端末外リソースの仮想化と UGC 型のアプリ開発をサポートする IoT フレームワーク

松野宏昭[†1] 丸島晃明[†1] 有村汐里[†2] 可児潤也[†3] 二村和明[†3]
峰野博史[†4] 飯田一朗[†5] 西垣正勝[†4]

概要：近年，様々なデバイスを連携させる IoT（Internet of Things）サービスの開発が進みつつある．IoT サービスには一般住環境での省エネ・快適化やオフィス・工場での業務効率化といった活用が期待され，様々な環境で普及させていくことが望ましい．しかし，IoT サービスの普及を妨げる要因として，(a) 環境によって物理的なコンテキストが多種多様であるため，それぞれに応じた IoT サービスを網羅的に開発することは個人や企業が単独で行うには難しいという問題と，(b) ユーザのニーズには多様性や変動性が存在するため，IoT サービスとのマッチングが難しいという問題がある．これらの問題を解決するため，我々は，現在のスマホアプリ市場が成功させている UGC（User Generated Contents）型のエコシステムの形成を目指している．UGC 型エコシステムを形成することで，問題 (a) については様々なユーザ（アプリ開発者）が人海戦術的にサービスを開発することで解決が期待できる．問題 (b) についてはユーザ自身がアプリストアの中から利用したいものを選択することで解決できる．本稿ではアプリ開発の生産性を高めてエコシステム実現するために，IoT デバイスを機能単位で仮想化する OS サポート機能をフレームワークとして提案する．提案フレームワークでは，アプリ・デバイス間の依存関係を，フレームワーク側で動的に注入することで IoT アプリの再利用性・可搬性を担保する．また，排他制御やアクセス制御などの IoT におけるセキュリティ課題を，デバイス単位でカプセル化することで効率的に管理する．このように IoT デバイスを仮想化することで，アプリ開発者はデバイスの機能をマッシュアップすることにのみ集中して物理的なコンテキストに依存しない IoT アプリを容易に開発できる．

IoT Framework To Promote Remote Resource Virtualization and UGC App Development

HIROAKI MATSUNO[†1] KOUMEI MARUSHIMA[†1] SHIORI ARIMURA[†2]
JUNNYA KANI[†3] KAZUAKI NIMURA[†3] HIROSHI MINENO[†4]
ICHIRO IIDA[†5] MASAKATSU NISHIGAKI[†4]

1. はじめに

近年，様々なデバイスを連携させる IoT（Internet of Things）サービスの開発が進みつつある．インターネットにつながるデバイスは増加し続けており，2020 年には 304 億個にまで増大すると予測されている[1]．特に，スマート家電やスマートトイといった IoT デバイスが一般向けに発売されるようになり，特別な設備や環境でなくとも IoT サービスを実現することが可能になった．

IoT サービスには住環境の省エネ・快適化や，オフィス・工場の業務効率化といった効果が期待されており，様々な IoT サービスが提案，導入されている[2][3]．生活の向上や産業の発展のためには高度な IoT サービスを様々な環境で普及させ活用していくことが望ましい．

しかし，現在 IoT サービスが住居やオフィスといった一般的な環境にまで広く普及しているとは言い難い．普及を妨げる原因として大きな二つの問題があると我々は考えている．一つ目の問題は，多様な IoT デバイスと膨大なコンテキストに起因する IoT サービスの種類の爆発である．オフィス・工場・一般住宅など様々な環境ごとに利用可能なデバイスは多種多様であり，デバイスごとにその利用形態が状況や環境に応じて変化する．膨大なデバイスに対してあらゆるコンテキストを想定し，それぞれに応じた IoT サービスをすべて用意することは，個人や企業が単独で行うには難しい．二つ目の問題は，IoT サービスの提供におけるニーズとシーズのマッチングの難しさである．仮に一つ目の問題が克服され，多様な IoT サービスがユーザに提供されるようになった場合，今度はユーザが「サービスの洪水」に溺れることになる．また，同一のサービスに対してもユーザによっては「気が利く」と感じたり「大きなお世

†1 静岡大学大学院 総合科学技術研究科
Graduate School of Integrated Science and Technology, Shizuoka University
†2 静岡大学大学院情報学研究科
Graduate School of Informatics, Shizuoka University
†3 株式会社富士通研究所
Fujitsu Laboratories Ltd.
†4 静岡大学創造科学技術大学院
Graduate School of Science and Technology, Shizuoka University
†5 秋田県立大学
Akita Prefectural University

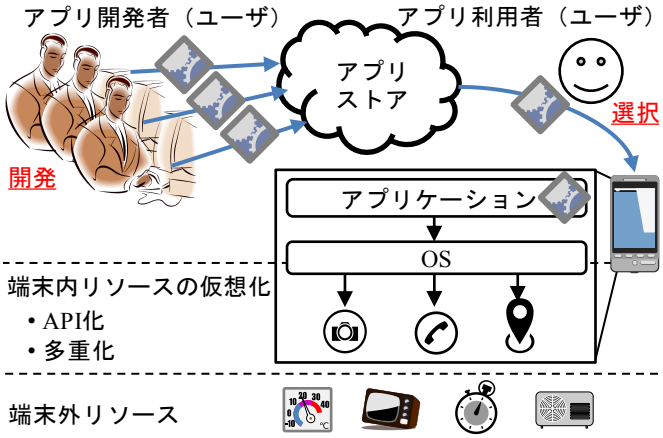

図 1　スマホアプリの UGC 型エコシステムの概観

話」と感じたりすることがあるように，ユーザによってニーズは異なり得る．更に，ユーザのニーズはコンテキストに応じて都度変化する可能性がある．このようにユーザのニーズにはこうした多様性や変動性が存在するため，ユーザの代わりにサービス提供者が一律な方法で推奨サービスを提示するようなことも難しい．

