intdash Edge Agent 用
デバイスコネクター
デベロッパーガイド

Device Connector Framework v2.0.0

第 1 版（2023 年 4 月）

aptpod

Rev. 4e7ecd6

目次

01 はじめに

重要:

- このドキュメントに記載されている仕様は予告なく変更される場合があります。このドキュメントは情報提供を目的としたものであり、仕様を保証するものではありません。
- 説明で使用している画面は一例です。ご使用の環境やアプリケーションのバージョンによって、表示や手順が一部異なる場合があります。

注釈: このドキュメントに記載されている会社名、サービス名、製品名等は、一般に、各社の登録商標または商標です。本文および図表中には、「™」、「®」は明記していません。

本ドキュメントは、intdash Edge Agent 用デバイスコネクターを使用・開発する方のために、以下を説明するものです。

- Device Connector Framework を使って開発されたデバイスコネクターの設定方法（エレメントを使ってパイプラインを組む方法など）
- Device Connector Framework を使って独自デバイスコネクターを開発する方法

02 デバイスコネクター開発の概要

2.1 デバイスコネクターとは

デバイスコネクターは、intdash エッジにおいて、intdash Edge Agent と外部デバイスとの間でデータを仲介するソフトウェアコンポーネントです。

デバイスコネクターは intdash Edge Agent と同じエッジデバイスにインストールされ、以下のような機能を担います。

- センサーなどの外部デバイスからデータを受け取り、intdash Edge Agent に渡す。
- 逆に、intdash Edge Agent からデータを受け取り、ロボットなどの外部デバイスに渡す。

デバイスコネクターと intdash Edge Agent の間のデータの受け渡しには、FIFO（名前付きパイプ）を使います。

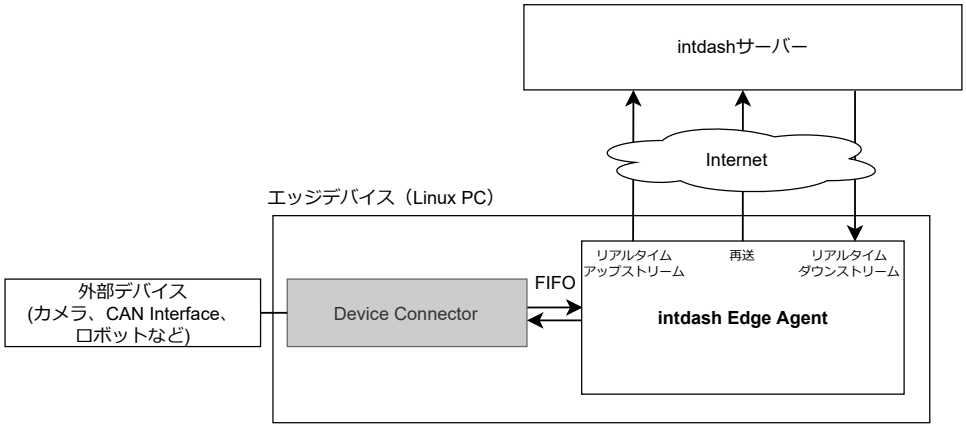

図 1　デバイスコネクター

2.2 Device Connector Framework とは

Device Connector Framework は、デバイスコネクターを開発する方のための開発用フレームワークです。

- Device Connector Framework により作成されたデバイスコネクターでは、デバイスへの接続やデータ収集、収集したデータに対する処理は、それぞれ「エレメント (Element)」として定義されます。ユーザーは、エレメントを組み合わせて、データが流れるパイプラインを設定することができます。設定はパイプライン設定ファイルで行います。
- デバイスコネクターを実行すると、パイプライン設定ファイルで指定された各エレメントは、同期または非同期に動作する「タスク（Task)」として実行されます。また、タスク間でのメッセージパッシングはデバイスコネクターにより管理されます。

これにより、柔軟にエレメントを組み合わせて、データ（メッセージ）が流れるパイプラインを作成することができます。

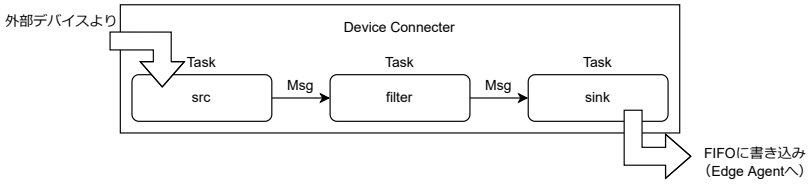

図 2　デバイスから intdash Edge Agent へ入力するパイプラインの例

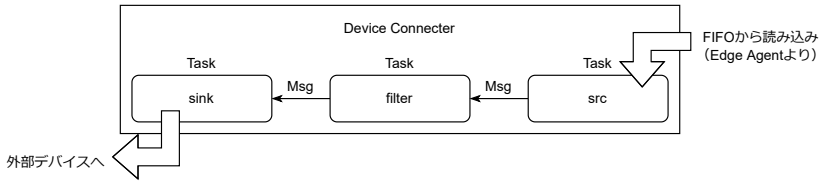

図 3　intdash Edge Agent からデバイスへ出力するパイプラインの例

なお、ファイルからの読み込み、ファイルへの書き出しなどの基本的な処理を行うエレメントは Device connector Framework に含まれています（基本エレメント）。

2.3 Device Connector Framework で使用される用語

Device Connector Framework で使用される用語について説明します。

2.3.1 メッセージ

タスク間で送受信されるデータです。

タスク間で送受信されるメッセージは、何らかの型を持ちます。型情報は、MIME タイプ（例：`MIME` タイプ `"image/jpeg"`）またはカスタムタイプ（例：カスタムタイプ `"iscp-v2-compat-msg"`）です。

