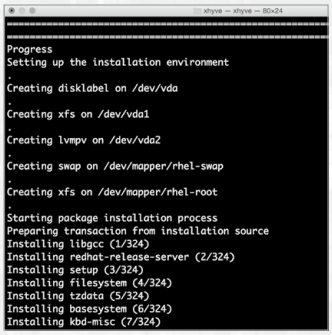
```
$ ssh root@192.168.64.14
[anaconda root@localhost ~]#
```

すべての項目の設定が完了したらインストールを開始しましょう。ただしインストールが完了してもファイルを転送する必要があるため再起動

注4) Fedoraの場合はハイブリッドDVD-ROMイメージなのでマウントできるが、これはMacにFedoraをインストールする際に必要なパーティションがマウントされるだけで、xhyveで必要となるファイルを抜き出せない。

注5) Modifierキーの転送ができないため。

▼図10　インストール開始



はしないでください(**図10**)。

インストールが完了したらMac OS Xのターミナルで RHELからMac OS Xにファイルを転送します(**図11**)。

また、xhyveではGRUB 2の1stステージを利用しないためxhyveにカーネルパラメータを指示する必要があります。Mac OS Xのターミナルでgrub.cfgを転送しておくと良いでしょう(**図12**)。

▼図11　ファイル転送の様子

```
$ scp root@192.168.64.14:/mnt/sysimage/boot/initram* ./xhyve/rhel71/boot/
initramfs-0-rescue-faabd67daf984767aea59626a4 100%   38MB  38.0MB/s   00:01
initramfs-3.10.0-229.el7.x86_64.img           100%   17MB  16.9MB/s   00:00
$ scp root@192.168.64.15:/mnt/sysimage/boot/vmlinuz* ./xhyve/rhel71/boot/
vmlinuz-0-rescue-faabd67daf984767aea59626a491 100% 4909KB   4.8MB/s   00:01
vmlinuz-3.10.0-229.el7.x86_64                 100% 4909KB   4.8MB/s   00:00
```

▼図12　grub.cfgの転送

```
$ scp root@192.168.64.14:/mnt/sysimage/boot/grub2/grub.cfg ./xhyve/
```

▼リスト3　rhel71run.sh

```
#!/bin/sh

KERNEL="rhel71/boot/vmlinuz-3.10.0-229.el7.x86_64"
INITRD="rhel71/boot/initramfs-3.10.0-229.el7.x86_64.img"
CMDLINE="console=ttyS0 BOOT_IMAGE=/vmlinuz-3.10.0-229.el7.x86_64 root=/dev/
mapper/rhel-root ro rd.lvm.lv=rhel/swap console=ttyS0 crashkernel=auto
rd.lvm.lv=rhel/root acpi=off LANG=en_US.UTF-8"

MEM="-m 1G"
#SMP="-c 2"
NET="-s 2:0,virtio-net"
#IMG_CD="-s 3,ahci-cd,/Users/rio/Downloads/rhel-server-7.1-x86_64-dvd.iso"
IMG_HDD="-s 4,virtio-blk,rhel71/hdd.img"
PCI_DEV="-s 0:0,hostbridge -s 31,lpc"
LPC_DEV="-l com1,stdio"

build/xhyve $MEM $SMP $PCI_DEV $LPC_DEV $NET $IMG_CD $IMG_HDD -f kexec,$KE
RNEL,$INITRD,"$CMDLINE"
```

RHEL 7.1の起動スクリプト

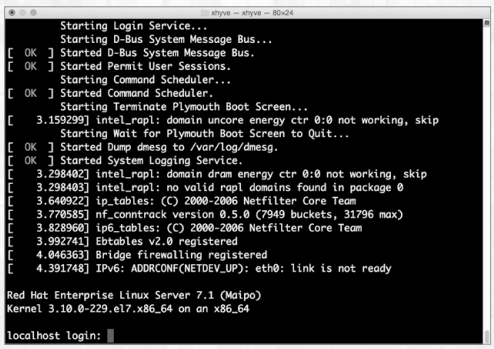
先に転送したgrub.cfgを参照しながら作成した"rhel71run.sh"スクリプトが**リスト3**です。

スクリプトを実行するとRHEL 7.1が起動します。

```
$ sudo ./rhel71run.sh
```

なお、RHEL 7.1とほぼ同様の手順でFedora 22/CentOS 7.1でもインストールができます。Fedora 22の場合、起動用のvmlinuz/initrd.imgはFedoraのミラーサイト[注6]からダウンロードするのが最も簡単ですが、インストーラDVDイメージに含まれるものと同じである必要があるため、うまくいかない場合はいずれかのLinux上でインストーラDVDイメージをループバックマウントして当該ファイルを抜き出してください。

▼図13　RHEL 7.1 on xhyve



まとめ

昨今はMacBook Air/MacBook Proを持ってセミナーや勉強会に参加するエンジニアが増えています。xhyveならば、手軽にLinuxをゲストOSとして動作させられるので、メリットがいろいろありそうです。SD

注6）たとえば、http://ftp.iij.ad.jp/pub/linux/fedora//releases/22/Server/x86_64/os/images/pxeboot/

Be familiar with FreeBSD.

チャーリー・ルートからの手紙

第22回 ❖BSDCan 2015で知る今後の動向

◉著者プロフィール

後藤 大地（ごとう だいち）

BSDコンサルティング（株） 取締役／（有）オングス 代表取締役／FreeBSD committer

エンタープライズシステムの設計、開発、運用、保守からIT系ニュースの執筆、IT系雑誌や書籍における執筆まで幅広く手がける。カーネルのカスタマイズから業務に特化したディストリビューションの構築、自動デプロイ、ネットワークの構築など、FreeBSD/Linuxを含め対応。BSDコンサルティングでは企業レベルで求められるFreeBSDの要求に柔軟に対応。

これまでで最大規模、BSDCan 2015

*BSD[注1]コミュニティは年に1回、または2年に1回といったペースで世界各地でカンファレンスを開催しています。開催されるカンファレンスの数は増え続けており、この数年だけでもBSDCan、vBSDCon、EuroBSDCon、BSDCon Brazil、AsiaBSDCon、MeetBSD California、BSDCam、NYBSDCon、ruBSD、KyivBSD、BSDDayといったカンファレンスが開催されています。

会議の規模、参加者の規模、発表内容の質、開催の頻度、募集と選定の確からしさなどから、AsiaBSDCon、BSDCan、EuroBSDConの3つのカンファレンスが代表的と言えます。とくにカナダのオタワで開催されるBSDCanは年々順調に参加者を増やし、もっとも規模の大きな*BSDのカンファレンスになっています。スポンサードも増やし、カンファレンスの質も年々向上しています。

今年からBSDCanの開催月が5月から6月に変更されました。今年の参加者はこれまでで最大となり、発表トラックも4トラックになりました。BSDCanの前にはFreeBSD DevSummitも開催され、FreeBSDの今後の動向が見える会議になりました。BSDCan 2015の発表から、とくに興味深かった講演や各プロジェクトによる発表をピックアップして紹介します（**写真1**）。

基調講演はあのBourne Shellの開発者：Stephen Bourne氏

BSDCan 2015の基調講演にはStephen Bourne氏が登壇しました。Bourne氏といえば、*BSDやLinuxで広く使われているBourne Shell（/bin/sh）の開発者として有名です（ディストリビューションに含まれているのはBourne Shellそのものではなく互換シェルです）。本誌の全読者にもっとも強い影響を与え続けている人物の一人といえます。

Bourne氏からは1975年の終わりごろから、それまで使っていたシェルの代わりとしてBourne Shellを開発するに至った経緯や、最初の設計と実装がどういった指針に基づいて行われたものであるかなどが語られました。最初のバージョンは1976年にリリースされていますので、Bourne Shellの登場からもう40年が経過しようとしていることになります。

Bourne Shellの特徴的なシンタックスである、**if**

▼**写真1 BSDCan 2015**

注1 BSDと名の付くものを総称して、このように「*」を付けて表記させてもらいます。

各コマンドのうしろについている括弧書きの数字は、manコマンドで見ることができるマニュアルに記載されている章番号を表しています。

... fi、**case ... esac**といった表記やプログラムのフロー、置換の機能などはALGOL 68のコンセプトをそのまま使用したと説明がありました。Bourne氏はALGOL 68のコンパイラの開発者でもあるからです。1977年には現在のBourne Shellの機能の大半が実装されていたようです。1975年当時はUnixの6th editionを使っていて、必要がなかったので実装にはCのライブラリは使っていなかったと説明しています。

Bourne氏はBourne Shellの開発を通じて実施したことや感じたことから、次のような項目を紹介していました。

- 他人が読みやすいコードを書くこと
- とにかくはじめてから細かく繰り返すこと
- 実際に使っているリアルユーザの声に耳を傾けること
- 機能をたくさん追加することで逆に扱いにくくなる現象に陥らないようにすること

なお、Bourne氏が開発したshのソースコードは、http://minnie.tuhs.org/cgi-bin/utree.pl?file=V7/usr/src/cmd/shで閲覧できます。C言語をALGOL 68風に記述するためのマクロが使われ、一見するとCのソースコードとは思えないような作りになっています。

JailとQEMU UEでARMおよびMIPS向けのパッケージビルド：Sean Bruno氏／Stacey Son氏

FreeBSDプロジェクトは、amd64とi386をTier 1[注2]プラットフォームとしてサポートしています。そしてarm、ia64、pc98、powerpc、sparc64はTier 2、mipsはTier 3と位置づけています。現在プロジェクトではarmとmipsに関しても需要が高いことから、Tier 1と同じようにパッケージの提供へ向けた作業を行っていますが、どういった方法で取り組まれているのかの発表がありました。

armやmipsはamd64ほどは汎用的でパワフルなマシンが存在しませんので、通常であれば次の2つのアプローチを取ることになりますが、どちらもビルドが完了するまで長い時間がかかるという問題があります。

- 実機でビルドする
- QEMU上でビルドする

Sean Bruno氏とStacey Son氏はこれに対し、Jailでビルド環境を作成するとともに、CPUの処理だけエミュレートするQEMUのユーザモードエミュレーションを使用する方法を紹介。こうすることでamd64の高速な環境でのクロスビルドが可能で、現実的なパッケージクラスタとして使用できることを紹介しました。

プロセッサの機能を使ってCapsicumを強化するCheriBSD：Brooks Davis氏

FreeBSDは10.0からCapsicumと呼ばれるケーパビリティの実装系を導入しています。これはプロセスが自発的に自身のアクセスできるリソースやシステムコールを制限することで、より安全に動作できるようにしようという試みで、既存のPOSIX APIと高い相性を実現しているという特徴があります。

Capsicumの研究者らはその開発を通じて、Capsicumが実現しようとしているリソースの隔離を行うには、現在のプロセッサでは実装がトリッキーにならざるを得ないと指摘しています。これを改善するために、より簡潔にリソースの区画化を実現できるように命令セットアーキテクチャのデザインそのものを模索する取り組みとして、CHERI（Capability Hardware Enhanced RISC Instructions）の開発に乗り出します。

CheriBSDは、実験機として開発されたCHERIを実装したハードウェアに対応した、FreeBSD派生のオペレーティングシステムです。CHERIを利用できるようにカーネルが変更されているほか、ユーザランドのソフトウェアからCHERIの機能が利用できるように実装が追加されています。

CHERIそのものも興味深いのですが、Brooks

注2 FreeBSDプロジェクトは複数のアーキテクチャをサポートしていますが、そのサポートのレベルをTier 1～4で表現しています。Tier 1はもっともサポートが手厚い対象で、リリースごとにインストーラもパッケージもすべてが提供されています。数字が大きくなるほどサポート内容が減っていきます。

チャーリー・ルートからの手紙

Davis氏の発表ではリポジトリの管理にGitHubを使ったこと、この結果としてマージ作業がとても簡単になったこと、などが紹介されていた点が特徴的でした。FreeBSDプロジェクトはSubversionをバージョン管理システムとして採用し、開発者向けにはPerforceを提供しています。Gitへのエクスポートも実施しているのでGit経由でもソースコードを取得できますが、あくまでも中心はSubversionです。

CheriBSDでは、GitHubのFreeBSDからソースコードを派生させてプロジェクトを開始しています。HEADに導入される変更を随時マージしながら、CheriBSDの変更部分に関しても保持し続けるという取り組みは興味深いものがありました。プロジェクトの最新の成果物を取り込み続けながら、カスタムコードも保持し続ける方法として参考になりました。ソースコードはhttps://github.com/CTSRD-CHERI/cheribsd/tree/masterで閲覧できます。

カーネルメモリ保護機能 KCoFIとVirtual Ghost：John Criswell氏

ユーザランドで動作するソフトウェアはアクセスできるメモリ領域や利用できる機能が限られており、システムを破壊するような操作はできないようになっています。一方、カーネルはどのリソースにも自由にアクセスできるため、カーネルにバグがあったり、バッファオーバーフローなどを仕込まれると、セキュリティはあってないような状態になります。

カーネルにおけるメモリアクセスを制限したり、問題のある動作をモニタリングして事前に処理を排除する方法などいくつかの方法がありますが、John Criswell氏からは「KCoFI」および「Virtual Ghost」という2つの研究成果が発表されました。

アイディアの根幹にあるものは、カーネルとユーザランドの間にすごく薄くて軽量なハイパーバイザのようなものを挟み込んで、このレイヤでカーネルによるほかの領域へのメモリアクセスを制御しようといった内容になっています。アイディアはまったく異なるのですが、カーネルの中にさらにカーネルを動作させてシステムコールの不正な実行を排除するという研究成果が、FreeBSD DevSummitでも発表されていました。このあたりは今ホットな研究領域のようです。

ケンブリッジスタイルOSコース L41：George Neville-Neil氏／Robert Watson氏

大学におけるオペレーティングシステムの授業はさまざまですが、BSDCan 2015ではRobert Watson氏によってケンブリッジ大学で開講されている「L41 Advanced Operating Systems」が紹介されました。オペレーティングシステムの授業ではシンプルな模造オペレーティングシステムを使ったり、実際のオペレーティングシステムを使っているものの難しすぎるといったこともあります。

「L41 Advanced Operating Systems」ではほかのコースとは異なり、教材として本物のFreeBSDを採用。教科書としては「The Design and Implementation of the FreeBSD Operating System（2nd Edition）」や「DTrace: Dynamic Tracing in Oracle Solaris, Mac OS X and FreeBSD（Oracle Solaris Series）」などを利用しつつ、システムをモニタリングする方法や分析する方法を通じて動きを探り、それに合わせて内部構造やしくみを解説する方法を取っていることを紹介していました。この取り組みの一部は、先に東京で開催されたAsiaBSDCon 2015において、その一部分がDTraceのチュートリアルという形で実施されています。

とくに発表では、DTraceを使ったシステムの動作モニタリングが授業において重要なポジションを占めていることが示されていました。実際にカーネルの内部でどういったことが起こっているのかを実際に手でコマンドを打って確認し、それを受けて設計や実装の講義を受けるスタイルです。カーネルの動きが実感として理解でき、論理的に系統立って頭に中に整理されるのではないかと感じました。

次世代設定ファイルフォーマット「UCL」：Allan Jude氏

FreeBSDユーザや管理者がもっとも影響を受け

ることになるであろう取り組みが、このUCL（Universal Configuration Files）でしょう。FreeBSDプロジェクトは今よりモダンで、さまざまなデバイスで利用できる「プラットフォーム」としてFreeBSDを開発すべく会議や開発を繰り返しています。その1つがUCLです。

現在iXsystemsでCTOを務めるJordan Hubbard氏は、以前FreeBSDの開発者会議で、FreeBSDはプラットフォームとして変わるべき時を迎えているとし、たとえば設定ファイルを統一されたXMLにするなどの取り組みを進めるべきだと提案したことがありました。この発言を受けてAllan Jude氏が開発をはじめたフォーマットであり、言語がUCLです。

UCLの直接の発想はNginxの設定ファイルから来ているそうです。JSONのようなフォーマットですが、JSONのように厳密なルールにはなっておらず、より人間でも書きやすいようなフォーマットになっています。JSONはよく利用されるフォーマットですが、人間が直接記述するには規制が強いフォーマットで、適切なJSONを手動で書き続けるのは難しいことです。

Allan Jude氏はいくつかのサービスでUCLを使い始めているほか、ほかの開発者もツールの設定ファイルにUCLの使用をはじめています。UCLは設定のオーバーレイやインクルード、ほかのフォーマットへの変更が可能で、コマンドからも操作ができます。Jude氏は将来的に、現在/etc/の下に展開されているデフォルトの設定ファイルをUCLフォーマットに変更するとともに、/etc/defaults/以下へ移動させ、システムのデフォルトとユーザがあとから追加した設定などを分離することで、管理とアップデートを容易化するアイディアを紹介していました。

実際にどのファイルを対象とするのか、ディレクトリ配置をどうするのか、さらに多くのコミッタ間で議論を進める必要がありますが、開発者の間にとくに強く反対する雰囲気はなかったように思います。少なくとも、大幅な変更が行われるのはFreeBSD 12以降になるのではないかと思いますが、より少ない手間でアップグレードの容易化や管理ツールからの管理の容易化を進める取り組みに関して、議論が進んでいることは知っておくと良いかもしれません。

ベースシステムのpkg(8)化：Baptiste Daroussin氏

FreeBSDプロジェクトは、ベースシステムのアップデートにfreebsd-update(8)、サードパーティ製ソフトウェアのインストールやアップデートにpkg(8)を採用しています。これに関して、より細かいアップデートやインストールなどを実現できるようにするために、ベースシステムも個別のパッケージpkg(8)として実装しようという取り組みが進められています。PC-BSDではすでにそのしくみでベースシステムの提供を開始しています。