以上二つの問題の解決には，現在のスマホアプリ市場において成功している UGC（User Generated Contents）型のエコシステムの形成が効果的と考える．スマホアプリの UGC 型エコシステムの概観を図 1 に示す．UGC 型エコシステムでは，アプリの利用者であるユーザ自身に，アプリを開発する手段が提供されている．様々なアプリ開発者（ユーザ）が思い思いにアプリを開発し，開発されたアプリはストアで公開される．アプリ利用者（ユーザ）は，自身のニーズに合うアプリをアプリストアの中から自由に選択して利用する．無数のアプリ開発者によって人海戦術的に多様なアプリが開発されることで，一つ目の問題が解決される．二つ目の問題については，アプリストアが全アプリを分類した形で陳列するとともに，アプリ検索機能やレビュテーション機能を提供することによって，アプリ利用者が自身の求めるアプリを容易に選択できるようにすることが可能である．

しかし，IoT サービスの場合は，ユーザがサービスアプリ（以下，IoT アプリ）を開発するにあたって障壁が存在しており，スマホアプリのような UGC 型のエコシステムを構成することが難しいという現状にある．図 1 に示すように，アプリが端末内のリソースを利用する際の競合管理やアクセス制御を OS が請け負うことによって端末内のリソースを仮想化し，アプリの生産性，可搬性，セキュリティを確保している．これに対し，IoT デバイスは，OS にとって「端末外のリソース」となる．現在のスマホアプリの開発フレームワークにおいては，端末外のリソースである IoT デバイス（以下，IoT リソース）を仮想化して扱う仕組みが確立していない．このため，IoT リソースの利用に際しては，物理コンテキストに依存する部分をアプリ側で記述してやる必要が生じ，これが IoT アプリに対する開発の生産性を大きく低下させる原因となっていた．

そこで本稿では，IoT リソースを仮想化するために必要となる機能を，OS をサポートするフレームワークとして提供する．提案フレームワークでは，IoT リソースが機能単位で仮想化されており，IoT リソースの機能の実体は，アプリが IoT リソースを使用する時点で，その時々の物理コンテキストに応じた形でアプリに動的に注入（IoT リソースの依存性の動的注入）される．また，提案フレームワークでは，IoT デバイスごとに仮想オブジェクトが設けられており，アプリが IoT リソースを使用する際の競合管理やアクセス制御を IoT デバイス側で自律的に行う．これらにより，アプリ側で IoT リソースの物理コンテキストを考慮する必要がなくなり，アプリ開発者が IoT リソースを端末内リソースと同じ感覚で取り扱うことが可能となる．このように，内部リソースの仮想化を OS が担い，外部リソースの仮想化をフレームワークが担うことによって，IoT アプリの生産性が向上し，UGC 型のアプリ開発が達成される．

以降の構成は次の通りである．2 章では端末外リソースの仮想化に関する課題と，仮想化が不十分なことでアプリ開発の生産性が低下する問題について述べる．3 章では，提案フレームワークの機能と 2 章で挙げた課題へのアプローチについて説明する．4 章では，IoT アプリの UGC 開発に関する先行研究・技術について調査し，提案フレームワークとの比較を行う．5 章では，まとめと今後の方針について述べる．

2. リソースの仮想化

2.1 端末内リソースの仮想化

現在のスマートフォンなどの OS では，メモリ・カメラ・GPS・通話機能といった端末内のリソースを仮想化している．ここでは，リソースの仮想化について，「リソースの API 化」と「リソースの多重化（共有）」の二つの観点で述べる．

「リソースの API 化」については，例えばスマホ搭載カメラの場合，カメラの機能を API として抽象化・標準化することで端末機種による搭載カメラの差異を吸収している．各リソースの機能の実体は，OS によってアプリに動的に注入（リソースの依存性の動的注入）される．これにより，アプリ開発者は，端末機種の違いを気にすることなく，常に Camera クラスの open メソッドを用いてアプリから容易にカメラ機能を利用できる．かつ，異なる機種の端末でも，同一のアプリを動作させることができる．この結果，アプリの生産性や可搬性が確保される．

「リソースの多重化」については，OS がリソースを時分割・空間分割することで，複数のアプリケーションが一つのリソースを同時に利用する際に起こる競合を調停する．

表 1　端末外リソース仮想化の課題

	課題	影響
1	機能の定義	API 化
2	依存性の動的注入	
3	制御機構	多重化
4	排他制御	
5	アクセス制御	
6	プライバシ制御	

例えば，一つの物理メモリを時分割することで，複数のアプリがメモリを疑似的に占有できる[4]．また，メモリ領域の空間分割によってサンドボックスを実現し，データの漏洩を防止する．これらにより，アプリ開発者は，リソースの競合やアクセス制御を気にすることなく，リソースを随意的に利用できる．この結果，アプリの可用性や機密性が確保される．