各ポートがどの型のメッセージを受け取れるかは、エレメントにおいて定義されます。

送信元のタスクは、メッセージを最初に送信するとき、そのメッセージの型をデバイスコネクターに通知します。デバイスコネクターは、メッセージを受信するポートがその型を受け入れ可能かどうかを判定し、受け入れができない場合はエラーとします。

データを、MIME タイプ "image/jpeg"、MIME タイプ "text/plain"、カスタムタイプ "iscp-v2-compat-msg" 等で送出するエレメントの場合は、1 枚の画像、1 行のテキスト、1 つのデータポイントのように、意味を持つ単位を 1 つのメッセージとして送出します。

それに対し、MIME タイプ "application/octet-stream" で送出するエレメントの場合は、送出するデータは区切りのないバイナリストリームであるため、意味的な区切りとは関係なく送信上の都合でセグメントに分割して送信します。この場合のセグメントもここではメッセージと呼びます。

2.3.2 エレメント

メッセージの生成やメッセージの処理を定義したものです。デバイスコネクター内にメッセージを生成する「src エレメント」、受け取るだけの「sink エレメント」、送受信を共に行う「filter エレメント」があります。

エレメントは、実行時には「タスク」として実行されます。

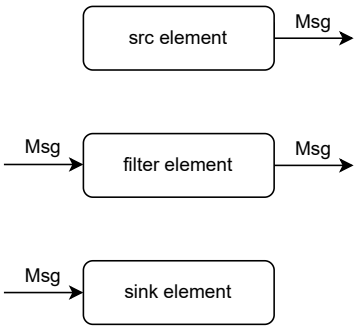

図 4　エレメントの種類（src、filter、sink）

デバイスコネクター内のパイプラインでは、以下の順序で処理が行われます。

- src エレメントにおいて、デバイスコネクターの外部（例えばセンサー）からデータを取得して、メッセージを生成
- filter エレメントにおいて、メッセージを意味のある単位に区切る、変換する、などの処理を行う
- sink エレメントにおいて、デバイスコネクターの外部（例えば intdash Edge Agent）にデータを出力する

注釈: アプトポッドにより提供されるエレメントは以下のような名前になっています。

- src エレメント: `*-src`
- filter エレメント: `*-filter`
- sink エレメント: `*-sink`

例えば、入力を jpeg のフレームごとのメッセージにするのは filter エレメントの役目で、jpeg-split-filter と名付けています。

2.3.3 タスク

エレメントにおいて定義された処理が実際に実行されるとき、その処理はタスクと呼ばれます。タスクは、メッセージの送受信が必要になったときに起動されます。

2.3.4 ポート

エレメントには、0 個以上 255 個以下の受信ポートと、0 個以上 255 個以下の送信ポートを定義することができます。複数のポートを使うことによって、ポートごとに別の型のメッセージを送受信させることができます。

- 複数の送信ポートからのメッセージを、1 つの受信ポートで受信することができます。
- 1 つの送信ポートから、複数の受信ポートに向けてメッセージを送信することはできません。

注釈: Device Connector Framework v1.0 現在、送受信ともに 0 番ポート 1 つのみを使用することができます。今後のリリースで、順次複数のポートに対応する予定です。

03 デバイスコネクターをビルドし、動かしてみる

この章では、Device Connector Framework を使用してデバイスコネクターをビルドし、動作確認します。ここでビルドされるデバイスコネクターには、基本エレメントが含まれます。基本エレメントについては、Device Connector Framework に含まれる基本エレメント (p. 14) を参照してください。

3.1 開発環境を準備する

Device Connector Framework を使ってデバイスコネクターをビルドするには、Rust の開発環境が必要です。Rust の開発環境をインストールしてください。

3.2 基本エレメントのみのデバイスコネクターをビルドする

Device Connector Framework には基本エレメントのソースコードが付属しています。

以下のコマンドを実行することにより、Device Connector Framework を取得し、基本エレメントを内部に含んだデバイスコネクターをビルドすることができます。

```
git clone https://github.com/aptpod/device-connector-framework.git
cd device-connector-framework
cargo build --release
```

生成された、`./target/release/device-connector-run` という実行ファイルが、基本エレメントを含むデバイスコネクターです。

3.3 パイプライン設定ファイルを作成する

Device Connector Framework により作成されたデバイスコネクターを使用するには、行いたい処理をパイプラインとして定義する必要があります。

例として、以下のパイプライン設定ファイル `conf.yml` をカレントディレクトリに用意します。ここでは、文字列を 100ms ごとに送出するエレメント (text-src) に、受信内容を標準出力に書き出すエレメント (stdout-sink) をつないで、パイプラインを作成しています。

```yaml
tasks:
  - id: 1
    element: text-src
    conf:
      text: "Hello, World!"
      interval_ms: 100

  - id: 2
    element: stdout-sink
    from: [[ 1 ]]
    conf:
      separator: "\n"
```

3.4 デバイスコネクターを実行する

以下のコマンドでデバイスコネクターを実行します。

```
./target/release/device-connector-run --config conf.yml
```

すると、パイプライン設定ファイルで指定した文字列が 100ms ごとに出力されます。

```
Hello, World!
Hello, World!
Hello, World!
Hello, World!
Hello, World!
```

終了するには Ctrl-C を押してください。

以上で、デバイスコネクターの動作確認は終了です。

この例では文字列を標準出力に出力しましたが、代わりに、intdash Edge Agent 用のバイナリフォーマットで FIFO に出力し、intdash Edge Agent がその FIFO から読み取るようにすることでデータを渡すことができます。