この取り組みは10系にバックポートされることはないと思いますが、早ければ11で、強い反対の理由がなければ12には基幹機能として取り込まれるのではないかと見られます。

BSDCanオークション

BSDCanはクロージングセッションでオークションを実施するという習慣があります。オークションの売上げはFreeBSD Foundationへの寄付になるわけですが、このオークションはなかなか面白いものです。BSDCanのDan Langille氏が丁々発止にオークションを進めていくのですが、書籍やTシャツから実にくだらないものまで、調子に乗せてどんどん値上げしていきます。これは凄いものだと値を上げるだけ上げて落札させたあとに、実はもう3枚あると言い出したり、会場は笑いの渦に包まれます。

BSDCanは開催場所も毎年同じですし、宿泊には費用の安い大学のレジデンスが使えます。海外の*BSDカンファレンスとしては参加しやすいので、興味がある方はBSDCan 2016の参加を検討してみてください。SD

29

Debian Developer　やまねひでき　henrich@debian.org

リポジトリの役割を理解して Debianを快適に使おう

Debian Hot Topics

開発の流れとリポジトリの関係

今回は、Debianのstable/testing/unstableバージョンの話と、用途ごとに存在するリポジトリ[注1]の立ち位置を説明します。「自分の用途とスキルでは、どのリポジトリを指定すれば、Debianを快適に利用できるのか」がおのずと見えてきます。

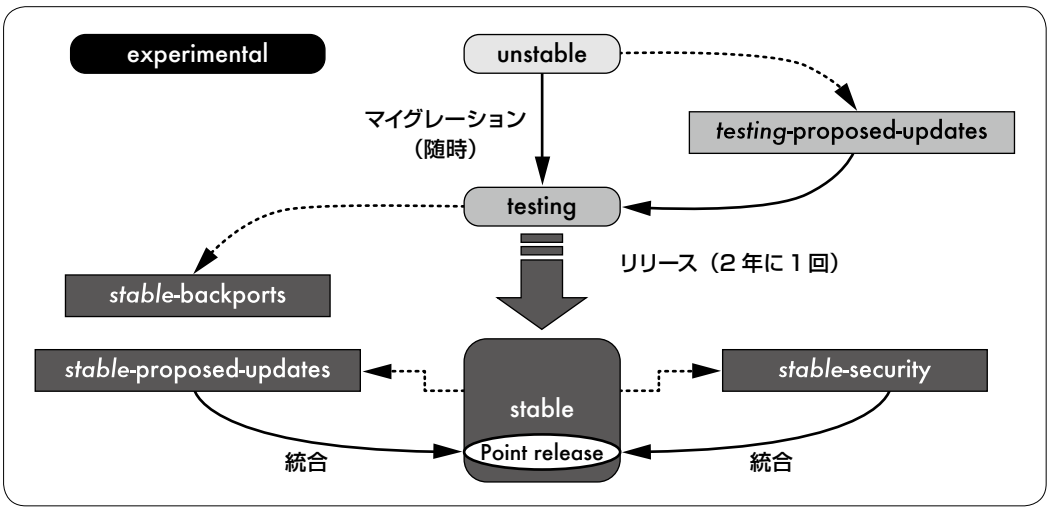
Debianではstable/testing/unstableにリポジトリが分かれていて、「unstable→testing→stable」という流れで開発が進みます。これに加え、その他さまざまなリポジトリが存在するのですが、何がどう違っているのかを理解されている方は、おそらく少数でしょう。今回は**図1**を見ながら、各リポジトリの関係について順を追って理解を深めていきましょう。

unstable

unstable（コードネーム：sid）は、Debianパッケージが最初に投入されるリポジトリです。最新のパッケージはここへアップロードされます。unstableのリポジトリは1日に4回更新され、逐次新しいパッケージが利用できるようになっていますが、数日経つとデスクトップ環境では「アップデートが数百MBほどある……」という

注1）リポジトリはパッケージの集積場所です。目的に合わせて分かれており、リポジトリ内ではパッケージ間の整合性がとられていますが、ほかのリポジトリを不用意に混ぜて利用すると依存関係が破綻する可能性があります。

▼図1　Debianの各リポジトリとその関係



ことは珍しくありません。

また、チェック不足によるバグなども多くあります(そのため「unstable」という名前が付いて、リリース版であるstableとは分けられているのですが[注2])。ここで勇敢な利用者による多くの確認を経て、重大な問題が見つからなければ、通常5日後[注3]にパッケージがtestingへ「マイグレーション」されます。

testing

testing(現在のコードネーム[注4]：stretch)はstableリリースをベースに、unstableからマイグレーションが行われたパッケージで構成されたリポジトリです。「これまで使い続けてきたタレに適宜新しいタレを混ぜている」という感じでしょうか。一部のパッケージは残念ながらunstableで「RCバグ」[注5]が見つかり修正されていないためにマイグレーションが実施されず、stableの古いバージョンのまま……ということもそれなりにあります。あとからRCバグが見つかったものは削除されるので、ユーザとしてはとくに不具合なく使っていたソフトウェアがいつの間にかtestingリポジトリから消えていてインストールができなくなっている……ということもしばしば見られます。

ここで2年に1回実施される半年程度の「フリーズ」期間の間に大きなバグを修正、あるいはパッケージ自体を削除などして調整がなされ、ある時点でstableとして「リリース」されます。

stable

一度stableとしてリリースされるといっさいの変更が加えられないので、変更に振り回されず安定して利用できます。ただし、これには例外が2つあります。セキュリティ更新とポイントリリースです。

Debianにおける「セキュリティ更新」は、基本的にセキュリティ修正だけを含む最低限の変更を加えたものです[注6]。セキュリティ問題が発見される[注7]と、

① メンテナがパッケージに適用するパッチを用意
② セキュリティチームによるパッチの精査
③ セキュリティチームによるパッケージのビルドおよびアップロードが行われる。これにより「*stable*-security[注8]」リポジトリが更新され、DSA(Debianセキュリティ勧告)が固有の番号付きで発行される。そして、GPGで署名されたメールがdebian-security-announceメーリングリストに送信され、Webサイトに情報が掲載される

という流れでユーザの手元までやってきます。実は「*stable*-security」リポジトリ自体はstableリポジトリとは別になっているのですが、インストール時点でapt lineに記述されているので、気づいてない方がほとんどでしょう(ちなみに*stable*-securityリポジトリは特殊なところがあり、通常のミラーサーバとは扱いが別になっています。security.debian.org以外ではミラーされていません)。

注2) unstable(不安定)という怖そうな名前が付いていますが、近年は「not stable」というぐらいの安定度になっていますので、そこまで怖がることはありません。地雷になるパッケージはだいたい特定のものに限られますし、Twitterなどを見ていれば地雷を踏んだ誰かが悲鳴をあげているので、それを回避することもできます。

注3) パッケージのchangelog(/usr/share/doc/パッケージ名/changelog.Debian.gz)を見るとわかるのですが「urgency」という項目があり、これがlowならば10日、mediumなら5日、highなら2日でマイグレーションが実施されます。以前はlowが開発ツールによる標準値でしたが、より開発を加速させるためにmediumに変更されています。

注4) unstableのコードネームはずっとsidですが、testing、stableのコードネームはリリースに合わせて変わります。

注5) Release Criticalバグ。機能面だけではなく「ソースからビルドできない」「ライセンス違反がある」などでもRCバグとされます。

注6) upstreamがセキュリティ修正以外の機能追加も含めたリリースしかしない場合(例：Firefox)はこの限りではありません。

注7) 多くの場合はCVE(Common Vulnerabilities and Exposures)が発行されて、各ディストリビューションベンダでハンドリングが容易になるように調整されます。CVEが発行される前の未公開の脆弱性の場合は、セキュリティチームが各メンテナに対してGPG鍵で暗号化したメールを送るなどして、第三者に漏れないように慎重に作業が進められます。

注8) *stable*の部分は、その時々のコードネームに置き換えてください。現在だとjessie-securityという名前です。以降、*stable*の表記が出てきた場合は同様に読み替えてください。

もう1つの「ポイントリリース」についてですが、「*stable*-proposed-updates」というリポジトリがあります。これはいわば「β版」のリポジトリです。stableに対して何かしらの修正を加えたアップデートを適用したい場合、stableリリースチームによる差分のチェックが行われたうえでこちらにアップロードを行い、ユーザによるテスト期間を経てポイントリリースの時点でstableに統合されます[注9]。

最近ですと、Debian 8に対する8.1のリリースがポイントリリースにあたります。proposed-updatesのリポジトリは公開されているので、大規模な運用を行っている場合はリリース前に限定した環境で適用してテスト運用を行い、問題があればフィードバックを行うと利用者は幸せになれるでしょう。

stableに対してのリポジトリには、もう1つ「*stable*-backports」があります。新しく更新したパッケージがtestingまで降りてきても、stableのリリースは2年ごとなのでタイミングが合わないとstableでは利用できません。とはいえ、2年というのは短くない期間ですので、「stableを使っているけど、testing/unstableにあるパッケージの機能を使いたい！」というジレンマが発生します。

これを解消する方法としては、testing/unstableにあるパッケージを「借りてくる」やり方(apt pinning)があります。しかし、この方法は依存関係を破壊してせっかく安定して運用できている環境を壊す危険性も秘めているため、頻繁に利用したくはありません。ここで、「testingにあるバージョンのソースをもとにstableに合わせてビルドしなおしたパッケージのリポジトリ」＝*stable*-backportsの登場です。既存の環境に悪影響を与えることなく新しいバージョンを利用できます。

非常に有用ではありますが、望むパッケージがbackportsに用意されているかはメンテナの気力しだいであるのが難点です(testingからbackportの自動化ができれば良いのですが……)。また、セキュリティアップデートがケアされているかというと、これも気力しだいですので追いつかないこともあり、サーバで運用する場合は注意が必要になります。

experimental

これまで説明したもの以外では「experimental」があります。experimentalは独立したリポジトリで、「大規模な変更のためそのままではunstableに入れてtestingにマイグレーションするのは怖い、もしくはリリース前でunstableへの投入がためらわれる[注10]」けれど、「継続的にリリース自体は行っておきたい(手元に死蔵させておきたくはない)」などの場合に利用されます。

experimentalにあるパッケージを利用するにはapt lineに追加するだけではダメで、インストール時に「experimentalのパッケージをインストールする」ことを明示的に指定する必要があります(**図2**)。

注9) 加えられるのはあくまでも修正であり、機能追加ではありません。

注10) 今後ですが、フリーズ期間中の新しいバージョンのアップデートについてはtesting-proposed-updatesリポジトリへアップロードすることにより、unstableへの投入が禁止されなくなるかもしれません。

▼図2　experimentalのパッケージをインストールする手順

```
$ sudo sh -c "echo deb http://ftp.jp.debian.org/debian exerimental main >> /etc/apt/
/sources.list"
  ↑experimentalリポジトリをapt lineに追加し、
$ sudo apt-get update
  ↑aptのデータベースをアップデートして変更を反映
$ sudo apt-get install -t experimental hello
  ↑-tオプションでexperimentalを使うことを明示し、ここではhelloパッケージのインストールを指示
```

また、ここに入れられたパッケージはリリースされることはないため、同じ内容のパッケージであっても必要に応じてバージョン番号を上げてunstableにアップロードしなおされます。

で、私はどれを利用すればいいの?

基本はstable、現在であればDebian 8「jessie」を利用するのが一番です。「まだDebianに慣れていない」「安定して利用したい」「時間と手間はかけたくない」方はstableを利用し、新しいパッケージが必要な場合は、適宜backportsを取り入れるのが良いでしょう。大規模サーバ用途で利用する場合は、proposed-updatesを有効にしたテスト環境も用意しておくことを忘れずに。

とにかく最新のバージョンが好きな人や、ソフトウェアの開発を行ったり[注11]、Debianを「いじる」のが好きで時間が取れる人はunstableを検討するのが良いでしょう。unstableは重大な問題が見つかっても報告の2、3日後には修正されていることはザラで、当日にアップデートが出ることもしばしばあります。ただし、タイミングしだいでは(かなり稀(まれ)ですが)起動しなくなるような問題が紛れ込むこともあり得る「じゃじゃ馬」ですので、その点は覚悟が必要です。利用する場合は、インターネットで情報検索するために別に予備機を1台持っておくと安心です。

参考までに筆者の場合ですが、自身のメンテするパッケージの開発も目的ではあるものの、Debian全体のドッグフーディング[注12]も兼ねてunstableをメインに利用することにしています(一方でリポジトリサーバなどの利用者が多く重要なインフラについては基本的にはstableで、サードパーティリポジトリなどの兼ね合いで一時的に1つ前のstableである「oldstable」を利用します)。unstableを利用していても、特定のバージョンが必要な場合、動作させるときはVirtualBoxやchrootを利用することで検証を行い、パッケージビルドするときはcowbuilder上でunstable/stableに対してそれぞれビルドを行えますので、不都合はありません。また、リスクヘッジとしてデスクトップPCとノートPCの2台を用意し、パッケージのアップデートは若干時期をずらして適用することで、バグで何もできなくなることを防いでいます。

結論:基本はstable、慣れてきて余裕があればunstableが楽しい

testingについては、unstableほど最新とは言いがたいものの、stableとは違って更新がそれなりにやってくるという「どっちつかず」なものになっているのですが、これは「リリース前のクッション役」というリポジトリの性格上、致し方がないところです。

testingにはもう2つほどマイナス点があります。unstableでは直っているのにtestingでは直っていない問題(セキュリティ修正含む)は、開発者からはほぼまったくケアされません(問題があってもマイグレーションで修正される予定なので。testingに個別にケアするような、そんな潤沢なリソースはどこにもありません)。同様に、RCバグが直っていないパッケージは自動で削除され、利用したいときにはtestingに存在しないということがしばしば起こります(これもunstableで修正されないと直りません)。

とはいえ、致命的な問題を避けつつ「全体的に」新しめのパッケージを使いたいという場合はtestingも選択肢になり得ます(筆者からするとunstable/stableでのそれぞれの良い部分が潰されてしまっていて、あまり常用をお勧めできません。ただ例外があって、リリース前のフリーズ期間はtestingを一番美味しくいただけます。この場合はstableに近い形で安定して利用ができるでしょう)。SD

注11)「ソフトウェア開発は行うが、枯れた環境での開発でかまわない」場合はstableでいきましょう。

注12)「eat your own dogfood」とも言い、開発者自身が自分の開発するソフトウェアを利用することで、早期に問題を発見し修正するようにする開発手法。Microsoft社のWindows NT開発を描いた『闘うプログラマー』でも、開発者に対して中期のまだ不安定なNTの利用を強制することで重要な問題を修正するように仕向け、安定性を高める描写があります。

第64回 Ubuntu Monthly Report

インプットメソッドと変換エンジンの遠くて近い関係

Ubuntu Japanese Team　あわしろいくや

今回はインプットメソッドと変換エンジンの発展に関するレポートです。両者は相互に独立して開発されていますが、発展に何らかの関係はあるのでしょうか。はたまたないのでしょうか。調査すると意外なことがわかりました。

歌は世に連れ 世は歌に連れ

Ubuntu 15.10から、インプットメソッドがFcitxに変更されることになりました。中国語では15.04からすでにデフォルトだったので[注1]、これに日本語、韓国語、ベトナム語が続くことになります。

Ubuntuで日本語の入力ができるようになったのは2006年6月リリースのUbuntu 6.06 LTSからでした。最初のインプットメソッドはSCIMであり、9.10からIBusに変更され、15.10からさらにFcitxとなります。

一方、変換エンジンは一貫してAnthyです。しかし、新しい変換エンジンが生まれてこなかったのかというと当然そのようなことはありません。変換エンジンが変わらなかったのは後述するUbuntu特有の事情からで、調査するとインプットメソッドと変換エンジンの発展には切っても切れない関係がありました。まさに「歌は世に連れ、世は歌に連れ」ということです。実際は「歌は世に連れ」は事実ですが「世は歌に連れ」は事実ではないのもまた同じです。というのも、変換エンジンのためにインプットメソッドに手を加えられることは実のところほとんどないのです。

注1）Ubuntuの中国向けフレーバーであるUbuntu Kylinは、以前よりFcitxでした。

インプットメソッドという用語

インプットメソッドという用語は、今回はSCIM/IBus/Fcitxを一般化して呼称したものとして使用していますが、本来これは正しくありません。変換エンジン部分、すなわちAnthyも含めてインプットメソッドと呼称するのが正しいのです。というわけで、今回はSCIM/IBus/Fcitxの一般名称をIMF（Input Method Framework）、変換エンジンは変換エンジン、双方のブリッジをIME（Input Method Engine）と呼称することにします[注2]。

Ubuntu特有の事情とは

Ubuntuのほかにはない特徴として、リポジトリのコンポーネントがmain、universe、restricted、multiverseの4つに分類されていることがあります。後者2つは、いわゆるnon-freeですのでさておき[注3]、mainはCanonicalによるサポートがあり、universeにはありません。それはいいのですが[注4]、Ubuntuにデフォルトで含まれるパッケージは依存するパッケージ、ビルド

注2）WindowsのIMEはInput Method Editorであり、また別のものです。

注3）とはいえmain/universeとほぼ同じなのですが。

注4）すべてのパッケージをCanonicalがメンテナンスするのは非現実的ですし、またそのようなことをするべきではないからです。

に必要なパッケージも含めてすべてmainに存在する必要があります[注5]。universeにあるパッケージをmainに入れる場合はMIR(Main Inclusion Request)[注6]というプロセス[注7]を経る必要があります。