2.2　IoT リソース仮想化の課題

IoT アプリは，スマート家電やスマートトイといった端末外のリソースを多く利用するが，現在の OS ではこれらを仮想化できていない．端末外リソースの仮想化に対する課題を表 1 に示す．

2.2.1 API 化の課題

IoT リソースの API 化に関する一つ目の課題が「課題 1：機能の定義」である．あらゆる種類の IoT デバイスがネットワークを介して接続される IoT の世界では，IoT リソースの種類は膨大となるため，これらが持つ様々な機能を，どのような抽象度・粒度で API 化するのかが課題となる．また，次々と新しい IoT デバイスが開発されていくため，柔軟に API を追加することが可能な枠組みが必要となる．

IoT リソースの API 化に関する二つ目の課題が「課題 2：依存性の動的注入」である．IoT リソースは端末外にあるため，端末内のアプリが API を呼び出したときに，その API コールと API 機能の実体（IoT リソース）を動的に結び付けてやる必要がある．これについて，IoT アプリのユースケースを例に採り説明する．

図 2 に不快指数計アプリのユースケースを示す．このアプリは，気温センサと湿度センサの値から不快指数を監視し，設定値以上ならば警告イベントをユーザに提示するサービスである．図 2 のユーザ 1 は，子供の寝室である部屋 A に配置したセンサに外部からアクセスして，部屋 A の不快指数をモニタリングしたいと考えている．このようなユースケースの典型的なプログラムの例として，接続先をハードコードしたプログラムを図 3 に示す．これは，1〜2 行目で部屋 A の気温センサ，湿度センサに接続し，3〜14 行でセンサをポーリングして不快指数を計測し，値がしきい値以上ならば警告を出すというプログラムである．このプログラムは，センサの URI がハードコードされており部屋 A のセンサにしかアクセスできないため，A 以外の部屋の不快指数を監視したい他のユーザはこのアプリを利用することができず，アプリの可搬性が低い．これを解決するには，1，2 行目の接続先センサの URI を設定ファイルに記述するなどして変更できるようにする必要がある．このように，コンポーネント間の依存関係をプログラムコードから排除して，設定ファイルなどによって別途指定するデザインパターンは「依存性の注入」と呼ばれている．

```
1.  tSensor = connect("https://room_a/t_sensor001");
2.  hSensor = connect("https://room_a/h_sensor001");
3.  while( true ){
4.      if( tSensor .isDisconnected ||
            hSensor.isDisconnected ){
5.          exit();
6.      }
7.      t = tSensor.getValue();
8.      h = hSensor.getValue();
9.      v = 0.81*t+0.01*h*(0.99*t-14.3) + 46.3;
10.     if( THRESHOLD < v ){
11.         警告イベントを送信();
12.     }
13.     wait(POLLING_INTERVAL);
14. }
```

図 3　接続先をハードコードしたプログラム

しかし，IoT アプリでは，IoT リソースの URI を設定ファイルに記述しておくような，静的な依存性の注入だけは不十分である．これについて，図 2 中のユーザ 2 を例に述べる．ユーザ 2 は，滞在先の部屋の不快指数を監視したいと考えている．この場合，ユーザ 2 が部屋を移動するたびに接続するセンサを動的に切り替える必要がある．ここで，部屋 B には二つの気温センサが存在するため，どちらの気温センサを利用するか決定しなければならない．使用中の

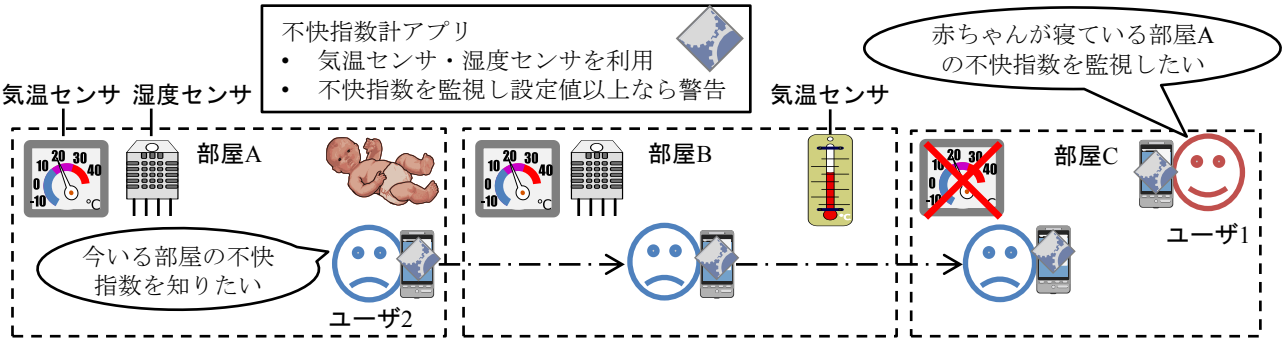

図 2　不快指数計アプリケーションのユースケース

表 2　提案フレームワークの機能一覧

コンセプト		機能	
I	仮想空間と物理空間の分離	1	機能を単位とした API【課題 1 の解決】
		2	依存性の動的注入【課題 2 の解決】
II	仮想空間における UGC 型アプリ開発	3	機能のマッシュアップ（メタ機能）
		4	機能・メタ機能のアプリ化
III	カプセル型リソース管理	5	物理デバイスと常に接続されたシャドウ【課題 3 の解決】
		6	デバイス単位の排他・アクセス・プライバシ制御【課題 4〜6 の解決】
IV	仮想空間からの UGC 型デバイス開発	7	デバイス機能の実装