04 パイプライン設定ファイル

4.1 パイプライン設定ファイルの基本

パイプライン設定ファイルは、デバイスコネクターで行いたい処理をエレメントの組み合わせで定義したものです。

以下の設定例では、`text-src` エレメントにより定義されたタスクから 100ms ごとに“Hello, World!”という文字列を送出し、`stdout-sink` エレメントにより定義されたタスクがそれを受け取って標準出力に出力します。

また、プラグインファイル `libdc_my_plugin0.so` をロードするように設定しています。

```yaml
runner:
  channel_capacity: 16

plugin:
  plugin_files:
    - /path/to/my/plugin0/libdc_my_plugin0.so

tasks:
  - id: 1
    element: text-src
    conf:
      text: "Hello, World!"
      interval_ms: 100

  - id: 2
    element: stdout-sink
    from: [[ 1 ]]
    conf:
      separator: "\n"
```

設定項目の詳細は以下のとおりです。

4.2 runner の設定

パイプライン設定ファイルの `runner:` には、デバイスコネクターの実行パラメータを記述します。

項目	設定値	説明
channel_capacity	整数値	スレッド間のメッセージ送信に用いるチャンネルの容量（メッセージの個数）です。これを超える容量のメッセージを送信しようとすると、受信側でメッセージが処理されるまで送信側のスレッドは処理がブロックされます。省略した場合デフォルトの値が使用されます。

4.3 bg_processes の設定

```
bg_processes:
  - command: /background/process/path args
```

バックグラウンドで実行するコマンドを指定します。タスクの開始時に `before_task` の前に起動されます。

4.4 before_task の設定

```
before_task:
  - command1
  - command2
```

タスクの開始前に実行されるコマンドを指定します。`bg_processes` と異なり、起動されたプロセスが終了するまでタスクの開始はブロックされます。

4.5 after_task の設定

```
after_task:
  - command1
  - command2
```

デバイスコネクターの終了時、タスクの終了後に実行されるコマンドを指定します。

4.6 plugin の設定

パイプライン設定ファイルの `plugin:` には、実行時にロードするプラグインについて設定します。

独自エレメントをプラグイン形式で作成した場合、ここでそのファイルを指定します。独自エレメントについては、独自エレメントの開発の概要 (p. 18) を参照してください。

項目	設定値	説明
plugin_files	文字列の配列	ロードしたいプライグインファイル (例:libdc_xxx.so) へのパスのリストです。

4.7 tasks の設定

タスクの設定は、`tasks:` 内に並列に記述します。

項目	設定値	説明
id	整数値	タスクに割り当てる ID です。これはこのパイプライン上で一意でなくてはなりません。
element	文字列	このタスクに割り当てるエレメントを指定します。
from	文字列の二次元配列	このタスクが受け取るメッセージの送信元を指定します。 `[["<送信元タスク ID>:<送信元ポート番号>"], ... ]` メッセージは非同期に送られます。 例えば、`[[ "1:0" ], [ "2:1" ]]` は、以下を意味します。 • 0 番ポートが「タスク ID1 の 0 番ポート」からメッセージを受け取る • 1 番ポートが「タスク ID2 の 1 番ポート」からメッセージを受け取る ポート番号が 0 の場合、ポート番号を省略して `"1:0"` を `"1"` のように記述することができます。したがって、`[[ "1:0" ], [ "2:1" ]]` は `[[ "1" ], [ "2:1" ]]` と同義です。

重要: `from` に設定する送信元のタスク ID は、自身のタスク ID より小さい数字である必要があります。

エレメントによっては、独自の設定項目も存在します。エレメント独自の設定項目は、`conf:` に記述してください。

05 Device Connector Framework に含まれる基本エレメント

Device Connector Framework には、パイプラインを作成するための基本的なエレメントが付属しています。

5.1 FileSinkElement (`file-sink`)

ドキュメント

指定されたパスのファイルに、受け取ったメッセージを書き出します。

エレメント独自の設定項目は以下のとおりです。`conf`: 以下に記述します。

エレメントの種類

sink

前のエレメントから受信するポートの数

1

受信ポートでの受信形式

任意（受け取ったデータをそのままファイルに書き出します）

項目	設定値	説明
path	文字列	出力先のファイルパス
create	true/false	true の場合、出力先のファイルがない場合は作成する。デフォルトは false
separator	文字列	メッセージ出力ごとに挿入するテキスト。デフォルトは空
flush_size	整数	入力をフラッシュするまでにバッファにためるデータ量 (バイト)。デフォルトは 0

5.2 FileSrcElement (`file-src`)

ドキュメント

指定されたパスのファイルを読み込み、内容をメッセージとして送出します。

エレメントの種類

src

次のエレメントに送信するポートの数

1

送信ポートでの送信形式

MIME タイプ "application/octet-stream"

エレメント独自の設定項目は以下のとおりです。`conf`: 以下に記述します。

項目	設定値	説明
path	文字列	読み取るファイルのパス

5.3 NullSinkElement (`null-sink`)

ドキュメント

メッセージを受け取りますが、それを何にも使わずに破棄します。このエレメントに設定項目はありません

エレメントの種類

sink

前のエレメントから受信するポートの数

1

受信ポートでの受信形式

任意

5.4 ProcessSrcElement (`process-src`)

ドキュメント

プロセスを起動し、起動したプロセスの標準出力をそのまま送出します。

エレメント独自の設定項目は以下のとおりです。`conf`: 以下に記述します。

エレメントの種類

src

次のエレメントに送信するポートの数

1

送信ポートでの送信形式

MIME タイプ "application/octet-stream"

