秋葉原ラボ技術報告

volume 3

AKBL

アキハバラボ

Akihabara Lab

巻頭言「問われる基盤力」

秋葉原ラボ 研究室長 福田 一郎

技術報告　Vol.3

新型コロナウィルス感染症の影響下で外出自粛やリモート勤務，短縮営業，営業自粛などという言葉とともに，仕事と私生活の双方で今までに経験したことがない環境になっている．我々のようなインターネットサービス開発に関わる者がただちにできるのは，状況を改善するために可能な限りのリモート勤務の実施と私生活においても不要不急の外出を避けることくらいである．最前線のいわゆる「エッセンシャルワーカー」の方々には感謝の念しかない．

このような緊急事態で問われるのは基盤がいかに整備されているかということであろう．今回は医療・検査体制やリモート勤務が可能な環境に目が向けられている．日ごろからどこまで基盤を整備し，あらかじめ緊急事態への対応能力を確保しておくべきかは判断が難しい問題だ．

秋葉原ラボ技術報告の巻頭言なのでやや強引に関連付けるが，近年流行りの機械学習に関わるシステムにも基盤整備は欠かせないものとなっている．アルゴリズムに注目が集まりがちではあるが，システムの裏側にデータ収集・集計・分析基盤やモデル管理・サービング基盤などが整備されていることで，より高品質かつ迅速に機械学習をサービスに適用できる．

共通化や基盤化をどこまで進めるかは想定される効果をもとに考えれば良いと思われるところだが，秋葉原ラボが所属する当社のメディア事業では新しいサービスが次々と作られ，それらが対象とする領域も多岐にわたるので，効果の程度の予測も難しいのが実情だ．そのためにどのようなサービスであっても共通となるであろう要素を抽出し，できる限り抽象化したうえでシステムとして落とし込むことを心がけた開発を行っている．

Vol.3 となる今回の秋葉原ラボ技術報告では機械学習システムを支えるデータ基盤を中心に取り上げ，さらに画像処理技術のサービス適用を機械学習の応用事例として紹介する．

機械学習システム向けの基盤整備という課題に対する私達の取り組みをご覧いただくことで，この分野の技術発展に多少なりとも寄与することを期待している．特に基盤を作る際に抽象化レイヤを意識して開発しているところに注目して読んでいただけると幸いである．

目次

1

データ活用のための基盤システム

秋葉原ラボは大規模データ処理を支える基盤システムを開発しデータ活用を促進している．これらの基盤システムは当社のインターネットサービスの多様性や，コンテンツ推薦・検索などの様々なデータ用途をふまえ改善されてきた．本章では基盤システムの取り組みとして，システムの礎となるクラウドネイティブ環境と，ストリーム処理基盤について解説する．

Akihabara Lab

1.1 秋葉原ラボの研究開発とシステム運用を支えるクラウドネイティブ環境

岩城 洋一　佐藤 栄一　松原 周平

概　要　秋葉原ラボでは各サービスを支援するための基盤や機械学習システムの開発運用を行っている．当社のプライベートクラウド環境はインフラコストやハードウェア選定などの面でメリットがあり，このプライベートクラウド環境との親和性が高いクラウドネイティブ環境を提供することが重要となる．本稿ではクラウドネイティブ環境の構築と運用の取り組みについて紹介する．

Keywords　クラウドネイティブ，Kubernetes，OSS，OpenStack，プライベートクラウド

1　はじめに

秋葉原ラボでは ABEMA や Ameba ブログ，広告プロダクトなどから日々生成される大量のデータを活用したレコメンデーションシステム，サービスの有害なコンテンツを効率良く検出するためのフィルタリングシステムなどを開発し，サービス改善に貢献している．

これらのシステムのベースとなる機械学習やデータ分析の分野は進歩が著しく，オープンソースソフトウェア（OSS）やクラウドサービスが日々登場する．このためこのような日々新たに登場するものや手法を効率的に取り込みながら研究開発およびシステム運用の改善を進めていくことが重要となる．

開発および運用の改善，効率化などについては近年 Docker や Kubernetes などによるクラウドネイティブ技術が注目されており，秋葉原ラボでもこれらの技術を活用することにより本問題の解消を進めている．

秋葉原ラボのクラウドネイティブ環境を実現するにあたって重要な要件の一つがプライベートクラウドが提供する各種インフラリソースと親和性の高い環境の構築と運用を行うことである．これはクラウドネイティブ環境を Amazon Web Services（AWS）や Google Cloud Platform（GCP）などのパブリッククラウドのみで運用する場合，当社が運用する社内向けプライベートクラウドの恩恵を受けられないためである．社内向けプライベートクラウドには，(i) インフラコストが低い，(ii) ハードウェアの選定などの自由度が高い，(iii) 既存のデータが集約されておりデータ活用との相性が良い，といった特徴がある．

本稿では，秋葉原ラボが運用するクラウドネイティブ環境である秋葉原ラボ向け Kubernetes 環境の概要，プライベートクラウド上のインフラリソースとの連携，Operator やアドオンなどの活用事例，ワークロード最適化の取り組みについて紹介する．

2　関連用語

本章では Kubernetes に関する用語が多く登場するため，本節で主な用語について説明する．

- kubelet — kubelet は各ノードで実行される

Kubernetes の主要コンポーネントの一つである．Kubernetes API を提供するコンポーネントである api-server と連携し，PodSpec で宣言されたコンテナの実行に対して責務を負う．また，Kubernetes クラスタへの Node の登録，Node や Pod の状態の api-server への通知も行う．

- Cloud Provider — Cloud Provider[*1]は，Kubernetes がロードバランサなどの外部のインフラと連携するためのしくみの総称．Cloud Provider では，主に Cloud Controller Manager と呼ばれるコンポーネントによって，Kubernetes 外のインフラに対して設定の同期などを行う．AWS 用のもの，GCP 用のもの，OpenStack 用のものなど，動作環境に合わせてさまざまな実装が提供されている．
- CNI — Container Network Interface[*2]の略称．CNCF（Cloud Native Computing Foundation)[*3]のプロジェクトで，コンテナに対して任意のネットワークを公開するための標準インタフェースである．Kubernetes ではネットワークプラグインとして CNI プラグインを利用できる．
- CSI — Container Storage Interface[*4]の略称．コンテナに対して任意のストレージを公開するための標準インタフェースである．CSI を実装した Kubernetes の CSI プラグインを導入することにより，ストレージを Kubernetes の PV/PVC リソースとして扱うことができる．多くの主要なストレージシステムの CSI プラグインが Kubernetes チームやサードパーティベンダによって提供されている．
- コントローラ — Kubernetes リソースを監視し，期待する状態と実際の状態を一致するように処理を行うコンポーネント．この処理ループは Reconcile ループと呼ばれる．
- CRD — Custom Resource Definition の略称．ユーザが定義可能な Kubernetes リソースの定義を指す．たとえば Deployment や CronJob のような Kubernetes リソースを独自に定義できる．
- Operator — Operator[*5]は Kubernetes の CRD とコントローラによって構成される「アプリケーション指向のコントローラ」である．Operator を使用することにより，Prometheus や MySQL などを CRD として宣言的に記述して登録し，複雑なコンフィグ，デプロイ，クラスタなどの管理運用を Operator 側で自動的に行わせることができる．
- Pod — Kubernetes リソースの一つ．Kubernetes でデプロイできる最小かつ最も基本的なオブジェクト．Docker コンテナなどのコンテナが 1 つ以上含まれており，Pod 内のコンテナはストレージやネットワークなどを共有する．
- PV — PersistentVolume の略称．Kubernetes リソースの一つ．ストレージクラスによってプロビジョニングされるクラスタのストレージリソース．
- PVC — PersistentVolumeClaim の略称．Kubernetes リソースの一つ．ユーザによって要求されるストレージリソース．新しく PVC を作成すると，その設定内容を満たす PV が確保されて PVC と紐づけられる．

3 概要と構成

本節では，秋葉原ラボ向け Kubernetes 環境（以降は lab-k8s と呼称）の全体の方針と構成を簡単に紹介する．

[*1] https://kubernetes.io/docs/concepts/architecture/cloud-controller/
[*2] https://github.com/containernetworking/cni
[*3] https://www.cncf.io/
[*4] https://github.com/container-storage-interface/spec/blob/master/spec.md
[*5] https://coreos.com/blog/introducing-operators.html

3.1 要件・設計方針

秋葉原ラボで開発・運用されるアプリケーションは多岐にわたるため，さまざまなワークロードへの対応が必要となる．構築時には下記の内容を考慮した．

- ステートフルなワークロードを扱えること．データベースなど，秋葉原ラボで扱うワークロードには永続的なストレージを必要とするものも多いため，そのデータ移行を考慮した場合，Kubernetes クラスタ間でワークロードを移行することは難しい．バージョンアップは，新しいバージョンの Kubernetes クラスタを作りワークロードを移行する方法ではなく，運用するクラスタを継続的にアップデートしていく方法を前提とする．
- バッチワークロードへの対応．API サーバやデータベースなどの長時間起動しているワークロードに限らず，CI のジョブや定期バッチなどで起動・終了を繰り返し，瞬間的なリソース消費が激しいワークロードにも対応する必要がある．
- ネットワーク処理のオーバーヘッドが少ないこと．低レイテンシ・高トラフィックを要求するワークロードも多いため，オーバーレイネットワークおよびネットワークアドレス変換（NAT）は原則使用しない．これは，トラブルシューティングの複雑化を避ける目的を兼ねている．
- マルチデータセンター構成に対応できること．当社には複数のデータセンターにそれぞれ OpenStack クラスタが存在するため，それらを透過的に使用できることが望ましい．また，データセンター間でのワークロードやストレージのポータビリティを実現できることが望ましい．
- マルチテナンシーの実現．秋葉原ラボには複数のチームが存在するため，権限や計算リソースを適切な粒度で分離し，利用者間・ワークロード間でお互いに影響を及ぼさないようにコントロールする必要がある．

3.2 コンポーネント構成

ここでは，それぞれのコンポーネントについて簡単に説明する．独自に実装しているコンポーネントには（*）を付けている．詳細については後の節を参照されたい．図 1.1.1 のように，Kubernetes 本体といえる部分は少なく，多くのコンポーネントが協調して動作している．

ネットワーク・DNS

- **OpenStack Port CNI(*)**: Pod に OpenStack の Port を直接割り当てる構成の CNI プラグイン．
- **lab-k8s-cloud-provider(*)**: 主に，Kubernetes の Service リソースから自動で外部のロードバランサの設定を行うためのコンポーネント．後述の ExternalDNS によって，作成されたロードバランサにはクラスタ外からでも参照可能な DNS レコードが自動設定される．
- **CoreDNS, nodelocaldns**: クラスタ内の DNS を提供するアドオン．CoreDNS への負荷を軽減するために各 Node にキャッシュを配備している．
- **ExternalDNS**: `type: LoadBalancer` などの Service を lab-k8s 外から参照できるようにするために，Amazon Route 53 に DNS レコードを登録している．

ストレージ

- **Cinder CSI Plugin**: OpenStack の Cinder ストレージを利用するための CSI プラグイン．
- **PV GC Controller(*)**: 解放された PV を一定期間後に自動削除するためのコントローラ．

ログ・モニタリング

- **Fluent Bit**: 各 Node で動作して，コンテナのログを Stackdriver Logging に転送している．
- **Prometheus Operator, Prometheus, Alertmanager**: 監視には主に Prometheus

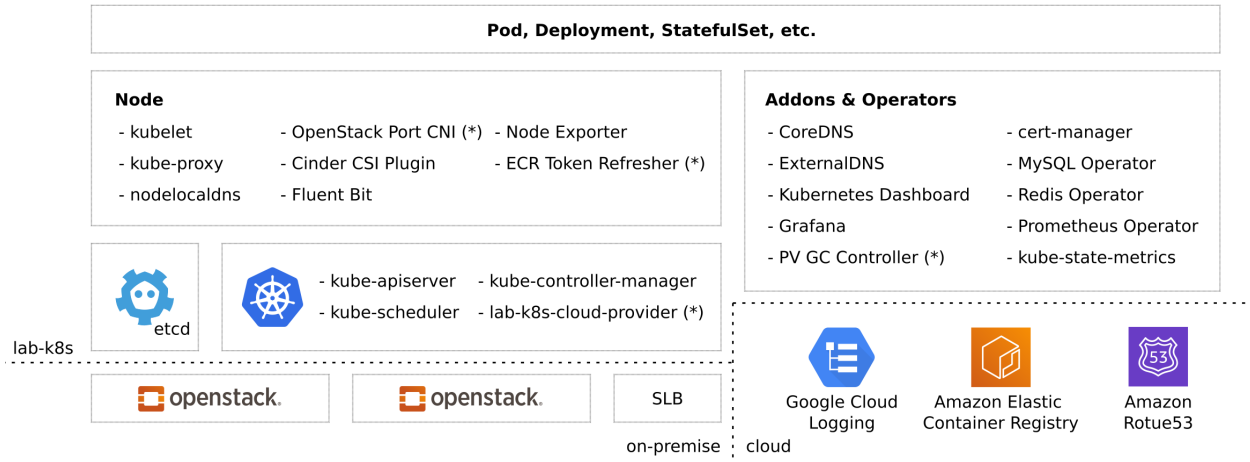


図 1.1.1　コンポーネント構成

を利用している．

- **Node Exporter, kube-state-metrics, etc.**: Kubernetes や Node の状態の監視のためのメトリクスを公開するコンポーネント．これらのメトリクスは Prometheus によって収集される．
- **Grafana**: Prometheus で収集したシステムのメトリクスなどの可視化を行う．
- **Kubernetes Dashboard**: Kubernetes の状態を確認するためのダッシュボード．

その他

- **Redis Operator, MySQL Operator**: 利用者が簡単に MySQL や Redis Sentinel を構築・運用するための Operator を導入している．
- **cert-manager**: 証明書の自動発行・更新を行うためのコンポーネント．一例として，発行した証明書は lab-k8s-cloud-provider によってロードバランサに自動設定できる．
- **ECR Token Refresher(*)**: コンテナレジストリとして利用している Amazon Elastic Container Registry（ECR）の imagePullSecret を自動更新するための DaemonSet．

4　インフラとの連携

当社ではインフラチームが複数のオンプレミスのデータセンターに世代の異なる OpenStack ベースのプライベートクラウドを構築・運用している．lab-k8s はそれらのプライベートクラウドを横断する形でクラスタを構築しており，Kubernetes 上からは Node ラベルを使用することでワークロードを実行するデータセンターを選択できる．明示的な affinity の指定がない場合，ワークロードは任意のデータセンターで実行される．また，インフラチームによって提供される SLB と呼ばれるソフトウェアロードバランサを Kubernetes の External Load Balancer として利用している． 本節ではノードの構築や設定，Cloud Providor，CSI や CNI などのプラグインによるインフラとの連携について記載する．

4.1　ノードの作成と準備

Kubernetes にはクラスタ内のオブジェクトの状態を制御するコアコンポーネントが稼働する Control Plane ノード，コンテナ化されたアプリケーションが実行される Worker ノードおよびメタデータを保存する etcd が必要になる．Kubernetes クラスタのブートストラッピングには kubeadm を利用している．etcd は Kubernetes の Control Plane ノードとは異なるインスタンス VM で実行する HA 構成を採っている．Worker

ノードは Terraform からインスタンス VM の起動を行うと自動的にクラスタに参画するように構成しており，新規ノードの追加を容易にしている．

etcd，Control Plane ノード，Worker ノード，ノードで実行される Pod など，それぞれの間の通信は OpenStack の SecurityGroup を利用して制御している．

4.2 クラスタ設定

秋葉原ラボ内の多様なアプリケーションやバッチ処理を実行する上で，リソースの分離や可用性の向上のための設定を行っている．

権限管理

RBAC と PodSecurityPolicy による権限管理を行う．lab-k8s は複数のチームが単一の Kubernetes クラスタを共有するマルチテナント環境であるため，利用者が Pod を実行する場合には原則，特権コンテナや root 昇格，root ユーザでの実行，ホストパスボリュームのマウント，ホストネットワークの使用など多くの権限を制限した PodSecurityPolicy のみ使用できる設定となっている．アプリケーションを開発・運用するチームの単位で Namespace を分割し，Namespace 内の管理者権限を割り当てている．

クラスタ内のサービスルーティング

クラスタ内のサービスルーティングには Service 数と Pod 数に比例し負荷が高くなる可能性がある iptables プロキシではなく IPVS プロキシを利用している．IPVS プロキシによりスケーラビリティの向上や柔軟なロードバランシングが可能になる．

リソース管理

リソースの競合を回避するために以下の管理を行っている．

- ユーザが利用する Namespace には ResourceQuota の設定を行っている．これによって，誤ったマニフェストによる意図しない過度なリソースの消費を避けられる．
- kube-reserved や system-reserved を設定することでユーザの実行するワークロードと kubelet やコンテナランタイムなどの Kubernetes のシステムコンポーネントとの間でリソースの競合が起きないよう，安定稼働に必要なリソースをあらかじめ確保している．
- cpu-management-policy を static に設定し，特定のアプリケーションが CPU を排他的に利用できるようにしている．コンテキストスイッチや CPU キャッシュミスの影響を大きく受けるワークロードのパフォーマンスの向上が期待できる．

4.3 Cloud Provider の実装

Cloud Provider は，Kubernetes が環境ごとに異なるロードバランサなどの外部のインフラと連携するためのしくみである．たとえば，`type: LoadBalancer` の Service を作成した際に，実際のロードバランサの作成・更新を行ったり，OpenStack などの IaaS から，Node に関する情報（状態，インスタンスタイプ, etc.）を取得して Kubernetes で利用できるようにする． Cloud Provider を作成するためのフレームワークが提供されているため，Go 言語のインタフェース `cloudprovider.Interface`[*6] を実装することで，独自の Cloud Provider を作成できる．

lab-k8s では，インフラチームによって提供されているロードバランサや複数の OpenStack クラスタとの連携のため，独自に Cloud Provider を実装して利用している．

ロードバランサとの連携

ロードバランサとの連携は，Service/Pod の増減や状態の変化を監視（Watch）して，ロードバランサの設定を適切な状態に同期することで実現している．SLB では，ノードの追加などの各種操作

[*6] `https://godoc.org/k8s.io/cloud-provider#Interface`

を API 経由で行うことができるため，これを利用している．

多くの type: LoadBalancer の実装では，まずは Kubernetes の Node にトラフィックを分散させてからそこからさらに Pod にトラフィックを振り分ける構成を採るが，この方式では経由する Node が多いためオーバーヘッドなどが大きくなってしまう（図 1.1.2）．lab-k8s では，SLB を含む社内のインフラから直接ルーティング可能な IP アドレスを Pod に付与しているため（詳細は CNI の項を参照），ロードバランサから直接トラフィックを Pod に届けるコンテナネイティブロードバランシングと呼ばれる構成を採用している．

OpenStack との連携

Kubernetes は，Cloud Provider のしくみにより，Node の情報（IP アドレス，インスタンスタイプ）や Node の存在性，起動状態などを，OpenStack などの IaaS から取得して利用する．lab-k8s は OpenStack の VM インスタンス上で稼働しているため，OpenStack API からこれらの情報を取得する必要がある．すでに OpenStack Cloud Provider[*7]が存在するが，lab-k8s では以下の機能などを実現するために Cloud Provider を独自に実装している．

- 複数の OpenStack クラスタに対応する必要があるため，Node の FQDN を基に，Node の情報を問い合わせるべき OpenStack API のエンドポイントを決める．
- OpenStack のインスタンス VM に複数の Port が付与される（詳細は CNI の節を参照）ため，Node 自体に設定されている IP アドレスがどれなのか OpenStack API からは判別できない．このため，プロビジョニング時に設定されたインスタンス VM のメタデータから Kubernetes の Node の IP を参照する．

4.4 ネットワークとストレージの連携

OpenStack が提供するネットワークとストレージのリソースを Kubernetes で利用するため，Kubernetes の CNI と CSI を経由して各種リソースとの連携を行っている．それぞれの概要と，より高可用性を実現するための取り組みや工夫について紹介する．

CNI

Kubernetes では既存の CNI プラグインがいくつか用意されており，大きくオーバーレイネットワーク方式とアンダーレイネットワーク方式のものに分類されるが，それぞれ以下の欠点，制約がある．

- Flannel などのオーバーレイネットワーク方式の CNI プラグインの場合，ネットワーク処理のオーバーヘッドが発生したり，トラブルシューティングが複雑化する．
- Calico などのアンダーレイネットワーク方式の場合，L3 ネットワーク間の通信を行うために BGP などのプロトコルが必要となったり，ルータ側の設定が必要になる．

このような問題を回避するため，lab-k8s では「各 Pod に対して OpenStack Port をアタッチ／デタッチする方式」の独自の CNI プラグインを開発・運用している．また OpenStack が提供する Port を使用することにより，OpenStack の SecurityGroup によるネットワークセキュリティ制御を利用可能になるといった利点もある．

OpenStack Port CNI は CNI プラグイン，CNI エージェント，CNI サーバの 3 つにより構成される．（図 1.1.3）

- CNI プラグイン: kubelet から実行される OpenStack Port CNI のエントリポイント．OpenStack Port を確保するための CNI プラグイン，および適切な iptables/IPVS を設定するための CNI プラグインが順番に実行される．

[*7] https://github.com/kubernetes/cloud-provider-openstack

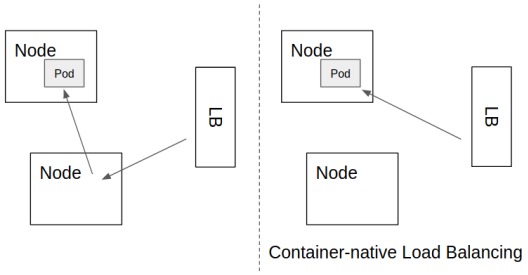


図 1.1.2 コンテナネイティブロードバランシング

- CNI エージェント: 各 Kubernetes ノードに配置される DaemonSet のエージェント．ノード上の OpenStack Port の管理を行う．また後述する Port のプーリングにより OpenStack への負荷を軽減する機能を備えている．
- CNI サーバ: OpenStack Port の作成，削除，アタッチ，デタッチを管理する．CNI サーバは OpenStack 環境ごとに配備される．

OpenStack Port CNI を運用するにあたり，OpenStack の負荷や障害などの予兆も含めて検出するために，CNI エージェントおよび CNI サーバでは複数のメトリクスを公開している．たとえば CNI 自体のメトリクス，OpenStack API に関連するメトリクスなどが含まれる．これを監視することにより障害時の問題の切り分けを行うことができる．

OpenStack の不具合などにより OpenStack Port が適切にデタッチ，削除されずにリークした状態となる特殊ケースがあるため，CNI エージェントではこれらのリークを検出して CNI サーバにリーク情報を通知するしくみも導入している．

以下に Pod 作成時の OpenStack Port CNI を含めた処理の流れを示す．

1. Kubernetes が Pod 作成リクエストを受信し，特定の Node に Pod をスケジューリングする．
2. スケジュールされた Node 上の kubelet が Pod に必要な IP を確保するために，OpenStack Port CNI の CNI プラグインを実行する．
3. CNI プラグインが実行されると，前述した CNI エージェントと CNI サーバが協調して OpenStack から OpenStack Port を確保し，必要な IP アドレス/iptables/IPVS 等の設定を行う．これを Pod で使用可能な IP アドレスなどの情報としてを kubelet に応答を返す．
4. kubelet は CNI プラグインの応答として受け取った IP アドレスなどの情報を利用して Pod を作成する．

CSI

lab-k8s ではプライベートクラウドの OpenStack の Cinder Volume を利用するために，OpenStack Cloud Provider の中で提供される Cinder CSI プラグインを導入している．しかし lab-k8s に導入した 2019 年当初は OpenStack に関連する不具合などにただちに対応する必要があったため，Cinder CSI プラグインのソースコードに独自の修正を加えている．その後は lab-k8s において StatefulSet の可用性を高めたり，OpenStack の UI 上から lab-k8s の PVC リソース情報との対応付けによって可視性を高めるために機能追加や改善も行っている．一部の機能追加を実現するために Cinder CSI プラグインに加えて，Kubernetes の storage-sig が提供する external-provisioner も独自に拡張を加えている．

- 作成した Cinder Volume に PVC や Namespace などのメタ情報を付与することにより，OpenStack と Kubernetes の対応関係を明確にする．
- 同一インスタンス VM 上の並列アタッチ/デタッチの排他制御を実装することにより，

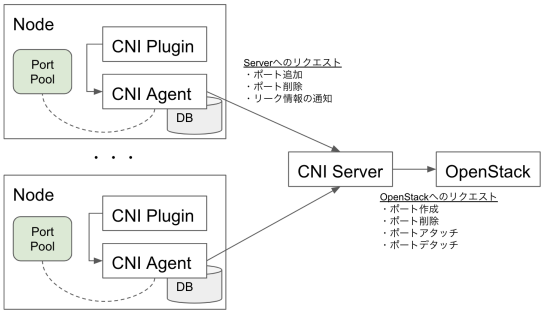


図 1.1.3 OpenStack Port CNI の構成

OpenStack で不整合状態が発生するバグを回避する.