気温センサが利用不可能になった場合は，もう一方の気温センサに切り替えるといった処理も必要であろう．また，部屋 C には湿度センサが存在しておらず，気温センサは存在するものの利用できない状態となっており，なんらかの例外処理が必要となる．

このように，IoT アプリでは，ユーザ 1 の場合のような依存性の静的注入だけでなく，ユーザ 2 の場合のような依存性の動的注入の仕組みが求められる．

2.2.2 多重化の課題

IoT リソースの多重化に関する一つ目の課題が「課題 3：制御機構」である．端末内リソースの多重化では，OS がリソースを時分割・空間分割し，各アプリに利用権限を割り当てて排他制御とアクセス制御を実現している．しかし，IoT デバイスは端末外のリソースであり，リソースとアプリが一つの OS の配下で管理されていない．したがって，OS と連携して IoT リソースの制御をサポートする機構が必要である．

IoT リソースの多重化に関する二つ目の課題が「課題 4：排他制御」である．複数の IoT アプリを同時に実行した際に，IoT リソースに対する IoT アプリの要求が競合し，ユーザの意図しない動作が発生することが起こり得る．IoT アプリの競合は，機器競合と環境競合に分類される．機器競合は，異なる IoT アプリが一つの機器に対して同時に両立し得ない機器操作を要求した場合（例えば，ある IoT デバイスに対し，IoT アプリ 1 が電源 on を，IoT アプリ 2 が電源 off を要求）に発生する競合である．環境競合は，異なる IoT アプリが異なる機器に対して機器操作を要求したとき，それらを取り巻く環境の観点から両者が矛盾・干渉する場合（例えば，IoT アプリ 1 がクーラーの電源 on を要求し，IoT アプリ 2 がその部屋のストーブの電源 on を要求）に発生する競合である[5]．これらの競合を調停する必要がある*.

IoT リソースの多重化に関する三つ目の課題が「課題 5：アクセス制御」である．IoT リソースを利用することができるのは，そのリソースの利用を許可されているユーザ（が使用している IoT アプリ）のみである．そして，ユーザの利用権限は，ユーザや機器の状況やコンテキストによって変わり得る．これらを自動的に調停するアクセス制御の仕組みが必要である．

IoT リソースの多重化に関する四つ目の課題が「課題 6：プライバシ制御」である．端末内のリソースと異なり，IoT リソースは様々な環境に偏在しており，不特定（利用権限は有する）多数のユーザに利用される．このとき，IoT リソースの設定，利用ログなどからセンシティブな情報が流出する可能性がある．このため，プライバシ制御の仕組みが必要である．

3. 提案フレームワーク

2 章で述べた課題 1〜6 を解決し，IoT リソースの仮想化をサポートするフレームワークを提案する．提案フレームワークの機能一覧を表 2 に示す．四つのコンセプトに基づく七つの機能により，課題 1〜6 が解決される．

3.1 コンセプト I：仮想空間と物理空間の分離

提案フレームワークでは，IoT リソースが機能単位で仮想化されており，IoT リソースの機能の実体は，アプリが IoT リソースを使用する時点で，その時々の物理コンテキストに応じた形でアプリに動的に注入される（図 4）．これにより仮想空間と物理空間が分離され，IoT アプリと IoT デバイスをそれぞれ独立に開発できるようになる．

3.1.1 機能 1：機能を単位とした API

IoT デバイスの種類は膨大であり，複数のデバイスベンダが同種の機器を製造・販売しており，また，異なる種類の機器であっても同じ機能を提供するデバイス（例えば，扇風機とクーラー）も多々存在する．このような状況に鑑

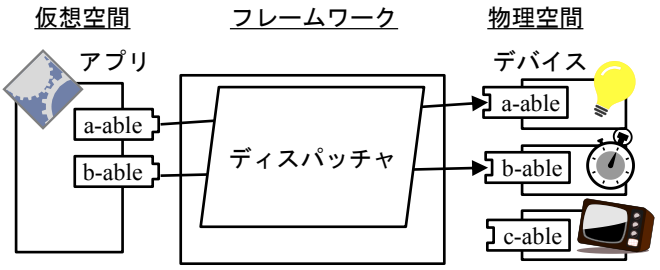

図 4　仮想空間と物理空間を分離して記述

*道路を 3 メートル四方の空間にグリッド分割し，各グリッドを IoT リソースと考える．車 1 を IoT アプリ 1 と考え，車 2 を IoT アプリ 2 と考える．IoT アプリ 1 と IoT アプリ 2 による IoT リソースの利用に対し，排他制御が実現できれば，車 1 と車 2 の衝突を回避できる．