当然といえば当然ですが、そのジャンルにおけるパッケージは厳選されます。すなわち、IBusがmainになるとSCIMはuniverseに戻ることになり、実際にそうなっています。今のところはIBusもFcitxもmainにありますが、いずれはIBusもuniverseに戻ることになるでしょう。Anthyも、ほかの変換エンジンがmainになったら、同じくuniverseに移ることになるでしょう。このことにより何が起きるのかというと、非常に単純な話ですが言語ごとの事情を斟酌(しんしゃく)しません。Debianではどうなっているのかというと、中国語ではFcitx[注8]、日本語ではuim[注9]です。また、変換エンジンもMozcとAnthyがインストールされ、Mozcが優先的に起動するようになっているようです。

MIRというプロセスがあるのならば、そこを通過すればいいじゃないかというのはまったくそのとおりなのですが、現実的にはなかなか難しいのです。というのも、Mozcはたくさんの機能があるので依存関係が複雑です。前述のとおりビルドに必要なパッケージもMIRを通過する必要があるので、膨大な作業量が見込まれます。MIRがどれほどたいへんな作業なのかというのは、Fcitxでの作業の様子[注10]を見れば一目瞭然です。英語の読み書きが不自由なくできることはもちろん、パッケージのクオリティに問題があればそれを修正するだけの知識も必要です[注11]。

SCIMとAnthy

Anthyの開発が始まったのが2000年、SCIMの開発が始まったのは2002年ごろですが、SCIMの開発者は中国人であり、日本語のことはあまり重要視していませんでした[注12]。同じく2002年に開発が始まったIMFであるuimはライブラリとしても使用できたので、scim-uimというIMEが開発され、SCIM + scim-uim + uim + uim-anthy + Anthyという組み合わせでAnthyを使用することはできましたが、uimもIMFですのでSCIMをフロントエンドにする積極的な理由はありませんでした。

そんな中登場したのがscim-anthyです(**図1**)。初のパブリックリリースは2004年11月29日で、バージョンは0.2.0でした。2005年も開発は継続し、その間に実用レベルに達しました。というわけで、Ubuntu 6.06 LTSはscim-anthyを採用することによって最初から「普通に」日本語の入力ができたのです。

今から考えるとscim-anthyのインパクトはとても大きかったように思います。SCIMより前、uimを入れても同様ですが、Linuxディストリビューションでは「普通に」日本語を入力するということは困難だったのです。SCIMとscim-anthyが安定していたのは当然のこととして、たとえばMicrosoft IMEのキーアサインとローマ字テーブルに慣れているのでこれをLinuxディストリビューションでも引き続き使用したいという場合、ATOKと同様の場合、JIS配列キーボードではないキーボード、具体的にはNICOLA(親指シフト)配列を使用したいという場合、それらの要求を一気かつ容易にかなえてくれたのがこのscim-anthyでした。もちろんAnthyの開発も積極的に行われており、当時はベストな変換エンジンでした。

SCIMの品質が十分に高かったこと、Anthyはとてもシンプルな機能しか提供しないのでIME側の実装の自由度が高かったこと、そして何よりscim-anthy開発者の慧眼(けいがん)と開発力の高さが、「普通に」入力できるレベルまで引き上げることができたのでしょう。

とはいえSCIMには構造的な問題がいくつかあり、IBusの登場となったわけですが、デフォルトの座を奪われた後どうなったのかというと、Tizen[注13]のIMFとして採用され、現在でもメンテナンスが継続しています。

注5) 当然各種フレーバーはまったく別です。
注6) 現在鋭意開発中のディスプレイサーバ、Mirとはまったく別です。
注7) https://wiki.ubuntu.com/MainInclusionProcess
注8) https://packages.debian.org/wheezy/task-chinese-s-desktop
注9) https://packages.debian.org/wheezy/task-japanese-desktop
注10) https://bugs.launchpad.net/ubuntu/+source/fcitx/+bug/1356222
注11) とはいえ、つい最近なんとかなるかもしれない動きがありました。
注12) 厳密に言えばできなかったのでしょう。
注13) https://www.tizen.org/ja

IBusとMozc

IBus

IBusの開発が始まったのは2008年5月ごろで、最初のリリースは同年8月10日でした、IBusとibus-anthyは同一の開発者によって同時に開発が進められ、最初のリリース日も同一です[注14]。IBusの開発者は中国人ですが、おそらくscim-anthyを参考に実装したのではないかと思います。scim-anthyはC++でibus-anthyはPythonと開発言語が異なるのでソースコードの流用は不可能ですが、機能はよく似通っています。

IBusの偉大だったところは、開発開始からあまり間をおかずに各種Linuxディストリビューションでデフォルトになったことです。Ubuntuでは9.10からでした。それだけSCIMの構造的な問題が深刻だったということではあるのですが、単純にAnthyを使いたいだけであればSCIMのままのほうが都合がよく、実際に筆者もUbuntu 9.10以降IBusからSCIMに戻して使用していた期間もありますが、あまり長くは続きませんでした。それはMozcが登場したからです(**図2**)。

Mozcの最初のリリースは2010年5月11日です。その時点でAnthyの変換効率を超えていたと記憶していますが、今となってははっきりとは思い出せません。時期的にはリリース日からも明らかなようにUbuntu 10.04 LTSリリースの直後です。

Mozc

MozcはGoogleが開発しており、そのGoogleの戦略をいろいろなところで感じます。まずはGoogle日本語入力のオープンソース版ではあるものの、辞書やそれに付随するファイルは非公開であり、その部分はGoogle日本語入力で採用されているものとは違ったものになっています。ライセンスはオープンソース(3条項BSDライセンス)ですが、開発はオープンではなく、原則として外部からのパッチを受け取ることができません[注15]。変換エンジンだけではなくIMEも含まれているのですが、現在はibus-mozcのみです。scim-mozcはいったんは追加されたものの、ほどなくして削除されました。前述のとおりfcitx-mozcやuim-mozcは取り込むことができないため、Mozcへのパッチという形式で公開されています。もともとはChromium OSのために公開されたのだと推測します。当初はChromium OSもIBusを採用していたため、ibus-mozcが最初からリリースに含まれているのは当然のことでした。その後IBusの使用を取りやめ、専用のIMFが実装されると、ibus-mozcをメンテナンスする理由もなくなり、ibus-mozcはソースツリーから削除される予定です。2013年3月29日のリリースからはAndroidでもビルドできるようになり、最近ではこのAndroid対応が開発のメインになっています。

Anthyの開発を継続するのは困難であり、MozcはGoogle社員以外が手を入れるのは難しいという理

注14) リリースタグが打たれた日はなぜかibus-anthyのほうが1日早いのですが、8月10日リリースと考えていいと思います。

注15) Google Codeが閉鎖されるにあたってプロジェクトがGitHubに移行されましたが、Pull Requestをmergeできないという珍しいプロジェクトになっています。

図1　Ubuntuで最初にデフォルトになったscim-anthy

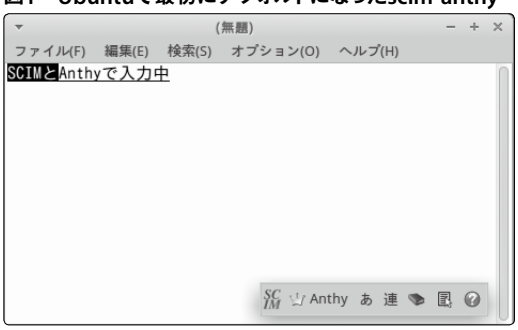
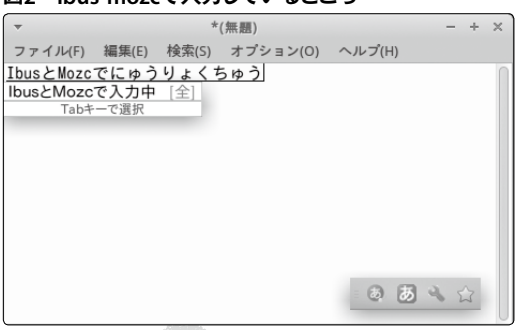


図2　ibus-mozcで入力しているところ



由[注16]から開発が始まったのがlibkkc[注17]です。これはイマドキのアプローチで開発された変換エンジンで、最初のリリースは2013年1月24日です。libkkcの興味深いところは、AnthyのようにライブラリなのですがAnthyよりもサポートする機能が多く、たとえばキーバインドの追加はlibkkc自身に行います。それによってIMEによる違いがあまり出ないようになっています。今後新しいIMFがリリースされても容易に対応ができ、また開発もオープンで[注18]、今後も継続的に発展することが期待できます。

Fcitx(小企鵝輸入法)はパッと出てきたような印象がありますが、元となるバージョンの開発が始まったのは2002年と、かなりの老舗です。とはいえ日本語でも使えるようになったのは2010年にリリースされたバージョン4.0からですので、そこだけ見ると5年間の歴史、と言えなくもないのですが。

fcitx-mozcの初リリースが2012年3月、fcitx-anthyの初リリースが同年7月、fcitx-kkcの初リリースが2013年6月、fcitx-skkの初リリースが同年10月と、日本語にも対応できるようになってそこそこの時間が経ってから各種IMEがリリースされるようになりました。ここで興味深いのは、すべて開発者(中国人)によって開発が開始されたことです。そして実用レベルに達したのはここ数ヵ月～1年くらいのことです[注19]。日本語の翻訳が一通り完了したのが2013年4月ごろであることをつけ加えると、だいたいの流れが見えてくるのではないでしょうか。

個人の主観はさておき、客観的に日本語でFcitxを使用するメリットがどのあたりにあるのかは、筆者にはよくわかりませんでした[注20]。しかし中国語でFcitxが便利に利用できるのであれば、そちらに移行するのは当然のことのように思われましたし、事実そうなったわけです。

では、中国語ではFcitxのどういうところが好評なのでしょうか。筆者は中国語のことはまったくわからないのでなんとも言えない部分はありますが、細やかなニーズを拾っている部分はあるように見受けました。たとえばスキン機能ですが、日本ではあまり馴染みのない機能ではあるものの、Baidu IME[注21]ではサポートしているなど、中国語圏では割にポピュラーな機能なのかもしれません。また、設定項目が多いのもそれだけのニーズがあることの証左のように思います。設定画面をよく見ると、中には日本語ではどういう効果があるのかよくわからない項目もあります。

筆者は中国語の変換エンジン事情にも明るいわけではありませんが、UbuntuではSunPinyinほかいくつかのピンインがデフォルトでインストールされています。そしてUbuntu Kylinの特徴[注22]を見ていると、Sogou Input Method(搜狗輸入法)[注23]が使えることが謳われています(**図3**)。たしかにUbuntu用のパッケージしかないため、Ubuntu/Ubuntu Kylinの特徴と言えるでしょう[注24]。さらにSogou Input MethodはFcitxにしか対応しておらず、このあたりもIBusからFcitxに変更した理由ではないかと思われます。やはりFcitxでも変換エンジン(正確にはピンインですが)との関係が発展の鍵になっていると言えそうです。SD

図3　Sogou Input Method(搜狗輸入法)で入力しているところ

注16) https://fedoraproject.org/wiki/Features/libkkc
注17) https://github.com/ueno/libkkc
注18) 筆者のPull Requestがmergeされる程度にはオープンです。この場を借りて御礼申し上げます。
注19) ただし筆者にはfcitx-skkが常用に達するのかはわかりません。
注20) Fcitxを推進した立場にある筆者が言っていいことなのかはわかりませんが……。
注21) http://ime.baidu.jp/type/
注22) http://www.ubuntu.com/desktop/ubuntu-kylin
注23) http://pinyin.sogou.com/linux/
注24) プロプライエタリなライセンスですので、デフォルトでインストールしておくことはできません。

第41回 Linux 4.0の機能 ～lazytimeとDAX

Text：青田 直大　AOTA Naohiro

6月24日にLinux 4.1がリリースされ、その直後からさっそくLinux 4.2の新機能のcommitが始まっています。4.2でも、f2fsの暗号化サポートやVFSのパス名解決ルーチンの書き換えなど多くの更新が入っています。

今月はLinux 4.0の新機能の中から、ファイルシステム関連の機能であるlazytimeとDAXについて紹介していきます。

タイムスタンプ更新の問題

ファイルシステムでは、ファイルのタイムスタンプとしてctime（inode情報が変更された時刻）、mtime（データが変更された時刻）、atime（アクセスされた時刻）を記録しています。タイムスタンプの取り扱いはファイルシステムにとって悩ましい問題です。

atimeの取り扱いはとくに重大で、これまでにもnoatimeやrelatimeというmountオプションを追加するというトリックが使われてきました。atimeはファイルがアクセスされた時刻を記録するので、ファイルを読むだけでもatimeの更新を行う必要があります。この「読み込みによって書き込みが起きる現象」のため、atimeの更新はとくにパフォーマンスへの影響を与えていました。そこでatimeの更新を行わないnoatimeが昔から使われてきました。しかし、noatimeではファイルが読み込まれたかどうかで既読管理を行うmutt（メールクライアント）のようなatimeが更新されることを期待するアプリケーションが誤作動するという問題が発生します。

そこでrelatimeという折衷案が考え出されました。これは現在のatimeが、mtimeやctimeよりも前だったとき（または、現在のatimeが24時間以上前だったとき）にのみ、atimeを更新するという設定です。この設定であれば、muttのように「ファイルの最終更新時刻が、アクセスされた時刻よりも前かどうか」で既読判定を行っているアプリケーションの動作が壊れることはありません。現在では、noatimeまたはstrictatime（atimeを厳密に毎回記録する）を指定しない限り、デフォルトでrelatimeの動作が行われます。

タイムスタンプの遅延書き込み：lazytime

このnoatime、relatimeをさらに進化させた実装が、Linux 4.0で導入されたlazytimeです。これまでのnoatime、relatimeは「atimeの更新をさぼる」ことで、更新書き込みの問題を避けてきました。そのため、atimeの更新を期待しているアプリケーションはうまく動作しません。たとえば、よくアクセスされるファイルをSSD上に置き、逆に1時間以上アクセスされていないファイルはHDD上に置くといったアプリケーション

は、まったくatimeを更新しないnoatimeでも、24時間に1回しかatimeを更新しないrelatimeでも思ったようには動作しません。

lazytimeでは、この問題を「ディスク上のatimeの更新を遅延する」というアプローチで解決します。lazyatimeを有効にすると、タイムスタンプの情報はメモリ上でのみ更新されるようになります。ディスクへのタイムスタンプの反映は次の3つの条件のいずれかの場合にのみ行われます。

- タイムスタンプ以外でinodeの更新が行われるとき
- ユーザスペースのプログラムがfsync()やsyncを行ったとき
- 削除されていないinodeの情報がメモリから解放されるとき

すなわち、ほかの情報が更新されることでinodeが更新されるときについでにタイムスタンプも更新するときか、syncなどでプログラムからディスクへの更新を強制されたときか、メモリ上から消えてディスクに書き戻すときということになります。

lazytimeのパフォーマンス測定

それでは、lazytimeのパフォーマンスについて見ていきましょう。**リスト1**のようなスクリプトを動かします。このスクリプトは、ext4を

▼リスト1　lazytimeのパフォーマンス測定用スクリプト

```
#!/bin/bash

DEV=/dev/libvirt_lvm/ktest

run() {
    option=$1
    echo $1
    wipefs -a ${DEV} >/dev/null
    mkfs.ext4 ${DEV} >/dev/null
    mount -o $1 ${DEV} /mnt/test || exit 1

    dd if=/dev/urandom of=/mnt/test/file bs=256M count=16 iflag=fullblock
    echo 3 > /proc/sys/vm/drop_caches
    sync
    seekwatcher -t ${option}.trace -o ${option}.png ?
      -p 'for x in `seq 5`; do echo 1 > /proc/sys/vm/drop_caches; cat /mnt/test/file ➲
>/dev/null; done' ?
      -d ${DEV} >${option}.result.txt

    file=file
    cat /mnt/test/${file} >/dev/null
    stat /mnt/test/${file} |grep 0900
    sleep 1

    cat /mnt/test/${file} >/dev/null
    sync
    stat /mnt/test/${file} |grep 0900
    umount ${DEV}
    echo
}

run strictatime
run noatime
run relatime
run lazytime,strictatime
run lazytime,relatime
```

次の5つのmountoptionをそれぞれ用いてmountします。

❶常にアクセス時刻を更新する“strictatime”
❷まったくアクセス時刻を更新しない“noatime”
❸編集時刻が現在のアクセス時刻よりも前だった場合にアクセス時刻を更新する“relatime”
❹strictatimeと同じ挙動だが、ディスクへの書き込みは遅延する“lazytime,strictatime”
❺relatimeと同じ挙動だが、ディスクへの書き込みは遅延する“lazytime,relatime”

そのあと、ファイルシステム上に4Gのファイルを作成し、そのファイルをキャッシュを落としてから読み込む作業を5回行い、1秒間隔で2度ファイルを読み込み、各読み込み後のアクセス時刻を見ています。