- Kubernetes の PVC 作成時に Cinder Volume の Availability Zone（以降は AZ と呼称）を自動分散することにより，StatefulSet などにおける高可用性を実現する.

この独自拡張した Cinder CSI を使用する場合は，PVC を作成する際に適切なストレージクラスおよび設定を指定する．以下にストレージクラスおよび設定の特徴を示す.

- Cinder Volume のタイプを指定可能．SSD, HDD などを指定する.
- Cinder Volume の AZ を指定可能．各 AZ を指定するか，自動分散を指定する.
- reclaimPolicy は Retain/Delete を指定可能．Retain の場合は Cinder Volume の削除は後述する PV のガベージコレクトによって一定期間を経てから削除されるため，誤って削除した場合でもデータを復旧することが可能.

5 Operator, アドオンの運用

Kubernetes 上にデータベース，キャッシュレイヤなどのミドルウェアやアプリケーションを導入することは比較的容易である．しかし HA 構成としたり各種メンテナンスを行ったりするところまで考慮した場合，対象となるミドルウェア／アプリケーションに加えて Kubernetes のアーキテクチャ，ツールなどの各種知識とノウハウが必要となるため導入するための敷居が高くなる．このため本節で紹介する Operator, アドオンを導入して運用を行っている.

5.1 Operator

本項では lab-k8s で使用する代表的な Operator を紹介する.

MySQL Operator

MySQL Operator[*8]は MySQL の HA クラスタ環境を提供する Operator である．MySQL Operator ではクラスタの管理および運用のために，MySQL Orchestator および Percona Server を使用しているため，MySQL クラスタの管理，運用，監視においてこれらのツールや UI などを利用できる.

lab-k8s では MySQL Operator の可用性などを高めて運用するために，以下の独自の拡張および修正を加えている.

- バックアップ，サイドカーなどの各種コンテナの resources や securityContext を CRD から指定できるように拡張.
- いくつかの Kubernetes 関連バグを回避するための修正.
- バックアップ関連のバグを修正.
- 各クラスタノードのボリュームの AZ が分散されるように対応（Cinder CSI プラグイン側の

[*8] https://github.com/presslabs/mysql-operator

拡張により対応).

上記の対応により，リソース面，セキュリティ面，可用性面の課題を解決した MySQL クラスタの構築および運用を実現している．なお MySQL クラスタの可用性を高めるため，podAntiAffinity により MySQL クラスタの Pod を同じホストに配置しないように設定している．また前述した Cinder CSI プラグインの AZ 自動分散を利用することによりストレージ面での可用性を高めている．

Redis Operator

Redis Operator[*9]は Redis Sentinel[*10]を利用することで Redis の HA 環境を提供する Operator である．

Redis Sentinel による HA 構成を実現可能するためのソリューションは Redis Operator 以外にもある．たとえば Kubernetes のアプリケーション管理ツールである Helm には Redis Sentinel ベースの Helm パッケージがいくつか存在する．しかしこれらの Helm パッケージは一般的に以下の課題と制約がある．

- ノードダウンなどにより Pod の IP アドレスが変わった場合，Redis Sentinel では IP アドレスの変更を把握できない．このためフェイルオーバーを繰り返すことにより見かけ上はダウン状態の Pod が増え続けることになり，最終的にフェイルオーバーができなくなる．
- 上記の問題を解消するために各 Pod にサイドカーコンテナを配置して定期的に各 Sentinel の状態を管理および制御することはできる．しかしこの場合は各 Pod の中ですべての Sentinel の状態監視，更新などの処理を実装する必要があるためしくみが複雑となる．

Redis Operator で Redis Sentinel による HA 環境を構築した場合，上述した Pod の IP アドレス変更などを検知したうえで Redis の HA 構成を適切に管理運用できる．なお Redis を構成する Pod の可用性を高めるため，MySQL Operator と同様の podAntiAffinity 設定を行っている．

Prometheus Operator

Prometheus Operator[*11]は「Prometheus インスタンスのデプロイ」と「Prometheus で行うモニタリングの設定」とを分離するというアイデアに基づき設計され，デプロイやライフサイクル管理を意識せずにモニタリングの設定を可能にする Operator である．

たとえば Kubernetes 自体のモニタリングでは多くのコンポーネント（kubelet，cAdvisor，apiserver，controller-manager，scheduler など）のメトリクスをスクレイピングする必要がある．しかしメトリクスを収集する対象が多くなると Prometheus の設定が複雑になり，監視対象やアラートルールを追加, 変更する際のコストが上がってしまう．Prometheus Operator が提供する ServiceMonitor や PodMonitor などの CRD により，スクレイピング対象が多い場合でも監視対象やアラートルールの追加，変更が容易になる．

5.2 メンテナンス管理

Kubernetes 環境上でのアプリケーションなどを運用する場合，その利用者がメンテナンスなどを手動かつ定期的に行うことは極力排除することが望ましい．本項ではメンテナンス管理に関連する取り組みの一例を紹介する．

コンテナレジストリ認証トークンの自動更新

lab-k8s では共通コンテナレジストリとして ECR を使用している．しかしオンプレミス上の lab-k8s から ECR の Docker イメージを Pull する場合，それぞれの利用者が認証トークンを imagePullSecret として指定する必要がある．さらに

[*9] https://github.com/spotahome/redis-operator
[*10] https://redis.io/topics/sentinel
[*11] https://github.com/coreos/prometheus-operator

ECR の認証トークンには期限が設定されているため，利用者は認証トークンの指定および自動更新を各自で導入して運用する必要がある．

各自でそのようなしくみを導入する煩雑さを解消するために，ECR の認証トークンを自動的に設定および定期更新するためのエージェントを各ノードに DaemonSet として配置している．Kubernetes ではイメージの Pull は各ノード上の kubelet が担当しているため，エージェントでは kubelet が参照する認証トークンを定期的に更新している．以下にエージェントの処理内容を示す．

1. AWS から新しい ECR 認証トークンを取得
2. 取得した ECR 認証トークンをもとに kubelet 向けコンフィグファイルを作成
3. 生成した kubelet 向けコンフィグファイルを kubelet に反映

PV のガベージコレクト

Cinder CSI 経由で OpenStack の Cinder Volume を作成した場合，Kubernetes の PVC リソースを削除した時点でこれに対応するボリュームである Cinder Volume は Kubernetes からは利用できなくなる．しかし PVC リソースの reclaimPolicy が Delete 以外となっている場合，OpenStack では Cinder Volume が削除されず残ったままとなり，不要なストレージリソースのコストがかかる問題がある．

このような問題に対処するために，PV ガベージコレクト機能を controller-runtime[*12]をベースに開発して導入している．

1. Release 済みの PV リソースがある場合，該当 PV リソースに Release 日時をアノテーションとして設定する．
2. Release 済みの PV リソースにすでに Release 日時がある場合，Release 日時から一定期間経過したかをチェックしてガベージかを判断する．
3. ガベージと判定された PV リソースの reclaimPolicy を Delete に更新することで OpenStack 側の Cinder Volume を削除する．

6　ワークロードに対する最適化

秋葉原ラボのアプリケーションは機械学習などにより生成されたモデルなどを活用するものがあるため，ETL のようなワークフロー／バッチ関連のワークロードを大量かつ並列に動かすことが要求される．ここでは最適化に関する取り組みの一部について紹介する．

6.1　OpenStack Port CNI の Port プーリング

前述した OpenStack Port CNI は Pod 単位に OpenStack Port を作成してノードにアタッチするため，バッチなどの大量かつ並列に Pod の作成および削除が発生するようなワークロードを運用した場合は OpenStack 側に大きな負荷を与えることになり，Pod のネットワーク作成に非常に時間がかかったり失敗したりする問題を引き起こすことがある．特に秋葉原ラボでは機械学習などに関連する定期処理バッチが運用されているため，これらのバッチが増えるにつれて OpenStack API の遅延やエラーが増加していった．

この問題に対応するため，前述した CNI エージェントに対して OpenStack Port をプーリングするしくみを導入している．具体的には CNI エージェントがインスタンス VM 上の OpenStack Port をプーリングすることにより OpenStack Port を再利用できるように改良している．また Kubernetes では各ノードのリソース利用状況によって特定のノードに対して Pod のスケジューリングが集中する場合があるため，これを考慮して OpenStack Port のプーリング数に下限と上限を設けている．下限を超えて確保された OpenStack Port は一定期間未使用の状態が続くと自動的に削除される．

これによりバッチ処理のような大量かつ並列に

[*12] https://github.com/kubernetes-sigs/controller-runtime

Pod の作成および削除が発生するようなワークロードの安定運用を実現している．

7 まとめ

本稿では，秋葉原ラボにおけるクラウドネイティブ環境である lab-k8s におけるプライベートクラウドのインフラリソースとの連携，および改善のための各種取り組みについて紹介した．lab-k8s では Kubernetes エコシステムの各種ツールや Operator などの導入に加えて，独自のアドオンを開発したり OSS に対して独自の拡張や変更を行ったりすることによって，より効率的で信頼性の高い環境を提供している．これらの独自の拡張や変更については，各 OSS プロジェクトにフィードバックする予定である．

現在秋葉原ラボの主なサービスは Docker コンテナ化および Kubernetes 対応が完了しており，lab-k8s の本番環境に移設されて運用されている．今後は機械学習などの各種実験のためのアドホックなワークロードや CI/CD などの用途での利用を促進することで研究開発およびシステム運用のさらなる改善を目指していく．また，GPU などの専用ハードウェアを搭載したマシンをクラスタに組み込むことにより，機械学習など大きな計算リソースを必要とするワークロードの高速化および効率化を行う予定である．

Akihabara Lab

1.2 スキーマ定義に基づき key-value ストアにアクセスする API サーバの構築と運用

渡邉 敬之

概　要　システムの整理統合をするために，宣言的にスキーマを定義することで key-value ストア上のデータをテーブルとしてアクセスする API サーバ（開発コード:Zumwalt）について紹介する．key-value ストアを使う場合，データを key-value の形式に変換する．この変換処理はアプリケーションによって異なり，アクセスパターンの変更には変換処理自体の修正が必要となる．そのため技術的負債や属人化の原因となる．紹介する API サーバはスキーマ定義に基づき変換を自動化することで key-value ストアの知識がない開発者でも key-value ストアを利用したアプリケーション開発・保守が可能となる．本章では，key-value ストアを用いたアプリケーション開発の課題と秋葉原ラボで開発運用しているスキーマ定義に基づき key-value ストアにアクセスする API サーバの構築と運用について述べる．

Keywords　key-value ストア，HBase，Kubernetes

1　はじめに

当社のメディア事業では ABEMA や Ameba ブログなどのサービスを提供しており，ユーザの行動ログなどさまざまなデータが大量に発生している．このデータをうまく活用することがサービスの改善を行う上で重要になる．大量のデータを効率的に処理するためには key-value ストアの利用が有用である．key-value ストアは Key と Value のみで構成されるシンプルなデータ構造をもち，Key を指定するとそれに紐づいた Value に対して操作できる機能を提供する．データ構造がシンプルなため，大量のデータを低コストで容易に分散管理でき，かつ高速にデータの読み書きができる．そのため従来の関係データベースに比べて高いスループットを出すことができる．

一方で，key-value ストアを利用したアプリケーション開発は関係モデルで表現されるアプリケーションデータと key-value の変換処理が必要になるため，関係データベースのみを用いたアプリケーションに比べて複雑になる．また，アプリケーションデータをどのように key-value として変換するかはアプリケーション全体の性能に影響を与える．そのため，アプリケーションのアクセスパターンに基づいて適切に key-value を設計する必要がある．

各アプリケーションで適切に key-value の設計をするということは，アプリケーションごとに異なった変換処理を実装しなければいけないということである．また，アプリケーションの仕様の変更でアクセスパターンが変わってしまった場合には，key-value の設計も変更する必要があり，その結果，変換処理の実装まで書き換える必要がある．これらのことから key-value ストアを用いたアプリケーションは実装や保守にかかるコストが大きいという問題がある．

本章では，スキーマ定義に基づき変換処理を自動化する技術 [1] についての概要を説明し，その技術を組み込んだ API サーバの構築と運用に関して記述する．2 節では，key-value ストアのスキーマ

設計について述べる．3 節では，スキーマ定義に基づく API サーバの実装について述べる．4 節では，スキーマ定義に基づく API サーバの構築と運用について述べる．5 節では，まとめと今後の取り組みについて述べる．

2 key-value ストアのスキーマ設計

key-value ストアを利用したアプリケーションでは，関係モデルで表現されるアプリケーションデータを key-value として変換する必要がある．その際，どのようにアプリケーションデータを key-value へと変換するかは重要である．アクセスパターンにあった設計をできれば，効率的に key-value を読み書きできるからである．本稿ではこの変換方法の定義をスキーマと呼ぶこととする．

本節では，key-value ストアの一種であり秋葉原ラボにおいても利用されている Apache HBase（以下，HBase と呼ぶ）を例として説明し，スキーマ設計がアプリケーション性能に与える影響と実装における課題について議論する．

2.1 HBase のデータ構造

HBase[*1]は BigTable [2] の設計を元にして作られたデータベースである．HBase の論理データ構造は関係データベースと同じ表形式であり，各行は RowKey で一意に識別され，各列は Qualifier で構成される．また，Qualifier をグルーピングする ColumnFamily と呼ばれる概念がある．RowKey と Qualifier が交差する部分にデータ（Value）とタイムスタンプを保持するようになっている．

一方で物理データ構造ではデータは key-value 形式で格納され，RowKey, ColumnFamily, Qualifier，タイムスタンプの 4 つの Key を構成するフィールドに対して Value が対応している．また，RowKey，ColumnFamily，Qualifier，タイムスタンプの優先度で Key がソートされて格納されている．

HBase では Table のデータを Region という単位で分割しており，Region は RowKey の範囲によって分割される．そのため，同じ RowKey をもっているデータは必ず同じ Region に所属する．さらに，同じ Region 内のデータは ColumnFamily 単位でさらに分割される．そのため，同時にアクセスする Qualifier を同じ ColumnFamily にまとめるとよい．

2.2 HBase のスキーマ設計

key-value の設計はアプリケーションの性能に大きな影響を与える．そのため，開発者はアクセスパターンに基づいて適切に設計する必要がある．HBase においても前述の通り RowKey と ColumnFamily がインデックスとなっているため，アプリケーションデータをマッピングする際に意識して設計する必要がある．

図 1.2.1 を例として HBase におけるスキーマ設計を説明する．これはアプリケーションデータと key-value の変換の例であり，1 種類のアプリケーションデータを 2 種類の異なる key-value へと変換している．

- **変換 1：** RowKey にキー属性である key1 と key2 をセパレータ文字（,）で連結して配置している．Value に非キー属性である value1, value2 をセパレータ文字（,）で連結して配置している．
- **変換 2：** RowKey に *key1* をハッシュ化した値の 3byte を先頭に付与し，それに続けてそれぞれの文字数を接頭辞として key1，key2 をつなげたものを配置している．Qualifier に非キー属性である value1，value2 を異なる key-value に配置している．

変換 1 は key1 を使った範囲検索をするのに適している．また，アプリケーションデータが 1 つの key-value で表現されるので空間効率がよくなる．一方で，同じ値をもった key1 が連続するためアクセスが集中してしまい，またセパレータ文字（,）が key1 の値に含まれてはいけない制約ができる．非

[*1] https://hbase.apache.org/

キー属性が 1 つの key-value で表現されているので不要なデータの読み取りが必要になる場合があるといったデメリットもある．

変換 2 は先頭にハッシュを付与しているため負荷分散される．key-value は RowKey でソートされているため，ハッシュなどを先頭に付与していない場合には，似た RowKey が連続してしまいアクセスが集中する可能性がある．また，非キー属性ごとに key-value を分けているため，属性ごとの読み書きや新しい属性の追加が容易にできる．たとえば，アプリケーションデータに新たに非キー属性である value3 を追加したい場合には，value3 を Qualifier に指定した key-value を追加すれば良い．一方で，ハッシュを付与することで Key が連続しなくなり範囲検索ができなくなる．さらに，非キー属性ごとに key-value を分けているために RowKey が重複し Region が分割しにくくなってしまう．

このようにアプリケーションデータを key-value に変換するさまざまな設計手法がある．あるキーに対して正規表現を用いて検索したい場合には変換 1 のように先頭に検索したいキーを置く．キーに時系列データがある場合には最新のデータが同じリージョンに集まり負荷が集中する可能性があるので変換 2 のようにハッシュを先頭に付与するなど，アクセスパターンを元に適切にスキーマ設計することが重要である．

2.3　変換処理の実装

HBase のような key-value ストアはデータ構造がシンプルなために，アプリケーションの複雑なデータとのギャップが存在してしまう．そのため前述のような HBase のスキーマ設計を実現するためには，アプリケーション開発者がアプリケーションデータと key-value 間の変換処理の実装を行う必要がある．変換処理はアプリケーションデータから key-value への変換を行うシリアライズとその逆に key-value からアプリケーションデータへの変換を行うデシリアライズがある．

HBase における RowKey のデシリアライズの例を図 1.2.2 に示す．これは図 1.2.1 の変換 2 の RowKey をアプリケーションデータへ変換する部分の処理を表している．key-value は内部でバイト配列として扱われている．バイト配列を String 型へと変換し，セパレート文字 (,) を探索して key1, key2 に分解する処理を行っている．この処理は変換 2 にのみしか使用できず，変換 1 ではセパレータ文字ではなくキー属性の文字数で分割するなどの異なる変換処理を実装する必要がある．一方で，セパレート文字で分割したり，ハッシュを先頭に付与して負荷分散するなどスキーマ設計によく使われる共通のテクニックが存在する．

3　スキーマ定義に基づく API サーバの実装

本節では，スキーマ設計によって異なる変換処理の実装コストを下げるための，アプリケーションデータと key-value のマッピングの汎用化について述べる．本節ではスキーマ定義の記述のみで key-value ストアにアクセスできる API サーバの実装を紹介する．

API サーバの概要を図 1.2.3 に示す．API サーバはアプリケーションデータ（Tuple）を入力として key-value（Record）を挿入する，もしくは SQL-like なクエリ（Query）を入力としてアプリケーションデータを探索する操作が可能である．変換処理を行うコンポーネントをそれぞれ説明する．

3.1　Schema Mapping Definition

Schema Mapping Definition はどのようにスキーマをマッピングするかを定義したファイルである．スキーマ定義の例を図 1.2.4 に示す．この図での例は図 1.2.1 のアプリケーションデータと key-value の変換を定義したものである．このファイルには 3 つの定義が書かれている．1 つ目はリレーションの属性を定義しており，残りの 2 つはスキーマを定義している．スキーマでは HBase の key-value の各フィールドに対して配置する属性をコロン（:）でつなげている．`suffix（,）{key1}`

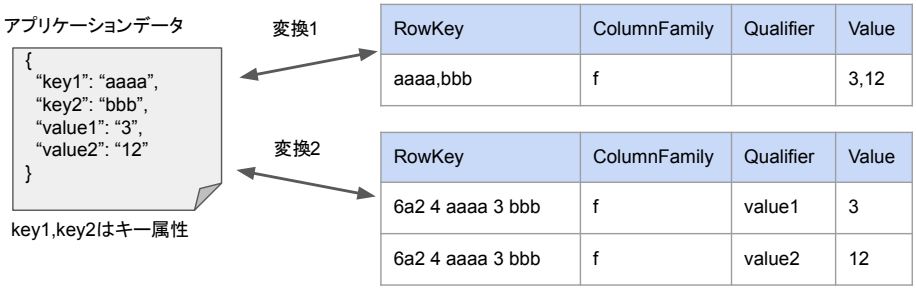


図 1.2.1　スキーママッピングの例

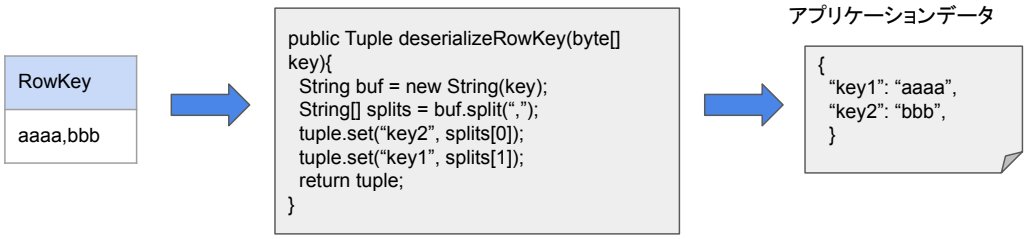


図 1.2.2　HBase における RowKey のデシリアライズの例

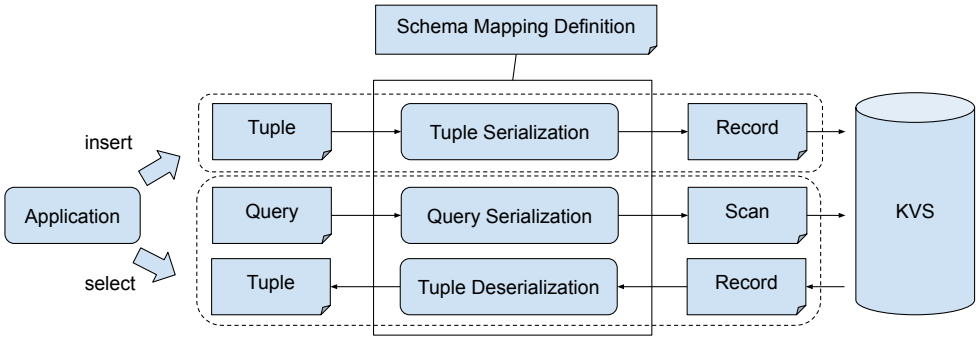


図 1.2.3　スキーママッピング技術を用いた API サーバの概要

や `size{key1}` は各属性をセパレータ文字や文字数の接頭辞などで分割することを表している．`hash[key1]` は key1 をハッシュ化することを表している．

3.2　Tuple と Record 間の変換処理

変換処理は 2.3 項で説明したように，シリアライズとデシリアライズを行う必要がある．それに加えて API サーバの実装では SQL-like なクエリでデータを操作できるようにクエリを key-value の問い合わせにシリアライズするコンポーネントを実装した．Tuple（アプリケーションデータ）と Record（key-value）間の変換処理を行うコンポーネントは以下の 3 つがある．

- **Tuple Serialization:** insert 操作においてスキーマ定義に従い Tuple を Record に変換する処理を行う．
- **Tuple Deserialization:** select 操作においてスキーマ定義に従い Record を Tuple に変換す

```
define relation order {
  key1 string key,
  key2 string key,
  value1 string,
  value2 string
};
define schema layout_1 for order
  on hbase(“localhost”) {
  table “s1”,
  row hash[key1]:size{key1}key1
    :size{key2}:key2
  family attr_name[key1, key2],
  value attr_value[key1,key2]
};
define schema layout_2 for order
  on hbase(“localhost”) {
  table “s2”,
  row suffix(,){key1}:key2,
  family “f”,
  qualifier “”,
  value suffix(“,”){value1}:value2
};
```

図 1.2.4　スキーママッピング定義の例

る処理を行う．

- **Query Serialization:** select 操作において SQL-like なクエリをスキーマ定義に従い key-value の問い合わせ（Scan）に変換する処理を行う．なお，ジョインや集計操作には対応していない．