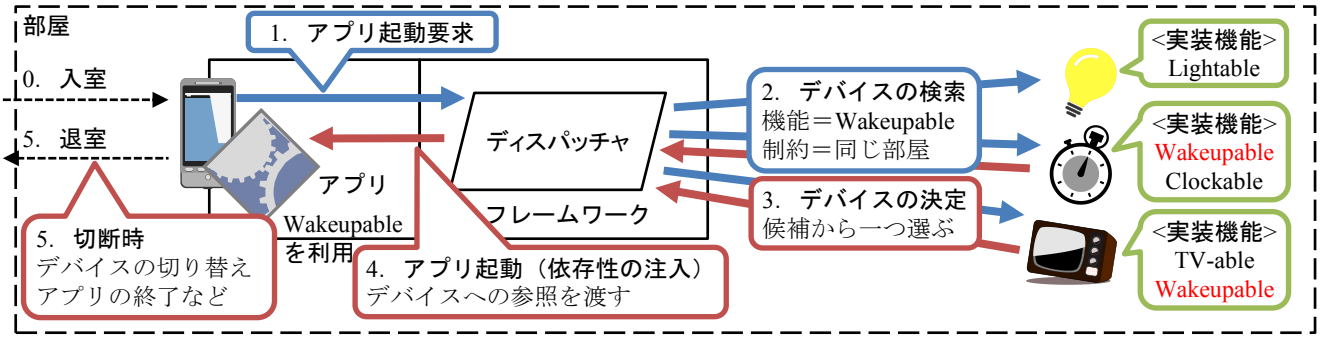

図 5　ディスパッチャの機能（動的な依存性の注入）

みるに，IoT リソースを機器ごとに個別に API 化するのではなく，IoT リソースを機能単位で API 化するアプローチが得策と言える．

API の名前は「xxx 機能を持っている」という意味で「xxx-able」という命名規則で定める．例えば，照明機能を意味する API は「Lightable」とする．xxx-able の API はメソッド，プロパティ，シグナルの 3 種類の要素で表現する．

このように機能単位で API を定義することで，IoT デバイスの機種の違いを隠ぺいできる．例えば，異なるメーカの照明機器を Lightable という一つの API で利用できるようになる．また，ポリモーフィズムの利用も可能である．例えば人を起こすという機能「Wakeupable」を考える．物理空間側においては，目覚まし時計がベルを鳴らすことで Wakeupable を実装したり，テレビが大音量で番組を流すことで Wakeupable を実装したりすることができる．仮想空間側においては，IoT アプリは，テレビや目覚ましといった具体的な物理デバイスのことは意識せずに，Wakeupable の API コールを用いて「人を起こす」という機能を利用できる．

3.1.2 機能 2：依存性の動的注入

提案フレームワークでは，IoT アプリが IoT リソースを使用する時点で，IoT リソースの機能の実体がアプリに動的に注入される．これを「バインディング」と呼ぶこととする．この機能を提供する機構が図 4 のディスパッチャである．物理空間は適切な範囲で区画化されており，区画ごとにディスパッチャが稼働している．ディスパッチャは，自区画内に存在する IoT デバイスを常にモニタしている．

ディスパッチャの仕組みを図 5 を用いて説明する．ユーザの端末が当該区画に入室すると，ディスパッチャは端末 OS のサポートを開始する（図中 0）．端末には，Wakeupable の API を利用する IoT アプリがインストールされていると する．ユーザによって当該アプリが起動されると（図中 1），その時点で当該区画内に存在しており，かつ，Wakeupable の機能を有する IoT デバイスの中から，当該ユーザによる Wakeupable の利用が許可されている IoT デバイスを，ディスパッチャがリストアップする（図中 2）．これが，この IoT アプリが API コールを発行した際に，Wakeupable の機能を実際に提供し得る IoT デバイス（IoT リソースの実体）の候補である．候補デバイスが複数ある場合は，ユーザにその旨を通知して選択してもらったり，履歴から利用頻度の高いものを選択するといった方法によって，一つの IoT デバイスを決定する（図中 3）．利用するデバイスが決まったら，ディスパッチャは，「デバイスの URI（Wakeupable への参照）」を IoT アプリへ渡した上で，アプリを起動する（図中 4）．アプリ実行中に当該デバイスが利用不可能になった場合には，ディスパッチャはアプリを終了させたり，別の利用可能デバイスに切り替えたりする（図中 5）．

このように，ディスパッチャにより，IoT リソースと IoT アプリがバインディングされる（IoT リソースの機能の実体が IoT アプリに動的に注入される）．ディスパッチャは，これら一連の処理を IoT アプリから透過的に行う．

3.2 コンセプト II：仮想空間における UGC 型アプリ開発

仮想空間と物理空間を分離したことにより，IoT アプリ側では，IoT リソースの実体を考慮することなく，API コールという形で IoT リソースの機能を利用してアプリを開発することができるようになる．この結果，IoT アプリの開発負荷が減少し，UGC 型の IoT アプリ開発が促進される．

3.2.1 機能 3：機能のマッシュアップ（メタ機能）

UGC 型の IoT アプリ開発を促進するために，既存のプログラム資産を再利用・マッシュアップして，より高度な IoT アプリを次々と開発できるような仕組みを構築する．マッシュアップによって作成されたより高度な API を「メタ API」と呼ぶこととする．

Alphaable.java

```
1  interface Alphaable{
2     void methodAlpha();
3  }
```

MetaAPI1.java

```
1  class MetaAPI1 implements Alphaable {
2     Aable a;
3     Bable b;
4     MetaAPI1 (Aable a, Bable b){
5        this.a = a;
6        this.b = b;
7     }
8     @Override
9     public void methodAlpha(){
10       // a,b を用いて Alphaable のメソッドを実装
11    }
12 }
```