項目	設定値	説明
program	文字列	実行ファイル
args	文字列の配列	実行ファイルに渡す引数
command	文字列	実行ファイルと引数を 1 文字列で指定したもの。program と args が指定されなかった場合に使用。

5.5 StatFilterElement (`stat-filter`)

ドキュメント

データを受け取った回数やサイズを確認するためのエレメントです。メッセージを受け取り、それをそのまま送出します。受け取ったメッセージの回数とサイズを記録し、その統計情報を標準エラー出力に書き出します。

エレメントの種類

filter

前のエレメントから受信するポートの数

1

次のエレメントに送信するポートの数

1

受信ポートでの受信形式

任意

送信ポートでの送信形式

受信ポートに入力されたデータをそのまま出力

エレメント独自の設定項目は以下のとおりです。`conf`: 以下に記述します。

項目	設定値	説明
interval_ms	整数	出力頻度をミリ秒にて指定

5.6 StdoutSinkElement (`stdout-sink`)

ドキュメント

受け取ったメッセージを標準出力に書き出します。

エレメントの種類

sink

前のエレメントから受信するポートの数

1

受信ポートでの受信形式

任意

エレメント独自の設定項目は以下のとおりです。`conf`: 以下に記述します。

項目	設定値	説明
separator	文字列	メッセージ出力ごとに挿入するテキスト。デフォルトは空

5.7 TextSrcElement (text-src)

ドキュメント

指定されたテキストを送出します。

エレメントの種類

src

次のエレメントに送信するポートの数

1

送信ポートでの送信形式

MIME タイプ "application/octet-stream"

エレメント独自の設定項目は以下のとおりです。`conf:` 以下に記述します。

項目	設定値	説明
text	文字列	メッセージのテキスト
interval_ms	整数値	メッセージ生成の間隔をミリ秒にて指定
repeat	整数値	「メッセージ送出のタイミングごとに、メッセージを何回送出するか」を指定します。デフォルトは 1 です例えば `interval_ms: 100` かつ `repeat: 5` の場合、100 ミリ秒経過ごとに 5 回メッセージを送出します。

06 独自エレメントの開発の概要

Device Connector Framework では、独自エレメントを開発し、パイプラインに組み込むことができます。ここでは、新たにエレメントを開発する方法を解説します。

独自エレメントを使用する方法には、以下の 2 つがあります。

- プラグインにしてデバイスコネクター実行時にロードする（Rust または C）(p. 18)
- デバイスコネクターの実行ファイルに含める（Rust）(p. 19)

独自エレメントを Rust 言語で開発する場合は、どちらの方法も使用可能です。

独自エレメントを C 言語で開発する場合は、プラグインにする方法のみ使用可能です。

6.1 プラグインにしてデバイスコネクター実行時にロードする（Rust または C）

Device Connector Framework で作成されたエレメントは、プラグインの形式（共有ライブラリファイル、.so 形式）にビルドすることができます。1 つのプラグインには、複数のエレメントを定義することが可能です。

ビルドされたプラグインを使用するには、パイプライン設定ファイルの `plugin.plugin_files` に、プラグインファイルのパスの配列を指定します。

```
runner:
  ..

plugin:
  plugin_files:
    - /path/to/my/plugin0/libdc_my_plugin0.so
    - /path/to/my/plugin1/libdc_my_plugin1.so

tasks:
  - id: 1
    element: my-plugin0-foo-element # defined in loaded plugin
  ..
```

独自エレメントのプラグインは、Rust または C で開発します。詳細は以下を参照してください。

- Rust による独自エレメントの開発 (p. 20)
- C による独自エレメントの開発 (p. 28)

6.2 デバイスコネクターの実行ファイルに含める（Rust）

Device Connector Framework で作成されたエレメントは、デバイスコネクターの実行ファイルに含めてビルドすることも可能です。

必要なエレメントがデバイスコネクターの実行ファイルに含まれていればプラグインは必要ないため、パイプライン設定ファイルに `plugin` の記述は不要になります。

ただしこの場合、独自エレメントを追加・修正するたびにデバイスコネクターの実行ファイルを再ビルドする必要があります。

独自のエレメントをデバイスコネクター実行ファイルに含めるには、エレメントを Rust で開発する必要があります。エレメントの開発方法はプラグインの場合と同様です。詳細は以下を参照してください。

- Rust による独自エレメントの開発 (p. 20)

07 Rust による独自エレメントの開発

Device Connector Framework では、Rust で独自エレメントを開発することができます。

> **注釈:** エレメントをプラグインにする場合も、デバイスコネクター実行ファイルに含める場合も、この後の エレメントをプラグインにする／実行ファイルに含める (p. 24) の前までは同じ手順です。

7.1 テンプレートを利用した準備

エレメントの開発のために、cargo-generate 用のテンプレートが用意されています。

> **注釈:** cargo-generate は、既存の Rust プロジェクトをテンプレートにして新しいプロジェクトを作成するためのツールです。インストールしていない場合、次のコマンドでインストールしてください。
>
> ```
> cargo install cargo-generate
> ```

次のコマンドでデバイスコネクターの新しいプロジェクトを作成します。新しいプロジェクトの名前の入力を求められるので、任意の名前を入力してください。

```
cargo generate --git https://github.com/aptpod/device-connector-template.git
```