4　スキーマ定義に基づく API サーバの構築と運用

本節では，スキーマ定義に基づく API サーバの構築と運用方法について紹介する．

4.1　設計方針

API サーバの構築と運用をする上で次の内容を考慮して設計を行った．

- アプリケーションの仕様の変更によるアクセスパターンの変更に対応するため，変換処理の更新を容易に行えるようにする．前節で説明したように API サーバはスキーマを定義するだけでデータストアへ接続しデータにアクセスすることができる．そのため，スキーマを定義したファイルを容易に差し替えられるようにすればこの要件は達成できる．
- 新たに API サーバを利用するアプリケーションを追加できるようにする．また，不要なアクセスを防ぐため，アプリケーションごとにアクセスできるスキーマを制限できるようにする．
- 運用や保守の負担を減らすため，スキーマの変更やアプリケーションの追加が行われても監視や障害対策の設定を変更する必要がないようにする．

4.2　構築環境

API サーバを構築している環境について説明する．図 1.2.5 は構築環境のコンポーネントを示している．API サーバはコンテナ化され，コンテナレジストリである Amazon Elastic Container Registry (Amazon ECR) にイメージが保存されている．

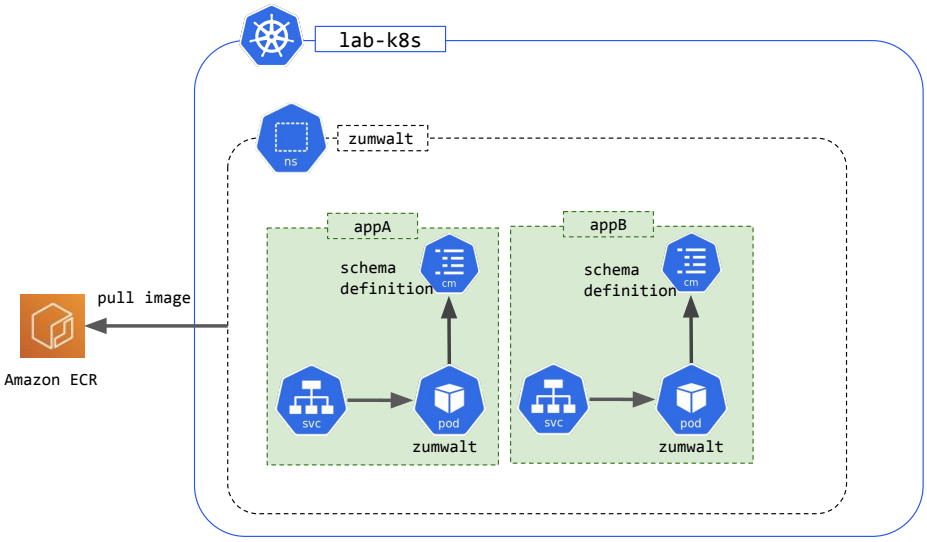


図 1.2.5 コンポーネント図

Kubernetes

Kubernetes はコンテナ化されたアプリケーションのデプロイなどの管理を自動化するためのプラットフォームである．秋葉原ラボでは独自の Kubernetes 環境 (以下，lab-k8s と呼ぶ) を運用している．環境構築や運用の負担を減らすために lab-k8s クラスタ上に API サーバを構築する．使用する Kubernetes のリソースは以下の通りである．

- **Pod:** コンテナを実行するためのリソース．Amazon Elastic Container Registry から API サーバのイメージを pull し実行している．
- **ConfigMap:** 設定情報などを Key-Value として保持するためのリソース．スキーマ定義ファイルを保存するために使用している．
- **Service:** Kubernetes クラスタ外と通信を行うためのリソース．lab-k8s では Service リソースを作成すると自動で外部のロードバランサーの設定を行い，lab-k8s 外から参照できるような仕組みになっている．

API サーバは起動時にスキーマディレクトリを参照し，スキーマ定義を読み込む．そのためスキーマディレクトリに ConfigMap として保存してあるスキーマ定義ファイルをマウントするようにしている．ConfigMap でスキーマ定義を管理することにより，API サーバのコンテナイメージに依存することなく容易にスキーマ定義を差し替えることが可能となっている．

また，アプリケーションごとに異なる API サーバを構築している．つまり，それぞれに対して Pod，ConfigMap，Service のリソースを作成している．図 1.2.5 では lab-k8s 内に appA と appB の 2 種類のアプリケーションが構築されている．アプリケーションごとに構築することにより，アクセスできるスキーマを制限することが可能となる．それぞれの Service リソースで Pod へのアクセス制限を設定することで，限られたアドレスからしかその Pod の API を利用できないからである．

Helm

Kubernetes のパッケージマネージャである．ここでは，テンプレートからマニフェストを作成するテンプレートエンジンとして Helm を利用している．

Kubernetes では YAML 形式で書かれたマニフェストファイルを作成してデプロイする必要がある．しかし，システムの規模が大きくなると大

量のマニフェストを作成することになり，管理や変更を行うことが困難になる．そこで，Helm を利用することでマニフェストを汎用化して扱うことができるため，マニフェストの管理を簡単に行うことができる．

Helm を利用してマニフェストを作成するには，Template と values の 2 種類のファイルが必要になる．

- **Template:** Kubernetes のマニフェストのテンプレートと変数を定義する．つまり API サーバ共通の部分をテンプレートとし，アプリケーションごとに異なる部分を変数として定義する．
- **values:** Template で定義した変数の設定値を記述する．アクセス元 IP や接続先の情報などアプリケーションごとに異なる設定値を記述する．

これらのファイルからアプリケーションごとのマニフェストを作成し，lab-k8s クラスタへ API サーバをデプロイしている．

4.3 監視と障害対策

API サーバの監視と障害対策に利用しているツールとその設定を紹介する．

Prometheus

オープンソフトウェアの監視ツールであり，時系列で様々なメトリクスを収集することができる．Prometheus は様々なコンポーネントで構成されているが，API サーバの監視は lab-k8s クラスタ上に Prometheus サーバと AlertManager を構築している．Prometheus サーバはメトリクスの収集と保存を行うコンポーネントであり，API サーバのスループットやレイテンシなどのメトリクスを収集している．Prometheus サーバに設定してあるモニタリングの閾値を越えると，AlertManager へアラートのリクエストが送られ，AlertManager はそのリクエストを元に外部のシステムに通知を送る．

Grafana

Prometheus で収集したメトリクスの可視化を行うツールである．lab-k8s では全プロジェクト共用の Grafana が提供されている．共用の Grafana から Prometheus サーバに接続することでメトリクスを可視化している．

Circuit Breaker

データストアへのリクエストを監視し，障害などの発生でレスポンスが返ってこなくなった場合にデータストアを一時的に切り離すことができる．Circuit Breaker を実現するためのライブラリとして Resilience4j[*2]を導入している．Circuit Breaker はスキーマ定義単位に作成しており，スキーマ定義に記述されているデータストアに障害が発生した場合にはデータストアを切り離しエラーをすぐ返すようにする．エラーをすぐ返すことで障害発生時でも応答時間を維持することができ，他のスキーマ定義に記述されている正常なデータストアへのリクエストの影響を少なくすることができる．

5 まとめ

本稿では，スキーマ定義に基づく API サーバ（開発コード: Zumwalt）を紹介した．アプリケーションデータと key-value 間の変換処理を自動化しスキーマ定義として宣言的に記述することで，key-value ストアを利用したアプリケーションを統合でき，開発や運用の負担を減らすことができるようになった．

今後の予定として，属人化を排除するためにスキーマ定義登録から API サーバ立ち上げまでの自動化をすることで管理者の負担を減らす予定である．

[*2] https://resilience4j.readme.io/

参考文献

[1] 善明 晃由, 津田 均. スキーマ定義に基づく SQL ライクな Key-Value ストアクライアント, 第 8 回データ工学と情報マネジメントに関するフォーラム (DEIM2016), 2016.

[2] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. In Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation — Volume 7, OSDI '06, p.15, 2006.

Akihabara Lab

1.3 ストリーム処理を考慮したデータ転送管理システムのレイヤ化

新平 和礼

概　要　インターネットサービスを提供するシステムは日々さまざまなデータを出力する．これらのデータはBI ツールによるデータ分析や効果測定，推薦や検索などのさまざまな用途で利用される．データを活用するためには大量のデータを目的のシステムに転送しなければならない．その際，データの種別によって利用目的が異なるため，データごとに適切なシステムへ転送する必要がある．また，転送中のデータに対し，不正なフォーマットのデータを排除するフィルタリングや転送先システムに応じたデータ加工などの処理が求められる．適用する処理についてもデータを生成したシステムやデータの種別，用途などの条件によって異なるためデータに応じた設定が求められる．データ転送用のミドルウェアを用いることでデータ転送や転送中のデータに対する処理の適用が可能となるが，単一のミドルウェアでさまざまな経路を管理することや，転送と転送中のデータに適用する処理が深く結び付くことなどから転送設定が複雑化するという課題がある．本稿ではこれらの課題を解決するためデータ転送の責務を分離する階層構造を提案し，本階層構造に基づくデータ転送を提供する基盤について述べる．

Keywords　ストリーム処理，データマネジメント，データ転送

1　はじめに

システムが生成するデータは BI ツールによるデータ分析や効果測定，外部システムとの連携などさまざまな用途で利用される．また，システムが出力するデータの利用目的は多岐にわたり，利用目的によってリアルタイム処理やバッチ処理など異なるシステムでデータを処理する必要がある．そのため，データを活用するためにはさまざまなシステムが出力したデータを目的に応じたシステムに転送しなければならない．

データ転送や転送中の処理の適用を実現するソフトウェアとして Apache Flume[*1]や Fluentd[*2]などのデータ転送ミドルウェアがあげられる．これらのミドルウェアは到達保証設定に応じたデータ転送の信頼性確保や，ミドルウェアを起動するマシンの追加による処理性能のスケールアウトの機能を提供する．ミドルウェアの機能を利用することで，各転送経路の信頼性やスケーラビリティを向上できる．

しかし，単一のミドルウェアで多数の経路を管理する場合，経路の増加とともにデータ転送ミドルウェアの設定が複雑になる．その要因として，データの形式や経路選択の設定方法，適用する処理が経路ごとに異なる点があげられる．データ転送の経路には異なる用途，形式のデータが混在するため，それぞれのデータから転送先の情報を抽出し適切な宛先に転送する必要がある．また，データ転送においては転送先システムへのデータ投入

[*1] https://flume.apache.org
[*2] https://www.fluentd.org/

前に不正な形式のデータを排除したり，重複したデータを排除するなどの前処理が必要となるケースも存在する．これらの条件が異なる複数の経路が同一のミドルウェアの設定に混在することで設定が複雑化する．

また，転送ミドルウェアにおける設定では，転送設定と処理が密に結合する課題がある．適用する処理内の関数の戻り値ごとに転送先を設定するなど，転送先の設定と処理の実装が混在し，転送経路の把握が困難となる．

本課題を解決するために我々はデータの転送情報を抽象化してレイヤ構造として定義する取り組みを行っている．データ転送のレイヤ化ではデータ転送の責務を 4 つの階層で定義することで，データ転送における経路の選択方法を共通化するとともに，アプリケーションレベルの処理とデータ転送を分離する．

本稿ではレイヤ構造に基づくデータ転送処理基盤を試作し，本基盤上でデータ転送を行う事例を紹介する．本稿の構成を以下に示す．2 節で既存のデータ転送における課題について述べる．3 節では本稿で提案するデータ転送の定義について説明し，4 節で開発したデータ転送処理基盤のアーキテクチャについて述べる．5 節で本システムの適用事例について述べ，6 節で本稿をまとめる．

2 背景

本節ではデータ転送の設定が複雑になる要因について述べる．

2.1 転送設定の異なる経路の混在

転送設定の異なる経路が混在することにより複雑化するデータ転送の構成例を図 1.3.1 に示す．本構成では，拠点 1 に構築されたシステム A，B が生成するデータをストレージとメッセージキューの 2 ヵ所へ転送する．拠点 1 に配置されたシステム A，システム B はそれぞれ異なる運用が行われ，データの形式も異なる．システム A が生成するデータは JSON 形式となっており，メッセージキューへ転送を行う必要がある．また，システム B が生成するデータは TSV 形式となっており，ストレージへデータ転送を行うものとする．

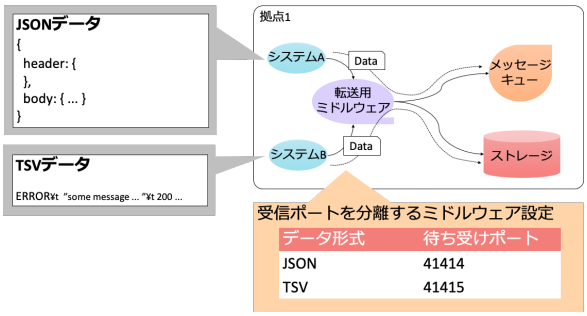


図 1.3.1 転送設定の異なる経路が混在する転送構成例

一般的なデータ転送ミドルウェアでは，異なる複数のポートでデータを受信することでこのような異なる条件や宛先の転送設定を実現できる．図 1.3.1 の例では JSON 形式のデータを 41414 番ポートで待ち受け，TSV 形式のデータを 41415 番ポートで待ち受けることで条件の異なる形式のデータをそれぞれ異なる宛先へ転送できる．

しかし，この設定方法ではデータ形式の差異がミドルウェアのネットワーク設定にまで波及している．送信側でデータの形式に応じてポートを選択する必要がある点やポート番号の変更時にデータの送信側と受信側で同期的に設定変更する必要があることなど，データ転送の管理が複雑化する．[*3]

2.2 処理と転送の結合

データ転送用のミドルウェアでは転送中のデータに対しストリーム処理を適用できる．2.1 項で示した条件の転送は本機能を利用することでも実現可能である．

図 1.3.2 にストリーム処理を適用する場合の転送構成例を示す．本転送設定では 41414 番ポートでデータを受信し，受信データに対し独自に実装した処理である DataSelector を適用する．DataSelector は受信データの形式に応じた分岐処理を行う機能を有し，これにより受信ポートの設定を分離せずにデータ形式に応じた転送先の選択を実現

[*3] 独自のプラグインを実装することでネットワーク設定への影響を軽減できるが，後述する転送と処理の一体化の問題がある．

可能となる．

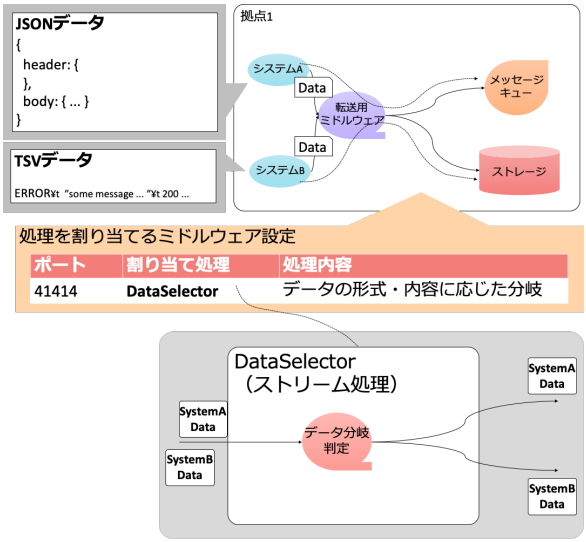


図 1.3.2　プラグイン処理によるデータ転送の構成例

一方で本設定では転送設定が処理の実装に依存し，一体化するという問題がある．たとえば本設定では 41414 番で受信したデータに対し処理を適用する設定を行うが，その処理内容は実装に依存する．処理の内容を理解した設定が必要になるなど，転送時に行う処理と転送設定が一体化することで経路設定の難易度が上がり，経路の把握も難しくなる．

2.3　複数の拠点をまたぐ転送経路の管理

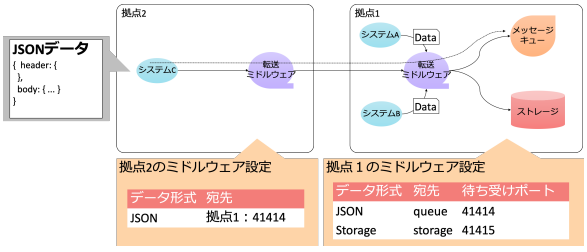


図 1.3.3　経路を追加する場合のデータ転送の構成例

複数の拠点をまたぐ転送経路の管理が複雑化する例を図 1.3.3 を用いて説明する．図 1.3.3 では図 1.3.1 の例に拠点 2 のシステム C からのデータ転送が追加されている．追加する経路では，システム C のデータを拠点 1 のメッセージキューに格納する．対象となるシステム C が送信するデータは JSON 形式であり，メッセージキューへ転送するものとする．

この転送設定を行うためにはまず拠点 1 の転送設定を確認し，どの形式のデータを何番ポートで待ち受けているかを把握する必要がある．本例では JSON 形式は 41414 番ポートで待ち受けているため，拠点 2 に対し JSON 形式のデータを拠点 1 の 41414 番ポートへ送ることで転送が成立する．

このように複数拠点にまたがって転送を成立させる場合に複数拠点の設定を理解して設定する必要がある．また，拠点 2 の設定には拠点 1 の 41414 番ポートへ送信するという情報しか記述されないため，システム C の転送経路全体を把握するには拠点 1，拠点 2 両方の設定をすべて確認する必要が生まれる．このように経路選択ロジックが局所化することで全体の転送設定の把握が困難となる．

3　データ転送処理のレイヤ化

データ転送における課題の要因として，共通的な転送先の決定手順や宛先情報の定義が存在しないこと，転送中のデータの加工やフィルタ，分岐といった処理内容と転送が密に結合していることがあげられる．

これらの課題を解決するため，データ転送の責務を 4 層に分離するレイヤ構造（以降，データフローレイヤ）を提案する．データフローレイヤではデータ転送における手続きの共通的な規定と責務の分離を行う．本節では，データフローレイヤの階層構造とレイヤ化による課題解決について説明する．

3.1　データ転送におけるレイヤ構造

データフローレイヤは下位レイヤから順にインフラストラクチャレイヤ，リンクレイヤ，ルーティングレイヤ，アプリケーションレイヤによって構成される．各レイヤは最下層のレイヤであるインフラストラクチャレイヤから順に，レイヤ 1，レイヤ 2，レイヤ 3，レイヤ 4 と呼称する．データフローレイヤにおけるデータ転送処理のフローとデータの変化を図 1.3.4 に示す．

送信処理では上位レイヤから下位レイヤに向かって処理を行い，処理したデータを下位レイヤ

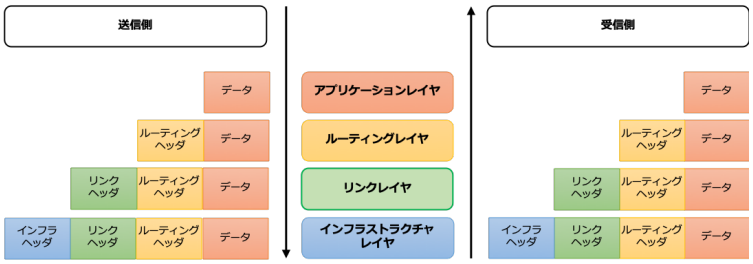


図 1.3.4 データフローレイヤの送受信処理イメージ

に引き渡す．また，各レイヤは上位レイヤから受け取ったデータに対し，それぞれのレイヤで必要となる転送情報を付与する．ここで各レイヤで付与する情報をヘッダと呼ぶものとする．

受信処理では下位レイヤから上位レイヤに向かって処理を行い，処理したデータを上位レイヤに引き渡す．上位レイヤでの処理内容の変更が下位レイヤの転送処理に影響を与えてはならない．受信処理では各レイヤは上位レイヤにデータを引き渡す際，それぞれのレイヤのヘッダ情報を削除することで上位レイヤに必要なデータのみを伝達する．各レイヤの責務の詳細について以下で述べる．

アプリケーションレイヤ — アプリケーションレイヤはアプリケーション固有の規定を定める．サービスや用途に応じたデータ形式の規定はアプリケーションレイヤの責務となる．データ転送中のデータの変換やフィルタなどのロジックについてもアプリケーションレイヤで処理される．

ルーティングレイヤ — ルーティングレイヤはデータ転送における経路選択の責務を担う．本レイヤで管理する経路情報に従いデータの宛先に対する経路を選択し，次の送信先を決定する．

リンクレイヤ — リンクレイヤでは隣接するデータ転送装置間のデータ送受信を管理する．上位レイヤから受領したデータおよび送信先を解釈し，ミドルウェアレベルの宛先情報に変換したうえでインフラストラクチャレイヤにデータと宛先を渡す．

インフラストラクチャレイヤ — インフラストラクチャレイヤはデータ転送に利用するミドルウェアを管理する責務を担う．ミドルウェア実装によるデータの送受信を担当し，ミドルウェアレベルでの送達管理を行う．

3.2 レイヤ化による課題解決

転送設定の異なる経路の混在

2.1 項では転送設定が異なる経路が混在する構成を例に，ミドルウェアの待ち受けポート番号を送信側が意識する必要があることを示した．また，これによりミドルウェアの増設やポート番号変更などの設定変更が送信元の設定にも影響を与えることを述べた．

データフローレイヤではこれらの課題を解決するため隣接する転送装置間のデータ転送をインフラストラクチャレイヤとリンクレイヤに分離する．リンクレイヤでは隣接する転送機能間のデータ転送の責務を担い，インフラストラクチャレイヤは具体的なミドルウェア実装による転送処理を管理する．

リンクレイヤはミドルウェアの IP アドレスやポート番号で識別される送信先を抽象化する．同一設定で実行される複数のミドルウェアをリンクレイヤでは 1 つの宛先としてみなすことが可能となり，インフラストラクチャレイヤのミドルウェアにおける設定変更や増設などの変更をリンクレイヤから隠蔽できる．

転送と処理の密結合

2.2 項で転送設定と処理の結合に関する課題を例示した．本課題の原因はデータ処理における分岐や加工が転送設定と密に結合することにある．たとえば，加工処理によって設定される値をミドルウェアの転送設定で参照するなど，ミドルウェア

の設定において実装によって決定される値の記述が求められるケースが存在する．データフローレイヤではアプリケーションレベルの処理をレイヤ 4 として定義し，転送処理を行うルーティングレイヤ以下と分離する．これにより，アプリケーションレベルの加工や分岐などの処理と転送処理を分け，転送設定の複雑化を解消する．

複数拠点間の経路の混在

2.3 項では経路選択の方法が標準化されていないことによる転送設定の複雑化の例を示した．本課題の要因としては転送先を選択する条件がデータ間で統一されていないことや，データの最終的な宛先情報および宛先に対する次の転送先の情報が管理されていないことからミドルウェアの設定に異なる条件の経路設定が混在し，複雑化することがあげられる．データフローレイヤではルーティングレイヤからインフラストラクチャレイヤでデータ転送における経路選択に必要な情報と責務を分離し，共通化することで本課題を解決する．

ルーティングレイヤは経路情報を管理し，最終的な宛先と宛先に対する次の転送先の対応関係を保持する．リンクレイヤは隣接する転送機能間のデータ転送を管理し，インフラストラクチャレイヤは具体的なミドルウェアを利用したデータ転送を行う．データ転送に関わる情報と責務を定義することで，データ転送の経路全体を通して共通的な設定方法を提供し，設定の複雑化を防ぐことができる．

4　データ転送処理基盤

当社では 3 節で述べたモデルに基づいたデータ転送処理基盤の開発を行っている．本節では試作した基盤のシステム構成について記載する．

4.1　データ転送処理基盤の概要

図 1.3.5 に本転送処理基盤によるデータ転送の概要を示す．本基盤は各拠点に配置され，それぞれのデータ転送をストリーム処理として実行する．本基盤はリンクレイヤからアプリケーションレイヤの処理を担当し，インフラストラクチャレイヤとして自身が接続するストレージやミドルウェアを制御している．基盤上のそれぞれの転送処理はレイヤ 1 のミドルウェアによって接続される．

本基盤では，転送経路やストレージの差異を吸収するため，リンクレイヤにおいて Source，Sink と呼ばれるインタフェースを提供する．Source はデータの入力を定義するものであり，Sink はミドルウェアやストレージへのデータの出力を定義するものである．Source，Sink によって転送経路やミドルウェア，ストレージの実装の詳細が隠蔽されるため，上位レイヤを特定の実装に依存しない形で定義できる．

ルーティングレイヤにおける経路の選択やアプリケーションレイヤにおけるアプリケーション固有の処理の実装を定義するため，本基盤は Interceptor と呼ばれるインタフェースを提供する．Interceptor は入力データに対する出力データを返す関数として定義される．