図 6　メタ API の実装例

図 6 にメタ API の実装例を示す．「Alpha-able.java」が，メタ API をインタフェースとして定義しているプログラムコードである．提案フレームワークでは機能単位で API を定義することとなっており，メタ API も API の一種であるので，メタ API も「(API のマッシュアップによって作成された高度な）機能」として扱われる．これを「メタ機能」と呼ぶこととする．この例では，Alpha-able というメタ機能を定義している．「MetaAPI1.java」が，メタ API のクラス実装に関するプログラムコードである．クラスはメタ機能を実装するクラスとして作成する（1 行目）．ここでは，Alpha-able を実装する MetaAPI1 というクラスを作成している．メタ機能の構成要素となる API は，コンストラクタの引数に記述する（4 行目）．この例では，A-able という API と，B-able という API をマッシュアップしている．メタ機能の中身は，引数で受け取った A-able と B-able の参照を用いて実装する（10 行目）．実行時には，ディスパッチャによって，「バインディングされる IoT リソース（物理デバイス）への参照」が引数として与えられ，コンストラクタが実行される．

メタ API を更にマッシュアップして，更に高機能なメタ API を実装することも可能である．この様子を図 7 に示す．図中の黒矢印はディスパッチャによるバインディングを表

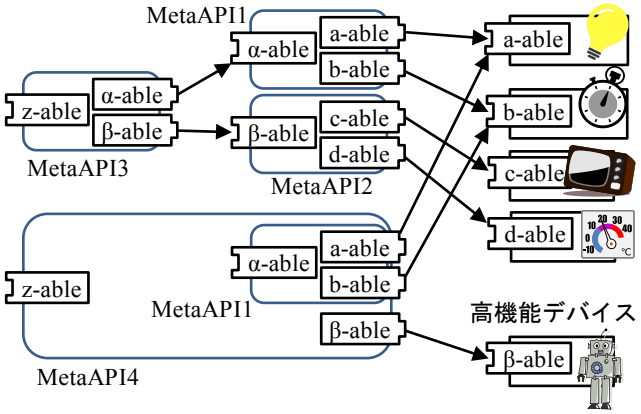

図 7　メタ API のマッシュアップ

Zable.java

```
1  interface Zable{
2     void methodZ();
3  }
```

MetaAPI3.java

```
1  class MetaAPI3 implements Zable {
2     Alphaable alpha;
3     Betaable beta;
4     MetaAPI3 (Alphaable alpha, Betaable beta){
5        this.alpha = alpha;
6        this.beta = beta;
7     }
8     @Override
9     public void methodZ(){
10       // alpha, beta を用いて Zable のメソッドを実装
11    }
12 }
```

MetaAPI4.java

```
1  class MetaAPI4 implements Zable {
2     Alphaable alpha;
3     Betaable beta;
4     MetaAPI4 (Aable a, Bable b, Betaable beta){
5        this.alpha = new MetaAPI1(a, b);
6        this.beta = beta;
7     }
8     @Override
9     public void methodZ(){
10       // alpha, beta を用いて Zable のメソッドを実装
11    }
12 }
```

図 8　メタ API を再利用するプログラムの例

している．a-able や b-able のような「IoT デバイスが提供する API」をマッシュアップすることによって，α-able やβ-able のようなメタ API が実装され，更に，α-able やβ-able をマッシュアップすることによって，より高度なメタ API である Z-able が実装されている．図 8 に高度なメタ API の実装例を示す．「MetaAPI3.java」では，α-able とβ-able というメタ API をマッシュアップして z-able というメタ API を実装している．「MetaAPI4.java」では，ほかのメタ API を内部的にインスタンス化して，これを再利用する形で z-able を実装している．

3.2.2 機能 4：機能・メタ機能のアプリ化

UGC 型の IoT アプリ開発を促進するために，API やメタ API を UI （User Interface）とつなげるだけで容易にアプリ化できるような仕組みを構築する．概観を図 9 に示す．a-able のような抽象度の低い API も，α-able，z-able のよ

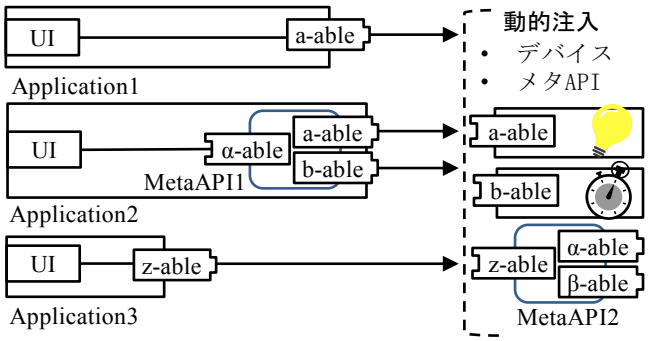

図 9　xxx-able のアプリケーション化

```
Application3.java
1  class Application3 extends IoTApplication {
2      Alphaable alpha;
3      Application3 (Alphaable a){
4          this.alpha = a;
5          // alpha を UI で操作する部分のみを記述
6          addButtonListener(BUTTON_ID1, ()->{
7              alpha.methodAlpha();
8          });
9      }
10 }
```

図 10　IoT アプリのプログラム例

うな高機能なメタ API も, UI を結合するだけですべて統一的にアプリ化できる. これによって, メタ API の開発と IoT アプリの開発はほぼ等価となり，アプリ開発のコストが削減される.