これで新しいプロジェクトが作成されます。

このプロジェクトには、以下の 2 つのエレメントのソースコードがあらかじめ含まれています。

- `src/hello.rs` (`hello-src`)
- `src/hexdump.rs` (`hexdump-sink`)

7.2 `ElementBuildable` トレイト

Rust では、型に何らかの属性や実装を付与するためにトレイトと呼ばれる機能が提供されています。独自エレメントを定義するには、そのエレメントにおいて `ElementBuildable` トレイトを実装する必要があります。

`ElementBuildable` トレイトの定義は以下のとおりです。

```rust
pub trait ElementBuildable: Sized + Send + 'static {
    type Config: DeserializeOwned;
    const NAME: &'static str;
    const RECV_PORTS: Port = 0;
    const SEND_PORTS: Port = 0;
    fn acceptable_msg_types() -> Vec<Vec<MsgType>> {
        Vec::new()
    }
    fn new(conf: Self::Config) -> Result<Self, Error>;
    fn next(&mut self, pipeline: &mut Pipeline, receiver: &mut MsgReceiver) -> ElementResult;
    fn finalizer(&mut self) -> Result<Option<ElementFinalizer>, crate::error::Error>;
```

(次のページに続く)

(前のページからの続き)

```
}
```

Config

このエレメントを構築するのに必要な設定項目を指定します。デシリアライズ可能なデータ型が指定できますが、一般的には `#[derive(Debug, Deserialize)]` を指定した構造体にします。設定を受け取らない場合、`device_connector::EmptyElementConf` を指定します。

NAME エレメントの名前を指定します。他のエレメントとの重複は許されません。他のエレメントの名前と重複していると、実行時にエラーになります。

RECV_PORTS

メッセージを受け取るポートの数です。src エレメントの場合は 0 を指定します。filter または sink エレメントの場合は 1 以上を指定します。デフォルトは 0 です。

SEND_PORTS

メッセージを送出するポートの数です。sink エレメントの場合は 0 を指定します。src または filter エレメントの場合は 1 以上を指定します。デフォルトは 0 です。

acceptable_msg_types()

受け取ることのできるメッセージの型情報の配列を返す関数です。メッセージを受け取らない src エレメントの場合は実装する必要はありません。

new()

`Config` で指定した設定情報を受け取り、エレメントを実際に構築して返すメソッドです。

next()

データを受け取るための `pipeline` を受け取り、エレメントの実行結果を返します。`next()` は、`MsgBuf` に有効なデータを書き出すか終了するまで、処理を返さないように実装します。

finalizer()

プロセス終了時に実行されるクロージャを返すメソッドです。このクロージャではエレメント固有の終了処理を定義します。このメソッドを実装しない場合、終了処理を行わない挙動（デフォルト）となります。

7.3 HelloSrcElement を作る

実例として、device-connector-template に含まれているサンプルのエレメント `HelloSrcElement` を見ていきます。

このエレメントは、一定時間ごとにテキストを送信します。

```rust
use device_connector::{error::Error, ElementBuildable, ElementResult, MsgType, Pipeline, Port};
use serde_derive::Deserialize;
use std::io::Write;
use std::thread::sleep;
use std::time::Duration;

// ElementBuildable を実装するための対象となる型
pub struct HelloSrcElement {
    conf: HelloSrcElementConf,
```

(次のページに続く)

(前のページからの続き)

```
}

// HelloSrcElement が受け取る設定の定義
#[derive(Debug, Deserialize)]
#[serde(deny_unknown_fields)]
pub struct HelloSrcElementConf {
    text: String,
}

// ElementBuildable の実装
impl ElementBuildable for HelloSrcElement {
    type Config = HelloSrcElementConf;

    const NAME: &'static str = "hello-src";

    const SEND_PORTS: Port = 1;

    fn new(conf: Self::Config) -> Result<Self, Error> {
        Ok(HelloSrcElement { conf })
    }

    fn next(&mut self, pipeline: &mut Pipeline, _receiver: &mut MsgReceiver) -> ElementResult {
        pipeline.check_send_msg_type(0, || MsgType::from_mime("text/plain").unwrap())?;

        sleep(Duration::from_millis(100));

        let mut buf = pipeline.msg_buf(0);
        buf.write_all(self.conf.text.as_bytes())?;

        Ok(ElementValue::MsgBuf)
    }
}
```

まず、本体である HelloSrcElement を定義します。

```
pub struct HelloSrcElement {
    conf: HelloSrcElementConf,
}
```

エレメントを実装する場合、実行に必要なデータを構造体のメンバとして持ちます。ここでは、送りたいテキストを含む HelloSrcElementConf をメンバとして持ちます。

HelloSrcElementConf は、エレメントを構築・実行するための設定を持ちます。

```
#[derive(Debug, Deserialize)]
#[serde(deny_unknown_fields)]
pub struct HelloSrcElementConf {
    text: String,
}
```

構造体の宣言に付随する各属性は、パイプライン設定ファイルからのデシリアライズを可能にするためのものです。基本的に、この例の通りの属性を付与すれば問題ありません。

エレメントを構築可能にするために、ElementBuildable を実装します。

```
impl ElementBuildable for HelloSrcElement {
    type Config = HelloSrcElementConf;

    const NAME: &'static str = "hello-src";

    const SEND_PORTS: Port = 1;

    fn new(conf: Self::Config) -> Result<Self, Error> {
        Ok(HelloSrcElement { conf })
    }

    ..
}
```

Config にはさきほど定義した HelloSrcElementConf を、NAME にはエレメントの名前である "hello-src" を指定します。この名前はパイプライン設定ファイルにおいてエレメントを指定する時に使われるもので、他のエレメントと重複しないものを選ぶ必要があります。