4.2　データ転送処理モデル

本項では本基盤において定義できるデータ転送処理のモデルについて述べる．データ転送処理の構成例を図 1.3.6 に示す．

本基盤では Source，Interceptor，Sink の総称を Task と呼称し，Task の入出力の接続を Port と呼称する．転送処理は Task と Port の組み合わせで定義する．Source はミドルウェアから受信したデータ列を出力 Port に書き込み，Sink は自身の入力 Port に書き込まれたデータを接続するストレージやミドルウェアに書き込む．Interceptor は入力 Port で受信したデータを処理し，結果に応じて出力する Port を選択する．なお，入力データに対し常に 1 つの出力 Port を選択する Interceptor を Transform と呼び，入力データに対し複数の出力 Port を選択可能な Interceptor を Router と呼称する．また，Router において出力 Port を 2 つ保持する Interceptor を Filter と呼ぶこととする．

開発した基盤では本項で述べた Task の構成情報を DSL で記述することで転送処理を構築し実行

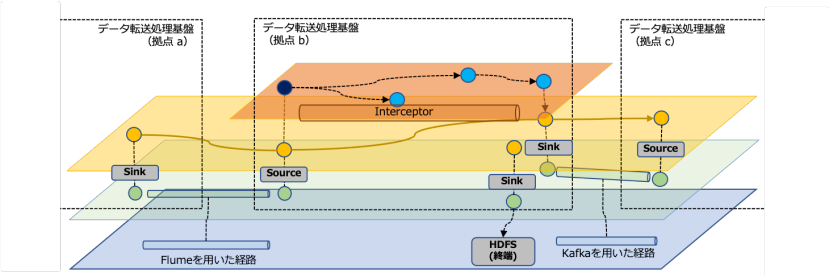


図 1.3.5 データ転送処理基盤による転送処理の概要

できる．DSL の記述例については 5 節で述べる．

4.3 システム構成

開発した基盤の構成を図 1.3.7 に示す．

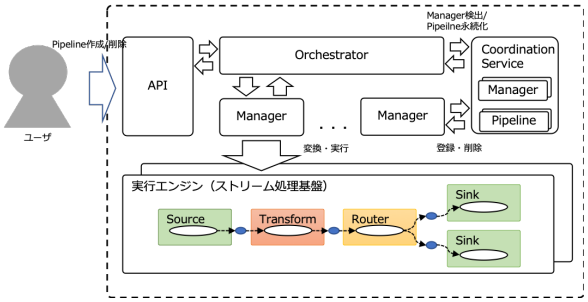


図 1.3.7 データ転送処理基盤のシステム構成

本基盤はユーザが定義したデータ転送処理を受領し, 構成情報をパース, ストリーム処理に変換し, Apache Flink などの実行エンジン上で実行することでデータ転送処理を実現する．本基盤はユーザからのリクエストを受け付ける API，実行エンジンの管理を行う Manager，複数の Manager や実行中の Pipeline を統括管理する Orschestrator で構成される．Manager は実行エンジンを管理しデータ転送処理を実行する機能部であり，実行エンジンごとの差分を Manager が吸収することで本基盤では複数の実行エンジンの利用を可能とする．

ユーザはデータ転送処理を行う Pipeline 構成を定義し，API にリクエストを送信する．API は Orchestrator に Pipeline の実行を要求し，Orchestrator は管理する複数の Manager からユーザリクエストに応じて適切な Manager を選択する．Manager はユーザが定義した Pipeline 構成を検証し，実行エンジンに適した処理に変換して転送処理を起動する．

5 適用事例

本節では 4 節で述べたデータ転送基盤によるデータ転送の適用例について記載する．以降の説明では適用例としてデータの品質チェックを伴うデータの転送を用いる．本処理の構成を図 1.3.8 に示す．

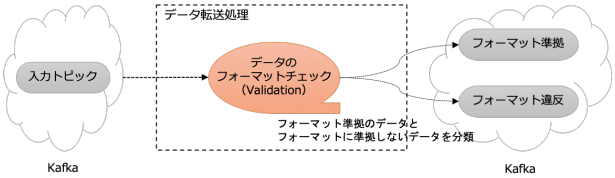


図 1.3.8 処理を伴うデータ転送の構成例

本処理では Kafka からデータを取得し， 取得

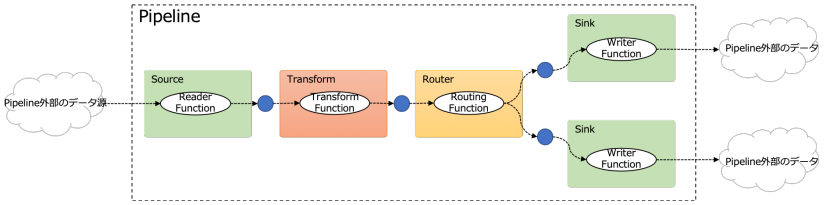


図 1.3.6 ストリーム処理の構成

したデータが定められたフォーマットに準拠しているかを判定する．判定の結果に基づき，フォーマットに準拠しているデータとフォーマットに準拠しないデータを異なる宛先へ転送する．

図 1.3.8 の転送を試作したデータ転送処理基盤で実行する際の設定を図 1.3.9 に示す．なお，ここで利用した Task は簡単のためルーティングレイヤによる経路選択は行わず，Sink による固定的な宛先へのデータ送信を行う処理として実装した．

図 1.3.9 の上部は転送処理の実行時にユーザが記述する DSL を示している．本基盤の DSL では tasks セクションで転送処理の接続関係を定義する．14–19 行目に記述した Filter 処理でフィルタリング処理を行い，Filter の出力ポートである valid_port，invalid_port でポート名を定義することで後続の処理との接続関係を記述する．ValidationFilter は type が Filter であることから，分岐を伴う処理であることは本設定から明らかであり，ユーザは出力ポートに対して後続処理を設定することで，実装で決定する値を意識することなく分岐を含むデータ転送を定義することが可能となる．

ここで，図 1.3.9 に示す Source，Sink の設定は基盤に事前登録することが可能となっている．これらの設定を基盤に事前登録しておくことでデータ転送処理を記述する利用者からインフラストラクチャレイヤの転送情報を隠蔽できる．これらは基盤管理者によって設定されるため，データ転送処理を起動する利用者はインフラストラクチャレベルレイヤの接続情報を意識することなく転送設定を記述可能となる．

6　まとめ

本項では既存のデータ転送が複雑化する要因について述べた．また，データ転送管理における設定の複雑化を改善する枠組みとしてデータ転送の情報を階層化して定義するデータフローレイヤを提案し，本定義に基づくデータ転送処理を行う基盤を試作した．試作した転送処理基盤におけるデータ転送処理の適用事例を示し，データ転送における関心の分離ができることを確認した．

今後は，データフローレイヤの仕様を詳細に定義し，より大規模な転送管理に適用する予定である．

```yaml
---
global_option:
  name: "example"
  description: "pipeline for example"
  runner:
    type: FLINK
    parallelism: 3

tasks:
  kafkasource:
    type: Source
    ref: "kafka_source"
    output_port: "kafka_source"
  validation:
    type: Filter
    implemented_class: |-
      com.example.task.filter.ValidationFilter
    input_port: "kafka_source"
    valid_port: "validation_ok"
    invalid_port: "validation_ng"
  ok_kafkasink:
    type: Sink
    input_port: "validation_ok"
    ref: "ok_kafka_sink"
  ng_kafkasink:
    type: Sink
    input_port: "validation_ng"
    ref: "ng_kafka_sink"
```

```yaml
kafaka_source:
  type: Source
  implemented_class: com.example.source.KafkaSource
  output_port: "OVERRIDE_AT_RUNTIME"  # 実行時に決定される
  properties:
    topic: "test_input"
    bootstrap_servers: "kafka01.server:9092
ok_kafka_sink:
  type: Sink
  implemented_class: com.example.sink.KafkaSink
  input_port: "OVERRIDE_AT_RUNTIME"  # 実行時に決定される
  properties:
    topic: "ok_topic"
    bootstrap_servers: "kafka01.server:9092
ng_kafka_sink:
  type: Sink
  implemented_class: com.example.sink.KafkaSink
  input_port: "OVERRIDE_AT_RUNTIME"  # 実行時に決定される
  properties:
    topic: "ng_topic"
    bootstrap_servers: "kafka01.server:9092
```

図 1.3.9 Validation を行う DSL 定義例

Akihabara Lab

1.4 ストリーム処理におけるステートフルデータの管理手法

斎藤 貴文

概　要　ストリーム処理においてステートフルなデータを簡潔に処理するための解析基盤 Phalanx について解説する．ストリーム処理ではイベントの到着順序が保証されないため，更新順序を考慮する必要があるステートフルなデータの扱いが難しい．Phalanx では CRDT のデータモデルに基づいて，ステートフルなストリーム処理を実現する．CRDT で用いられるデータ型は更新の順序を問わず同じ状態となることが保証されるため，データの到着順序が保証されない状況でのステートフルなデータへの管理を簡素化できる．そのような結果整合性をもつデータ型を利用することで，Phalanx ではイベントを CRDT の操作に対応付ける記述のみでステートフルなデータの変更を実現できる．

Keywords　ストリーム処理，CRDT

1　はじめに

近年，インターネットサービスのデータ解析においてストリーム処理は重要視されている．ストリーム処理ではイベントという単位のデータを入力として受け取り逐次処理する．そのため入力データを時刻で区切った単位でまとめて処理を行うバッチ処理に比べてリアルタイム性の高い処理を行うことが可能である．

実運用におけるストリーム処理のイベントはさまざまなライブラリやシステムの転送パイプラインを経てストリーム処理エンジンに届けられるため，ストリーム処理ではイベントが発生した時刻と同じ順序でストリーム処理エンジンに届くという保証はない．たとえばパイプライン中の転送経路を並列化した場合，負荷の不均質が原因で経路間で転送速度が異なり到着順序が入れ替わる可能性がある．

転送順序が保証されないためストリーム処理で扱うことが困難なデータとしてステートフルデータがある．ステートフルデータとは現在の状態に依存して次の状態が決まるデータである．インターネットサービスにおけるステートフルデータにはソーシャルゲームのユーザの所持アイテムリスト，Web サイトの PV やセッション数など多岐に渡る．

ストリーム処理でステートフルデータを更新することでリアルタイムで最新の状態を参照できるようになる．しかしステートフルデータの変更は現在の状態に依存し，ストリーム処理ではイベントの順序が保証されないのでステートフルデータの整合性を維持するのが難しい．

本章ではストリーム処理においてステートフルなデータ処理を簡潔に記述するための処理基盤 Phalanx について説明する．Phalanx では CRDT（Conflict-free Replicated Data Type）[1] のデータモデルを参考に更新順序が保証されない場合も同じ状態になることを保証する結果整合性をもつデータ型を用意している．結果整合性を担保するデータ型を利用することで更新の順序を考慮する

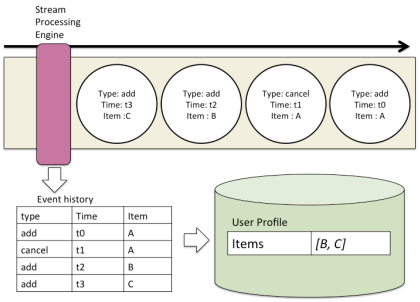


図 1.4.1 イベントの順序に変更がないときの例

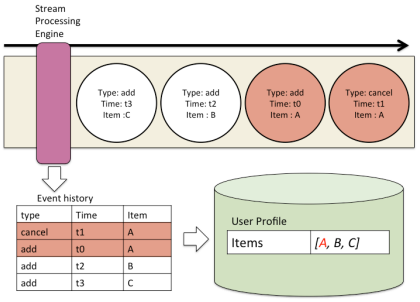


図 1.4.2 購入取消イベントが購入追加イベントより先に到達した例

ことなく，イベントとステートフルデータへの操作を対応付けるのみでストリーム処理でステートフルデータを扱うことが可能になる．

2 背景

2.1 ステートフルストリーム処理について

バッチ処理でステートフルデータの変更を行う場合，実際のイベントの発生時刻とステートフルデータへの変更が反映されるまでの時刻に差異が生じる．たとえば日次バッチ処理によってステートフルデータを変更する場合，発生したイベントは翌日にならなければステートフルデータに変更が反映されない．対してステートフルストリーム処理ではリアルタイムにステートフルデータの変更が可能なので最新のデータを参照できる．

しかし，ストリーム処理でステートフルデータを扱うことは難しい．それはストリーム処理ではイベントの到着順が保証されないためである．

ソーシャルゲームにおけるユーザのアイテムリストをストリーム処理で更新する例を挙げる．イベントにはアイテムの購入を表すイベント（`type=buy`）とアイテムの購入の取り消しを表す購入取消イベント（`type=cancel`）の 2 種類が流れると仮定する．2 つのイベントにはアイテム名が付与される．ストリーム処理エンジンは購入イベントが到達するとアイテムリストにアイテムを追加する．購入取消イベントが到達するとアイテムリストに該当のアイテムが存在する場合アイテムをリストから削除する．図 1.4.1 はイベントが時刻順に到達した場合を表している．このときアイテムリストは正しい状態でデータストアに反映される．一方，図 1.4.2 では時刻 t_0 のアイテム購入イベントが時刻 t_1 のアイテム購入取消イベントより遅れて到達した場合を表している．このときアイテムリストには本来削除されるはずのアイテム **A** が存在する．これはアイテムの購入取消をアイテムの購入後に対して行うはずであるのに，更新の順序が入れ替わることでアイテムが存在しない状態のリストに対してアイテムの購入取消を試みるためである．

このようにイベントの順序が入れ替わることで更新の順序も入れ替わりステートフルデータに対して誤った変更を行うことが起こりえる．

3 Phalanx のシステムデザイン

本節ではステートフルストリーム処理基盤である Phalanx のシステムについて説明する．

3.1 システムの概要

ステートフルストリーム処理を実現するためにはイベントが順不同で到達する状況で整合性を保つ必要がある．Phalanx では CRDT [1] というデータの更新の順序を問わず結果整合性を担保するデータ型を参考にしてデータ型を用意した．

CRDT（Conflict-free Replicated Data Type）は分散環境における個々の複製への更新を複製間で衝突させずに結果整合性を担保するためのデータ型である．CRDT には Operation-

based（Commutative Replicated Data Types, CmRDTs）と State-Based（Convergent Replicated Data Types, CvRDTs）の 2 種類が存在する．CmRDT は複製の更新操作を可換にすることで更新が順不同であっても必ず同じ状態になることを保証する．CvRDT は参照する際に可換かつ結合的である集約関数を用いてすべての複製を結合する．集約関数は可換であるため CmRDT 同様，各複製への更新が順不同であっても整合性を担保する．

Phalanx では CmRDT を参考にして，更新時にステートフルデータを変更するのではなく更新操作を保存し[*1]，更新操作を集約してステートフルデータとして提供する．このデータ型によってストリーム処理においてイベントの発生と到着の順序が不同であっても参照時に集約することで結果整合性を担保する．また更新時にイベントの到着順を考慮する必要がなくなるため，ステートフルデータの更新の記述を簡略化できる．

図 1.4.3 は Phalanx のシステムの概要図である．Phalanx はメタデータを保存する **Metadata Store**，入力イベントを更新操作に変換する **Updater Layer**, ステートフルデータへの更新操作を保存する **Storage**，ストレージのデータをステートフルデータとして提供する **Serving Layer** で構成される．

まず入力ストリームのイベントは Updater Layer に渡される．イベントはイベントが発生した時刻を表すタイムスタンプをもつ．Updater Layer は Metadata Store から **Table Schema（テーブルスキーマ）**と **Updater Rule（更新ルール）**の 2 種類のメタデータを参照して処理を行う．Phalanx ではステートフルデータはテーブルスキーマで規定される **Table（テーブル）**ごとに管理する．更新ルールは Updater Layer に届いたイベントを更新操作に変換する際に用いられる．テーブルスキーマ・更新ルールについては 3.2 項で説明する．Updater Layer では更新ルールに基づいてイベントを更新操作に変換し Storage に追加する．

ステートフルデータを参照する際には Serving Layer に問い合わせる．Serving Layer ではテーブルスキーマに従って Storage の更新操作をデータ型の集約関数を用いて集約した結果を返す．

3.2 Phalanx のデータモデル

本項では Phalanx のデータモデルについて説明する．まず Phalanx のデータ型について説明する．次にデータ型を踏まえて Storage のデータ構造について説明する．最後に Phalanx のメタデータについて説明する．

Phalanx のデータ型

Phalanx は更新時にステートフルデータを変更するのではなく，参照時に更新操作を集約してステートフルデータを返す．Phalanx のデータ型 D は以下のタプルで構成される．

$$D = (P, p_0, U, q) \tag{1.4.1}$$

P はステートフルデータへの変更の集合を表すペイロードである．P はバージョン V と更新値 Δ の直積 $P = V \times \Delta$ とする．ペイロードのバージョン V は更新の順序を定める値であるので, ペイロード P はバージョンに従って順序付けられる順序集合である．更新値 Δ はデータ型によって規定される．

$p_0 = (ver_0, val_0) : ver_0 \in V, val_0 \in \Delta$ はペイロードの初期値を表す．ver_0 は V の最小値とする．

U は入力イベント E をペイロード P に変換する更新メソッドの集合である．更新メソッドはデータ型によって複数存在することがある．更新メソッドは入力イベントとペイロードの間で単調性を満たす関数である．

q はペイロードを状態に変換する参照メソッドである．

Phalanx のデータ型は CRDT と同様，データ型

[*1] Phalanx では複製の扱いについては考えない

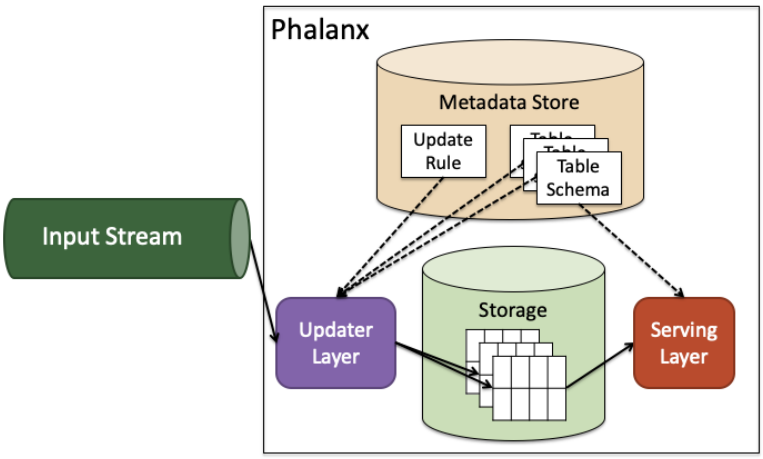


図 1.4.3 Phalanx のシステム概要

に更新メソッドや参照メソッドが用意されている．しかし CRDT と違い，Phalanx では状態を保持するのではなくペイロードを保持し，参照メソッドを用いてペイロードをステートフルデータに変換する．

次に Phalanx でサポートするデータ型についていくつか説明する．

Register 型 — Register 型は更新時に与えられた値を保存するための型である．Phalanx では Last Writer Wins ベースの Register（LWW-Register）を用意している．LWW-Register では最新の更新操作によって与えられた値を返す．Phalanx ではバージョンを元に最新の値を返す．

Register 型は下記の様に表すことができる．

```
payload: V v, X x
  initial: ver0, null
update: set(V u, X y):V v, X x
  let v = u
  let x = y
query: query(P): X x
  mv = ver0
  mx = null
  for p ∈ P do
    if p.v > mv
      then mx = p.x; mv = p.v;
  let x = mx
```

Set 型 — Set は更新時に与えられた値を要素とする集合を保存するための型である．

2P-Set 型 — Set は更新時に与えられた値を要素とする集合を保存するための型であり，2P-Set(Two-Phase-Set) は要素の追加だけでなく削除も可能な集合である．2P-Set は add と remove の 2 つの更新メソッドが用意されている．2P-Set 型は下記の様に表すことができる．

```
payload: V v, set AS, set RS
  initial: ver0, ∅, ∅
update: add(V u, X x): V v, set AS, set RS
  let v = u
  let AS = {x}
  let RS = ∅
update: remove(V u, X x): V v, set AS, set RS
  let v = u
  let AS = ∅
  let RS = {x}
query: query(P): set S
  let S = ∪P.AS \ ∪P.RS
```

Storage のデータ構造

ペイロードはバージョンによって順序付けられる集合である．そのためペイロードの要素は Storage においてイベントの到着順序にかかわらずイベントの発生時刻に基づいたバージョン順に配置される．参照時はペイロードをスキャンしデータ型で規定される参照メソッドを用いてステートフルデータへの変換を行う．

カラム名	データ型	要素型
user_id		String
last_login_date	Register	String
items	Set	String

表 1.4.1　Phalanx のテーブル例

メタデータ

テーブルスキーマは Phalanx で扱うアプリケーションのデータ構造を定義するための記述である．表 1.4.1 はソーシャルゲームにおけるユーザのステータスを表すテーブルスキーマの例である．Phalanx のテーブルスキーマは主キーとステートフルデータの 2 種類のデータを扱うカラムによって構成される．主キーはレコードを一意に表す値でステートフルデータを参照する際に使われる．また複数のカラムを用いて複合主キーとしても利用可能である．ステートフルデータを扱うカラムは必ず 1 つのデータ型と紐付く．またデータ型のペイロードの要素値を規定する要素型も定義する．Phalanx では主キーを指定することでステートフルデータを参照できる．その際, ステートフルデータを扱うカラムに紐づくデータ型の参照メソッドに従ってペイロードをステートフルデータへと変換する．

更新ルールはストリームで流れてくるイベントをペイロードに変換する記述である．Phalanx ではデータ型によって順序に関係なくステートフルデータへの更新を行うことが可能である．そのため入力イベントをデータ型の更新メソッドに変換するようなシンプルな更新ルールを記述するだけで更新を実現することが可能になる．

4　Phalanx の実装

本節では Phalanx の実装について説明する．

4.1　HBase におけるペイロードのフォーマット

Phalanx の Storage には Apache HBase[*1]を用いた．HBase のデータは Cell という単位で構成される．Cell は Row, ColumnFamily, Qualifier, Timestamp, Value をフィールドとしてもつ．HBase の Cell の配置は行キー（Row）と列属性群（ColumnFamily）と列属性値（Qualifier）で決められる，また Cell は Qualifier の辞書順にソートされる．Phalanx の実装では Row に主キーの値とステートフルデータを扱うカラム名の順で構成されるバイト配列を与える．また ColumnFamily は常に単一の値を固定して使用する．そして Qualifier にバージョンを付与し Value にデータ型に合わせたペイロードの更新値もしくはスナップショットの集約値を与える．そのため Row ごとのペイロードはバージョンでソートされた状態で配置される．

4.2　ストリーム処理の到達保証

ストリーム処理を実装する際にイベントの転送の到達保証について考慮する必要がある．イベント転送の到達保証は大きく 3 つに分けられる．

- **At Most Once Semantics**
- **Exactly Once Semantics**
- **At Least Once Semantics**

At Most Once Semantics はイベントが最大で 1 回到達，Exactly Once Semantics はイベントが一度だけ到達, At Least Once Semantics はイベントが少なくとも 1 回以上到達を保証する．つまり At Most Once Semantics はイベントが転送途中で失われる可能性があり, At Least Once Semantics はイベントが重複して届く可能性があることを示す．

[*1] https://hbase.apache.org/

一番望ましいのは Exactly Once Semantics であるが，Exactly Once Semantics を実運用で実現するためには厳密な送達制御が必要となり，送達制御のオーバーヘッドが性能劣化を引き起こすことが起こりうる．MillWheel [2] ではイベントにあらかじめユニークな ID を付与し ID による重複除去の機構を設けている．Phalanx では MillWheel を参考に，イベントの到達保証が At Least Once Semantics のときでも同じイベントに対する更新を繰り返し行わない仕組みを設けている．Phalanx の入力イベントには必ずユニークな UUID の付与を必須と定めており，ペイロードのバージョンにはタイムスタンプに加え UUID を追記している．もし重複するイベントが Updater Layer に届いてもペイロードは同じバージョンになるため，HBase の同一の Cell として上書きされる．そのため Phalanx では参照時にはペイロードの重複が除去されていることを保証する．

4.3 更新ルールの実装

```
# イベント内のフィールドをで指定し変数として宣言JQ
params:
  type: ".type"
  user_id: ".user_id"
  time: ".time"
# 条件に合ったイベントをペイロードに変換
when:  $type == "login"
tables:
  # テーブル指定
  - table_name: user_status
    find:
      # 主キー指定
      - by:
          user_id: "$user_id"
        # ステートフルデータへの変換を記述
        apply:
          - method: set
            column_name: last_login_date
            args: $time
```

図 1.4.4 更新ルールの例

更新ルールは入力ストリームから受け取ったイベントをペイロードに変換する．入力ストリームからはさまざまな種類のイベントが Updater Layer に届くため，適切なイベントを選択して更新操作に変換できるようにする必要がある．

図 1.4.4 は更新ルールを YAML 形式で記述した時の例である．1 つの入力ストリームから取得したイベントを user_status テーブルへの更新操作に変換する．このとき入力イベントは JSON フォーマットであることを仮定している．この更新ルールではイベントから更新値を抽出する際に jq[*2]を JSON 操作用 DSL として利用している．

5 まとめ

ステートフルデータをストリーム処理で扱うための処理基盤 Phalanx を開発した．Phalanx ではイベントの到着順が保証されないストリーム処理においてステートフルデータの変更を可能にするために更新時にステートフルデータを変更するのではなく参照時に更新操作を集約しステートフルデータに変換するデータ型を用意した．このデータ型では結果整合性を保証する．Phalanx のデータ型は順不同で更新が可能であるのでステートフルデータの変更をイベントとデータ型への操作を対応するだけの平易な更新ルールを記述するのみで実現可能にした．

参考文献

[1] Marc Shapiro, Nuno Preguica, Carlos Baquero, and Marek Zawirski. Conict-freereplicated data types. In Proceedings of the 13th International Conference on Stabilization, Safety, and Security of Distributed Systems, SSS'11, pp.386–400, Berlin, Heidelberg, 2011. Springer-Verlag.