まず出力の抜粋から見ていきましょう(**図1**)。

▼図1　出力結果抜粋

```
strictatime
Access: 2015-06-09 20:30:45.254000000 +0900 ①
Modify: 2015-06-09 20:30:35.426000000 +0900
Change: 2015-06-09 20:30:35.426000000 +0900
Access: 2015-06-09 20:30:46.399000000 +0900 ②
Modify: 2015-06-09 20:30:35.426000000 +0900
Change: 2015-06-09 20:30:35.426000000 +0900

noatime
Access: 2015-06-09 20:30:48.456000000 +0900 ③
Modify: 2015-06-09 20:30:55.162000000 +0900
Change: 2015-06-09 20:30:55.162000000 +0900
Access: 2015-06-09 20:30:48.456000000 +0900 ④
Modify: 2015-06-09 20:30:55.162000000 +0900
Change: 2015-06-09 20:30:55.162000000 +0900

relatime
Access: 2015-06-09 20:31:26.374000000 +0900 ⑤
Modify: 2015-06-09 20:31:15.927000000 +0900
Change: 2015-06-09 20:31:15.927000000 +0900
Access: 2015-06-09 20:31:26.374000000 +0900 ⑥
Modify: 2015-06-09 20:31:15.927000000 +0900
Change: 2015-06-09 20:31:15.927000000 +0900

lazytime,strictatime
Access: 2015-06-09 20:31:47.527000000 +0900 ⑦
Modify: 2015-06-09 20:31:37.172000000 +0900
Change: 2015-06-09 20:31:37.172000000 +0900
Access: 2015-06-09 20:31:48.671000000 +0900 ⑧
Modify: 2015-06-09 20:31:37.172000000 +0900
Change: 2015-06-09 20:31:37.172000000 +0900
```

※lazytime,relatimeの結果は省略

stritatimeでは①、②に見られるように、2回のファイル読み込みで、たしかに20:30:45から20:30:46にアクセス時刻(Access)が更新されています。

一方noatimeでは③、④のようにアクセス時刻は20:30:48で変わっていません。relatimeにおいても、⑤、⑥の間ではアクセス時刻は20:31:26と変わっていませんが、noatimeでは更新時刻(Modify)が20:30:55であるのにかかわらず、アクセス時刻は20:30:48のままであるのに対して、relatimeでは更新時刻は20:31:15で、アクセス時刻は20:31:26と、更新時刻よりは後になるように一度は更新されていることがわかります。

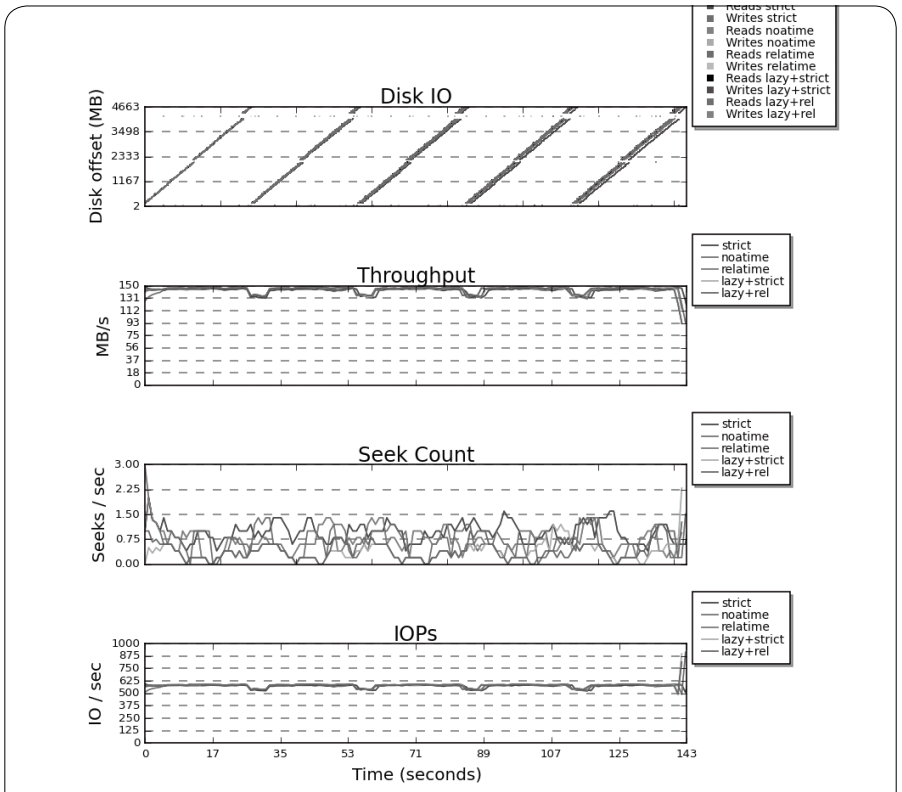
lazytime、strictatimeでは、ディスクへのタイムスタンプ更新が遅延される以外はstrictatimeと変わらないので⑦、⑧のように20:31:47から20:31:48へとアクセス時刻が更新されています。

次にseekwatcherによって出力される比較グラフを見てみましょう(**図2**)。一番下の表を見ると、strictatimeがほかのものより一秒平均シーク回数が一段多く、その影響でスループットやI/O回数も悪化しています。lazytimeを用いた2つはnoatimeとほぼ同等のパフォーマンスをアクセス時刻をきちんと更新しつつも達成していることがわかります。

また、一番上の“Disk IO”のグラフを見ると、4,000MBのあたりに、飛び飛びの点があるのがわかります。dump2fsでジャーナルファイルのinode番号を取得し、debugfsでそのファイル領域を取得してみるとわかりますが、この部分はジャーナル領域にあたります(**図3**)。すなわちstrictatimeにおいてアクセス時刻の更新によりinodeが更新され、ジャーナルへの書き込みが起きていること、それによってシーク回数が増えていることが“Disk IO”のグラフでも確認できるということになります。

このようにlazytimeを使うことで、パフォーマンスを損なうことなく、これまでは困難であっ

▼図2　seekwatcherによる比較グラフ



	Avg Seeks/s	Avg MB/s	Avg IO/s	Run time (s)
strict	0.96	143.06	572.54	143.35
noatime	0.69	144.15	576.6	142.18
relatime	0.7	144.43	577.7	141.92
lazy+strict	0.38	145.07	580.29	141.19
lazy+rel	0.4	145.05	580.2	141.22

▼図3　ジャーナル領域の取得

```
# dumpe2fs /dev/libvirt_lvm/ktest|grep 'Journal inode'
dumpe2fs 1.42.13 (17-May-2015)
Journal inode:            8          # ジャーナルのinode番号=8
# debugfs -c -R "stat <8>" /dev/libvirt_lvm/ktest     # inode 8の情報をダンプ
debugfs 1.42.13 (17-May-2015)
/dev/libvirt_lvm/ktest: catastrophic mode - not reading inode or group bitmaps
Inode: 8   Type: regular    Mode:  0600   Flags: 0x80000
Generation: 0    Version: 0x00000000:00000000
User:     0   Group:     0   Size: 134217728
File ACL: 0    Directory ACL: 0
Links: 1   Blockcount: 262144
Fragment:  Address: 0    Number: 0    Size: 0
 ctime: 0x558a0d2a:00000000 -- Wed Jun 24 10:51:38 2015
 atime: 0x558a0d2a:00000000 -- Wed Jun 24 10:51:38 2015
 mtime: 0x558a0d2a:00000000 -- Wed Jun 24 10:51:38 2015
crtime: 0x558a0d2a:00000000 -- Wed Jun 24 10:51:38 2015
Size of extra inode fields: 28
EXTENTS:
(0-32766):1081344-1114110, (32767):1114111          # ジャーナルのデータ領域は、1081344 blockから1114111 block
$ echo $((1081344 * 4096 / 1024 / 1024)) $((1114112 * 4096 / 1024 / 1024))
4224 4352                                           # すなわち4224MBから4352MBがジャーナルの領域
```

た正確なアクセス時刻の記録を行うことができるようになります。今回はアクセス時刻にフォーカスした実験としましたが、VMイメージやDBのファイルへの書き込みなどファイルサイズは更新されません。したがって、ファイル更新時刻以外のinode情報は変わらない操作においてもパフォーマンスの改善が見られるはずです。

ファイルデータへのダイレクトアクセス

現在ファイルシステムはHDDやSSD上に作られています。これらのデバイスはメモリよりは低速であるため、ディスク上のデータを一度メモリに読み込みメモリ上でデータの読み書きを行い、ディスクへの変更の適用は後でまとめて行っています。この時使われるメモリ上のキャッシュをpage cacheと言います。page cacheを用いることでディスクの遅さをユーザから隠蔽できます。

ところが、この機能はより新しく高速なデバイス上ではかえって障害となります。たとえばNV-DIMMというデバイスがあります。これは通常のDDR4メモリのようにメモリスロットに挿入でき、通常のメモリのように振る舞います。しかし、NV-DIMMにはNANDフラッシュがバックアップとしてついていて、通常のメモリと異なり不揮発性があります。すなわち再起動時にNANDからメモリへとデータの復元が行われます。このデバイス上にファイルシステムを作れば、高速なストレージにすることができます。ここでpage cacheの存在が問題となります。もともと低速なディスクのためにあった機能であるpage cacheは、メモリ並の速度のデバイスの前ではただの「無駄なコピー」にしかなりません。

こうしたデバイスへの最適化として実はext2(だけ)にはXIPという機能があります。これはExecute In Placeの意味で、フラッシュデバイス上の実行可能ファイルをメモリにロードせずに直接実行するという機能になります。これはフラッシュからメモリへのロードの手間と、ロードされる分のメモリ容量を節約できるというメリットのある機能でした。

このXIPはその"execute"の名のとおり、実行ファイルに特化していました。Linux 4.0で実装されたDAXは、XIPをさらに進めすべてのデータアクセスをpage cacheの介在なしに行うものです。ちなみにDAXとはDirect Accessの意味で、XはeXcitingのXとのことです:-)

DAXが有効なファイルシステム(ext2やext4)をmount option daxをつけてmountすると、inode情報にS_DAXというフラグが立ちます。このフラグの付いたファイルへの読み書きは自動的にO_DIRECTが立ったファイルへの読み書きと同等に扱われるようになり、page cacheを迂回するようになります。さらに、ファイルシステムはS_DAXのフラグの立ったinodeへの読み書きおよびmmapをDAXのI/O関数を通じて行います。ここで呼ばれるDAXのI/O関数であるdax_do_io()やdax_mkwrite()は、ファイルに対応するデバイス上のブロック位置をファイルシステムから取得し、そのブロック位置をデバイスドライバのdirect_access関数を用いてメモリ上のアドレスへと変換します。こうしてデバイスへの読み書きがブロックレイヤを通さずメモリ上でのコピーだけで完了します。

では、実際にDAXを使ってみましょう。先ほど書いたとおり、DAXはデバイスドライバのdirect_access関数を用います。今のところ、3つのデバイスドライバにこの関数が実装されてますが、メモリ上にブロックデバイスを作るbrdが一番お手軽に使えるものでしょう。"modprobe brd"にリストのように引数を指定することで1GiBのメモリ上ブロックデバイス/dev/ram0が作成されます(**図4**)。

こうしてmountしたファイルシステム上で読み書きを行い、①ダイレクトI/Oになっていること、②ブロックI/Oが使われていないことを確認します(**図5、図6**)。

①はperf-tools[注1]のtpointを使って“ext4/ext4_direct_IO_enter”というイベントをトレースすることで確認できます。これはダイレクトI/Oが実行されるときのイベントです。

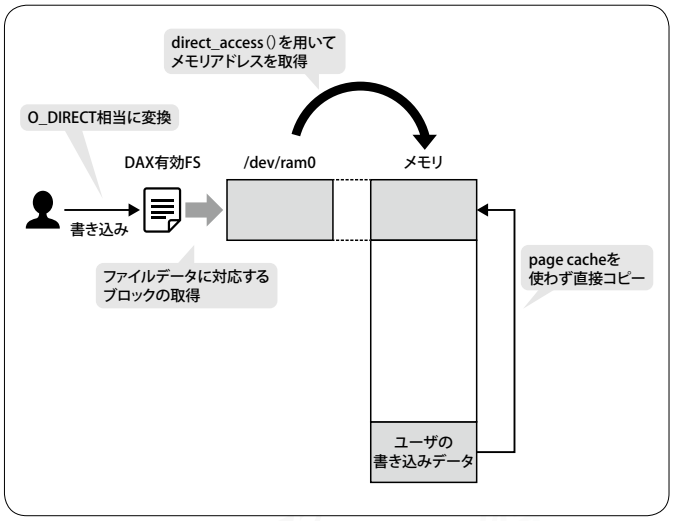
②はbtraceコマンドを用いると確認できます。通常のファイルシステムであれば、ダイレクトI/O時にリストのようにブロックI/Oのトレースが出てきますが、DAXを使ったファイルシステムではこの出力がありません。

まとめ

今月はLinux 4.0の新機能から、ファイルシステム関連の機能にフォーカスしてタイムスタンプ書き込みを遅延するlazytimeと、page cacheとブロックI/Oレイヤを迂回してデバイスへの直接アクセスを可能にするDAXについて紹介しました。Linux 4.1もリリースされていますが、来月ももう少しLinux 4.0の機能について見ていきます。SD

注1）https://GitHub.com/brendangregg/perf-tools

▼図6　DAX有効時の動作



▼図4　メモリ上ブロックデバイスの作成とmount

```
# modprobe brd rd_nr=1 rd_size=$((1024 * 1024))
# fdisk -l /dev/ram0
Disk /dev/ram0: 1 GiB, 1073741824 bytes, 2097152 sectors
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 4096 bytes
I/O size (minimum/optimal): 4096 bytes / 4096 bytes
# mkfs.ext4 /dev/ram0
 ⋮
# mount -o dax /dev/ram0 /mnt/test                    # DAXを有効にしてmount
```

▼図5　ダイレクトI/Oとブロック I/Oのトレース

```
(...DAX使用時...)
# ~/src/perf-tools/bin/tpoint ext4/ext4_direct_IO_enter
Tracing ext4/ext4_direct_IO_enter. Ctrl-C to end.
      dd-19250 [002] .... 76718.267275: ext4_direct_IO_enter: dev 1,0 ino 12 pos 0 len 4096 rw 1
      dd-19250 [002] .... 76718.267290: ext4_direct_IO_enter: dev 1,0 ino 12 pos 4096 len 4096 rw 1
      dd-19250 [002] .... 76718.267294: ext4_direct_IO_enter: dev 1,0 ino 12 pos 8192 len 4096 rw 1
 ⋮
(...DAX未使用時...)
# btrace /dev/ram0
  1,0    2        1     0.000000000 19148  Q  WS 266240 + 8 [dd]
  1,0    2        2     0.000010329 19148  Q  WS 266248 + 8 [dd]
  1,0    2        3     0.000015178 19148  Q  WS 266256 + 8 [dd]
 ⋮
```

August 2015

NO.46

Monthly News from jus

Japan UNIX Society

日本UNIXユーザ会　http://www.jus.or.jp/
法林 浩之　HOURIN Hiroyuki　hourin@suplex.gr.jp

RubyによるLLプログラマのためのUNIX勉強会

今回は、6月に札幌で行った研究会の模様をお伝えします。

jus研究会　札幌大会

■なるほどUnixプロセス
——RubyでUnixの基礎を学ぼう
【講師】島田 浩二（Ruby札幌）
【司会】法林 浩之（日本UNIXユーザ会）
【日時】2015年6月13日（土）15:15～16:00
【会場】札幌コンベンションセンター 201会議室

■LLプログラマのためのUNIX書

2015年度としては最初の研究会となる札幌大会は、札幌を拠点にRubyのコミュニティ活動を行っている島田さん（**写真1**）を講師にお迎えし、UNIXのプロセスをRubyのプログラムで勉強する方法を紹介していただきました。参加者は38人でした（**写真2**）。

はじめに、今回の発表の元ネタとなった電子書籍『なるほどUnixプロセス——Rubyで学ぶUnixの基礎』（**図1**）[注1]の紹介がありました。これまでUNIXプロセスの解説書は、プログラムがCやC++で書かれたものばかりで、いわゆるLLプログラマにとっては勉強するためのハードルが高いのが難点でした。そこをRubyのプログラムで解説することで、LLプログラマにも平易に理解してもらおうというのが本書ならびに今回のセッションの主旨です。

■UNIXの機能をRubyで書いてみる

続いて本題に入り、島田さんがirbというRubyの対話型プログラムを用いてライブコーディングを

注1） Jesse Storimer（著）、島田浩二（訳）、角谷信太郎（訳）、達人出版会、2013年発行、EPUB/PDF/ZIP
URL http://tatsu-zine.com/books/naruhounix

写真1　島田浩二氏

写真2　研究会の様子

しながら解説を進めていきました。

まず、プロセスとは何かという話がありました。プロセスは実行中のプログラムの実体であり、psコマンドで見ると実行中のプロセスが表示されます。個々のプロセスはIDを持っていて、RubyではProcess.pidで確認できます。プロセスには標準入力、標準出力、標準エラー出力という3つの標準ストリームがついていて、RubyではそれぞれSTDIN、STDOUT、STDERRで参照できます。また、UNIXでは標準入出力もファイル入出力も同様のリソースとして扱うことができ、それらはファイルディスクリプタという整数値で表現されます。ファイルディスクリプタは、Rubyではfilenoを使うと確認できます。さらに、UNIXのシェルでよく用いられるリダイレクションは、Rubyではreopenを使うことで実現できます。

下記はreopenを使ったプログラム例と実行結果です。reopenを使って標準出力先をout.txtというファイルに変更してから文字列を出力しています。

プログラム例：redirection_example.rb

```
file = File.open("out.txt","W")
STDOUT.reopen(file)
puts "標準出力先が変わっているはず"
```

実行結果

```
$ ruby redirection_example.rb
$ cat out.txt
標準出力先が変わっているはず
```

これら前半の話をまとめたあと、後半はプロセスの親子関係や、プロセス同士のつながり方を解説しました。まず、UNIXではすべてのプロセスに対して親プロセスが存在します。RubyではこれをProcess.ppidで確認することができます。子プロセスの生成にはforkを使います。forkにより作られたプロセスは、forkが宣言されたブロックの中だけで実行されます。似たようなものにexecがあります。execを使うと別のプログラムを実行するプロセスが作られます。それから、UNIXでよく利用されるパイプは、Rubyではpipeを使って実現できます。