このエレメントは 1 つの送信ポートを持つので、SEND_PORTS に 1 を指定します。

new() には、受け取った設定 (HelloSrcElementConf) を元に、HelloSrcElement を生成するためのコードを記述します。

new() は HelloSrcElementConf 型の値 conf（パイプライン設定ファイルからパースされた設定）を受け取り、その設定に基づいて HelloSrcElement を作成して返却します。

filter エレメントまたは sink エレメントを実装する場合、どの型のメッセージを受け取れるかを示すために acceptable_msg_types() を実装しなければなりません。実装しない場合、どのようなデータも受け取れないエレメントとなります。

最後に、HelloSrcElement がメッセージを生成するためのメソッド next を定義します。

```
impl ElementBuildable for HelloSrcElement {
    ..

    fn next(&mut self, pipeline: &mut Pipeline, _receiver: &mut MsgReceiver) -> ElementResult {
        pipeline.check_send_msg_type(0, || MsgType::from_mime("text/plain").unwrap())?;

        sleep(Duration::from_millis(100));

        let mut buf = pipeline.msg_buf(0);
        buf.write_all(self.conf.text.as_bytes())?;

        Ok(ElementValue::MsgBuf)
    }
}
```

src エレメントとして実装するために、next() メソッドの内部を記述していきます。

```
pipeline.check_send_msg_type(0, || MsgType::from_mime("text/plain").unwrap())?;
```

送出メッセージの型情報を pipeline.check_send_msg_type に渡し、受信側で受け取りができるのかを判定します。このエレメントはテキストを送出するので、"text/plain" という MIME 情報を渡します。

```
sleep(Duration::from_millis(100));
```

データ量を適当な量に制限するため、ここでは 100ms の間 sleep します。

```
let mut buf = pipeline.msg_buf(0);
buf.write_all(self.conf.text.as_bytes())?;

Ok(ElementValue::MsgBuf)
```

設定で与えられたテキストをメッセージにするために、まずは Pipeline::msg_buf() を呼び出して、MsgBuf 型のバッファを用意します。

Pipeline::msg_buf() では、メッセージの送信に用いるポート番号を引数にします（大抵の場合は 0 を指定します）。

MsgBuf は std::io::Write を実装しているため、write_all() を用いてバッファにテキストを書き込みます。

関数の最後の Ok(ElementValue::MsgBuf) は、MsgBuf に書き込んだデータが送信先のタスクに送信されるよう指定するためのものです。

返却されたメッセージは、エレメントが独立したスレッドで動作している場合、関連付けられたタスクへチャンネルを用いて送信されます。エレメントが同期的に呼び出されている場合、呼び出し元のタスクの実行へ戻ります。

7.4 エレメントをプラグインにする／実行ファイルに含める

Rust で実装したエレメントを動作可能にする方法は、プラグインにする場合と、デバイスコネクターの実行ファイルに含める場合とで異なります。

7.4.1 プラグインにする場合

実装したエレメントをプラグインにする場合、define_dc_load!() マクロを使用します。例として、テンプレートの src/lib.rs を見てみます。

```
mod hello;

// Implement plugin interface.
device_connector::define_dc_load!(hello::HelloSrcElement);
```

ここでは、さきほど実装した HelloSrcElement をプラグインに登録しています。define_dc_load!() マクロには、複数のエレメントを渡すことも可能です。プラグインを開発したい場合、このような記述を lib.rs に記述します。

7.4.2 デバイスコネクターの実行ファイルに含める場合

Rust で実装したエレメントをデバイスコネクターの実行ファイルに含めるには、ElementBank に登録します。

ElementBuildable を実装した型を append_from_buildable() により登録し、その後に Runner を構築することで、エレメントが利用できるようになります。

例として、テンプレートの src/main.rs は以下のようになっています。

```
fn main() -> Result<()> {
    // ログシステムの初期化
    env_logger::init();

    // パイプライン設定ファイルの読み込み
    let opts: Opts = Opts::parse();
    let conf = Conf::read_from_file(&opts.config)?;

    // ElementBank の作成
    let mut bank = ElementBank::new();

    // プラグインのロード
    let loaded_plugin = LoadedPlugin::from_conf(&conf.plugin)?;

    // 実装した HelloSrcElement の登録
    bank.append_from_buildable::<hello::HelloSrcElement>();

    // runner の作成
    let mut runner_builder = RunnerBuilder::new(&bank, &loaded_plugin, &conf.runner);
    runner_builder.append_from_conf(&conf.tasks)?;
    let runner = runner_builder.build()?;

    // 起動
    runner.run()?;

    Ok(())
}
```

これは通常の Rust の main 関数であるため、ここに追記することでデバイスコネクターが動作する前後の挙動をカスタマイズすることも可能です。

7.5 ビルド

テンプレートを使用した場合、実装した拡張は以下のコマンドでビルドできます。

```
cargo build --release
```

これにより、target/release に、以下の 2 種類の成果物が生成されます。いずれかを使用してください。

- libdc_xxxxxx.so：HelloSrcElement を含むプラグインファイル（共有ライブラリ）。デバイスコネクター実行時にこのプラグインをロードするように指定すれば、HelloSrcElement を使用できます。プラグイン

のロード方法については plugin の設定 (p. 12) を参照してください。

- xxxxxx-run：HelloSrcElement を含むデバイスコネクターの実行ファイル。このデバイスコネクターを実行すれば、プラグインなしで HelloSrcElement を使用できます。