[2] Tyler Akidau, Alex Balikov, Kaya Bekiroglu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, and Sam Whittle. Millwheel: Faulttolerant stream processing at internet scale. Proc. VLDB Endow., 6(11):1033–1044, August 2013.

[*2] https://stedolan.github.io/jq/

2

機械学習システム

機械学習を計算機システムに組み込むにあたっては，学習アルゴリズムと訓練データによる無数の組み合わせから生まれる学習済みモデルを統合的に管理しながら開発および運用・保守を行う必要がある．ここではそのための秋葉原ラボの取り組みとして，フィルタ基盤の開発の実践およびその設計思想と，機械学習に特有なモデル管理の課題とその解決のために開発している基盤システムについてそれぞれ紹介する．

Akihabara Lab

2.1 フィルタ基盤の開発における取り組みと利用されている技術

Connolly, Juhani

概　要　秋葉原ラボではさまざまなデータ活用システムを開発・運用しており，共通して必要となる機能としてコンテンツのフィルタリングがある．データ活用の普及に伴って，フィルタリング処理のスケーリングや管理の効率化が課題となっていた．そこで，秋葉原ラボでは，フィルタリングロジックの分散処理やリソース利用の効率化を可能にし，新しいフィルタの開発やデプロイを容易にするフィルタ基盤を開発した．本稿では新たに開発したフィルタ基盤の開発・運用や今後の改善点について紹介する．

Keywords　Filtering，Microservices，CI/CD

1　フィルタ基盤の背景

インターネットサービスを運営する上で必要不可欠な処理としてフィルタリングがある．当社でも ABEMA のコメントや Ameba ブログの有害コンテンツ・スパム投稿などを防ぐため，それらの検知機能をもつフィルタを開発している．フィルタが扱うデータはテキストデータ，画像データ，機械学習の特徴量データなど多様であり，ロジックもシンプルなルールベースのものから機械学習を使うものまでさまざまである．

秋葉原ラボではさまざまなフィルタを開発してきたが，フィルタの増加に伴い，フィルタ処理や機械学習モデルの管理の効率性が課題となっていた．たとえば，複数のフィルタが混在するシステムでは，1 つのフィルタロジックの変更によりほかのフィルタが影響を受ける可能性がある．また，複数の機械学習モデルが共存していると CPU やメモリなどの計算機リソースのスケジューリングも課題となる．

これらの課題に対応するために，フィルタ処理を効率的に管理するためのシステムの基盤化を行った．既存のシステムは，1 つのサーバですべてのリクエストを集中的に処理しており，スケーラビリティに課題があった．また，モノリシックな構成で 1 つのアプリケーションにさまざまなフィルタ処理やそれらに必要なモデルデータが混在し，管理が複雑化していた．そこで，以下を目的とした新しいシステムの開発を行った．

- 複数のフィルタの連続処理を定義・管理・実行する，簡単にスケールできるサービスを提供する．
- フィルタの開発・デプロイ・デバッグを簡単にし，インフラを開発者に意識させないようにする．
- リソースの利用を効率的にし，利用状況を把握できるようにする．
- フィルタごとにリクエスト数が異なるため，それに応じてリソースの割り当てを最適化する．
- 特定のサービスからのリクエストがスパイクして全体のパフォーマンスと可用性に影響を及ぼすことを防ぐ．

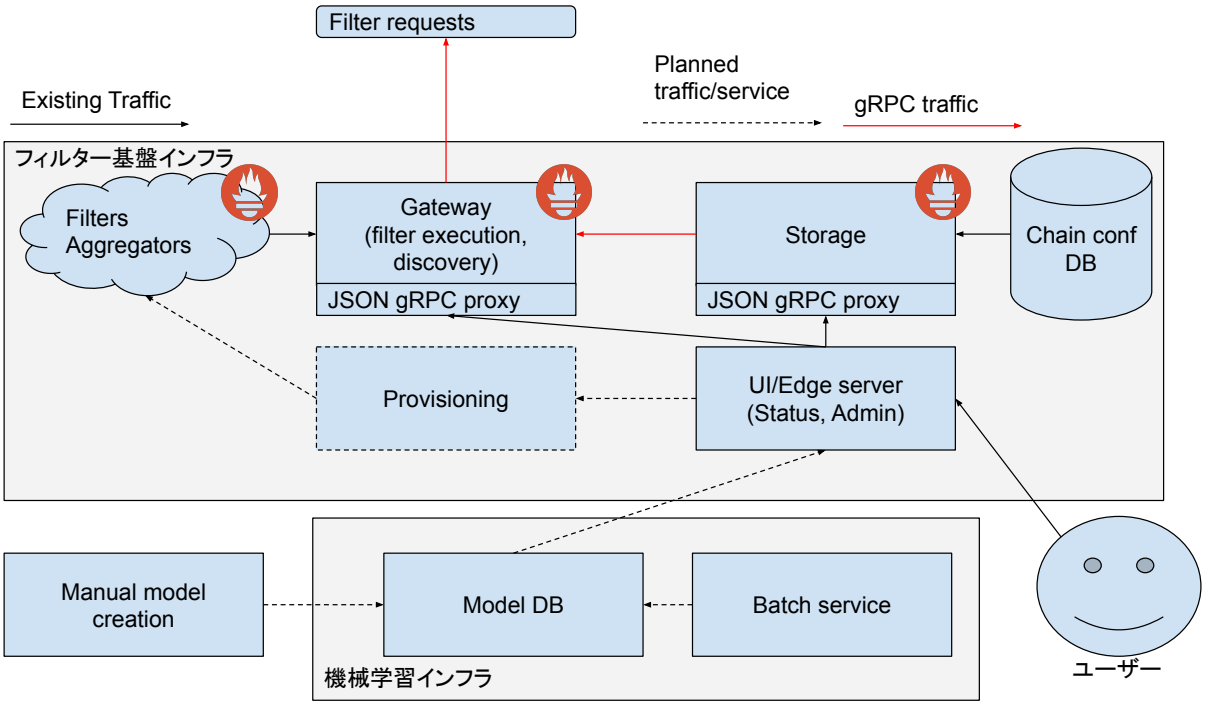


図 2.1.1　フィルタ基盤構成

1.1　マイクロサービスと UNIX Philosophy

フィルタ基盤の開発にあたっては，UNIX Philosophy[*1]に基づき，サービスだけでなく開発プロセスの分割も明確にした．現在，大規模なシステムを複数のパーツに分けるマイクロサービス型のシステムアーキテクチャが注目されている．このマイクロサービスというコンセプトのメリットは，大規模なシステムを複数の小さなモジュールに分ける事により，複雑なシステムを理解・メンテナンスしやすい小さなシステムに分解させる事にある．

これは，UNIX Philosophy の中心的な言葉である “Design programs to do only a single thing, but to do it well, and to work together well with other programs.” をインフラとアーキテクチャに適用したものとみなせる．

2　システム構成

フィルタ基盤の構成を図 2.1.1 に示す．図 2.1.1 は，現在提供しているサービスに加え，今後対応予定のサービスも含んでいる．フィルタ基盤は複数のプロトコルとサービスから構成される．このプロトコルは Protocol Buffers/gRPC（3.1 項）で定義されており，各サービスはそれにしたがってコミュニケーションを行う．

以下各サービスについて説明する．

2.1　ゲートウェイ

ゲートウェイはフィルタ基盤の中心となるサービスである．このサービスはコーディネーターの役割を持ち，定義されているフィルタチェインを実行する．

この実行は以下の処理により構成される．

- フィルタのディスカバリ
- フィルタのグループの並列実行とその結果の集計
- メトリック収集
- フィルタのリトライ・タイムアウトとエラー処理
- 集計結果をベースにした処理フローの分岐

これらの処理はサブシステムに分離されており，

[*1] http://www.linfo.org/unix_philosophy.html

リプレイスが容易である．また，ディスカバリはインフラへの依存を防ぐためプラガブルになっている．

基本的に gRPC での利用を想定しているが，grpc-gateway[*2]を使ってプロトコル定義ファイルから JSON のプロキシを自動生成することにより，JSON でのアクセスも可能となる．

ゲートウェイの動き

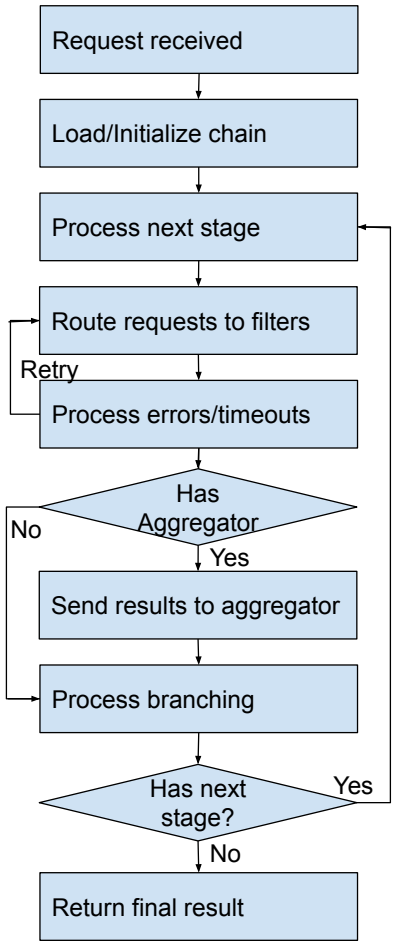


図 2.1.2 フィルタチェインの処理の流れ

ゲートウェイは複数のフィルタで構成されるチェイン（パイプラインとも呼ぶ）のオーケストレーションを行うシステムである．各チェインは複数のステージで構成され，各ステージは並行に処理される複数のフィルタからなる．図 2.1.2 にその流れを示す．また，ゲートウェイは，異なるサーバに存在するフィルタの実装を発見し，リモートフィルタのファクトリ機能を提供するディスカバリ処理をもつ．

フィルタ処理は，チェインやステージなどの各レベルごとに明確に役割が分離されたランナーにより実行される．

チェインレベルのランナーは，チェインのロードと初期化を行い，ステージのランナーへのリクエストの転送と処理結果の受信，およびクライアントへの返信の処理を行う．

ステージのランナーでは各フィルタのランナーへリクエストを転送し，その結果に対して必要な処理を行う． フィルタのランナーは定義されている外部のフィルタにリクエストを転送する．また，リトライやタイムアウトなどのエラーハンドリングをチェインの設定にしたがって実行する．

フィルタ処理の結果に対して集約処理が必要な場合は，対応するアグリゲーターの呼び出しと集約後の分岐処理を行う．アグリゲーターでは Min/Max/Mean などの組込みの集約処理に加えて，カスタムの集約処理を拡張することができる．また，必要に応じて，フィルタと同様に，リモートのアグリゲーターを用いて複雑な集約処理を定義できる．

分岐処理では結果に応じて次に実行すべきステージを決定し，チェインのランナーに通知する．次のステージを決定するために，BranchRule の確認を行う．BranchRule がない場合は次のステージを実行する．

2.2 チェインストア

チェインストアは，フィルタチェインの保存先であり，フィルタの具体的な実装を抽象化したものを提供する．また，フィルタチェインのバリデーションを行う．今後は，より複雑なバリデーションの対応や，チェインの変更検知などを実装する予定である．また，ゲートウエイと同じく，チェインストアは，gRPC もしくは JSON プロキシにより利用できる．

[*2] https://grpc-ecosystem.github.io/grpc-gateway/

2.3　UI/エッジサーバ

UI は SPA によって簡単にフィルタチェインの変更を行えるインタフェースを提供する．また，クラスタ外からの唯一のエントリポイントであり，リバースプロキシとしてほかのサービスの機能を外部に公開する．

2.4　フィルタ

フィルタ基盤における実際のフィルタ処理の実装はエンドユーザにより提供される．gRPC のプロトコルの実装とディスカバリの設定（実装しているフィルタの名義，サポートしているデータタイプなど）により，ゲートウェイはフィルタを検出することが可能となり，フィルタチェインで利用できる．

また，チェインの設定により各フィルタの設定を上書き可能であり，異なったパラメータでフィルタを再利用することも可能となる．また，フィルタ設定を変えるための再起動なども不要である．

3　利用技術

3.1　Protocol Buffers/gRPC

UNIX Philosophy における規範の 1 つとして明確なインタフェースがある．明確なインタフェースを確保するためには利用するものの開発のしやすさ，効率，周辺ライブラリの充実などが重要となる．また，複数の言語でコミュニケーションをするためには，特定の言語に依存しないシリアライズ方式が必要となる．

そこでフィルタ基盤では，データ（インタフェース）の定義に Protocol Buffers[*3]を用いた．Protcol Buffers は，Google 社のライブラリで複数のプログラミング言語間での構造化データの受け渡しを可能にする．また，gRPC[*4]を採用し，Protocol Buffers のデータ定義から生成されたクライアントやサーバインタフェースのテンプレートを基にフィルタ基盤を実装した．

これらを使う事によって，フィルタ基盤の開発を効率的に行うことができた．また，Protocol Buffers を用いることでデータ定義に後方互換性をもたせることが可能であり，データ定義に新しい項目の追加や削除があっても，変更前のインタフェースを用いてコミュニケーションを行える．このためクライアントとサーバのバーションミスマッチが発生しにくく，プロトコルの変更を計画的に行うことができる．

また，grpc-gateway ライブラリを用いることで，プロトコルの定義から JSON のプロキシを容易に生成できる．

OpenAPI

UI は TypeScript で書かれているため，サーバとは JSON 形式のデータでコミュニケーションできることが望ましい．

マニュアルで UI のコードを管理する場合，プロトコルが変更されるたびに更新が必要になる．
これを避けるため，grpc-gateway に加えて OpenAPI（旧 Swagger）の定義も生成する．OpenAPI の定義をベースに TypeScript の SPA のクライアントコードを自動生成を行う事ができ，JSON プロキシとのやりとりを効率的に実現できる．

gRPC と OpenAPI を使う事によって一つの定義で複数のシステム間のコミュニケーションプロトコルを明確にできる．また，周辺ツールが充実しており，プロトコルを拡張しやすいというメリットもある．インタフェースは明確かつ柔軟になっているため，利用コンポーネントの入れ替えが必要な場合も柔軟に対応できる．また，プロトコルに不備がある場合も，適切な計画を立てることによって変更や拡張はスムーズにできる．

3.2　Java/Spring/Jib

フィルタ基盤の開発では，ロジック，設定，パッケージングを分離しそれぞれの責務を明確にした．これにより，問題があった場合のメンテナンスや

[*3] https://developers.google.com/protocol-buffers
[*4] https://grpc.io/

リプレースを容易にした．

フィルタ基盤のコアの部分のアプリケーションコードはすべて Java で実装した．また，コアコードは特定のフレームワークに依存しないようにした．これにより，挙動が理解しやすく，明確なインタフェースをもったアプリケーションとして実装できた．

コア以外のモジュールでは，Spring をもちいたアプリケーションの設定とパッケージングを実装した．Spring の設定を 1 ヵ所に集約することで，モジュール間の関係の理解が容易になり，フレームワークの「魔法」に頼る所が少なくなる．

Spring Boot を使う一つの大きなメリットに Externalized configuration[*5]がある．これを使う事によって設定情報をパッケージング時に入れずに，デプロイ時に YAML/JSON ファイルや環境変数で設定できる．これにより，設定の自由度が高くなり，あらゆるインフラへのデプロイが可能になる．

Java のアプリケーションは，Google の Jib[*6]を用いることで Dockerfile など別の設定ファイルを記述することなく，コンテナ化してデプロイする．Jib を使う事によって，Docker のレイヤキャッシングの効率化や，変更がない時コンテナのハッシュも変わらないというメリットがある．

3.3 監視

フィルタ基盤は Prometheus とログベースによる監視を行っている．各コンポーネントでは Prometheus のメトリックの形でリクエストカウント，エラーカウント，同時リクエスト数，リクエストレイテンシなどのメトリックを記録している．フィルタ基盤は秋葉原ラボで開発・運用している Kubernetes クラスタ（以降，lab-k8s と呼ぶ．4.1 項参照）上にデプロイされている．lab-k8s では，Prometheus のサーバ立ち上げや管理をする prometheus-controller があり，監視を容易に管理できる．

適切なメトリックの利用により，システムの問題を容易に把握するだけでなく，デプロイフローの中での問題に対して，リグレッション検知を行ってロールバックも可能になっている（5.4 項）．

4 インフラの抽象化

前述のアプリケーションは，特定の環境に依存しないように実装されている．具体的にはアプリケーションの jar ファイルと，それを含む Docker イメージである．本節では，それらの Kubernetes 環境（lab-k8s）上での運用について説明する．

4.1 lab-k8s

秋葉原ラボは独自の Kubernetes 環境を開発・運用している．lab-k8s では，標準の Kubernetes に加えて，いくつかの Custom Resource Definition（CRD）がインストールされており，さまざまなタスクを効率化している．ただし，インフラの実装は変化する可能性があり，また，ほかの環境へのデプロイが必要となる可能性もあるため，本フィルタ基盤が lab-k8s の各機能を利用するかは切り替えることができる．

4.2 Helm

Helm[*7]は Kubernetes のパッケージマネージャである．1 つのアプリケーションとその依存などを抽象化し，パッケージ化するしくみを提供する．そのパッケージの中では Pod,Deployment,Service などのテンプレーティングを行い，エンドユーザからインフラの詳細を隠蔽し，1 つの設定ファイルに集約する．

フィルタ基盤では，このファイルに Spring の設定，Prometheus モニタリングの自動設定（CRD 依存あり）の有無，レプリカ数，DB の自動プロビジョニング（CRD 依存あり）などをまとめて

[*5] https://docs.spring.io/spring-boot/docs/1.2.2.RELEASE/reference/html/boot-features-external-config.html
[*6] https://github.com/GoogleContainerTools/jib
[*7] https://helm.sh/

いる．CRD のない Kubernetes 環境に移動しても，そのフラグをオフにするだけで lab-k8s 以外の Kubernetes 環境にも対応できる（ただし CRD でプロビジョニングされている DB などの導入や設定は必要となる）．

4.3 Kubernetes ディスカバリの実装

ディスカバリのシステムは抽象化されているが，実際に機能するかはインフラの実装に依存する．Kubernetes 環境では，ディスカバリは容易に実現できる．

まず，各フィルタのパッケージの設定にディスカバリの情報を含める．具体的にはポッドのメタデータのアノテーションとして設定する．そしてゲートウェイにより，Kubernetes の API にリスナを作成する．このリスナは Kubernetes のネームスペース内のポッドの変更を監視し，関連するアノテーションが付与されている場合，フィルタと認識して必要な処理を行う．

処理の内容としてはアダプタの処理を行ってディスカバリプロセッサに必要な接続情報（ホスト，ポート，フィルタの名称など）を送信する．プロセッサはマッチするフィルタの接続プールを探し（ない場合は作る），gRPC 接続を作成し，プールに追加する．フィルタが必要になった際に，FilterFactory はディスカバリプロセッサからリモートフィルタに必要なコネクションプールを取得し，フィルタを立ち上げる．

5 開発プロセス

1.1 項で述べた UNIX Philosophy は開発にも適用されている．

それぞれのリポジトリに明確な役割を与えることによって CI や CD の依存関係を疎にでき，CI や CD の実装に変更があったとしてもほとんどのコードに影響のない開発プロセスとリポジトリ構成を実現している．

開発プロセスの全体像を図 2.1.3 で示す．以降で詳細を説明する．

5.1 ビルドシステムとリポジトリ

フィルタ基盤の開発における主なリポジトリとして，アプリケーションのコードを管理するものがある．フィルタ基盤の開発においては，アプリケーションコードのリポジトリが，CI/CD のシステムやデプロイ先のインフラに依存する設定を含める必要がないように開発プロセスを設計した．

フィルタ基盤の大部分のコードは Java で実装されているため，現時点ではビルドシステムとして Maven[*8]を採用している．豊富なプラグインが提供されているため Java のコンパイルやテストだけでなく，Docker コンテナの作成やほかのビルダ（UI のコードのコンパイル）なども連携可能である．

アプリケーションコード以外のリポジトリとしては，インフラコードやデプロイのコードを管理するリポジトリがある．これらは，アプリケーションコードと依存関係をもたないように設計した．

インフラコードの管理を分離することによって，たとえば Kubernetes から VM の環境に移行する事があったとしても，アプリケーションコードの変更を不要にできる．別の環境への移行は，新環境用のインフラのコードだけを追加することで実現できる（ただしインフラに依存するディスカバリの実装の追加は必要になる可能性がある）．また，アプリケーションの変更など，さまざまなコミット履歴が混在されず，変更履歴が理解しやすいメリットもある．

同様に，デプロイのコードも独立に管理されている．CD のシステムが変われば，そのシステムの設定方法なども変更が必要となる．デプロイのコードを分離することにより，CD の実装とアプリケーションの疎結合を維持できる．

この分割によって各リポジトリの関心を分離させることができる．

[*8] https://maven.apache.org/index.html

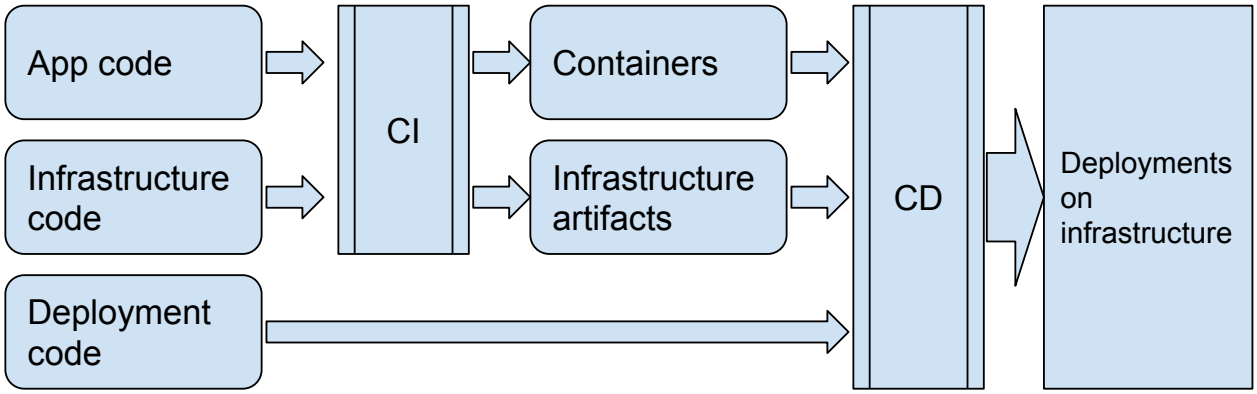


図 2.1.3 存在するリポジトリの種類と CI/CD の関係

本来のビルドシステムの問題

Maven では豊富なプラグインが提供されているが，Java 以外の言語のコンパイルは最終的に Exec plugin を使って実行する．このように，マルチ言語のプロジェクトでは Java 以外で書かれているソースは低いプライオリティーになり，可能なビルド処理に制限がある．