IoT アプリのプログラム例を図 10 に示す．アプリは，UI 機能を備えたアプリ開発用の基底クラスを継承したクラスとして作成する（1 行目）．コンストラクタには，アプリ化したい API（メタ API）を記述する（3 行目）．UI の実装は，UI 操作と API（メタ API）の操作を関連付けるコードとして記述する（6〜7 行目）．「Application3.java」では，ボタンイベントに Alpha-able のメソッドを対応付けている．アプリの実行時には，メタ API と同じようにディスパッチャによってコンストラクタの Alpha-able が動的にバインディングされる.

3.3 コンセプト III：カプセル型リソース管理

提案フレームワークでは，IoT デバイスごとに仮想オブジェクトが設けられており，アプリが IoT リソースを使用する際の競合管理やアクセス制御を IoT デバイス側で自律的に行う.

3.3.1 機能 5：物理デバイスと常に接続されたシャドウ

IoT デバイスには，電源やネットワーク接続が切れてしまうといった可用性の問題がある．IoT デバイスがオフラインになった場合，IoT デバイスが再度オンラインになり次第，IoT デバイスでの処理が再開されるべきである．よって, IoT デバイスがオフラインの間も, IoT アプリから IoT デバイスへ発行されたクエリなどについては，その損失を防ぐ必要がある.

これを実現するために，提案フレームワークでは，IoT デバイスと一対一で対応する仮想的なオブジェクトを作成し，これをクラウドなどの信頼性の高いストレージに配置する．IoT デバイスへのクエリはすべてこの仮想オブジェクトを仲介することで，クエリの保管や持続的な応答を実現する. この仕組みは AWS IoT [6]ではシャドウと呼ばれており，ほかの IoT プラットフォームでも同様の仕組みが用いられている.

シャドウは，物理的な IoT デバイスの機能を強化する仮想的なカプセルである．例えば，IoT デバイスはハードウエアリソースが限られている場合があるが，シャドウを用いることで，低リソースのデバイスに対しても，シャドウ側に機能を追加することで，仮想的にデバイスを高機能化できる.

3.3.2 機能 6：デバイス単位の排他・アクセス・プライバシ制御

複数の IoT アプリがある一つの IoT リソースを同時に利用しようとした際に，排他制御が行われることになる．このように IoT リソースの場合も，物理デバイス単位での排他制御が基本となる．提案フレームワークではデバイスごとにシャドウが用意されるので，排他制御の管理はシャドウに行わせることが可能である.

ただし，同じ部屋の中でストーブとクーラーを同時に動かしてしまうというような環境競合に対しては，ストーブ単体あるいはクーラー単体での対策は不可能である．この問題に対しては，部屋を IoT デバイスと見做してシャドウを設置し，室内気温という「プロパティ」に対する排他制御を行うことで競合を回避できると考えている.

IoT リソースに対するアクセス制御を実現するためには, IoT デバイスごとにアクセスポリシを設定した上で，「IoT アプリ利用者のユーザプロファイル」と「IoT デバイスのアクセスポリシ」の適否を検査することとなる.

端末内リソースにおいては，物理メモリを空間分割し，排他制御することでサンドボックスによるプライバシ制御を実現していた．IoT リソースの場合も，排他制御によるリソースのサンドボックス化によるプライバシ制御を実現できると考えられる.

3.4 コンセプト IV : 仮想空間からの UGC 型デバイス開発

シャドウによって，物理的な IoT デバイスを仮想的なオブジェクトとして扱うことが可能となる．これは，仮想空間側のプログラム資産である「IoT アプリ開発者が作成したメタ機能」を, シャドウに組み込むという方法によって，IoT デバイスの機能の仮想的に拡張できることを意味する．すなわち，UGC 型の IoT デバイス開発が実現する.

3.4.1 機能 7：デバイス機能の実装

図 5 の照明デバイスは Lightable を実装している IoT デバイスである. このデバイスは Wakeupable の機能を持っていないが,「照明を激しく点滅させる」ことで人を起こすことができる可能性がある．そこで，Lightable の API をマッシュアップすることによって,「照明を激しく点滅させる」というメタ機能を実装して，このメタ API を Wakeupable という名前で登録する．これを，この照明デバイスのシャドウに組み込めば, この照明デバイスを（Lightable に加え）Wakeupable の機能を実装する IoT デバイスとして利用することができるようになる．このように，デバイスベンダだけでは想定しきれないような IoT デバイスの機能を，利用者が自由に考えて実装することが可能である.

なお，IoT デバイス（物理デバイス）自体がプログラマブルであり，かつ，デバイスベンダから当該 IoT デバイスの SDK（Software Development Kit）が提供されている場

合は，IoT デバイスのデバイスドライバを直接改造することもできる．ただし，この場合は，デバイスベンダが用意した機種依存の命令を用いてのドライバ開発となるため，メタ API を用いる方法と比べ，コードの再利用性は低くなる．

4. 関連研究・技術

この章では IoT アプリの UGC 型エコシステムの開発に関する関連技術・研究を紹介する．IoT アプリのエコシステム形成を目指していると考えられるものについて，フレームワークの機能比較を表 3 に示す．

AWS IoT [6]，IBM Bluemix [7]，Kii Cloud [8]では，IoT デバイスをクラウドに接続・管理する機能を提供しており，クラウド上の仮想オブジェクトとして API を提供することが可能である．アプリは API を用いて仮想オブジェクトを利用する．これらのフレームワークでは，IoT デバイスの機能の定義を開発者自身がクラウド登録時に設定するため，アプリとデバイスの機能面での依存性が強くなりやすい．また，コンテキストに応じて動的に接続する IoT デバイスを切り替えるような仕組みがアプリ開発者に提供されていない．