7.6 （補足）sink エレメントの例

上の説明では例として src エレメントを作成しましたが、sink エレメントを作成する場合、例えば以下のようになります。

このエレメント（hexdump-sink）では、受け取った各メッセージを標準エラーに出力します。

> **注釈:** このエレメント（hexdump-sink）は device-connector-template に含まれます。

```rust
use device_connector::EmptyElementConf;
use device_connector::{
    ElementBuildable, ElementResult, Error, MsgReceiver, MsgType, Pipeline, Port,
};

pub struct HexdumpSinkElement {}

impl ElementBuildable for HexdumpSinkElement {
    type Config = EmptyElementConf;

    const NAME: &'static str = "hexdump-sink";
    const RECV_PORTS: Port = 1;
    const SEND_PORTS: Port = 0;

    fn acceptable_msg_types() -> Vec<Vec<MsgType>> {
        vec![vec![MsgType::any()]]
    }

    fn new(_conf: Self::Config) -> Result<Self, Error> {
        Ok(Self {})
    }

    fn next(&mut self, _pipeline: &mut Pipeline, receiver: &mut MsgReceiver) -> ElementResult {
        loop {
            let msg = receiver.recv(0)?;
            let bytes = msg.as_bytes();
            eprintln!(
                "msg ={}",
                bytes
                    .iter()
                    .map(|x| format!(" {:02X}", x))
                    .collect::<String>()
            );
        }
    }
```

(次のページに続く)

(前のページからの続き)

```
}
```

08 C による独自エレメントの開発

Device Connector Framework では、C インターフェイスを使って独自エレメントのプラグインを開発することができます。

8.1 開発用のファイルの準備

C による開発を行うためには、以下の 2 つのファイルが必要です。

- Device Connector Framework のリポジトリに含まれるインクルードファイル `common/include/device_connector.h`
- 静的リンク用のライブラリファイル `libdevice_connector_common.a` 。以下のコマンドによりビルドしてください。

```
git clone https://github.com/aptpod/device-connector-framework.git && cd device-connector-framework
cargo build -p device-connector-common --release
```

このコマンドにより、`target/release` 以下に `libdevice_connector_common.a` が生成されます。これを任意のディレクトリにコピーして使用してください。

8.2 エレメントの実装

"example-plugin" という src エレメントを実装する場合、以下のようになります。これを `example_plugin.c` という名前のファイルとして保存します。

```
#include <stdbool.h>
#include <stdlib.h>
#include <stdint.h>
#include <string.h>
#include <unistd.h>
#include <device_connector.h>

#define N_ELEMENT 1
#define PLUGIN_NAME "example-plugin"

typedef struct {
    const char* text;
} ExamplePlugin;

void *example_plugin_new(const char *config);
DcElementResult example_plugin_next(void* element, DcPipeline *pipeline, DcMsgReceiver *msg_receiver);
bool example_plugin_finalizer(void *element, struct DcFinalizer *finalizer);
void example_plugin_free(void* element);

bool dc_load(DcPlugin *plugin) {
    dc_init(PLUGIN_NAME);
```

(次のページに続く)

(前のページからの続き)

```
    plugin->version = "0.1.0";
    plugin->n_element = N_ELEMENT;

    DcElement *elements = (DcElement *)malloc(sizeof(DcElement) * N_ELEMENT);

    // Element settings
    elements[0].name = PLUGIN_NAME;
    elements[0].recv_ports = 0;
    elements[0].send_ports = 1;
    elements[0].acceptable_msg_types = NULL;
    elements[0].config_format = "json";
    elements[0].new_ = example_plugin_new;
    elements[0].next = example_plugin_next;
    elements[0].finalizer = example_plugin_finalizer;
    elements[0].free = example_plugin_free;

    plugin->elements = elements;
    return true;
}

void *example_plugin_new(const char *config) {
    ExamplePlugin *example_plugin = (ExamplePlugin *)malloc(sizeof(ExamplePlugin));
    example_plugin->text = "hello, world from plugin";
    return example_plugin;
}

DcElementResult example_plugin_next(void* element, DcPipeline *pipeline, DcMsgReceiver *msg_receiver) {
    ExamplePlugin *example_plugin = (ExamplePlugin *)element;

    if (!dc_pipeline_send_msg_type_checked(pipeline)) {
     DcMsgType msg_type;
     if (dc_msg_type_new("mime:text/plain", &msg_type)) {
         dc_pipeline_check_send_msg_type(pipeline, 0, msg_type);
     }
    }

    sleep(1);

    DcMsgBuf *msg_buf = dc_pipeline_msg_buf(pipeline);

    const uint8_t *data = (const uint8_t *)example_plugin->text;
    const size_t len = strlen(example_plugin->text);
    dc_msg_buf_write(msg_buf, data, len);

    return DcElementResult_MsgBuf;
}
```

(次のページに続く)

(前のページからの続き)

```
bool example_plugin_finalizer(void *element, struct DcFinalizer *finalizer) {
    return true;
}

void example_plugin_free(void* element) {
    ExamplePlugin *example_plugin = (ExamplePlugin *)element;
    free(example_plugin);
}
```

以下、このソースコードについて解説していきます。

```
#include <device_connector.h>
```

プラグイン開発のために device_connector.h をインクルードします。

```
typedef struct {
    const char* text;
} ExamplePlugin;
```

エレメントの本体を定義します。

```
bool dc_load(DcPlugin *plugin) {
    dc_init(PLUGIN_NAME);
```

プラグインをロードする時に呼び出される dc_load 関数を定義します。最初に、dc_init にプラグインの名称を渡します。

```
plugin->version = "0.1.0";
plugin->n_element = N_ELEMENT;
```