また，大規模なリポジトリではビルドにかかる時間が問題となる．モジュール間の細かい依存関係の定義は困難なため，ビルドがプロジェクト単位など大きな粒度で行われることが一般的である．単一の言語で書かれていないリポジトリでは Maven で詳細な依存情報を定義するのは不可能なため，システムによって変更のないアーティファクトまでビルドやテストが実行され，それらのパッケージが各リポジトリにプッシュされてしまう．

スクリプトなどによって一部のサブモジュールのビルドを除外するのは可能だが，それを開発するのは困難で新しい問題の原因となりうると考える．

全アーティファクトがプッシュされる事によって，CD にトリガなどがセットされている場合，本来不要なアーティファクトまでデプロイが実行されてしまう．このため，無駄が多く，ソースコードの変更からインフラで反映されるまでの間が長くなり，CI/CD による効率化が十分に行えない．

今後の可能性・Bazel

以上の問題に対して，Google は数年前に Bazel[*9] というシステムをオープンソースにした．

このシステムでは細かいレベルで依存関係を定義する事ができ，大きなリポジトリのビルドを効率化できる．変更箇所とその依存部分のみのビルドだけが実行されるようになっており，中間アーティファクトは決定的に作られているため，ビルドキャッシュをそのまま使うことが可能で無駄なくビルドを行う事ができる．変更に直接かかわる箇所のみの再ビルドが実行されることにより，ビルドやリリースの速度を向上させ，問題のあるコードも早期に発見できる．

ただし，Bazel への移行はコストが高く現在進めていない．

5.2 CI

CI の役割は再現可能な形ですべてのビルド，ユニットテストとパッケージングを行う事である．これを実現するためにはその実行を Docker コンテナ内で行う．フィルタ基盤は特定の CI ツールに依存しない構成であるが，ユニットテスト内の Docker の利用があるため，それをサポートしている CircleCI を採用している．

5.3 CD

デプロイの効率化と早期の問題の発見のために最近 CD（Continuous Delivery もしくは Contin-

[*9] https://bazel.build/

uous Deployment）が注目されている．

フィルタ基盤では，Harness[*10]を用いて CD を実現している．Harness はほかのコンポーネントと同じく適切な抽象化がなされており，既存のインタフェースを変えずに組み込むことができる．今後問題があったとしても，ほかのコードに影響なくリプレイスできる．

CD の目的

フィルタ基盤の CD では以下を目的とした．

- 自動化されたデプロイによるデプロイの操作ミスの防止
- デプロイフローにおける問題を防ぐためのスモークテストの統合
- エンドユーザからインフラを隠蔽するためのデプロイの抽象化
- デプロイの方法のバージョン管理（いわゆる GitOps に近い）
- rollback 可能なデプロイ戦略の実現

Harness での実装

前述のようにアプリケーションとインフラレイヤのコードのリポジトリとは別にデプロイ（CD）のリポジトリを構築してある．

デプロイリポジトリはハーネスのすべての設定を管理する．ハーネスではいくつかの抽象化された機能があり，それらを使う事によって再利用可能なワークフロー（1 つのサービスのデプロイの方法）とパイプライン（複数のワークフローの連携）を定義できる．

サービスレベルではサービスが使うコンテナイメージ, Helm パッケージの設定の overload（Helm の Values ファイルの中身）を定義している．また環境に特化した設定（たとえば利用するリソース・デバッグ設定）は，Values の環境別の overload により実現できる．

[*10] https://harness.io/

5.4 問題のあるリリースを避ける方法

CI の中でユニットテストやインテグレーションテストを行っているが，それらで検知できない問題もある．たとえば，ディスカバリのコードに問題があれば，Kubernetes 環境にデプロイされるまで問題が顕在化しない場合がある．また，パフォーマンスのリグレッションなどは CI の中でのユニットテストで検知することは難しい．

このような問題を検知するために，ステージング環境のデプロイを行う際に，Harness の Verification 機能を使って，負荷テストを組み込んでいる．負荷テスト中に Harness はログとメトリック（Stackdriver Logging/Prometheus のもの）を収集し，過去の実行と比較する．これらに問題があれば（たとえばスループットの低下），デプロイが rollback され，プロダクション環境へのデプロイが中止される．この取り組みによってリグレッションなどの問題をもつデプロイを防止することが可能になっている．

6 まとめ

本稿では秋葉原ラボで開発しているフィルタ基盤について紹介した．各レベルでは適切な抽象化を意識し，各システムとサブシステムのカプセル化を行っている．開発プロセスではコードベースをアプリケーション，インフラ，デプロイで分離し，メンテナンス性を向上させている．CI/CD のプロセスの中にテストを組み込むことによって，さまざまな段階で問題のあるコードを早期に検知することが可能になっている．開発プロセスについてもメンテナンスのしやすさを意識した設計をしており，新しいテストの追加が容易なフレームワークが実現できている．フィルタ基盤は継続的に開発していく予定であるが，シンプルなインタフェースで作る事によって，問題のある部分も容易に改善していくことができると考える．

今後は，より利便性を高めるために，いくつかの拡張を行う予定である．まず，秋葉原ラボで開発

しているモデル管理システムとの連携により，フィルタが使う機械学習モデルの管理を効率化することがある．さらには，特定のモデルを簡単なインタフェースで即時にプロビジョニングするサービングのシステムを検討する予定である．また，UIにはまだ改善の余地が多い．

Akihabara Lab

2.2 データと処理の依存関係を整理する機械学習モデル管理基盤の開発

大内 裕晃

概　要　秋葉原ラボではコンテンツ推薦，動画や画像の特徴量抽出，自然言語処理などのシステムで機械学習を活用している．機械学習モデルの再現性を担保するためのバージョン管理では，通常のソフトウェア開発のようにコードを管理するだけではなく，学習に利用したデータセットやほかの機械学習システムの出力を利用しているといった依存関係を含めて管理を行う必要がある．そこで，機械学習モデルの開発に関連する依存関係の整理を自動化するため，我々は機械学習モデル管理基盤 Etna を開発した．Etna は抽象化された機械学習のデータフローに基づき，モデルのバージョン管理およびデータの依存関係を管理するシステムである．本稿ではモデル管理基盤となる Etna 開発の背景やその機能について紹介する．

Keywords　機械学習，モデル管理，機械学習工学

1　背景

近年，様々な分野で機械学習システムが活用されるようになった．秋葉原ラボでもコンテンツ推薦や動画の特徴量抽出，不適切なコンテンツのフィルタリングなどに機械学習システムを用いている．しかし，フレームワークなどの発展により機械学習システムの開発や導入は容易になりつつある一方，長期的に機械学習システムを維持することは依然として困難であることが多い．従来のソフトウェア開発ではプログラムの変更に伴う保守性を向上させるために，技術的負債 [1] を定義しそれらを避けるための取り組みが行われていたが，機械学習システムの開発ではこの技術的負債を回避するための手法を適用することが困難であることに加え，機械学習システム特有の技術的負債も発生する．たとえば従来のソフトウェア開発におけるバージョン管理では最終的な成果物を作成するために用いたプログラムコードを Git[*1] などのバージョン管理システムで管理するだけで十分であったが，機械学習モデルの開発ではプログラムコードに変更がない場合でも用いるデータやハイパーパラメータ，実行環境が変わることで得られるモデルの性能が変化するため，プログラムコードのバージョン管理だけでは不十分である．また，機械学習システムにおける技術的負債の例としてデータの依存関係や宣言されていない消費者の存在がある [2, 3]．機械学習システムにおけるデータの依存関係はソフトウェア工学におけるコードの依存関係と比較できる．従来ではプログラムコードの更新を追従したり静的解析を用いることで依存関係を抽出できるが，機械学習モデルの更新ではプログラムコードの更新だけでなく，学習スクリプトの変更，学習データの変更，ハイパーパラメータの変更などの要因が挙げられ，これらの解析は困難である．データの変更についても前処理を変更した場合，生データを変更した場合などがあり，

[*1] https://git-scm.com/

コードの依存関係よりもデータの依存関係の方がより複雑である．このようなデータの依存関係を適切に管理できていない場合，モデル生成の再実験が困難になり，最新のモデルや実際に運用したモデルを作成した際に利用した学習データを誤って削除してしまうといった運用上の課題が発生する可能性がある．宣言されていない消費者は，ある機械学習モデル（親モデル）の出力を入力として学習や推論を行う機械学習モデル（子モデル）のことである．機械学習モデル間にこのような依存関係があるとき，親モデルの更新によって出力が変更された場合には子モデルには意図しない影響が発生する可能性がある．この依存関係による影響を把握するためには，子モデルの開発者は親モデルの更新を検知し内容を把握する必要があるが，それぞれのモデルが異なる開発者によって管理されている場合には検知が難しく，特に親モデルの開発者は依存をもつ子モデルが増えることで親モデルの更新時に考慮すべきことが増え保守性が低下することもある．

これらの課題を解決するために，我々はデータの依存関係を含めた状態で学習済みモデルを管理する基盤である Etna を開発した．Etna は学習済みモデルや学習データをバージョン管理し，モデル学習時に利用したコードのバージョンや学習データ，親モデルのバージョンを保存する．学習済みモデルと学習時の情報を管理することにより実験の再現性の向上やモデル間の依存関係を容易に把握できる．

本稿では，2 節で機械学習システムにおける問題点を具体的に述べる．次に 3 節で既存の機械学習ツールと比較を行い，さらに 4 節で課題を解決するためのデータフローの抽象化および Etna の設計と機能について説明する．そして 5 節でまとめと今後の課題を述べる．

2 機械学習におけるデータ依存の課題

機械学習はデータセットを利用し，それを学習プログラムで処理することでモデルを作成しているため，モデルはプログラムコードだけでなく利用するデータの変更によってその精度が変動する．特にデータの変更については CASE（Changing Anything Changes Everything）という法則 [3] があり，データ数の増減や分布の変更などわずかなデータの変更であっても，その予測の結果に大きな影響が出る可能性があるとされている．そのため，機械学習システムの推論の結果が意図したものでなくデバッグを行う際には，システムで利用されたモデルの学習プログラムコードにバグが含まれていないかだけではなく，どのデータを利用して学習したものなのかを知ることは重要である．しかし，このような情報は学習を行った環境にしか残っておらず，また機械学習エンジニアが自身の使いやすいような環境や使い捨ての仮想環境で行うため，属人化してしまうといったことが起こってしまう．その結果，実験の再現性が担保されなかったり，モデル間の依存関係が不明瞭になったりする課題がある．以下，これらの 2 つの課題について述べる．

2.1 実験の再現性が担保できない

1 点目の課題はモデル作成時に利用した学習データやスクリプトが把握できず，同じ性能のモデルを作成できないことである．機械学習モデルの開発では期待する精度のモデルが得られるまでパラメータやプログラムを変更させながら試行錯誤を行うため，開発中にどのようなパラメータで学習を実行したかを管理する必要がある．このような実験時の情報が管理されていない場合，別の開発者が追実験できずモデルの開発が属人化してしまったり，デバッグが困難になるといった問題が発生するため，実験の再現性を担保することは機械学習システムの運用上重要となる．実験を再現するためには，学習に利用したデータセット，プログラムコード，実行時のハイパーパラメータの情報が必要になる．またデータセットについては，単にデータセットのバージョンやデータの保存場所を管理するだけではなく，そのデータセットを生成するための前処理に問題がある場合も考慮し，前

処理プログラムの情報や前処理前の生データの情報も同時に管理する必要がある．

2.2　モデル間の依存関係が不明瞭

2 点目の課題はデータを通じたモデル間の依存関係が不明瞭になってしまうことである．学習済みモデル A を作成するために，別の機械学習モデル B から出力されたデータを学習データとして利用している場合に，その 2 つのモデルはデータを通じて依存関係をもつことになる．依存関係をもった機械学習のデータフローではモデル A の開発者，モデル B の開発者双方に負担が発生することがある．モデル A の開発者はモデル B が更新を検知できず，モデル A を更新する際に最新のモデル B を用いたデータセットが利用できるが古いデータセットで学習を行ってしまうため，精度が向上せず，効率的に学習を行うことができないといった問題が発生する．一方でモデル B の開発者はモデル B を更新する際に依存をもったモデルに対する影響分析ができず，モデルの変更や更新が困難になるといった問題が起きる．

3　既存ツールとの比較

2 節で挙げた課題は既存のモデル管理サービスである MLflow[*2]，Comet[*3]や ModelDB[*4]といったクラウドサービスや OSS ですべてを解消することが困難であった．既存のモデル管理ツールは学習実行時のデータとハイパーパラメータの管理に特化しており，モデル間の依存関係を把握できない．データについても学習実行時にスナップショットを保存するだけであったり，ファイル名を保存するのみであり，そのファイル自体がどのように作成されたのかまでを追跡することは困難である．またシステムは公開されていないが，Amazon が設計を公開しているシステム [4] と比較すると，データセットのバージョニングを行う点では同じだが，同様にモデルの生成過程しか管理できず，前処理など機械学習に関連する手続きを汎用的に管理できない．Etna はモデルだけでなくデータセットの作成過程も含めて管理でき，より汎用的に機械学習のデータフローを管理できる点で既存ツールより優れている．

4　Etna の概要

Etna は機械学習におけるデータフローをデータとデータへの処理という 2 つの観点から抽象化し，それらのバージョン管理および関連を管理するシステムである．開発者が作成した機械学習モデルと関連するデータセットと学習プログラムの情報とを関連付けて保存することで，機械学習のフローを容易に把握できるようにしている．本節では，Etna における機械学習データフローの抽象化の方法，システムの設計および依存関係を表示するグラフについて説明する．

4.1　機械学習データフローの抽象化

2 節で挙げた課題を解決するためには，機械学習モデルの作成に利用したデータセット，プログラムコード，ハイパーパラメータを管理する必要がある．さらにデータセットに加工を加える前処理や，機械学習モデルの推論結果を用いた学習もあることから，前処理，推論といったデータへの処理も管理することが求められる．しかしモデルやデータの定義は開発者や使用するライブラリによって異なりあいまいである [5]．それらすべてを個別に管理するようなシステムを開発・維持していくことは難しい．そこでデータセットやハイパーパラメータ，モデルファイルなどをデータとして，データへの処理を加える関数をプロセスとして抽象化し，汎用的にモデルの作成過程を管理できるような設計を目指し，次のようなデータ，プロセスの概念を導入する．これにより，前処理や学習，推論といった機械学習に関わる処理を，入力と出力をもつものとして抽象化できる．

[*2] https://mlflow.org/
[*3] https://www.comet.ml/site/
[*4] https://github.com/VertaAI/modeldb

データ

機械学習を行うために必要なデータセットやハイパーパラメータ，学習の結果によって得られた学習済みモデルファイル，推論結果やモデルの評価結果をデータとして定義する．データセットや機械学習モデルファイルをデータという同一の形式で抽象化している．これによって，前処理済みのデータセットの作成過程（処理前のデータ，処理内容が記述されたコードなど）やモデルの作成過程（データセット，学習処理が記述されたコードなど）を，「データ作成に利用したデータ」の管理として同一の方法で管理することが可能になる．

プロセス

入力をデータ，出力をデータとするような関数をプロセスとして定義する．機械学習においては前処理，学習，推論，学習済みモデルの評価などがプロセスに相当する．たとえば，ある文書から空白・改行を除く前処理では処理したい文書ファイルを入力，処理済みの文書ファイルを出力とするプロセスとして扱うことができ，機械学習モデルを利用してブログ内容が不適切コンテンツかを判別する推論では，学習済みモデルファイルとブログのコンテンツを入力，不適切コンテンツである確率を出力とするプロセスとして扱うことができる．なおプログラムの中には，プログラム中でランダムに文字列を生成するものや，データベースへの接続設定やクエリがハードコーディングされているようなものが存在する．このように実行に必要となる入力データが存在しない場合には例外的に入力のデータが存在せず出力のみをもつプロセスとして扱う．

4.2 Etna の設計

図 2.2.1 に Etna のデータベース設計の一部を示す．データはテーブル `Data` によってそのバージョンやフォーマット，保存場所などが管理される．また，ハイパーパラメータやモデルの評価結果などデータがファイルとして保存されないものについてはカラム `content` にデータの内容を保存する．データの名前やそのデータ自身が生データなのか加工済みのものなのか，モデルなのかといった種別の情報はテーブル `DataSeries` によって管理される．同様にプロセスはテーブル `Process` によってバージョンやコードの情報，そのプロセスの入出力型の情報を管理する．入出力型の情報は省略可能だが処理の内容を理解する際に開発者を助けるほか，登録された情報を基に 2 つのプロセスを連続して処理可能かを判別することに使われる．データとプロセスを利用したという情報はテーブル `Execution` によって管理される．`Execution` は利用したデータセット，プロセスに加えて実行したコマンドラインの情報も管理し実験の再現性を向上させている．またデータが特定の `Execution` によって作成されたという情報はテーブル `Data` のカラム `ExecutionId` によって管理される．この設計は Amazon のシステムでの例を参考にしている [4].

4.3 データ依存関係を示すグラフ

Etna は機械学習モデルに登録されたデータ，プロセスの依存関係に基づき，その作成過程のグラフを描画する．例として次のような機械学習モデルの開発を考える．

1. 学習用データセット（text-dataset）を用意する
2. text-dataset を用いて機械学習モデル A を作成する
3. 入力用データセット（raw-dataset）を用意する
4. raw-dataset と機械学習モデル A を用いて学習用データセットとテストセットを作成する
5. 学習用データセットとテストセットを用いて機械学習モデル B とそのメトリクスを得る

これを Etna に登録した場合に得られる依存関係グラフを図 2.2.2 に示す．図の中で四角形のノードはデータ，円形のノードはプロセスを示しており，データ名およびバージョンが記述されている．プロセスの左側にあるエッジはプロセスを実行した際に入力として利用されたデータを表し，右側にあるエッジは出力されたデータを表している．

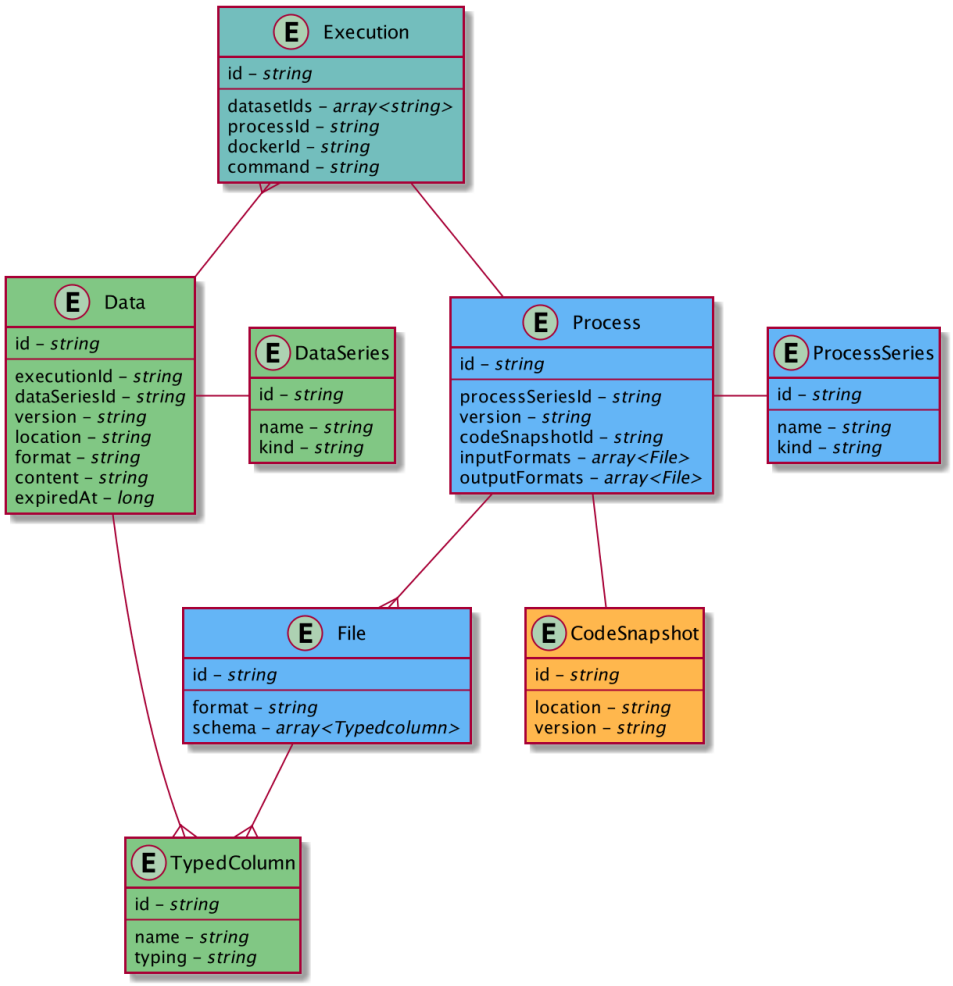


図 2.2.1　Etna の ER 図

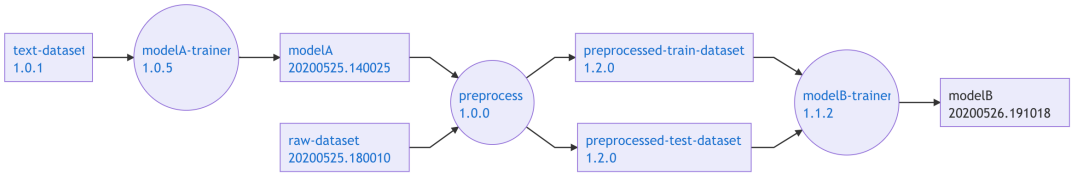


図 2.2.2　Etna でのデータ依存関係の表示例

これによって，機械学習のデータフローをデータとプロセスが交互に連続する 2 部グラフとして可視化することが可能になる．たとえば，機械学習モデル B の作成およびメトリクスの取得には 2 つのデータが利用されており，それらのデータがどのように作成されたのかといった情報が可視化されている．なお Etna は登録および依存関係表示のための Web インタフェースを備えており，登録に際しては Web インタフェース上からの登録以外に Python モジュールを介した登録，CLI ツールを介した登録の方法が存在する．

5　まとめと課題

本稿では機械学習におけるデータの依存関係を管理するためのシステムである Etna を紹介した．Etna は機械学習エンジニアが作成したモデルおよびデータセットのバージョン管理を行うこと，機械学習モデル間の依存関係の把握のために活用されている．

今後の課題としては，モデル間の差分抽出や機械学習の自動化に向けた取り組みがある．モデル間の差分抽出は 2 つのバージョンの間でどのような差異があるのかを簡潔に示すことである．たとえば更新前後のモデル間でメトリクスの向上が見られたとき，その要因がデータセットの変更なのか，プログラムコードの変更なのか，もしくはハイパーパラメータの変更なのかを示すことで，モデル精度向上の要因を簡単に知ることができるようになる．また，メトリクスの管理については 2 つのモデル間のメトリクスが比較可能なのかといった情報も重要である．たとえば評価に用いたデータが異なるなど同一の条件で評価が行われていない場合，それらのモデル性能を比較することは困難であり，評価の条件も含めてモデルを管理することで，モデルを正しく比較できることをモデル管理基盤を通じて担保することが可能になる．4.1 項で述べたように，Etna はデータやモデル，ハイパーパラメータ，メトリクスといった機械学習で扱われるものを同一のデータという概念で抽象化して管理している．そのため，メトリクスの変化であってもハイパーパラメータの変化であっても Etna 上ではデータの差分として扱うことができ，それを表示するようなインタフェースを提供すれば良い．また扱うデータの種別によって確認したい差分の内容が異なる場合には，それに応じてインタフェースを変更すれば良いため拡張しやすいといえる．

機械学習の自動化に向けた取り組みとしては，登録された依存関係情報の活用や利用状況の反映が考えられる．依存関係情報の活用については，現在の Etna では依存関係を依存関係グラフの描画にしか用いていないが，データの依存関係に基づき親データセットや親モデルの更新があった時に，モデルの再学習を実行したり，開発者に再学習のタイミングを通知するなど，モデルの継続的な更新の負荷を下げることで機械学習システムの安定化に貢献することも可能である．利用状況の反映については，機械学習モデルをデプロイする手順を標準化したうえで，本番環境やテスト環境で適用されているといったシステム運用上の情報をモデル管理基盤上に反映させることが考えられる．実際の適用状況を反映することでモデル開発者の意図しないモデルが本番に適用されてしまうことを検知しやすくするほか，それを防止することが可能になる．これを実現するためには，Etna とモデルをデプロイするシステムとの連携が必要だといえる．今後はこれらの課題の解決に取り組む予定である．

参考文献

[1] Ward Cunningham.: The wycash portfolio management system, Addendum to the Proceedings on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), pp.29–30, (1992).