Linking [9]では，IoT デバイスの機能の定義を標準化しており，IoT アプリと IoT デバイスを別々に開発可能である．アプリとデバイスのバインディングは Linking が提供する専用アプリが管理する．このバインディングの管理は静的なものであり，動的に依存性を注入する機能は提供されていない．また階層的なマッシュアップの部品間の接続の管理機能も提供されていない．

AllJoyn [10]，HomeKit [11]では IoT デバイスの機能の定義を標準化しており，同一ネットワーク内などから特定の機能を持ったデバイスを検索するモジュールをアプリから利用できる．これによって，動的なバインディングが可能となるが，これらの処理をアプリ側で記述する必要があり，動的な依存性の注入は行われていない．

WoT [12]では，IoT デバイス機能の標準化，フレームワーク側でのバインディング管理などを提供しているが，階層的なマッシュアップの部品間の接続などの管理機能は策定されていない．

5. まとめ

現在のスマートフォンアプリケーション市場のように UGC（User Generated Contents）型のサービスアプリケーション開発エコシステムを形成し，ユーザによる IoT サービスの開発を促進するためのフレームワークを提案した．提案フレームワークでは，機能を単位とした API を定義し，これを用いて IoT アプリ・IoT リソース間の依存性を動的に注入することによって，アプリ開発の生産性・可搬性を確保する．また，排他制御・アクセス制御・プライバシ制御などの IoT におけるセキュリティの課題を，デバイス単位のカプセル化を行うことで管理する．これらによって，アプリ開発者は機能をマッシュアップすることにのみ集中して物理的なコンテキストに依存しない IoT サービスを容易に開発できる．

表 3　フレームワークの機能比較

フレームワーク	機能						
	1	2	3	4	5	6	7
AWS IoT [6]	×	×	×	○	○	△	△
IBM Bluemix [7]	×	×	×	○	○	△	△
Kii Cloud[8]	×	×	×	○	○	△	△
Linking [9]	○	△	×	○	○	△	○
AllJoyn [10]	○	△	×	○	○	△	○
HomeKit [11]	○	△	×	○	○	△	△
WoT [12]	○	○	△	○	○	△	△
提案方式	○	○	○	○	○	○	○

今後の検討課題としては，アプリ開発の容易さの評価，マッシュアップによる性能への影響評価，様々な抽象度の API の中からアプリ開発者が必要なものを検索するための仕組み，ディスパッチャのインテリジェンス化，悪意のあるアプリ対策，ヒューマンファクターへの対応などがある．

謝辞　本研究において貴重なご意見を頂きました，株式会社富士通研究所 司波章氏に厚く御礼申し上げます．

参考文献

[1] 総務省：総務省｜平成 28 年版 情報通信白書｜IoT 時代における ICT 産業動向分析，入手先〈http://www.soumu.go.jp/johotsusintokei/whitepaper/ja/h28/html/nc120000.html〉（参照 2017-01-26）

[2] 総務省：総務省｜平成 28 年版 情報通信白書｜プロダクトにおける IoT の導入事例，入手先〈http://www.soumu.go.jp/johotsusintokei/whitepaper/ja/h28/html/nc123220.html〉（参照 2017-04-30）

[3] 地主岳史，知崎一紘，川上裕介：IoT 活用による工場の生産活動最適化，富士通，入手先〈https://www.jpo.go.jp/shiryou/kijun/kijun2/pdf/handbook_shinsa_h27/app_z.pdf〉（参照 2017-04-18）

[4] Andrew S. Tanenbaum（著）．水野忠則，太田剛，最所圭三，福田晃，吉澤康文（訳）: モダンオペレーティングシステム（原著第 2 版），pp.4-5，ピアソン・エデュケーション・ジャパン（2004）

[5] 池上弘祐，稲田卓也，まつ本真佑ほか：ホームネットワークシステムにおける環境インパクトの性質を考慮した環境競合の再定式化，電子情報通信学会技術研究報告，電子情報通信学会技術研究報告,Vol.111，No.255，pp.67-72（2011）．

[6] Amazon Web Services : AWS IoT プラットフォームの仕組み - アマゾン ウェブ サービス，入手先〈https://aws.amazon.com/jp/iot-platform/how-it-works〉（参照 2017-01-20）

[7] IBM Bluemix : IBM developer Works 日本語版 : IBM Bluemix，入手先〈http://www.ibm.com/developerworks/jp/bluemix〉（参照 2017-01-20）

[8] Kii Corporation : IoT クラウドプラットフォーム - Kii 株式会社，入手先：〈https://jp.kii.com/iot〉（参照 2017-04-10）

[9] Project Linking : Project Linking，入手先〈https://linkingiot.com〉（参照 2017-01-20）
[10] AllSeen Alliance : AllSeen Alliance，入手先〈https://allseenalliance.jp〉（参照 2017-01-20）
[11] Apple Developer : HomeKit デベロッパガイド，入手先〈https://developer.apple.com/jp/documentation/HomeKitDeveloperGuide.pdf〉（参照 2017-01-20）
[12] W3C : W3C Web of Things at W3C，入手先〈https://www.w3.org/WoT/〉（参照 2017-04-20）