下記はforkやpipeを使ったプログラム例と実行結果です。psコマンドを実行して得られた出力をwcコマンドに入力し、行数を数えて出力しています。

プログラム例：pipe_example.rb

```
IO.pipe do |read_io, write_io|
  fork do
    STDOUT.reopen(write_io)
    exec 'ps', '-x'
  end

  fork do
    STDIN.reopen(read_io)
    exec 'wc', '-l'
  end
end
Process.waitall
```

実行結果

```
$ ruby pipe_example.rb
      13
```

図1 『なるほどUnixプロセス――Rubyで学ぶUnixの基礎』

■最後に

最後にまとめとして今日やったこととやらなかったこと（たとえば環境変数など）を紹介し、「Rubyを使ってUNIXのプロセスを理解することで、より良いプログラミングに役立ててほしい」とコメントされました。

jusの本分であるUNIXの世界をRubyで勉強するという、とても良質なテーマの講演で、参加者にも満足してもらえたのではないかと思います。島田さんにより今回の発表の資料が公開されています[注2]ので、こちらもぜひご覧ください。SD

注2） URL https://speakerdeck.com/snoozer05/naruhounix

Hack For Japan
エンジニアだからこそできる復興への一歩

●Hack For Japanスタッフ
関 治之　Hal Seki
Twitter @hal_sk

第44回 CIVIC TECH FORUM 2015 レポート

2015年3月29日、東京の科学技術館で「CIVIC TECH FORUM 2015」が開催されました。今回はこのフォーラムで行われたいくつかのセッションをピックアップし、日本でのシビックテック活動の一端をお伝えします。

世の人にテクノロジをもっと活用してもらうには

Hack For Japanの関です。何度か本連載でも紹介していますが、筆者はCode for Japanという団体の代表理事もしています。Code for Japanは、「シビックテック」という新しい社会的な動きを推進するための活動を行っています。「CIVIC TECH（シビックテック）」とは、市民がテクノロジを活用して、公共サービスなどの地域課題解決を行うことを指す言葉です。

- 地方自治のありかたを、ITを使ってバージョンアップさせるような取り組みやサービス
- これまで行政が担っていた公共サービスを、地域コミュニティがITを使って効果的に運用する取り組みやサービス
- 地域コミュニティ内で官民問わず多様なプレイヤーが協働しながら地域課題を解決する取り組み
- 地域のリソースを効率的にシェアするようなサービス
- 自治体が情報プラットフォームになり、そこから生まれる新たなサービス

などが当てはまります。

本稿では、シビックテックを盛り上げる目的のために、今年3月に東京・千代田区の科学技術館で開催された「CIVIC TECH FORUM 2015　公共とITの新しい関係」についてのレポートをお送りします。筆者はこのフォーラムの実行委員の1人でもありました。

シビックテックは市民主体の多様なムーブメントである点、まだ黎明期であり明確な事例があまりない点から、今回のフォーラムでは地道に地域で活動をしている人達をなるべく多く集め、参加者と登壇者、参加者同士の交流を重要視していました。講演セッション以外にも、子供でも参加できる電子工作ワークショップや交流スペース（**写真1**）、講演の模様をイラストを使って表現するグラフィックレコーディング（**写真2**）など、さまざまな仕掛けを取り入れています。

このフォーラムの詳細なレポートは、メディア

◆写真1　交流スペース

Photo by Mika Suzuki @civictech forum（CC-BY）

◆写真2　グラフィックレコーディング

Photo by Toshiya Kondo @civictech forum（CC-BY）

パートナーであるfinderさんのほうで公開されていますので、興味を持たれた方はぜひ訪れてください[注1]。

シビックテックとオープンソース文化

オープニングセッション「シビックテックは何をもたらすのか？」では、筆者のほうからシビックテックとオープンソース文化の関連性について解説させていただきました。筆者のシビックテックに対する目覚めは、震災直後に始めたsinsai.info（クラウドソースによる震災情報収集サイト）の活動であり、Hack For Japanの活動の中でのさまざまな人々との対話でした。技術はツールにしかすぎません。技術が正しく使われるには、地域の課題やニーズを、地域に入っていくことで把握し、地域の人達が主体的にITを活用するための手助けをする必要がありました。また、活動の中でこれまでブラックボックスだった行政のしくみを知ることができました。

活動の中で、さまざまなステークホルダーが、それぞれの活動の制約の中で新たな解決策を作ることの重要性を知りました。そういった活動に面白さを見出したのと同時に、自治体がITをもっと戦略的に活用することによる社会的なインパクトを感じたからこそ、Code for Japanの活動につながっています。

講演の中では、コードを書くことやテクノロジ活用といった“How”ではなく、エンジニアリング的な思考と、オープンソースコミュニティ文化こそシビックテックの力だと伝えました。GitHubを使ってIssueの共有がされたり、Pull Requestを通じてこの流れが加速したり、といった部分こそがテクノロジの強みであり、その文化が、エンジニアだけでなくもっと多くの人に伝播していく。10年、20年といったスパンで見ていけば、このようなオープンソース的な考え方が、一般の人たちの中でも普通になる世の中が作れると感じています。

コミュニティデザインの重要性

その後のセッションでは、東京大学工学部都市工学科の小泉秀樹先生から、『地域を支えるコミュニティの変遷とこれからの姿「産官民の関わり合いに今起きている変化」』という講演をしてもらいました。シビックテックで忘れてはいけない要素、それはコミュニティビルディングです。市民のためのテクノロジである以上、技術者だけの活動だけで満足するのではなく、多様な人達の中に入っていき、課題を定義し、解決に向けた持続的な体制を作っていく必要があります。IT技術をどう使うかということだけではなく、従来のコミュニティデザイン論を学ぶことで、シビックテックをどのように社会の中に実装していくのかのヒントにしたいと思い、コミュニティデザインに造詣の深い小泉先生に基調講演をお願いしました。

コミュニティとは？という定義から始まり、自治会などに代表されるような地縁型のコミュニティから、NPOに代表されるようなテーマ型のコミュニティの誕生、そして現代的なコミュニティデザイン論の必要性などについての解説がありました。また、事例として紹介された、神戸市の真野地区や世田谷区の太子堂などでの住民たちによるまちづくりなどから、課題やビジョンを共有し、多様な住民が共に課題解決を行っていくための具体的な手法、ステークホルダー分析、ワークショップ、アウトリーチなどが紹介されました。

日本では、コミュニティマネージャーという職種があまり重要視されない側面があり、来場者もあまり体系立った話を聞いたことがない方が多く、勉強になったという意見が多かったです。

続くセッションでは、ルームを分けて、地域のコミュニティ活動をやってきた人々のセッションと、スタートアップとしてサービスを全国規模、世界規模にスケールさせていくようなことを模索する人々のセッションを行っていきました。

注1　http://fin.der.jp/civictechforum2015/

公と共の担う役割に変化が起きている

地域コミュニティ活動側のセッションからは、株式会社巡の環の信岡良亮さんの、『公共とITのあたらしい関係「公と共の担う役割に変化が起きている」』を紹介します。

セッションでは会場全員に立ってもらい、『「海士町がどこにあるかわからない人」に手を上げさせ、全員手を下ろすことができるまで座れない』というゲームから始まりました。そのルールが提示されると、海士町の位置を知っている人（手を上げていない人）が知らない人（手を上げている人）に海士町の位置を教え始めます。そのうち、海士町の場所を叫んで教えるような人も現れました。

じつは、このとき起きたことが、まさに公と共の話でした。一定の人が集まって生まれた場である「共」の課題は、その場の関係者の中で解決ができるのです。「公」である役所と、「共」を担う人々が合わさって「公共」なのに、いつのまにか公共サービスは「公」のみがやるものという考えになってしまった。「公共サービスがなくなるという言葉を耳にするけれど、公と共とは別のもの。公＝Public（役所）の反対は『私＝Private』であり、共は私たちのことで『Commons』です。公のサービスがなくなっても、人が存在すれば共というサービスはなくならないはず」という考えを教えてくれました。

少子高齢化と、それに伴う税収の低下により、公のサービスができなくなってきたとき、公に不満をぶつけるのではなく、「共」の枠組みを自分達で考え手を動かしていく。そういったときにシビックテックが必要とされるということでした。

「シビックテックスタートアップ」は成り立つのか？

欧米ではシビックテックは成長市場として認知されており、米国におけるシビックテック市場の市場規模は2015年で65億ドル、2013年から2018年にかけて、既存のIT投資に比べ14倍早く成長するという調査結果も出ています[注2]。そのような市場環境のもと、外部から投資を受け早いスピードで成長する会社「シビックテックスタートアップ」が出始めています。一方、日本ではシビックテックは市場としてはまだ認知さえもされていません。

筆者の担当するもう1つのセッションでは、日本でも上記のような市場が生まれるのか？というテーマについて、パネルディスカッションを行いました（**写真3**）。パネラーとして、世界的にもシビックテックの分野をリードしており、Google Impact Challengeという、最大5,000万円の助成金をNPOに支援する施策を日本で開始したばかりのグーグルの恩賀氏、日本を代表する大手企業の中で、共創をテーマにしたメディア「あしたのコミュニティラボ」を運営し大企業のオープンイノベーションをリードする柴崎氏、2015年を「シビックテック元年」と位置づけ、シビックテックフォーラムのセッションでもこの領域への会社としてのコミットを宣言したリクルートの麻生氏、そして起業家育成を20年にわたり行ってきたNPO法人ETIC.から佐々木氏に登壇いただくことにしました。

セッション後半にパネラーが共通して語ったのは、「身近な課題解決」と「インパクト思考」とのバランスでした。シビックテックコミュニティやNPOは、ともすれば目先の身近な課題解決にフォーカスしすぎて、スケールするビジネスモデルが考えられ

注2　参照：Knight Foundationによるレポート
http://www.knightfoundation.org/features/civictech/

◆**写真3　パネルディスカッションの模様**

Photo by Tomoki Yanagawa @civictech forum (CC-BY)

ていない場合があります。一方で、スケールやインパクトばかりのいわゆる“Big Thing”ばかり考えていても、現場感が失われた魂の通わないものになってしまう。起業家側にとってどちらが大事かと言えば、「この課題を解決したい」という情熱と、「実際に解決できている」という部分が、スケーラビリティよりも大事だという点で意見が一致しました。儲かるかどうかではなく、本当に役に立っているかが重要であり、それをスケールするしくみはパートナー企業側でも考えられるというわけです。

起業家側はあまり小利口になる必要はなく、真摯に課題について向き合い、愚直に活動をやり続けること、そして、その動きの延長線をはるか高みに持っていくインパクト思考との両立を意識することが大事なのだと感じました。そして、そんな活動を応援する企業や社会的なしくみは今後増えていくのだろうと思います。

シビックテック大国の、ちいさな取り組み

基調講演として、シビックテック先進地として名高いシカゴから、Smart Chicago Collaborativeのクリストファー・ウィテカー氏に、シカゴ市でのシビックテックの取り組みについて発表していただきました。

ウィテカー氏はもともとエンジニア出身ではなく、イリノイ州政府の職員として働いていました。しかし、旧来型の行政システムと、iPhoneやFacebookなどといった新しいITシステムとのギャップに疑問を感じ、ハッカソンなどのイベントに参加するようになります。その中で技術者と交流するうちに、コミュニティメンバーの中心的存在となっていったそうです。そして今では、シビックテックのコンサルタントとして自身でCivicWhitaker社を設立、シカゴ市のシビックテックコミュニティでも重要な存在となっています。

シカゴでは、このコミュニティ活動の推進を支援する財団などもあり、地域をあげてシビックテックを盛り上げるためのエコシステムができているということでした。また、ウィテカー氏は「オープン・ガブ・ハックナイト（The Open Gov Hack Night）」というイベントをシカゴ中心部に位置するコワーキングスペースで毎週開催しており、行政、企業、市民団体から広く参加者を募り、エンジニアとデザイナー同士をつなぐことの重要性を語りました。ウィテカー氏は、エンジニアと行政両方の気持ちがわかる、双方の「翻訳者」としての存在がシビックテックコミュニティには重要であると言います。シビックテックを通じて良い地域コミュニティを作るためのヒントがいろいろ散りばめられた講演でした。

◆写真4　クリストファー・ウィテカー氏

Photo by Toshiya Kondo @civictech forum（CC-BY）

今こそシビックテックに目覚める時

いかがでしたでしょうか。これからの社会の中で、シビックテック的な考え方は非常に重要なものになっていくと思います。今このような動きにかかわっておくことは、皆さんの将来の可能性を大きく広げるものだと思っています。エンジニアだけでなく、だれでも活動に参加することができます。

本稿で紹介したのは、フォーラムのごく一部のセッションのみであり、まだまだたくさんのテーマが語られています。ご興味をもってくださった方はぜひフォーラムのサイトを訪れていただき、他のセッションも見てみてください。セッションの動画なども公開されています。

Code for Japanの活動にも興味をもってくださった方は、Facebookページ[注3]にも訪れていただけましたら幸いです。SD

注3　https://www.facebook.com/codeforjapan

温故知新
IT むかしばなし

第45回

CPUのスピードレース CPUBENCH

速水 祐(HAYAMI You)　http://zob.club　Twitter : @yyhayami

はじめに

手元に1992年3月号の「ざべ[注1]」があります。そこで筆者はAdvanced Assemblerというx86のアセンブラに関する連載をしていました。毎月、新たな構想とプログラムを1つ作成するのは、時間のない中、忙しくも楽しい作業でした。この号の連載タイトルが「CPUのスピードレース」なのです。

そのころのパソコンCPUの環境

1992年、F1レースは91年までのマクラーレン・ホンダのコンストラクターズチャンピオン4連覇から、ウィリアムズ・ルノーに勝者が移っていく年であり、同じくパソコン界も、PC-9801全盛の時代から、流れが変わる足音が聞こえてくる、1つのターニングポイントの年でした。

その足音とは、性能の高いAT互換機とWindows 3.1です。当時のPC-9801DAに載っていたCPUは、i80386DX 20MHz。対して海外のAT互換機は、i486 33MHzから50MHzに性能アップしようとする状況です。当時新製品の記事に、「PC-9801FA i486SX 16MHz従来機に比べて約1.6倍高速化している」とあります。完全に周回遅れです。使用し始めたWindows 3.0の動作スピードにいらいら感が募る中、パソコンにおけるCPUのスピードに大きな関心が集まっていたのです。

注1) 正式名称はThe BASIC。弊社から1983年5月に創刊され1997年9月号で休刊になりました。

CPUのスピードを測るCPUBENCH

CPUのスピードを比べるプログラムをx86アセンブラでサクっと書いてみようと思い、前述の連載のために作成したベンチマークソフトウェアがCPUBENCHです。

基本は整数と文字列処理のDhrystone(ドライストーン)を利用しています。C言語で記述されたソースをコンパイラにより、最適化なしの設定でアセンブラソースとして出力させ、それを手作業で最適化しています。文字列処理関数のstrcpyなどは、標準ライブラリを使用せずにアセンブラで独自に作成しました(アセンブラの製作記事ですから)。それを3万回ループさせて、1/100秒の精度で、その実行時間を測っています。スピード比較が容易にできるように、処理時間を表示するだけでなく、その指標を我が国の16bitパソコンの原点であろう初代PC-9801(i8086 5MHz)とし、その何倍のスピードであるかを「Ratio to the first PC9801」で表示するようにしました。PC-9801だけでなくIBM-PCおよび富士通FM-Rでも動作できるようにして、バージョン0.98として公開したのです。

CPUBNECHの結果とCPUの載せ換え

表1にCPUBENCHのおもな結果を示します。PC-9801シリーズ2代目のPC-9801E/F(i8086 8MHz[1.8]。([]内はPC98比))でベースを築き、次にCPUをNEC製のV30に載せ換えたPC-9801VM(V30 10MHz[3])で98の優位性が確立しました。そして、i80286を載せたPC-9801VX(i80286 10MHz[6])と続き、PC-9801DA(i80386DX 20MHz[13])で1992年を迎えます。そのとき、AT互換機は、i486DX 33MHz[42]が標準になりつつあったのです。この

ためCPUの載せ換えをするマニアックなユーザが出てきました。

Cyrix社やAMD社が発売を始めたi80386互換で内部処理を高速化したCPUを、PC-9801のマザーボード上のCPUと換装して使用し始めたのです。その際の換装CPUのスピードチェックに、拙作のCPUBENCHの利用が広まったようです。

その後のCPUBENCH

表1の下2つに、その後発表されたx86 CPUの値が示されています。これは、ZOB/Vプロジェクト[注2]のために、バージョンアップしたCPUBENCH v0.99で測定したものです。CPUのスピードアップは予想以上に早く、1/100秒精度、30,000回ループのバージョン0.98では実用に耐えないものになっていました。そこで次のような改良を加え、バージョン0.99を作成しました。

❶測定タイムの100倍以上の精度アップ
❷最新のx86 CPUの自動判別
❸Dhrystone測定ルーチンを連続的に複数(8)並べて、64Kコードサイズの測定を追加

❶は、タイマ処理を行うLSIであるi8253を直接操作することで精度アップを図っています。また、CPUのキャッシュサイズの増加に対応するため、❸のように実行処理部のサイズを広げて測定するベンチマーク処理を追加しています。**表1**の最右列項目の64Kは、その値です。