[2] Sculley, D. et al.: Hidden Technical Debt in Machine Learning Systems, Advances in Neural Information Processing Systems 28 (NIPS2015), pp.2503–2511, (2015).

[3] D. Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, and Michael Young: Machine Learning: The High Interest Credit Card of Technical Debt, SE4ML: Software Engineering for Machine Learning (NIPS 2014 Workshop), 2014.

[4] S. Schelter, J.-H. Boese, J. Kirschnick, T. Klein, and S. Seufert.: Automatically Tracking Metadata and Provenance of Machine Learning Experiments, Machine Learning Systems workshop at NIPS, (2017).

[5] Sebastian Schelter, Felix Bießmann, Tim Januschowski, David Salinas, Stephan Seufert, and Gyuri Szarvas: On Challenges in Machine Learning Model Management, IEEE Data Eng. Bull., vol.41, pp.5–15, (2018).

3

画像処理

コンピュータビジョン（CV）の分野は畳み込みニューラルネットワーク（CNN）などの深層学習技術の普及により急速に発展しており，多くのタスクで目覚ましい成果が挙がっている．しかし，深層学習技術を実社会に応用することは簡単ではなく，データセットの品質や実社会の問題とのギャップ，実際のシステムへの組み込みなどの問題がある．これらの問題に対する取り組みを，実環境での実例を交えて紹介する．

Akihabara Lab

3.1 マッチングサービスにおける画像審査の自動化

上岡 将也

概　要　メディアサービスを運営する上でサービスの健全化は重要な課題である．当社のメディアサービスの多くはユーザによるコンテンツ投稿が可能であり，不適切なコンテンツの投稿により健全性が損なわれる危険がある．当社ではそれらのサービスの健全化のためにユーザが投稿したコンテンツを監視している．秋葉原ラボでは監視を効率的にかつ精度よく行うためにルールに基づくフィルタや機械学習を用いたフィルタを開発・運用している．本稿では，当社のマッチングサービスであるタップルで投稿されたプロフィール画像の審査を自動化するために開発したフィルタを紹介する．これは現在も実際に導入されており，複数の機械学習技術を組み合わせることで高い適合率を維持しながら，審査結果のブレの防止や監視コストの削減，審査時間の短縮などに貢献している．

Keywords　機械学習，画像処理，監視，CNN

1　はじめに

当社のメディア事業では，ユーザがコンテンツを投稿できるサービスを数多く展開している．たとえば，テレビ＆ビデオエンターテインメント ABEMA では，リニア配信中にコメントを投稿でき，ブログサービス Ameba ブログでは，テキストや動画像を含む記事やそれぞれの記事に対するコメントを投稿できる．

ユーザがコンテンツを自由に投稿できるサービスを運営する上では，不適切なコンテンツの投稿を監視し，サービスの健全性を維持することが重要となる．たとえば角田ら [1] は，Ameba ブログにおける不適切コンテンツの類型化を行い，その中でも比較的に多くの割合を占めているコピーコンテンツスパムを検出するフィルタを開発して Ameba ブログの健全化に取り組んでいる．

マッチングサービスではコメントなどのテキスト情報以外にもプロフィール画像の監視が重要である．プロフィール画像とはユーザがサービス利用開始時に登録する顔画像のことで，マッチングサービスのしくみ上，多くのユーザの目に触れるものであり，サービスの利用にあたっては必須となる．もし，顔のサイズが小さい画像や加工が強い画像などの本人を識別するには不適切な画像による登録ができると，ユーザからの信頼を失ってサービスの価値を損なってしまう．

当社のマッチングサービスであるタップルでは，プロフィール画像に対して厳格な基準をもうけて 24 時間体制で監視することによりサービスの健全性を維持している．しかしプロフィール画像審査には次のような課題があった．

- 多くの画像を 24 時間体制で迅速に審査するに

[*1] 監視オペレータの稼働時間だけを考えればよいわけではなく，厳密な基準の下に公平に監視できるように育成するコストやまたそういった人材を採用するコストも含む．

は，多くの人員を確保する必要があり，監視コスト[*1]が高い．
- 審査基準を明確にして監視オペレータへの共有を徹底しても，人の主観的な判断が入るために監視結果にばらつき[*2]が生じる．

これらの問題を解決するため，我々は機械学習を用いたフィルタを開発し，プロフィール画像審査の自動化に取り組んだ．

本稿では，まず 2 節でタップルのプロフィール画像審査やその基準について紹介し，そのフィルタを開発する上での課題を述べる．続いて 3 節では，タップルの画像審査自動化のために開発したフィルタについて説明する．次に 4 節でフィルタを用いた画像審査の処理の流れやフィルタ導入後の効果について紹介し，最後に 5 節で本稿を総括して今後の課題について述べる．

2　タップルの画像審査

タップルでは，なりすましや写真の悪用等を防ぐためにプロフィール画像の審査を行っている．本節では，タップルのプロフィール画像審査やその基準について説明し，画像審査を自動化する上での課題を述べる．

2.1　審査基準と監視作業

画像審査を公平かつ的確に行うためには，審査基準を適切に決める必要がある．公開されている一部の否認例[*3]について表 3.1.1 に，具体的なサンプル画像を図 3.1.1 に示す．実際には，表 3.1.1 のような公開している基準よりも細かく厳密に決められた基準を下に審査を行っている．

監視オペレータは，投稿されたプロフィール画像が承認か否認かを決めるだけでなく，否認する場合は，否認した理由を基準の中から選択する．これは，否認理由により投稿の悪質度が異なるので，それに応じてサービス側の対応を変える必要があるためである．

2.2　フィルタ開発の課題

1 節で述べたとおり，タップルのプロフィール画像審査では，監視コストやオペレータによる監視結果のばらつきが問題となっていた．我々はそれらの問題を解決するために機械学習フィルタの開発を検討したが，検討段階で次のような課題が予想された．

- 課題 1：解くべき問題は，ユーザが投稿した画像を承認すべきか否認すべきかを判別する 2 クラス分類だが，画像処理や機械学習の技術を用いてこの問題を解こうとすると，審査基準が多様であるために，そのすべてを満たせる 1 つの機械学習モデルを作ることが難しい．
- 課題 2：一部の否認理由に該当する画像において，外部の要因でデータの分布が頻繁に変わることが考えられる．たとえば，顔写真の加工アプリケーションなどによって加工された画像がそれにあたる．画像が加工されたものかを判別するモデルを学習させて判別できたとしても，加工アプリケーションのアップデートで新たな加工処理が追加された場合，データの分布が変わってしまい，以前のモデルではうまく判別できなくなってしまう．

3　機械学習フィルタ

今回開発したフィルタの概略図を図 3.1.2 に示す．2.2 項で述べたフィルタ開発の課題 1 は，図 3.1.2 のように，基準に沿った複数の機械学習技術を組み合わせてフィルタを構築することで解決した．課題 2 は，アップデートの頻度を考慮してフィルタを分割することで解決した．本節では，それらの解決方法の詳細を具体例を用いて説明する．

3.1　基準に沿ったフィルタの構築

2.2 項の課題 1 で述べたとおり，タップルのプロフィール画像審査は，審査基準が多様であるために簡単な分類問題としては解くことができない．

[*2] ぼかしが入っている画像など数値化が難しいものや，芸能人などの監視オペレータの知識依存になってしまうものなど．
[*3] https://support.tapple.me/hc/ja/articles/360007366514-メイン写真が否認されるのはなぜですか-

図 3.1.1 否認されるプロフィール画像の具体例

表 3.1.1 タップルの画像審査で否認される例

否認理由	詳細
顔を認識できない	顔の半分以上確認できないもの，目元が確認できないもの
加工	加工が強すぎるもの
顔のサイズが小さい	画像全体に比べてあまりにも顔の部分が小さもの
不鮮明	画素数が低く，画像が不鮮明なもの
写真の写真	カメラで写真やプリクラ，証明書などを撮影した画像
複数人	複数人が写っていて，誰が本人かがわからない
芸能人	芸能人や明らかに本人でない画像
人間以外	人物ではないイラストや動物，景色などの画像
個人情報	個人情報が書き込まれている画像
卑猥	卑猥な画像
グロテスク	グロテスクな描写を含む画像

フィルタを開発するにあたって，開発前に画像審査の基準書や実際のデータを分析することで，複数の機械学習技術の組み合わせが必要なことがわかった．以下，図 3.1.2 の複数のフィルタのうち，顔検出フィルタについて詳しく説明する．

顔検出

表 3.1.1 の否認理由のうち，「顔のサイズが小さい」と「人間以外」，「複数人」を自動で検知したい場合は，分類問題として解くのではなく，物体検出（顔検出）の問題として解くこととした．その理由は大きく 2 つある．一つは「顔のサイズが小さい」と「人間以外」，「複数人」の 3 つの否認理由を精度よく分類するためである．3 カテゴリ分類の問題をそのまま機械学習で解くよりも，顔検出の問題を機械学習で解き，検出した顔の候補領域の大きさや数をルールベースで 3 カテゴリに分類したほうが精度が高かった．もう一つは，顔検出の結果を，後段のフィルタで使用するためである．表 3.1.1 の否認理由は，「加工」や「不鮮明」といった顔の領域のみで判断できるものと，「卑猥」や「グロテスク」，「個人情報」のように顔の領域以外での判断が必要なものとに分けることができる．それらのうち顔の領域のみで判断できる否認理由を分類するフィルタは，顔検出の結果を使用して顔の領域のみを切り抜いた画像を入力することで精度よく分類できた．これは背景や体など判断に必要のない情報を取り除くことができて，かつ顔の位置も固定できたためである．

次に顔検出フィルタの詳細について述べる．顔検出には，モデルサイズや推論速度を考慮して CNN（Convolutional Neural Network）の有名な

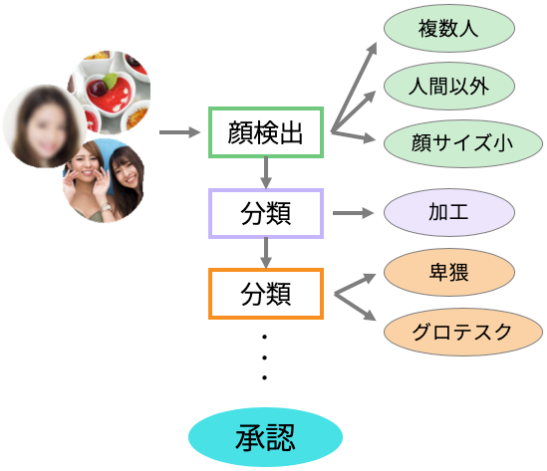


図 3.1.2　フィルタ構成の例

モデルである MobileNetV2 [2] を特徴抽出器とする SSD（Single Shot MultiBox Detector）[3] のモデルを使用した．顔検出の後処理には，Soft-NMS [4] を用いた．

ここで，後処理に Soft-NMS を用いた理由を簡単に説明する．一般的に物体検出の後処理によく使用されている NMS（Non-maximum Suppression）は，モデルが予測した候補領域の中に，信頼度が高い領域とその領域に重なっている候補領域があったときに，領域の重なり度合いを表す IoU がある閾値以上の場合に重なっている候補領域を削除するというものである．しかし NMS には次の問題がある．隣接した物体に対する候補領域の信頼度がどれほど高くても，IoU が閾値を超えてしまった場合には隣接した物体に対する候補領域が除外されてしまう．タップルのプロフィール画像として投稿されて，人の手によるラベリングで「複数人」という理由で否認された画像の中には，2 つの顔が非常に近いものが多く，自動処理による顔検出で 2 つの顔を高い信頼度で検出できても，後処理で NMS を使用すると，いずれかの候補領域が IoU の閾値処理によって除外されてしまい，顔が 1 つだと判定されて承認してしまうケースが多くあった．このことから，IoU の値を使って信頼度を減少させて信頼度に対して閾値処理を行うアルゴリズムである Soft-NMS を使用することにより，これらの課題を解決した．

3.2　アップデートを考慮したフィルタの構築

2.2 項の課題 2 で述べたとおり，一部の否認理由に該当する画像において，外部の要因でデータが頻繁に変わることによる機械学習モデルの陳腐化が問題となっていた．それを解消するためには機械学習モデルのアップデートが必要だが，適切な頻度は分類対象の否認理由によって異なる．仮に複数の否認理由を 1 つの機械学習モデルで多クラス分類した場合，ある 1 つの否認理由に該当するデータが外部要因によって変わったときに，それに伴って機械学習モデルをアップデートすると，ほかの否認理由に対する分類結果も変わってしまうおそれがある．これを避けるために，アップデートの頻度が多いフィルタは可能な限り分離した．これにより，ほかの否認理由を分類するフィルタの精度への影響を避けることができ，精度検証もしやすい．

タップルのプロフィール画像審査では，特に加工を検知する分類モデルの更新頻度が多いので，図 3.1.2 のように，「加工」を分類するモデルと，「卑猥」や「グロテスク」を分類するモデルとを分離し，それぞれを別々のフィルタとすることでアップデートしやすいしくみとした．

しかし，このようにいずれも分類問題として解こうとしているものを，否認理由ごとに異なる機

械学習モデルを学習させてフィルタを分けたときには次のような短所があると考えられる．一つは，複数の機械学習モデルが必要になるために，データセットの作成や学習および評価がそれぞれのモデルで必要となり，管理が難しくなってしまうことである．もう一つは，推論に必要となるマシンリソースや推論時間が増加することである．

タップルのプロフィール画像審査においては，これらの問題を考慮したうえでも，高頻度でアップデートするフィルタを分離するほうがアップデートの作業も最小限で済み，フィルタ全体の精度のブレが小さくなるため，加工を分類するモデルは分離することとした．

4 フィルタの導入と結果

開発した機械学習フィルタは，内製の総合監視基盤システム Orion（オライオン）[5] 上で動作している．監視オペレータは，日々 Orion の Web UI を通して，ユーザが投稿したコンテンツを効率的に監視している．

タップルの画像審査における Orion での処理の流れを，図 3.1.3 を用いて簡単に説明する．タップルで投稿されたプロフィール画像は，まずサービスを提供するサーバから Orion に転送され，本稿で説明した機械学習フィルタにより自動審査が行われる．機械学習フィルタが承認できると判定した場合は，そのまま投稿された画像が承認であることをサービス側に返す．もし否認と判定した場合は，今までと同様に監視オペレータが目視で審査を行ってサービス側にその結果を返す．これにより，機械学習フィルタで承認と判定される画像の分の監視コストを削減できる．ただし機械学習フィルタで自動承認した画像は，監視オペレータが目視で審査を行うことはないため，不適切なプロフィール画像が自動承認されてしまうと，サービス品質が低下する可能性がある．これを防ぐため，適合率（Precision）を最も重視して機械学習フィルタを構築した．ここでの適合率は，フィルタが承認する画像のうち，監視オペレータも承認する画像の割合を指している．現在稼働しているシステムにおける適合率は常に 99% を超えるものになっている．

実際にフィルタを導入することで，次の 3 つの効果があった．

- 監視コストの削減
- 審査結果のブレの防止
- 審査時間の短縮

高い適合率を保ちながら，1 節で述べたプロフィール画像審査の課題の 2 つを解決できただけでなく，審査時間の短縮も同時に実現できた．

またフィルタ導入後もフィルタの品質を担保することは重要であり，フィルタの精度の悪化や投稿されるデータの変化をただちに検知する必要がある．そのためにフィルタの予測結果の割合や自動承認率などのメトリクスを定常的に収集し，それらに大きな変化があった場合はアラートを通知するしくみも整えている．また図 3.1.3 にあるように，通常はフィルタが承認した画像を監視オペレータが目視で確認することはないが，定期的にその一部を監視オペレータが目視で確認することで，フィルタの精度の悪化やデータの変化をいち早く検知できるように努めている．

5 まとめ

本項では，当社のマッチングサービスであるタップルのプロフィール画像審査における課題を紹介し，それを解決するための機械学習フィルタの開発における工夫などを具体例を用いて紹介した．今後は Precision を維持したまま，自動否認の導入などを視野にコスト削減率の向上に取り組んでいきたい．

参考文献

[1] 角田 孝昭，數見 拓朗：“アメブロを護る：不適切コンテンツの傾向分析と検知手法の検討”，CyberAgent，CyberAgent 秋葉原ラボ技術報告 Vol. 2，2018.

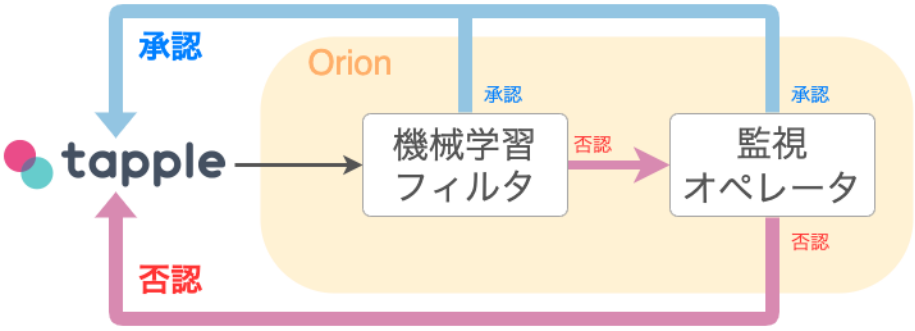


図 3.1.3　画像審査の簡略図

[2] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L. Chen: “MobileNetV2: Inverted residuals and linear bottlenecks,” The IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pp. 4510–4520, 2018.

[3] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, and S. E. Reed: “SSD: single shot multibox detector,” European Conference on Computer Vision (ECCV), 2016.

[4] N. Bodla, B. Singh, R. Chellappa, and L. S. Davis: “Soft-NMS—improving object detection with one line of code,” 2017 IEEE International Conference on Computer Vision (ICCV), 2017.

[5] Y. Fujisaka: “Orion: An Integrated Multimedia Content Moderation System for Web Services,” The Annual ACM International Conference on Multimedia Retrieval (ICMR), Industrial Session, 2018.

Akihabara Lab

3.2 NG 画像フィルタの設計から運用まで

岩井 二郎

概　要　ブログプラットフォームにおける有害画像の検出という技術課題を端緒にして，より一般的な課題として画像分類フレームワークの開発と推論 API のインフラ構築に取り組んだ．画像分類フレームワークとして複数サービスの要求に迅速に低コストで応える gmclf を提案する．個々のサービスの要求の共通性と相違点の分析に基づいて導出した gmclf のアーキテクチャの特徴について述べ，gmclf の適用事例として NG 画像フィルタを紹介し，評価結果を報告する．推論 API のインフラ構築については NG 画像フィルタを例にしてコンポーネントの構成と技術選定および現状の課題について述べる．

Keywords　機械学習, コンピュータビジョン，畳み込みニューラルネットワーク, CNN, 画像分類

1　はじめに

1.1　ビジネス課題

一般的に Web でサービスを提供する企業にとって魅力的なサービスを提供し続けるということは企業の存在理由そのものであり，当然のことながらステークホルダーにとって大きな関心事である．

サイバーエージェントのメディア事業においても Ameba ブログ，ABEMA，タップルなどのさまざまなインターネットサービスを展開しているが，ユーザにとって有益なコンテンツ，プラットフォームを提供することはいうまでもなく，低コストで高いサービス品質を維持することも常に優先度の高い課題である．たとえば，Ameba ブログのようなブログのプラットフォームではユーザがテキストだけでなく，静止画像，動画像を投稿するしくみが提供されている．ユーザの投稿するコンテンツは多くのサービス利用者の目に触れるため，有害なコンテンツが投稿されるとサービスの価値を毀損しビジネスに負のインパクトを与える．したがってサービス品質維持のために継続的にユーザの投稿を監視する必要がある．

従来 Ameba ブログでは Orion によるユーザ投稿の有人監視を行ってきたが，最近ではテキストについてはキーワードマッチングや機械学習などの技術を用いた監視の自動化が積極的に進められている．しかしながら画像については一日あたり数十万枚の投稿を 24 時間体制で有人監視しているというのが実情であり，サービス品質の維持とともにコスト削減のために監視の自動化が喫緊の課題となっていた．以上のような背景を踏まえて本稿では以下の 2 つの課題に焦点を絞って話を進める．

- サービス品質の維持
- コスト削減

1.2　技術課題

上記のビジネス課題を踏まえて秋葉原ラボの画像チームでは機械学習を用いて Ameba ブログの投稿画像から有害画像を検出するという技術課題に取り組んできた．

多くの画像の中から何らかの基準を満たす，ある

いは満たさない画像を検出するタスクはより一般的な既定クラスへの画像分類のタスクの一部としてとらえることができる．画像分類はコンピュータビジョンの分野では古くから扱われてきたタスクであるが，2012 年の CNN（畳み込みニューラルネットワーク）の登場 [1] 以降飛躍的に分類精度が向上して実用レベルになっている．CNN を実際のアプリケーションに適用する場合は分類器の学習に膨大な教師データが必要になるため，どのようにデータを収集するか，そのコストをいかに低減するかが課題となる．また，Ameba ブログ以外のサービスにおいても画像の何らかの属性に基づくサービス提供をしたいという需要は多く，今後さまざまなサービスへ機械学習による画像分類の適用が見込まれている．

複数のサービスへ機械学習による画像分類を適用する場合，個々のサービス要求に応じて最小限の開発コストで迅速に分類器を学習できる環境が整備されていることが開発効率を向上させてコストの削減につながるという意味で重要である．さらに，運用および保守という観点からはレイテンシやスループット，可用性についてサービスの要求に応える分類器の推論 API のインフラをいかに低コストで提供するかということが課題となる．技術課題をまとめると以下のようになる．

- 画像分類器開発の汎用的な環境の整備
- 推論 API のインフラの構築

以降の 2 節では上記の 1 つ目の技術課題を解決するために開発した画像分類フレームワーク gmclf について詳述し，3 節では gmclf を使って作成したブログの有害画像検出を行う NG 画像フィルタとその性能評価について述べ，4 節では推論 API とインフラの構築について簡潔に議論し，5 節ではまとめと今後の課題について述べる．

2　画像分類フレームワーク gmclf

前節で述べた技術課題を考慮し，特定のサービスに特化した画像分類器を作成するのではなく，今後需要が見込まれるさまざまなサービスの要求に合わせてその都度迅速に画像分類器を作成できる汎用的な画像分類器作成のためのフレームワークを開発することを試み，このフレームワークを gmclf と名付けた．以下に gmclf が備えるべき特徴を列挙する．

- 分類器のモデル変更が容易である
- 利用できる教師データが少ない場合でも実用的な分類器を作成できる
- データ収集を支援するしくみがある
- 精度を確認しながらデータを容易に選別するしくみがある
- 複数の分類器を作成してもコードが重複することがない
- コードとデータを分離して管理できる

上記の特徴が実現されるように個々の分類器への要求によって変化すること（共通性），変化しないこと（相違点）を明確にしてアーキテクチャを決定した．共通性と相違点とをまとめると以下のようになる．

共通性

- 画像分類の基本的な技術
- データ収集の手順
- 学習の手順
- 分類器の精度の評価，可視化の方法

相違点

- データ
- 分類器のモデル
- 学習のパラメータ

以降でこれらの共通性と相違点を実現する仕様と技術およびその選定理由について述べる．gmclf は画像のダウンロードや分類器の学習，テスト用 API，可視化ツールなどのデータによって変わらないアルゴリズムを Python で実装し，gmclf とは別のディレクトリ（以降プロジェクトディレクトリという）に，画像に加えて CNN の学習済みモデル