このプラグインがターゲットとするデバイスコネクターのバージョンと、読み込ませたいエレメントの数を指定します。

```
DcElement *elements = (DcElement *)malloc(sizeof(DcElement) * N_ELEMENT);
```

DcElement の配列を malloc で用意します。この長さは読み込ませたいエレメントの数 (N_ELEMENT) と同じです。

```
elements[0].name = PLUGIN_NAME;          // エレメントの名前（実行時に他のエレメントと重複しているとエラーになる）
elements[0].recv_ports = 0;              // 受信用ポートの数
elements[0].send_ports = 1;              // 送信用ポートの数
elements[0].acceptable_msg_types = NULL; // 受け取ることのできるデータ型。何も受信しない場合は NULL
elements[0].config_format = "json";      // 設定フォーマット (json or yaml)。このフォーマットが new_に指定した関数に渡される
elements[0].new_ = example_plugin_new;   // エレメントを生成する関数
elements[0].next = example_plugin_next;  // エレメントを実行する関数
elements[0].finalizer = example_plugin_finalizer;  // プロセス終了時呼び出されるファイナライザを設定する関数
elements[0].free = example_plugin_free;  // エレメントを終了・解放する関数
```

エレメントの詳細を定義します。

```
    plugin->elements = elements;
    return true;
}
```

plugin->elements に DcElement の配列を設定し、dc_load が成功したら true を返します。

```
void *example_plugin_new(const char *config) {
    ExamplePlugin *example_plugin = (ExamplePlugin *)malloc(sizeof(ExamplePlugin));
    example_plugin->text = "hello, world from plugin";
    return example_plugin;
}
```

エレメントを生成する関数です。config には、パイプライン設定ファイルに記述されたのエレメントの設定（conf フィールドの値）が、config_format で指定したフォーマットに変換され文字列として渡されます。（この例では config の値は使用していません。）

ExamplePlugin のための領域を malloc で確保し、初期化後に void ポインタとして返します。失敗時には NULL を返却します。

```
DcElementResult example_plugin_next(void* element, DcPipeline *pipeline, DcMsgReceiver *msg_receiver) {
    ExamplePlugin *example_plugin = (ExamplePlugin *)element;
```

example_plugin_next はエレメントを実行するための関数です。受け取った element を ExamplePlugin * にキャストします。

```
if (!dc_pipeline_send_msg_type_checked(pipeline)) {
    DcMsgType msg_type;
    if (dc_msg_type_new("mime:text/plain", &msg_type)) {
        dc_pipeline_check_send_msg_type(pipeline, 0, msg_type);
    }
}
```

dc_pipeline_send_msg_type_checked() で送信するメッセージの型チェックが行われているか調べ、行われていなければ、DcMsgType を作成して dc_pipeline_check_send_msg_type() に渡します。

```
sleep(1);

DcMsgBuf *msg_buf = dc_pipeline_msg_buf(pipeline);

const uint8_t *data = (const uint8_t *)example_plugin->text;
const size_t len = strlen(example_plugin->text);
dc_msg_buf_write(msg_buf, data, len);
```

1 秒間スリープした後、msg_buf を取得し、送信したいデータを dc_msg_buf_write() で書き込みます。ここで書き込むのは example_plugin_new で設定したテキストです。

```
    return DcElementResult_MsgBuf;
}
```

DcElementResult_MsgBuf を返し、msg_buf に書き込んだデータを送信することを示します。

```
bool example_plugin_finalizer(void *element, struct DcFinalizer *finalizer) {
    return true;
}
```

プロセス終了時に呼び出されるファイナライザを登録するための関数です。ここでは特に何も行いませんが、プロセス終了時にエレメントが占有するリソースを解放する必要がある場合、ファイナライザに記述します。

```
void example_plugin_free(void* element) {
    ExamplePlugin *example_plugin = (ExamplePlugin *)element;
    free(example_plugin);
}
```

終了処理を記述します。ここでは malloc() で確保した領域を free() に渡すだけです。

8.3 コンパイル

上記の example_plugin.c を、GCC でコンパイルするには以下のコマンドを実行します。

```
gcc -I/include_dir -Wall -O2 -fPIC -shared -L/library_dir \
    -o libdc_example_plugin.so example_plugin.c -ldevice_connector_common
```

-I オプションで、device_connector.h ファイルが含まれるディレクトリを、-L オプションで、libdevice_connector_common.a ファイルが含まれるディレクトリを指定してください。

これにより、プラグインファイル libdc_example_plugin.so が生成されます。プラグインのロード方法については plugin の設定 (p. 12) を参照してください。

09 Device Connector Framework の既知の制約

Device Connector Framework には、以下のような既知の問題・制約があります。

エレメント利用者向け

- タスクの設定において、メッセージの送信元のタスク `from` を設定するときは、送信元のタスク ID は自身のタスク ID より小さい数字である必要があります。
- 番号が 1 以上の送信ポートについては未実装です。

独自エレメントの開発者向け

- 本体とプラグインで、利用するメモリアロケータが異なる場合の動作は未定義です。この問題は、Rust の将来のバージョンでメモリアロケータを固定化できれば回避できます。メモリアロケータの設定を故意に変えなければ問題は起こりません。
- Rust の `Vec` と `String` を、プラグインの FFI（Foreign Function Interface）間で受け渡しているため、もしこの ABI（Application Binary Interface）に変更があれば、コンパイラのバージョンを本体とプラグインで合わせておかないと、プラグインが壊れる可能性があります。