しかし、その当時の新しいx86 CPUであるスーパースケーラ構造のPentiumや内部的RISC化を図ったPentium Proに対しては、改良したCPU BENCHでも、値にバラツキが出てしまい正確な測定ができない状況になっていたのです。

現在のCPUでCPUBENCHを動かす

第4世代 Core i7-4770K 3.5GHz (Turbo Boost時3.9GHz)でCPU BENCH v0.98をMS-DOSの実行環境を整えて動かしてみると、0で除算と表示されてしまい、次にv0.99で実行したところ、ここでも問題が生じました(**画面1**)。

通常でPC98比4,779、64Kでは、9,350となりました。なぜ通常の処理と64Kにこれだけの差が出たのかは不明です。やはり、16bitのコードをCore i7で実行してスピードを求めるのは無理があり、単純な比較にもならないようです。

もう、しかたがないので、32bit版をサクっと作ってみました。C++のコードをWindows上のMinGW開発環境で32bitコードを出力するようにして、最適化なしでビルドしただけのものです。ループ数は、3万から100倍の300万にして、1/1,000秒精度で測定しています。保有する3種類のCPUで実行した値が**表2**です。

Core i7-4770Kでは、PC98比で66,442となり、今度は値が大き過ぎます。ただ、3種類のCPUのクロック値でPC98比を割った値の比較では、ある程度は信用できそうな結果を出すことができたようです。

おわりに

次回以降は、1992年のターニングポイントを境に、現在という未来への歩みとともに、マイコン黎明期の過去へも遡(さかのぼ)ってみたいと思います。新しいIT社会につながる温故知新であるために。SD

▼表1　過去のCPUBENCH主な結果

CPU	CLOCK	Ratio to PC9801	64K
i8086	8MHz	1.8	
V30	10MHz	3	
i80286	10MHz	6	
i80286	12MHz	7	
i80386SX	16MHz	9	
i80386DX (PC-9801DA)	20MHz	13	
i80386DX	33MHz	24	
Cyrix486DRx2	40MHz	29	
i486DX	33MHz	42	
i486DX	50MHz	64	
Pentium	133MHz	219	135
Pentium Pro	200MHz	267	150

▼表2　CPUBENCH 32ビットバージョン結果
クロック()内は、Turbo Boost時

CPU	クロック	Ratio to PC9801	PC98比/クロックGHz
Core i7-4770K Haswell	3.5GHz(3.9GHz)	66442	18983
Core i7-3537U Ivy Bridge	2.0GHz(3.1GHz)	38820	19410
Core2 Duo D9400	1.4GHz	18983	13559

▼画面1　CPUBENCHでの実行結果

```
x86 CPU Speed TEST(16bits) v0.99c Copyright 1996 ZOBplus Hayami
DHRYSTONE 30000 LOOPS
CPU Type        : Pentium 3660 MHz
CPU ID    : family=6 model=c step=3
CPU mode        : real
Execute area    : 4000:0000 ->4000:1C5C / FC5C

                                        64K
Ratio to the first PC9801 : 4778.67     9350.46
            Execute time :  14.5 ms      7.4 ms
```

注2) PC-AT互換機を自作し、パソコン通信「ZOB Station BBS」のメンバ同士で情報とノウハウを共有しようという目的でスタートしたプロジェクトです。1994～96年の3年間、その時点で先進性・安定性ともにベストと思われる部品を使用して、共通のPC-AT互換機を製作しました。

この個所は、雑誌発行時には記事が掲載されていました。編集の都合上、
総集編では収録致しません。

この個所は、雑誌発行時には記事が掲載されていました。編集の都合上、
総集編では収録致しません。

この個所は、雑誌発行時には記事が掲載されていました。編集の都合上、
総集編では収録致しません。

この個所は、雑誌発行時には記事が掲載されていました。編集の都合上、
総集編では収録致しません。

この個所は、雑誌発行時には記事が掲載されていました。編集の都合上、
総集編では収録致しません。

この個所は、雑誌発行時には記事が掲載されていました。編集の都合上、
総集編では収録致しません。

この個所は、雑誌発行時には記事が掲載されていました。編集の都合上、
総集編では収録致しません。

この個所は、雑誌発行時には記事が掲載されていました。編集の都合上、
総集編では収録致しません。

グレープシティ、機能別グリッド2種の新バージョンを同時発売

グレープシティ㈱は5月25日、業務アプリに欠かせないデータの参照や登録を行う画面の開発を支援する2種のデータグリッド「SPREAD for Windows Forms 8.0J」と「MultiRow for Windows Forms 8.0J」を発売した。

「SPREAD」は多機能表計算コンポーネント、「MultiRow」は1レコード複数行表示を実現するツールとなっている。2製品とも高DPI対応で、今夏リリースが予定されているWindows 10、Visual Studio 2015などの最新環境についても、Service Pack（無償アップグレード）を迅速に公開することで対応していく予定とのこと。1開発ライセンス価格は、SPREADが172,800円（税込）、MultiRowが129,600円（税込）となっている。

業務アプリにおいてデータグリッドは欠かせない存在だが、データの集計・一覧表示の作成、1レコード複数行、チャート表示といったように、求められる要件は開発案件ごとに異なる。そのため同社が提供するWindowsフォームアプリ開発用のデータグリッドコンポーネントには今回提供の2製品をはじめ、複数の製品が用意されている。一覧画面やチャート表示はSPREADで開発、入力画面はMultiRowで開発するといったように、異なる機能を持つデータグリッドを併用することができる。

CONTACT
グレープシティ㈱ URL http://www.grapecity.com

リックソフト、ヤフーの約7,000人の情報共有システムに、アトラシアンの「Confluence」を活用

リックソフト㈱は、ヤフー㈱の全社約7,000人が利用する情報共有システムに、アトラシアン社製のビジネスツール「Confluence」を採用し、導入を支援した。

「Confluence」はアトラシアン社が開発する企業向けの情報共有のためのツール。チームがコンテンツを作成、共有、議論するための、シンプルかつ高機能なナレッジ管理ツールだ。プロジェクト、ドキュメンテーション、ファイル、アイデア、議事録、仕様、図、モックアップなどの情報をチームで効率的に共有できる。

ヤフーでは数年前より経営陣を含めた全社の約7,000人がConfluenceを利用し、1日あたり15,000ページもの情報が更新され、共有されている。

リックソフトは、運用においてのさまざまな課題を解決するために技術支援を提供し、ヤフーの情報共有システムを支えている。また、同社では現在、独自開発のアカウント管理システムとConfluenceのアクセス制御機能を連携させており、人事異動があった場合でもConfluence側に自動的に反映している。さらに、Confluenceの「社内FAQ」機能や、プロジェクト管理ツールである「JIRA」との連携も視野に、活用範囲を広げていく予定とのこと。

CONTACT
リックソフト㈱ URL http://www.ricksoft.jp

マップアール・テクノロジーズ、「Apache Drill 1.0」を提供開始

6月11日、マップアール・テクノロジーズ㈱は、同社が提供するHadoopディストリビューション「MapR」に含まれる「Apache Drill 1.0」の正式な提供を開始した。

「MapR」は、ビッグデータの分散処理技術「Apache Hadoop」の商用ディストリビューションの1つ。そのMapRのパッケージの中で今回正式提供が始まった「Apache Drill 1.0」は、事前のスキーマ定義を必要とせず、Hadoopに対してANSI SQLでのアクセスを可能にする製品。ユーザは本製品を活用することで、複雑なJSON形式などを含む幅広いデータタイプを、新旧を問わずあらゆるデータソースから、従来のSQL文で準備・検索できるようになる。

もともとHadoopは、スケーリング可能で費用対効果が高く、スキーマレスの書き込みを行うことが可能だった。今回の「Apache Drill」の提供により、データの読み出しにおいても、費用対効果の高いスキーマフリーでの読み出しが可能になる。

本製品の一番の売りは、企業内のビジネス部門のユーザが、IT部門に頼らずHadoopにアクセスできるという点で、既存のBIツールを使ってリアルタイムにビッグデータを活用できる。

CONTACT
マップアール・テクノロジーズ㈱ URL https://www.mapr.com/jp

Report

アカマイテクノロジーズ、「Akamai Conference 2015」開催

6月11日、ザ・プリンスパークタワー東京（東京都港区）にて、「Akamai Conference 2015」が開催された。

「Akamai Conference 2015」は、アカマイテクノロジーズ合同会社によって開催された、同社の技術セッション、ユーザ企業の事例紹介などを含むイベント。同社マーケティング本部の中西一博氏によって行われたプレセッション「DDos攻撃のトレンドとアカマイ・セキュリティソリューション」の模様を紹介する。

セッション始めに行われたのはアカマイの大規模セキュリティ調査の報告。これは、世界中に配置された17万台以上のエッジサーバからなるアカマイのネットワーク上で、2015年第1四半期に観測された、Webアプリに対する数百万件の攻撃データと、DDoS攻撃の分析結果である。おもな結果は次のとおり。

- 攻撃件数は1年で2倍以上の増加
- 平均攻撃時間は四半期で40%以上増加
- 100Gbps以上の攻撃は8件発生（最大は170Gbps）
- とくに「SSDPリフレクション攻撃」「WordPressプラグインを標的とした攻撃」が増加

また、社会情勢として懸念されているのが、DDoS攻撃請負業者の台頭と、業者間の価格競争の加速である。業者によっては1,000円ほどの代金で、100Gbpsを超える代理攻撃が可能になった。「子供がお小遣いを使って、企業へのクレームの代わりに大規模なDDos攻撃をしかけるといったことができてしまう」と、中西氏はこの状況を危惧している。セッションではほかに、脅迫メールでサイバー攻撃をほのめかし、ビットコインを要求する「DD4BC」という新しい脅威も紹介された。

セッションの最後にはアカマイが提供するDDoS攻撃対策ソリューション「Kona Site Defender」が紹介された。これは、世界中に設置されたアカマイのエッジサーバを使って攻撃トラフィックを「超分散」させるしくみ。攻撃元に近いエッジサーバ群でまだ小川の状態の攻撃を受け止めることで、大河になる前に緩和させられる。大量のサーバを抱え持つアカマイならではのDDoS対策と言える。

CONTACT
アカマイテクノロジーズ合同会社 URL http://www.akamai.co.jp

Service

リンク、拠点間VPNサービスの提供開始

㈱リンクは、物理サーバの追加・削除・コピーをコントロールパネルの操作からできるベアメタルクラウドサービス「ベアメタル型アプリプラットフォーム」において、6月24日より、「拠点間VPNサービス」の提供を開始した。

「ベアメタル型アプリプラットフォーム」は、セキュリティやパフォーマンスの面から、物理サーバを利用したい、あるいは物理サーバと仮想サーバをうまく使い分けたいといった要望を持つ多くのユーザから好評を得ている。また、コントロールパネル上から仮想サーバを利用する感覚で物理サーバを運用できるため、セキュリティ要件などによって他社とのリソース共有を望まないユーザからの評価も高い。物理サーバを提供しているサービスの特性上、データを完全に消去できるため「サービス終了後にエンドユーザの個人情報をサーバに残さず、完全に消去したという証明書がほしい」などの要望にも対応できている。「データを守る」だけでなく「データを完全に消す」という観点においても安全に運用できるサービスとなっている。

このように、セキュアな環境下で安定した稼働が実現できることから、新規案件を立ち上げる際に本サービスでのシステム構築を検討するユーザが増えるとともに、自社で構築した既存の環境からの完全移行を検討するケースも増えてきている。

一方、移行に際して一部のサーバなどを、アウトソースせずに社内環境に置いておきたいといったケースも多くある。その場合、自社に残したサーバと本サービスへ移行した環境が、いかに安全に通信できるかが大きな課題となる。そのような課題を解決するため、今回リリースされたのが「拠点間VPNサービス」である。これを利用することで、ユーザ企業は本サービスと各拠点間をセキュアな状態でローカル接続できるようになる。社内ネットワーク上にあるサーバとの接続など、既存の資産を活かした運用を望むユーザや、サーバのアウトソースの流れの中でどうしても社内に残ってしまうサーバと安全に通信したいと望むユーザにとって、最適なサービスとなっている。

CONTACT
㈱リンク URL http://www.link.co.jp

Report

レッドハット、「Project Atomic Meetup」開催

6月2日、レッドハット㈱主催のイベント「Project Atomic Meetup」が開催された。

「Project Atomic Meetup」は米Red Hat社が開発を進めているコンテナに特化した軽量OS「AtomicHost」に関する技術イベント。6月3～5日に開催された「LinuxCon Japan 2015」に合わせて来日した、「AtomicHost」のコミュニティマネージャを務めるジョー・ブロックマイヤー氏が招かれた。Project Atomicの最新情報を取材したのでその模様をレポートする。

ブロックマイヤー氏は、まずProject Atomicの概要について紹介した。Fedora、CentOS、それにRed Hat Enterprise Linux（RHEL）のそれぞれのプラットフォームに対応したAtomicHostが開発されていることを説明し、「なぜAtomic用にディストリビューションを自作しなかったのか？」という問いを自ら挙げたうえで、「すでにユーザが慣れているOS上でコンテナを動かすことがもっとも重要。そのためにもう1つのディストリビューションを作るのは賢い選択とは思えない」と説明、「AtomicHostの開発は、常に最新の機能を盛り込むFedoraで行われ、その次にCentOS、それからRHELに対応する順番になるだろう」とリリース方針を示した。

AtomicHost注目の新機能、SPCとCockpit

次に紹介されたのが、現在開発が進む新機能「Super Privileged Container（SPC）とCockpit」。氏によれば「コンテナはもともとアプリケーションをアイソレーションさせて依存関係をなくし、ポータブルにしたものだが、デバッグや管理のためにホストのシステムやほかのコンテナと接続できる特権を持ったコンテナが必要となった。そのためにSPCを開発した」とのこと。

また、Dockerコンテナの管理ツールであるCockpitは「AtomicHostが開発されるよりも前に、Red Hat社の中でプライベートなプロジェクトとして開発されていたソフトウェア。AtomicHostのプロジェクトを始めるときに『そういえばそんなツールがあったよね』みたいな感じで掘り起こされたもの」だそう。つまり、サーバを管理するためのツールではあるが、コンテナを管理対象にできるもののようだ。

このあと、実際にノートPCでAtomicHostとCockpitを稼働させるデモが行われ、いかに少ないリソースしか要求されないかをモニターしながら解説した。Meetupに参加した参加者からは「Cockpitはプロダクションのシステムに適用できるのか？」という質問が出たが、これに対して「Cockpitはまだベータの段階。あくまでも開発やテスト用として使ってほしい」との答え。

さらに、社内で開発されていた別のソフトウェアとして、rpm-ostreeというOSのアップデートのための機能も紹介された。rpm-ostreeはOSのアップデートなどを行う際に現在のOSのスナップショットを保存しておき、何か不具合が発見された際にリブートして以前のOSイメージへのロールバックを可能にするもの。これもProject Atomicの中で利用されている。

CoreOS RocketとAtomicHost

Meetup後の懇親会で、Dockerに対抗するコンテナ技術Rocketについて氏に伺った。

――***CoreOSが推進するコンテナ技術、Rocketについてどのように考えておられるのか教えてください。***

Rocketは、確かにコンテナ技術に対する1つの回答ではありますが、Red Hat社としてはDockerを主として実装することを考えています。

――***システムインテグレータなどから、「これからはDockerではなくRocketなのでは？」と聞かれることがあるのですが、それに関してはどう思われますか？***

コンテナは基本的にはアプリケーションを動かす技術であって、コンテナそのものが重要なのではありません。これから数年のうちにもっと別のコンテナ技術が出てくるかもしれないのです。ただしRed Hat社のスタンスで言えば、企業ユーザがすでにRHELやCentOSなどを使ってシステムを構築しているのが実態であると思います。そのため、現時点で動いているシステムから大きく変えることなくコンテナ技術を導入できるしくみが必要です。そのためには既存のプラットフォームを維持することが重要なのです。Rocketをすぐに導入したいというユーザとして考えられるのは、まったく新しくビジネスを起こすベンチャー企業ですね。しかし、実際にそういうユーザは少ないのではないのでしょうか。

◆ ◆ ◆

Project Atomicでは、活発に活動しているコミュニティメンバが現在20～30名程度なので、もっと多くのデベロッパに参加してほしいとのこと、興味のある方はプロジェクトにコンタクトしてはいかがだろうか。

・著者プロフィール

松下康之（まつしたやすゆき）
フリーランスライター＆マーケティングスペシャリスト。DEC、マイクロソフト、アドビ、レノボなどでのマーケティング、ビジネス誌の編集委員などを経てICT関連のトピックを追うライターに。オープンソースとセキュリティが最近の興味の中心。

CONTACT
レッドハット㈱ URL http://www.redhat.com/en/global/japan

Topic

ギットハブジャパン、GitHub初の海外支社として設立

GitHub社は6月4日、初の海外支社として日本支社「ギットハブ・ジャパン合同会社」を設立したことを発表した。同社は東京を本拠地とし、今後はジェネラルマネージャーである堀江大輔氏が運営を行う。

発表会にはGitHub社のCEOクリス・ワンストラス氏も参加し、「海外初のオフィスをオープンすることはGitHubにとってとても重要な節目です。イノベーションに富んだソフトウェアを生み続けてきた、今でも刻々と成長を続ける日本のデベロッパコミュニティをサポートする日本法人を設立できることを、とてもうれしく思っています。新しく就任したジェネラルマネージャーの堀江大輔は、豊富なビジネス経験とGitおよびGitHubの深い理解で、日本のお客さまにとって必ず役に立ってくれると信じています」と述べた。日本での利用のされ方について、2008年のまだ日本語サポートのないGitHub設立当初から、github.comへのアクセス数は上位10ヵ国に入り続けてきた。そして、日本のユーザは現在も増加し続けており、2014年の日本ユーザのGitHub上でのアクティビティは、前年比60%も増加したとのこと。