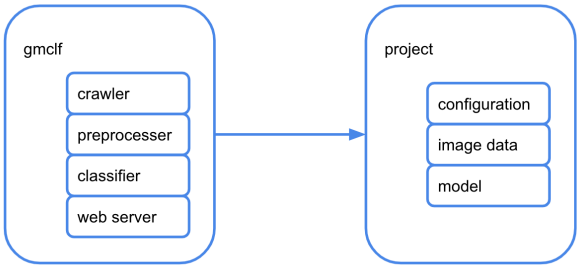


図 3.2.1 gmclf のアーキテクチャ

や学習パラメータなどを指定する設定ファイルを YAML で記述し配置する．この構成により上記の共通性と相違を実現するとともにコードとデータの分離を図っている．

画像分類の基本的な技術としては CNN を採用している．検討開始時点で実用的な精度を期待できる技術としては CNN 以外の選択肢がなかったというのが実際のところであるが，むしろ近年の CNN の精度向上によって gmclf の開発が可能になったという方が適切である．

CNN を実現する深層学習のフレームとしては Python の深層学習ライブラリ Keras[*1]，バックエンドとして TensorFlow[*2]を採用した．Keras では画像分類器として利用できる学習済みモデルが多数サポートされていること，記述の自由度は下がるが TensorFlow を直接に使うよりも少ないコード量で目的を達成できることなどが採用理由である．検討時はバックエンドとしては TensorFlow と Theano[*3]が選択できたが TensorFlow の方がシェアが高く今後の発展性があると判断して採用した．

Keras の画像分類のための学習済みモデルは代表的な CNN のモデルを ImageNet[*4]を使ってあらかじめ学習されており，目的に応じて精度とモデルの複雑さについてトレードオフを検討できる．さらに学習済みモデルを用いることの利点としては転移学習を使うことで初期状態から学習するよりも教師データの量や学習時間をはるかに削減できるということが挙げられる．学習済みモデルはほかの学習パラメータとともに設定ファイルで以下のように指定する．

```
NV_GPU: 0
BATCH_SIZE: 64
EPOCHS: 10
PATIENCE: 10
BASE_MODEL_NAME: vgg16
FIXED_LAYERS: ~
OPTIMIZER:
  name: adam
  param:
    lr: 1.0e-04
L2: 0.001
DROPOUT: 0.5
MODEL_NAME: catndog_vgg16_v1
```

ここで BASE_MODEL_NAME: が学習済みモデルを表す．データ収集の手順に関しては一般論としてはサービスごとに収集方法が異なるが，gmclf では画像は URL で示されるものとして定めて，画像の収集を統一的に扱うことができるようにしている．

画像分類のためのデータセットは画像を示す URL と教師ラベルを示すテキストの組の集合と

[*1] https://keras.io/ja/
[*2] https://www.tensorflow.org/
[*3] http://deeplearning.net/software/theano/
[*4] http://www.image-net.org/

して CSV 形式で表現する．ユーザは分類器の学習に先立ってあらかじめデータセットとして CSV ファイルを用意する．gmclf は分類器の学習の前処理として CSV ファイルにしたがって画像をダウンロードして Keras の形式に沿って画像を保存する．画像が少ない，あるいはまったくない場合でもキーワードを指定して Web クローリングにより画像を収集できる．キーワードは教師データのクラスごとに複数のキーワードを組み合わせて指定できる．たとえば犬と猫の画像を判別する分類器を作成する場合は設定ファイルに以下のように指定できる．

```
CLASSES:
- cat
-
 - ’三毛猫’
 - ’アメリカンショートヘアー’
 - ’マンチカン’
- dog
-
 - ’シェルティー’
 - ’柴犬’
 - ’ゴールデンレトリバー’
```

データセットを表す CSV ファイルと設定ファイルをプロジェクトディレクトリに用意し，gmclf を clone したディレクトリで以下のようにプロジェクトディレクトリを指定してコマンドラインから make を実行するとデータのダウンロード，重複チェック，分類器の学習までが自動的に行われる．

```
$ make PRJ_DIR="プロジェクトディレクトリ名"
```

学習終了後，api をターゲットとして make を実行するとテストと可視化のための Web サーバがローカル環境で動作する．

```
$ make PRJ_DIR="プロジェクトディレクトリ名" api
```

URL のパスおよびポート番号は以下のようにあらかじめ設定ファイルに指定しておく．

```
ROUTE_API: /catndog/api/v1/classify
ROUTE_WEB: /catndog/web/v1
PORT_API: 5003
```

図 3.2.2 に示すように Web UI では画像の URL を指定して分類器の推論結果を視覚的に確認できるほか，Grad-CAM [2] というアルゴリズムにより CNN のモデルで判断の根拠として画像のどの部分が重要視されているかをヒートマップで表示する．

さらに画像の下の追加ボタンを使って表示されている画像をプロジェクトディレクトリに追加できる．プルダウンメニューでは画像を学習データに追加するか検証データに追加するかの選択と，どのクラスに追加するかの選択とを行う．上部の検索タブを選択すると図 3.2.3 のように混同行列を表示し，行列要素を選択して検索ボタンを使って誤分類した画像を確認できる．画像の直下に表示されるプルダウンメニューでクラスを変更してアノテーションの誤りを修正したり，不適切なデータを学習データから除外したりできる．

犬画像と猫画像の分類器を作成する場合の例を挙げると，画像分類や CNN に関する知識のない人であってもまったく教師データがない状態から数十分程度で精度 90% 程度の分類器を作成できる．

3　NG 画像フィルタとその性能評価

本節では gmclf を Ameba ブログの有害画像検出に適用して作成した NG 画像フィルタについて述べる．

データセット

NG 画像フィルタの元のデータセットは 75 種類のラベルが付けられた 13400 枚の画像からなる．NG 画像フィルタを作成するにあたって画像分類タスクとしての難易度やクラス間のデータ数の不均衡などを考慮してクラスを 12 種類に統合した．さらにデータ数が十分にあり実際にブログへの投稿で頻発する 4 種類のクラスに属する画像のみを取り出し，データ数の不均衡を是正するようにダ

図 3.2.2 可視化ツールの UI（1）

図 3.2.3 可視化ツールの UI（2）

ウンサンプリングとクローリングによる画像の追加を行った．各クラスの意味とデータ数は図 3.2.4 左上のとおりである．

モデルの学習と精度評価

学習済みモデルとしては InceptionResnetV2 を使用し，conv 層の重みは固定した上で元の全結合を削除して新たに付加した全結合層のみを学習した．全結合層は Global Average Pooling 層，1024 ユニットの全結合層，4 ユニットの全結合層を付加

し，活性化関数としてはそれぞれ ReLU，softmax を使用した．

データセットを 8 対 2 で学習データと検証データに分割して，最適化手法として SGD および Adam を用い, 学習率を固定して NVIDIA の GPU V100 を使って 2000 エポック学習した．学習時間は 48 時間程度である．検証データの学習曲線と精度評価の結果を図 3.2.4 左下および右に示す．

Virtual Adversarial Training

CNN による画像分類は人間に認識できない画像のわずかな変化で分類結果が大きく変わってしまう脆弱性 [3] が指摘されているが，これを逆手に取って摂動を加えた画像を学習データとして用いることにより分類精度を向上させる Adversarial Training [4] [5] という手法が提唱されている．NG 画像フィルタの分類精度向上の一環として gmclf に Adversarial Training の一種である VAT（Virtual Adversarial Training）[6] を実装し，NG 画像フィルタのデータで精度評価を行った．以下のように VAT を用いるか否かの指定とパラメータの値もプロジェクトディレクトリの設定ファイルで指定できる．

```
VAT:
  USE: true
  PARAM:
    EPS: 5.0
    XI: 10.0
    IP: 1
```

学習済みモデルとしては ResNet50 を使用し 164 層までの重みは固定し 165 層以降と元の全結合を削除し新たに付加した全結合層を再学習した．全結合層の構造，データセットの分割，GPU は前述のとおり変更せず，最適化手法として Adam を用い，学習率を固定して学習した．学習時間は 12〜48 時間程度である．検証データの学習曲線と精度評価の結果を図 3.2.5 に示す．緑が VAT を使用しない場合を示し，赤と灰色が VAT を使用したときに画像に与える摂動のノルムが 1 の場合と 2 の場合とをそれぞれ示している．残念ながら今回試行したパラメータの範囲内では NG 画像フィルタのデータセットでは VAT の効果が確認できなかった．原因については転移学習で浅い層の重みを固定しているために精度向上につながる特徴が十分に学習できていないなどの理由が推測されるが，詳しい検証は今後の課題である．

データセットを CSV ファイルとして準備すれば，以降のモデルの学習，精度評価，データの追加と削除の作業はすべて gmclf だけで完結しているので，試行錯誤のイテレーションを迅速に行える．

実際の投稿画像での評価

実際に Ameba ブログに投稿された画像で NG 画像フィルタを評価した結果について述べる．評価データは 2019 年 11 月から 2020 年 1 月に投稿された画像からランダムに 10 万枚を選択した．

NG 画像フィルタは 1 枚の画像に対して 4 つのクラスについての確率を推論する．4 つのクラスはそれぞれ e3（卑猥），g3（グロテスク），id（QR コード），ok（適切な画像）であり，ok に分類される確率を OK 確率，1−OK 確率を NG 確率，ある画像について OK 確率が閾値以上の場合 OK 画像，閾値より小さい場合 NG 画像とここでは呼ぶことにする．

評価データについての OK 確率をヒストグラムで表示すると図 3.2.6 左上のようになる．ヒストグラムから，過半数の画像が OK 確率 0.9 から 1.0 に含まれることが分かる．ヒストグラム中で赤で示している部分は目視で有害と判断できる画像の枚数のサンプリングによる推定値である．サンプリングではヒストグラムの各レベルで 1000 枚の画像をランダムに抽出して目視によって有害画像かどうかを確認した．図 3.2.6 右上では閾値の変化に伴う NG 画像の割合を示している．閾値 0.5 で全画像の約 28% を NG 画像と判定している．これにより OK 画像について有人監視をしないことにすると 72% コスト削減が可能となることが分かる．

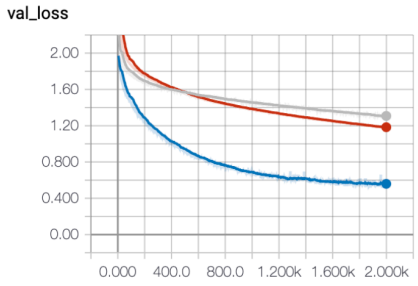
実際のコスト削減については，NG 画像フィルタの運用により 60% 程度の削減を確認している．

クラス名	意味	元データの画像枚数	ダウンサンプリング後の画像枚数	クローリング後の画像枚数
ok	問題のない画像	4350	659	1261
e3	実写卑猥画像	6559	659	797
g3	実写グロテスク画像	367	367	667
id	QRコード	593	593	732

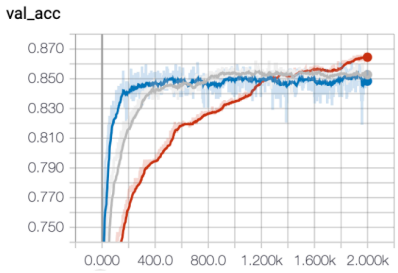
データセットの内訳

クラス	precision	recall	f1-score
e3	0.80	0.94	0.87
g3	0.83	0.78	0.81
id	0.86	0.97	0.91
ok	0.91	0.79	0.84
total	0.86	0.85	0.85

検証データの評価結果

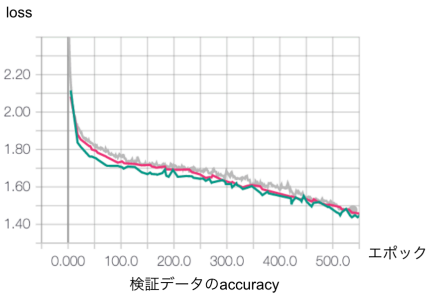


検証データのloss



検証データのaccuracy

図 3.2.4 データセットの内訳，精度評価結果



検証データのaccuracy

検証データのloss

図 3.2.5 VAT の精度評価

図 3.2.6 左下では閾値の変化に伴って OK 画像への有害画像の混入率の上限がどのように変化するかをヒストグラムから計算して示した．ヒストグラムの各階級における有害画像の枚数が正規分布に従うとし，有意水準 0.05 における信頼区間から混入率の上限を定めた．閾値 0.5 で OK 画像の 0.027% が有害画像となっている．これは Ameba ブログのサービス品質として十分許容できる値であり，実際の有害画像の含有率も上限を下回っている．

4 推論 API のソフトウェアとインフラ構成

本節では 2 つ目の技術課題である推論 API のインフラ構成と関連するソフトウェアの構成について述べる．NG 画像フィルタの推論 API は平均で 3RPS，ピーク時で 30RPS のリクエストに 3 秒以内に応答することが求められ，99% 以上の可用性を維持することが努力目標となっている．

インフラ構築に必要な時間，障害発生時に容易に復旧できること，適用範囲の拡大に伴うリソースの柔軟な増強ができることなどを考慮して Amazon

*5 https://aws.amazon.com/jp/

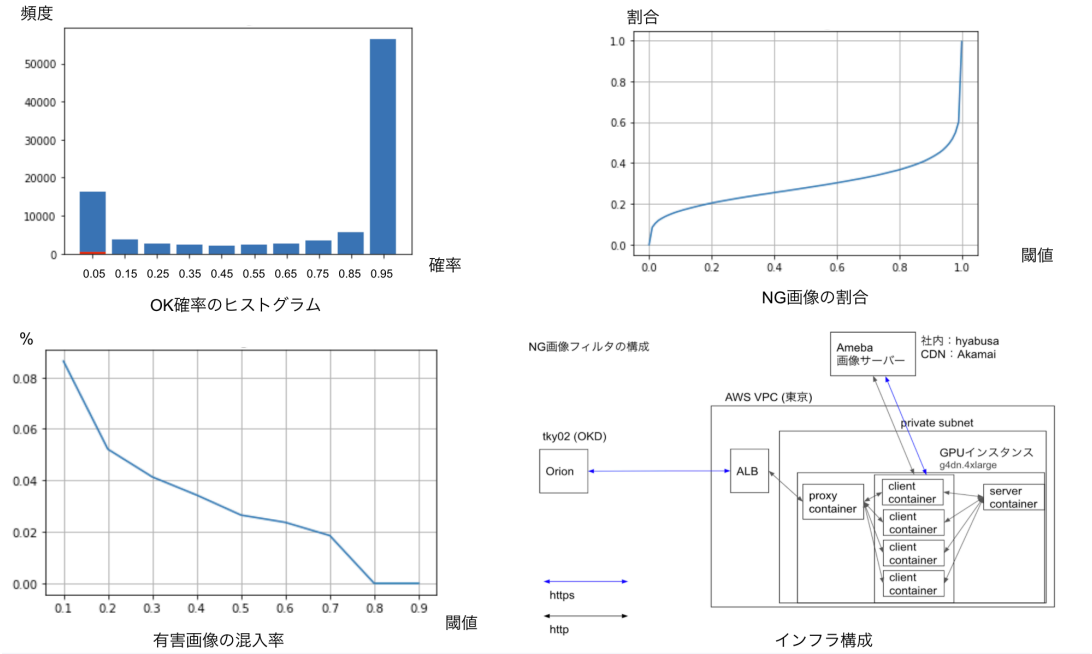


図 3.2.6　左上：OK 確率のヒストグラム，右上：NG 画像の割合，右上：有害画像の混入率，左下：インフラ構成

Web Services (AWS)[*5]にインフラを構築した．パブリッククラウドとしてはほかの選択肢もあるが社内でのサポートが得やすいことから AWS を採用した．

開発効率と汎用性を考慮しアプリケーションを構成する主要なコンポーネントは Docker コンテナとして実装している．RESTfull API は gunicorn, flask を使って実装し，推論サーバについては OSS の NVIDIA Triton Inference Server[*6]を採用している．採用理由としては汎用性の高さと GPU の特性を考慮したパフォーマンスチューニングができることが挙げられる．gmclf で作成したモデルをサーバの所定のディレクトリにアップロードすることにより推論サーバのローリングアップデートが可能であり，モデルを変更することで NG 画像フィルタ以外のアプリケーションでも動作可能な汎用的な構成となっている．

インフラ構成の概念図を図 3.2.6 右下に示す．本稿執筆時点での推論 API への要求，ソフトウェア，インフラ構成について述べたがこれは必ずしも最良の選択であるとは限らない．特にコストについては長期的にパブリッククラウドを用いるよりもオンプレミスの方が有利になる可能性が高いのでオンプレミス環境への移設を検討中である．

5　まとめと今後の課題

画像分類が必要となる Web におけるサービス品質の維持とコスト削減という課題に対する解決方法の一つとして画像分類フレームワーク gmclf を提案し共通性と相違点の分析に基づくアーキテクチャが複数のサービスの要求にどのように応えるか詳述した．gmclf の適用例として NG 画像フィルタを紹介し，ブログのプラットフォームにおける有害画像検出という技術課題をどのように解決するかについて精度とコスト削減を定量的に示すとともに，インフラ構成の技術選定とその課題について述べた．

gmclf の仕様は現状のサービス要求に十分に応えられるものであるが，実装面ではコードの冗長さ，構成の分かり難さがあることは否めず，CI/CD が整備されていないなどの課題もある．今後はオンプレミス環境への移設や，Kubernetes などのコ

[*6] https://docs.nvidia.com/deeplearning/triton-inference-server/user-guide/docs/

ンテナオーケストレーションの導入を意識し画像分類だけに限定しないより一般的な画像解析基盤として改良することを計画している．

本稿が汎用的なソフトウェアの開発に取り組む開発者にとって何らかの参考になれば幸いである．

参考文献

[1] Krizhevsky, Alex, *et al.*: “Imagenet classification with deep convolutional neural networks,” Advances in neural information processing systems (NIPS), pp.1097–1105, 2012.

[2] Selvaraju, Ramprasaath R., *et al.*: “Grad-cam: Visual explanations from deep networks via gradient-based localization,” Proceedings of the IEEE international conference on computer vision (ICCV), pp.618–626, 2017.

[3] Szegedy, Christian, *et al.*: “Intriguing properties of neural networks,” arXiv preprint arXiv:1312.6199, 2013.

[4] Alexey, Kurakin, *et al.*: “Adversarial machine learning at scale,” arXiv preprint arXiv:1611.01236, 2016.

[5] Tram è r, Florian, *et al.*: “Ensemble adversarial training: Attacks and defenses,” arXiv preprint arXiv:1705.07204, 2017.

[6] Miyato, Takeru, *et al.*: “Distributional smoothing with virtual adversarial training,” arXiv preprint arXiv:1507.00677, 2015.

Akihabara Lab

2019 年発表一覧

技術報告　Vol.3

書籍・解説記事

- 森下 壮一郎，水上 ひろき，高野 雅典，數見 拓朗，和田 計也：“データマイニングエンジニアの教科書”，C&R 研究所，2019.
- 水上 ひろき，福田 鉄也，山下 剛史，Juhani Connolly，武内 慎，數見 拓朗，福田 一郎：“AWA における類似プレイリスト探索システムの構築”，デジタルプラクティス誌 招待論文，Vol.10，No.2，2019.

論文誌・国際会議（査読付き）

- M. Takano, F. Taka, S. Morishita, T. Nishi, and Y. Ogawa: “Expressing Modern/Oldfashioned Racism against Zainichi Koreans in Japan depending on News Contents of Internet Television,” 5th International Conference on Computational Social Science (IC2S2), Extended Abstract, 2019.
- T. Masanori and T. Tsunoda: “Self-Disclosure of Bullying Experiences and Social Support in Avatar Communication: Analysis of Verbal and Nonverbal Communications,” The 13th in International AAAI Conference on Web and Social Media (ICWSM-2019), 2019.

国内学会/セミナー

- 松田 和己，鈴木 元也，和田 計也：“インターネットテレビ局 AbemaTV におけるコメントデータ解析 ～BI ツール Tableau を用いた可視化～”，日本計算機統計学会 第 33 回シンポジウム，2019.
- 水上 ひろき：“音楽配信サービスにおける推薦システムの概要と数理モデルについて”，日本数学会 2019 年度秋季総合分科会 ワークショップ「数学ソフトウェアとフリードキュメント XXIX」，2019.
- 高野 雅典：“アバターコミュニケーションにおける自己開示とソーシャルサポート —ピグパーティの行動ログ分析と質問紙調査—”，日本心理学会第 83 回大会 公募シンポジウム「サブリミナルソーシャルインパクト —環境に埋め込まれたエージェント性とその展開—」，話題提供，2019.
- 上岡 将也：“マッチングサービスの画像審査における機械学習の応用”，第 12 回 Web とデータベースに関するフォーラム（WebDB Forum 2019），2019.
- 上辻 慶典，大内 裕晃，角田 孝昭，數見 拓朗，善明 晃由：“Orion Annotator ：機械学習を支えるアノテーションシステム”，ソフトウェアエンジニアリングシンポジウム 2019，2019.
- 武内 慎：“他人への音楽推薦行動の拡散現象のモデリング”，ネットワーク科学セミナー，2019.
- 高野 雅典：“インターネットテレビの視聴データによるメディア・コミュニケーション研究”，マルチメディア，分散，協調とモバイル（DICOMO2019）シンポジウム，招待講演，p.223，2019.
- 西 朋里，小川 祐樹，服部 宏充，高 史明，高野 雅典，森下 壮一郎：“インターネットテレビのニュー

ス番組におけるコメント内容の分析”，第 33 回人工知能学会全国大会，2D4-OS-1a-03，2019.

- 高野 雅典，高 史明，森下 壮一郎，西 朋里，小川 祐樹：“現代的／古典的レイシズムの表出におけるニュース番組の影響：インターネットテレビに投稿される差別的コメントの分析”，第 33 回人工知能学会全国大会，2D5-OS-1b-05，2019.
- 森下 壮一郎：“メディアサービスにおけるユーザの継続の冪分布に基づくモデル化”，第 33 回人工知能学会全国大会，1J4-J-3-01，2019.
- 涌田 悠佑，善明 晃由，松本 拓海，佐々木 勇和，鬼塚 真：“Secondary index を活用する NoSQL スキーマ推薦によるクエリ処理高速化”, The 3rd cross-disciplinary Workshop on Computing Systems, Infrastructures, and Programming (xSIG), 2019.
- 高野 雅典：“AbemaTV の視聴データを用いたメディア・コミュニケーション研究の試み”，ウェブサイエンス研究会 オープンセミナー vol.8，2019.
- 山口 想，高野 雅典，森下 壮一郎，山本 悠二，福田 鉄也，福田 一郎，梁木 俊冴，高屋 英知，栗原 聡：“映像シーンに対応した視聴者の感情・印象抽出”, 社会システムと情報技術研究ウィーク（WSSIT19），ICS10，2019.
- 高野 雅典：“人と Web サービスの相互作用：ピグパーティ、AbemaTV がユーザと社会に与える影響”，HAI シンポジウム，招待講演，2019.
- 涌田 悠佑，善明 晃由，松本 拓海，佐々木 勇和，鬼塚 真：“Secondary index を活用する NoSQL スキーマ推薦による Query 処理高速化”，第 11 回データ工学と情報マネジメントに関するフォーラム（DEIM2019），H2-4，2019.
- 武内 慎：“Hawkes 過程を用いた人の音楽鑑賞行動の解析”，第 3 回計算社会科学ワークショップ（CSSJ2019），2019.
- 高野 雅典，高 史明，森下 壮一郎，西 朋里，小川 祐樹：“ニュースを起点とするレイシズム表出におけるニュース番組の性質と個人の性質の関連：インターネットテレビに投稿される差別的コメントの分析”，第 3 回計算社会科学ワークショップ（CSSJ2019），2019.

CyberAgent 秋葉原ラボ 技術報告 Volume.3

発 行 日　2020年11月1日 初版
発 行 所　CyberAgent 秋葉原ラボ 技術報告編集委員会
〒101-0021
東京都千代田区外神田1丁目18番13号
秋葉原ダイビル 13階

編　集　數見 拓朗
上辻 慶典
松井 美帆
善明 晃由
森下 壮一郎
デザイン　デザインファクトリー　柴 尚子
印 刷 所　昭栄印刷株式会社


CyberAgent®