◎「GitHub Enterprise」の日本展開

「GitHub Enterprise」は、オープンソースプラットフォームと同じプロジェクト環境を、自分たちの組織内にクローズドで再現できるサービス。日本法人設立に合わせて、これまで言語や決済などの観点から導入が難しかった国内企業に対しても、さらに迅速できめ細かいサービスやサポートを提供するため、マクニカネットワークス㈱と国内総代理店契約締結を行い、日本語による同サービスの法人向け導入サポートを開始した。この提携により、円建て決済や日本語のテクニカルサポートも受けられるようになる。

▲堀江大輔氏（左）、クリス・ワンストラス氏（右）

CONTACT
ギットハブジャパン合同会社 URL http://github.co.jp

Software

GitHub社、テキストエディタ「Atom 1.0」をリリース

GitHub社は6月26日、テキストエディタ「Atom 1.0」を正式にリリースした。

「Atom」は、GitHub共同創業者でCEOのクリス・ワンストラスの、"Webデベロッパー向けにEmacsのような自由にカスタマイズできる新世代のエディタを最新のWebテクノロジを用いて開発したい" という思いから開発が始まり、約1年前にβ版が一般公開された。β版時点でも、Atomのダウンロード回数は130万回、月間アクティブユーザは35万人に達していた。また、Atomのコミュニティでは、660種類のテーマ、2,090種類のパッケージが作られ、この中からはlinter、autocomplete-plus、minimapなどの人気作も生まれた。

Atomは現在に至るまで155回ものアップデートが行われ、パフォーマンス、安定性、機能、モジュラリティなどあらゆる面で大きく進歩を遂げており、スクロール、タイプ入力、起動すべてが高速化した。正式版は現在、Windowsインストーラ版やLinuxパッケージ版として提供されている。また、ウィンドウペインのリサイズやマルチ・フォルダ同時展開機能など以前から強く要望されていた機能が追加されている。

さらに、モジュラー化の面では、大量の新機能が追加された。主なものを挙げれば、ステーブル版のAPI、ECMAScript 6の新機能をパッケージに導入するためのbabelライブラリのサポート、サービスを通じてパッケージが通信する機能、コアエディタを拡大するための各種の表示オプション、そしてUIを自動的にシンタックスカラーに合わせる新しいテーマなどだ。一方、auto complete-plusなどいくつかの優れたコミュニティー版のパッケージに敬意を表して「Atom 1.0」ではこれに対応する機能を削除したとのこと。

▲ダウンロードページ（https://atom.io/）

CONTACT
GitHub, Inc. URL https://github.com

Report

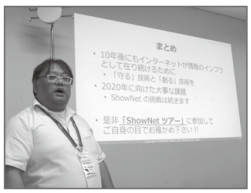
「Interop Tokyo 2015」開催

活気が感じられたInterop2015

6月10～12日、幕張メッセにて開催されたInterop Tokyo 2015は、総出展数が1,345、参加企業486社、来場者数はのべ136,341人（2014年132,609人）と、昨年よりも幾分賑やかな印象を受けるものだった。イベントの目玉でもあり、Ineteropの屋台骨ともいえるネットワーク環境の構築については、本誌連載記事「ShowNetが示すネットワークの近未来」を参照してほしい。

ShowNetの見どころとは

東京大学情報基盤センター関谷勇司氏によるとShowNet 2015の最大のテーマは「今のインターネットはあと10年生き残れるか」。2020年の東京オリンピックに向けた課題でもあり、そのためにどうすべきか、現時点の技術をベースに考えるというもの。今年のShow Netネットワークの規模は、機材総額がおよそ74億円、参加した技術者はのべ400名以上、設計に半年以上をかけ、2週間ですべてを構築したという。最新の機材と技術で作り上げたもので1つとしてムダなものがなく、誇れるものだという。そしてネットワークの見どころとして「セキュリティ」「Wi-Fi」「IoT（Internet of Things）」「NFV」「インタークラウド／マルチクラウド」の5点を挙げた。



▲関谷 勇司 氏

データとインフラの両面でセキュリティ展開

インターネットの機能拡張を続けてきて課題になるのは、“今のまま10年いけるのか”ということ。そのことに3年がかりで解決を求めてきており、今年で2年目。今年はバランス重視で設計した。多機能にしてインフラが複雑になると、故障が増えて実用に耐えられなくなる。そこで便利さとタフさのバランスを見極めたという。

データを守るセキュリティとして、標的型攻撃やマルウェアに対してはその前兆をとらえ、攻撃をブロックする「運用によるサイバーキルチェーン」を提示した。そして、インフラを守るセキュリティとして、DDoS攻撃対策を重視。DDoS攻撃は近年で400Gbpsに達するものもあり大規模化している。その対策には個別組織で行うのではなく、ISPやIX、そして自社網で防御する必要がある。さらにヨーロッパで導入が進んでいるRPKI（リソースPKI）にも注目した。そのため全対外接続ルータの全経路でRPKI相互接続実証実験を行った。最後に提唱されたのが「エンドポイントまでのセキュリティのための多層防御の重要性」である。この分野には多くの製品があり、本イベントでは、これらをどのように適用するのか示すことで、セキュリティオーケストレーションモデルを提示するのが目的。

今年は一般的な無線設計をせずに、「スタジアムアンテナを会場内に対角で設置」（アクセスポイントを減らすため）、「同軸漏洩ケーブルによる局所化」といった技術を使用し、“きれいな無線（Wi-Fi）”を目指したとのこと。これで従来と比較して劇的に改善を図ることができたという。Bluetoothや出展社所有のWi-Fiなどで通信が混乱し、SSIDのビーコンすら発信できないのが現状なので、あえて電波密度を疎にしたのだ。同時にNOC（ネットワークオペレーションセンター）では電波状態を計測し常時監視をしている。

また、IoTについては、会場内のゴミ箱の中にセンサーを設置し、ゴミの量をスキャンするセンサーネットワークの実験を実施した。

SDN/NFVでは、真にスケールアウトできるアーキテクチャとは何かという課題を挙げて検証を試みたという。SDN/NFVの利点は欲しいときに使いたい機能が使えることだが、それがタフなサービスとして使えるのかが問題という。そのため、ソフトウェアを多層化してスケールアウトできるアーキテクチャを構成した。インターコネクティビティのためにVxLAN、Ethrenet VPNやOpenStackなどの実証実験も行っている。

これらの検証環境はすべてNOCの中のラックに収められており、その隣にあるホワイトボードに書かれた吹き出し1つ1つに技術者の気持ちが込められている。来年はどのような技術が使われるのか、Interop 2016を訪れるときはまずはNOCから見学することを勧めたい。

Interop番外編「村井 純先生還暦を祝う会」

Interopはインターネットにかかわる多くの人間が集うイベントであるが、“Internetの父”慶應義塾大学村井純先生の還暦を祝う会が、6月11日の夕方より開催された。スターウォーズ風のビデオ画像が会場内巨大スクリーンに映写され、はなばなしくスタートした会は、永井美奈子アナウンサーが司会をつとめ、さんまのまんま風の舞台でぞくぞくとゲストが見えるという設定。SFCの1期生や夏野剛氏など村井先生にゆかりのある方々が続々と登場し還暦を祝った。

CONTACT
Interop Tokyo 2015
URL http://www.interop.jp/2015/

日本ヒューレッド・パッカード、ストレージサーバ「HP Apollo 4000シリーズ」を発表

日本ヒューレット・パッカード㈱は7月2日、ビッグデータに最適化されたストレージサーバ「HP Apollo 4000シリーズ」を発表した。

同シリーズは、ソフトウェアベースの分散型ストレージに最適な、高密度実装、シンプルな拡張性、そして柔軟性を持つサーバプラットフォームだ。2製品のおもな特徴は次のとおり。

・HP Apollo 4200 System

2Uサイズに最大224TBとなる28本の3.5インチ（LFF）ドライブ、または50本の2.5インチ（SFF）ドライブを内蔵可能。標準的な2Uラックサーバとして、既存のラックをそのまま利用できる

・HP Apollo 4530 System

4Uラック型シャーシに3台のサーバノードと各ノード15本の3.5インチ（LFF）ドライブを提供。CPUパワーとスピンドル数のバランスを重視した高密度サーバ

▲HP Apollo 4200 System（左）
HP Apollo 4530 System（右）

CONTACT
日本ヒューレッド・パッカード㈱ URL http://www8.hp.com/jp/ja

デベルアップジャパン、パケットキャプチャ専用機「Sonarman」を発売

㈱デベルアップジャパンは6月23日、同社の豊富なトラブルシューティング経験を活かした、障害対応に特化したパケットキャプチャ専用機「Sonarman」を発売した。

コンピュータネットワークにおいて、システムの入出力を直接解析する「パケットキャプチャ」はトラブルシューティングに非常に有効な手法。同製品はパケットキャプチャを継続的に取得し、障害対応に役立つ“ネットワークにおけるドライブレコーダ”となっている。

本製品は、ポートミラーリングによって常時監視対象サーバのパケットをキャプチャし、リングバッファに一定期間記録を保持、エラー発生時に投げられたSyslogメッセージをトリガーに、発生時のパケットをストレージに避難させるというしくみ。2GBのメモリ、128GBのSSDを搭載している。再現性の低い障害への対応策として、また外部業者とのやりとりにおけるエビデンス（証拠）を基本としたトラブルシューティングの精度向上のための利用を想定している。

価格は、初期設定および各種サポートを含め、35万円（税別）から。仮想アプライアンスとして動作する無償バージョンも公開されている。

CONTACT
㈱デベルアップジャパン URL http://develup-japan.co.jp

ネオジャパン、グループウェア「desknet's NEO V3.0」をリリース

㈱ネオジャパンは6月10日、「desknet's NEO」の新バージョン（V3.0）の提供を開始した。

同製品は累計316万ユーザ（2015年3月時点）の販売実績を持つ、純国産のWebグループウェア。製品コンセプトは「現場主義」で、ユーザへのヒアリング結果をもとにバージョンアップのたびに現場で必要な機能を積極的に実装してきた。V3.0では次の機能強化が行われた。

① 交通費／経費精算機能の搭載
② 動画の配信／画像編集ツールの搭載
③ WebメールのIMAP対応
④ Webメールの誤送信防止機能の再強化
⑤ データの自動保存／回復

とくに①については、ジョルダン㈱の「乗換案内Biz」との連携により、スケジュール登録と同時に訪問先までの交通経路を検索したり、交通費や経費を蓄積して経費精算申請を行ったりできるようになった。

同製品のクラウド版は1ユーザ月額400円（税別）。パッケージ版は5ユーザ39,800円（税別）～。今回追加された交通費／経費精算機能は、別途オプション費用が必要。

CONTACT
㈱ネオジャパン URL http://www.neo.co.jp

Software Design plus

Software Design plusシリーズは、OSとネットワーク、IT環境を支えるエンジニアの総合誌『Software Design』編集部が自信を持ってお届けする書籍シリーズです。

WordPressプロフェッショナル養成読本
養成読本編集部 編
定価 1,980円+税　ISBN 978-4-7741-6787-9

サーバ/インフラエンジニア養成読本 ログ収集〜可視化編
養成読本編集部 編
定価 1,980円+税　ISBN 978-4-7741-6983-5

フロントエンドエンジニア養成読本
養成読本編集部 編
定価 1,980円+税　ISBN 978-4-7741-6578-3

PHPライブラリ&サンプル実践活用[厳選100]
WINGSプロジェクト 著
定価 2,480円+税　ISBN 978-4-7741-6566-0

アドテクノロジープロフェッショナル養成読本
養成読本編集部 編
定価 1,980円+税　ISBN 978-4-7741-6429-8

[改訂新版] サーバ/インフラエンジニア養成読本 管理・監視編
養成読本編集部 編
定価 1,980円+税　ISBN 978-4-7741-6424-3

[改訂新版] サーバ/インフラエンジニア養成読本 仮想化活用編
養成読本編集部 編
定価 1,980円+税　ISBN 978-4-7741-6425-0

[改訂新版] サーバ/インフラエンジニア養成読本
養成読本編集部 編
定価 1,980円+税　ISBN 978-4-7741-6422-9

iOSアプリエンジニア養成読本
高橋俊光、諏訪悠紀、湯村 翼、平屋真吾、平井祐樹 著
定価 1,980円+税　ISBN 978-4-7741-6385-7

[改訂新版] Linuxエンジニア養成読本
養成読本編集部 編
定価 1,980円+税　ISBN 978-4-7741-6377-2

Webアプリエンジニア養成読本
和田裕介、石田絢一（uzulla）、すがわらまさのり、斎藤祐一郎 著
定価 1,880円+税　ISBN 978-4-7741-6367-3

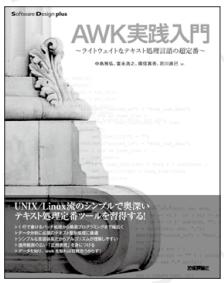
エンジニアのためのデータ可視化[実践]入門
森藤大地、あんちべ 著
定価 2,780円+税　ISBN 978-4-7741-6326-0

GPU並列図形処理入門
乾正知 著
定価 3,200円+税　ISBN 978-4-7741-6304-8

Zabbix統合監視徹底活用
TIS㈱ 池田大輔 著
定価 3,500円+税　ISBN 978-4-7741-6288-1

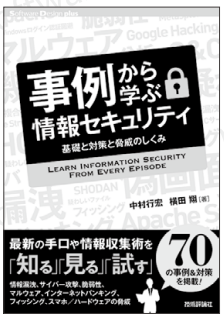
過負荷に耐えるWebの作り方
㈱バイプドビッツ 著
定価 2,480円+税　ISBN 978-4-7741-6205-8

上田隆一 著
USP研究所 監修
B5変形判・416ページ
定価 2,980円(本体)+税
ISBN 978-4-7741-7344-3

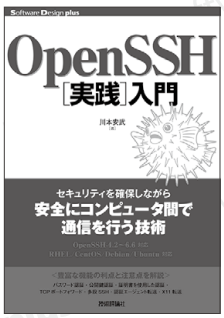


中島雅弘、富永浩之、國信真吾、花川直己 著
B5変形判・416ページ
定価 2,980円(本体)+税
ISBN 978-4-7741-7369-6

福田和宏、中村文則、竹本浩、木本裕紀 著
B5判・128ページ
定価 1,980円(本体)+税
ISBN 978-4-7741-7345-0



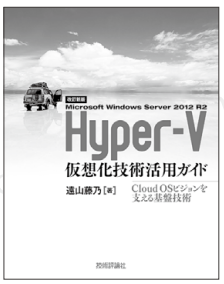
中村行宏、横田翔 著
A5判・320ページ
定価 2,480円(本体)+税
ISBN 978-4-7741-7114-2



川本安武 著
A5判・400ページ
定価 2,980円(本体)+税
ISBN 978-4-7741-6807-4

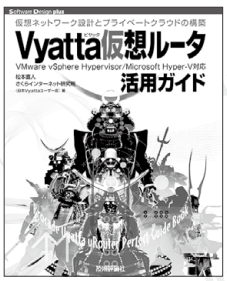
勝俣智成、佐伯昌樹、原田登志 著
A5判・288ページ
定価 3,300円(本体)+税
ISBN 978-4-7741-6709-1

倉田晃次、澤井健、幸坂大輔 著
B5変形判・520ページ
定価 3,700円(本体)+税
ISBN 978-4-7741-6984-2



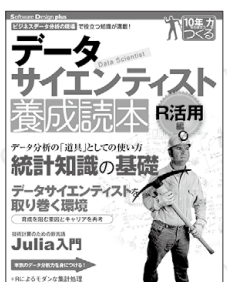
遠山勝乃 著
B5変形判・392ページ
定価 3,500円(本体)+税
ISBN 978-4-7741-6571-4

寺島広大 著
B5変形判・440ページ
定価 3,500円(本体)+税
ISBN 978-4-7741-6543-1



松本直人、さくらインターネット研究所（日本Vyattaユーザー会） 著
B5変形判・320ページ
定価 3,300円(本体)+税
ISBN 978-4-7741-6553-0

川瀬裕久、古川文生、松尾大、竹澤有貴、小山哲志、新原雅司 著
B5判・156ページ
定価 1,980円(本体)+税
ISBN 978-4-7741-7313-9



養成読本編集部 編
B5判・164ページ
定価 1,980円(本体)+税
ISBN 978-4-7741-7057-2

きしだなおき、のざきひろふみ、吉田真也、菊田洋一、渡辺修司、伊賀敏樹 著
B5判・168ページ
定価 1,980円(本体)+税
ISBN 978-4-7741-6931-6

吾郷協、山田順久、竹馬光太郎、和智大二郎 著
B5判・136ページ
定価 1,980円(本体)+税
ISBN 978-4-7741-6797-8

技術評論社　〒162-0846 東京都新宿区市谷左内町21-13
販売促進部　TEL 03-3513-6150 FAX 03-3513-6151

ひみつのLinux通信

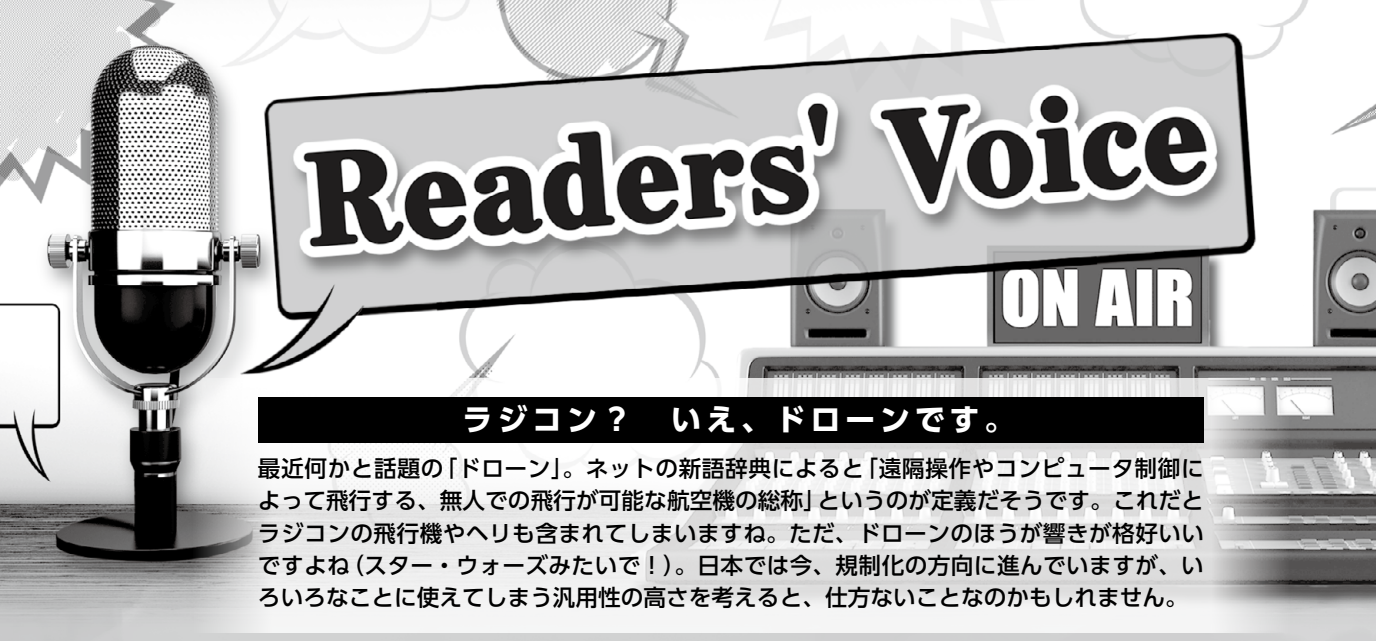
作)くつなりょうすけ
@ryosuke927

第19回 ライフログ

ライフログで自分の一日を振り返ると鬱になるから、怖くてそんなことできない担当編集にも愛のツイートを！

to be Continued

歩数や睡眠時間などが記録でき、スマホと連携するとメッセージやメールなども通知してくれる機能を持ったブレスレットを買ってみたら、これが楽しいの。歩くのは通勤程度のデスクワークな私、一応設定した目標に達すると幸せを感じるわけです。コレを始めてみたらログするのが楽しくなっちゃって、前からやってたランニングのGPSログに加えて、体重計の数値を記録するスマホアプリを導入したり、飲酒量とかも記録したりし始めました。振り返りって大切ですよ。「こんなに呑んでたんだ」って愕然とします。コマンド履歴もライフログみたいなものですが、コマンド実行回数より利用時間で統計を取りたいですね。そんなログツールありましたっけ？



ラジコン？　いえ、ドローンです。

最近何かと話題の「ドローン」。ネットの新語辞典によると「遠隔操作やコンピュータ制御によって飛行する、無人での飛行が可能な航空機の総称」というのが定義だそうです。これだとラジコンの飛行機やヘリも含まれてしまいますね。ただ、ドローンのほうが響きが格好いいですよね（スター・ウォーズみたいで！）。日本では今、規制化の方向に進んでいますが、いろいろなことに使えてしまう汎用性の高さを考えると、仕方ないことなのかもしれません。

2015年6月号について、たくさんの声が届きました。

第1特集 Git & GitHub入門

分散型バージョン管理システム「Git」、そのしくみを使ってコードやドキュメントを共有・公開できるWebサービス「GitHub」。特集ではその2つについて、新人向けに基礎の基礎から解説しました。第1章導入部分の、新入社員コントが印象的でしたね。

そろそろ新人が配属になるので、最初に読んでもらうのにちょうどいいです。
ganganさん／沖縄県

ちょうど知りたいと思っていたので、雑誌で特集してくれるのはありがたい。
野村さん／栃木県

「なぜGitが主流になったのか？」「なぜMercurialやBazaarよりGitなのか？」といった、Gitがほかより優れている点について説明があるともっと良かったと思います。　齋藤さん／神奈川県

個人で使い始めたところだったので非常にためになった。　杉岡さん／神奈川県

aicoさんのイラスト、相変わらずナイスです。　山下さん／東京都

GitHubについてあまり理解していなかったため使用していなかったが、便利そうなので導入を検討したいと思った。　marcosさん／愛知県

開発現場以外でのGitの活用法の紹介などがあるとおもしろいかなと思いました。　島さん／静岡県

Gitはローカルでチェックアウト、コミットができる点が魅力だと思う。Git以前のソース管理としてSubversionを使っている現場が多いと思うので、Gitへの移行時の注意事項や手順などを扱ってもらうと、現場に浸透しやすいと思う。　隼さん／岩手県

GitとGitHubの利用は企業・組織に限らず、地方自治体でも始まっています。最近の話ですと、和歌山県がGitHubのアカウントを取得してオープンデータの公開などに利用しているそうです。

第2特集 OpenLDAPの教科書

OpenLDAPは、ディレクトリアクセスプロトコル「LDAP」を実装するOSS。特集では、まずLDAPの基本事項について紹介し、OpenLDAPによるLDAPサーバの構築を解説、そしてCentOS、GitHubなどのクライアントでLDAPを構築する方法を説明しました。

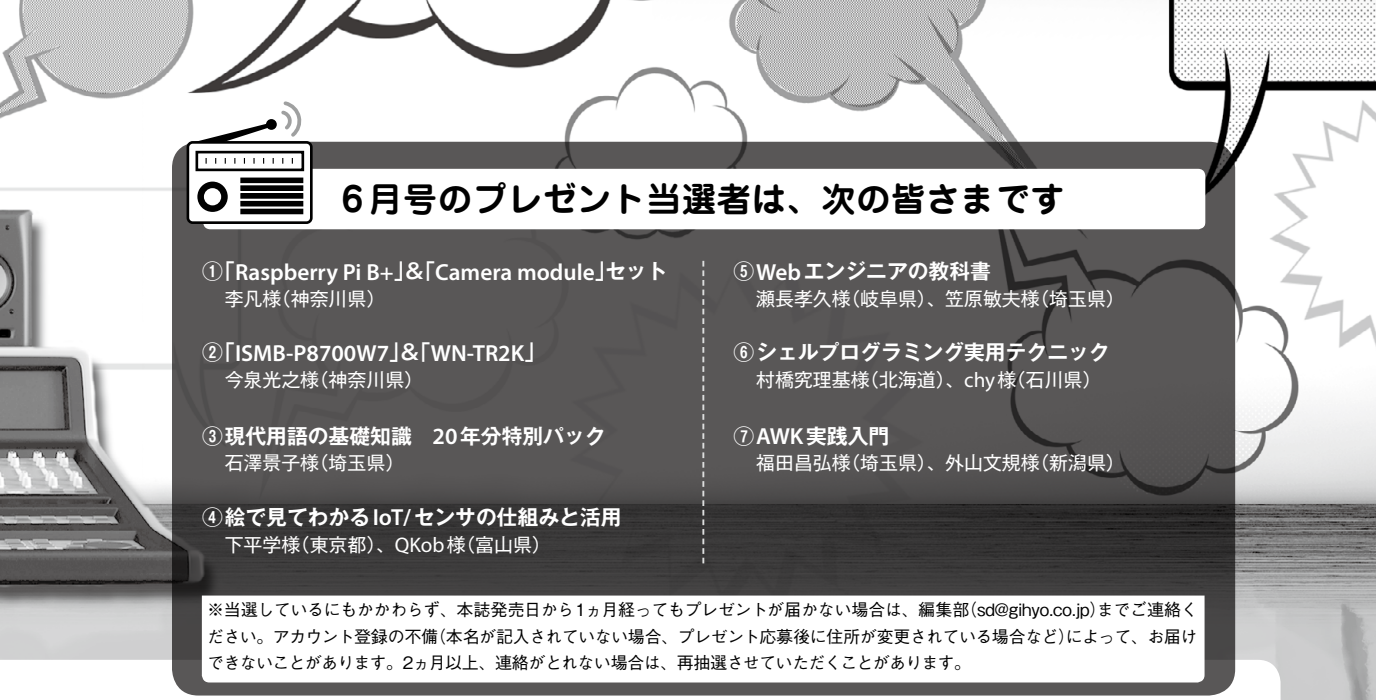
LDAPについて興味があったので購入しました。　菊地さん／愛知県

意外と使うので特集があって助かった。
片山さん／東京都

従来はしくみの説明に終始する記述が多かったが、とくにDIT設計の勘所を簡潔にまとめた個所が良かった。
若山さん／千葉県

OpenLDAPは、あまり情報がなく設定が呪文みたいでとっつきにくいのですが、今回の特集のように全体的に網羅されている記事は参考になります。呪文みたいな設定を理解したいので、できれば連載化・書籍化をお願いしたいです。　今井さん／千葉県

当たり前のように使われている技術だからこそ、その内部構成を理解しておくことはとても大切です。「とりあえず動けばいい」と普段からおまじない的に設定を書いていては、トラブル時に対応できなくなりますね。



6月号のプレゼント当選者は、次の皆さまです

①「Raspberry Pi B+」&「Camera module」セット
李凡様（神奈川県）

②「ISMB-P8700W7」&「WN-TR2K」
今泉光之様（神奈川県）

③現代用語の基礎知識　20年分特別パック
石澤景子様（埼玉県）

④絵で見てわかるIoT/センサの仕組みと活用
下平学様（東京都）、QKob様（富山県）

⑤Webエンジニアの教科書
瀬長孝久様（岐阜県）、笠原敏夫様（埼玉県）

⑥シェルプログラミング実用テクニック
村橋究理基様（北海道）、chy様（石川県）

⑦AWK実践入門
福田昌弘様（埼玉県）、外山文規様（新潟県）

※当選しているにもかかわらず、本誌発売日から1ヵ月経ってもプレゼントが届かない場合は、編集部（sd@gihyo.co.jp）までご連絡ください。アカウント登録の不備（本名が記入されていない場合、プレゼント応募後に住所が変更されている場合など）によって、お届けできないことがあります。2ヵ月以上、連絡がとれない場合は、再抽選させていただくことがあります。

緊急企画　SambaによるActive Directoryの機能性と移行性を検証する

Windows Server 2003のサポート終了を目前に控え、企画された記事。Windows Server 2003上に構築されたディレクトリサービスシステム「Active Directory」を、OSSであるSamba上へ移行する方法について、両者の機能比較も行いながら解説しました。

自分の働いているところでもWindows Server 2003がまだ動いているので、それの移行として考えてみることができた。今後予算が付かない場合、OSSでの構築も考えないといけないと思った。　**ももんがさん／静岡県**

SambaでADの代用が可能だということを初めて知りました。結構ハードルは高そうですね。　**NGC2068さん／愛知県**

SambaでのADが実用範囲内だと知り有意義です。　**森本さん／埼玉県**

テーマとしてこの量では少な過ぎたが、あまりこういった記事がないので良かったです。　**コメットさん／兵庫県**

まさにWindows Server 2003からの乗り換え中なので、とてもタイムリーなネタで助かります。　**くまーーーさん／神奈川県**

昔と比べると非常に使いやすくなっていると思え、また試してみたくなった。　**山添さん／東京都**

Windows Server 2003は、そのクライアントにあたるWindows XPとともに、あらゆる場所で使用されてきました。サポート終了を控えて、OSSへの乗り換えを検討しているユーザも多いのではないでしょうか。

短期連載　Kotlin入門【3】

Java仮想マシン上で動作するオブジェクト指向言語「Kotlin」についての短期連載。Androidアプリを作るのにも使いやすいプログラミング言語です。第3回ではKotlinの、言語としての文法・機能を1つずつ解説していきました。

高階関数が早くも出てきた、という感じがします。　**いつも計画倒れさん／奈良県**

いよいよ深いところまで踏み込んできた感じで、少し難しいけどおもしろかったです。　**オミオさん／宮城県**

言語の概要説明と環境構築を終え、いよいよコーディングの話題です。高階関数など出てきて、難易度がやや高くなってきましたね。

フリートーク

新種のウイルスは、出現してから対策ソフトのデータベースに反映されるまではどのように対処すれば良いのでしょうか。年金問題で自分の環境も見直したいです。　**澤下さん／大阪府**

年金関連で話題の、不審なメールや添付ファイルなどにどう対処するか……。特集記事を掲載してください。　**牧さん／大阪府**

日本年金機構の個人情報流失事件、何人かの方から心配の声が寄せられました。今回は人的な要因がかなり大きかったようですが、システムで防げる部分も多分にあったことと思います。

コメントを掲載させていただいた読者の方には、1,000円分のQUOカードをお送りしております。コメントは、本誌サイト**http://sd.gihyo.jp/**の「読者アンケートと資料請求」からアクセスできるアンケートにてご投稿ください。みなさまのご意見・ご感想をお待ちしています。

次号予告

Software Design
September 2015

2015年9月号
定価（本体1,220円+税）
192ページ

8月18日発売

［第1特集］エンジニアの夏期講習
特講「オブジェクト指向・SQL・正規表現」
苦手克服のベストプラクティス

［第2特集］しくみをご存じですか?
メールシステムの教科書
メール配信、メッセージ転送、MTAのしくみ、安全性など多面的にメールシステムを学ぶ

［特別企画］
なぜ俺の提案は通らないのか?
理想を実現するために必要なたった1つのこと

［好評!短期連載］
Jamesのセキュリティレッスン「新しいWireshark」

※特集・記事内容は、予告なく変更される場合があります。あらかじめご容赦ください。

お詫びと訂正

以下の記事に誤りがございました。読者のみなさま、および関係者の方々にご迷惑をおかけしたことをお詫び申し上げます。

■2015年7月号　連載「Linuxカーネル観光ガイド」第40回
- ●P.167　図4　SO_INCOMING_CPUの機能の2と3の順番が逆になります。
- ●P.168　写真1のリンク［URL］http://beagleboard.org/black
- ●P.168　写真2のリンク［URL］http://elinux.org/File:Bottom-LCD3.jpg

SD Staff Room

●ガチャピン先生に問われて連投ツイートした原稿執筆心得が結城浩先生の目にとまり、まとめサイトで公開されたら、フォロー数が一気に増えました。技術者としてまっとうに生き抜いていれば、誰にも負けない技術・知恵ができているはずです。それを原稿にして欲しいんです!　ただそれだけなのです。（本）

●今年も水耕栽培を始めました。ダシパックにバーミキュライトを入れ、それを大きめの器に入れて大塚ハウスEC2.6の液肥に浸し、ダイソーで買ってきた50円のレタスミックスの種を撒くだけの簡単なものです。2日目には発芽し、数週間で収穫できるくらい大きくなります。毎日成長を見るのも楽しみです。（撒）

●最近、「3行日記」なるものをつけてみてます（順天堂大医学部教授 小林弘幸先生提唱）。手書き推奨なのですが、私は仕事の終わりにテキストファイルにメモして帰宅することに。するとどうでしょう!ってほど劇的に何かが変わった感じはまだしませんが、「今日一日が終わった感」がいいです。（キ）

●この十何年間、コンピュータゲームをしていません。ふと「久々にやると新たな感動があるのでは」と思い、やってみることに。PS2のエミュレータとROMイメージを作成し、小一時間ほどかかってPCで動作させることに成功。何だかそれだけで心が満たされてしまい、結局ゲームはやらずじまい。（よし）

●カシオが昔出していた、多機能のデジタル腕時計が好きです。電卓がそのままくっついたような「データバンク・カリキュレータ」、大きなボタンが4つ付いた「フューチャリスト」などなど。僕の愛機「ワールドタイム」は、液晶に世界地図とアナログ風デジタル時計が組み込まれていてクレイジーです。（な）

●テレビの特集を見てから気になっていた江戸東京博物館に行ってきました。実物大の模型、当時の様子を再現した縮尺模型、実際に使っていた実物資料などの展示や、動く模型の演出と解説もあり、見ごたえがありました。分館で江戸東京たてもの園もあるそうなので、今度行ってみたいと思います。（ま）

ご案内
編集部へのニュースリリース、各種案内などがございましたら、下記宛までお送りくださいますようお願いいたします。

Software Design 編集部
ニュース担当係

［E-mail］
sd@gihyo.co.jp

［FAX］
03-3513-6173

※FAX番号は変更される可能性もありますので、ご確認のうえご利用ください。

Software Design
2015年8月号

発行日
2015年8月18日

●発行人
片岡　巌

●編集人
池本公平

●編集
金田富士男
菊池　猛
吉岡高弘
中田瑛人

●編集アシスタント
松本涼子

●広告
中島亮太
北川香織

●発行所
㈱技術評論社
編集部
TEL：03-3513-6170
販売促進部
TEL：03-3513-6150
広告企画部
TEL：03-3513-6165

●印刷
図書印刷㈱

この個所は、雑誌発行時には広告が掲載されていました。編集の都合上、
総集編では収録致しません。

この個所は、雑誌発行時には広告が掲載されていました。編集の都合上、
総集編では収録致しません。