SATySFi エコシステム再考(?)

2023 年 10 月 22 日
SATySFi Conf 2023

Takashi Suwa
gfngfn

概要

- 「SATγSF| v0.1.0 なかなかリリースできねえ！」
  - 非互換な変更を良い機会とし,
    処理系本体やパッケージの互換性を
    筋良く扱えるエコシステムが構築できてからリリースしたい
    - 長期的に保守する文書にとって互換性は重要
  - 既存ツールである **Satyrographos** [Sakamoto 2018] は
    現状では OPAM との共同利用の形態をとっているが,
    OPAM に起因する厄介ごとがいろいろある

- パッケージマネジメントやビルド制御の
  プロトタイプがどうあるべきかの設計・実装を個人的に検討中
  - 最終的に Satyrographos にフィードバックするか,
    別個につくるのかなどは未定（相談の上決定したい）

発表のねらい

- ユーザやコントリビュータの方々に
  OPAM に依存しないエコシステムの設計案や
  そのプロトタイプ実装を叩き台として共有する

- また，問題点を指摘してもらいながら
  おおよそユーザ全体として「これでよかろう」という
  ゆるやかな合意に最終的に至れるとうれしい

目次

現状のエコシステムの問題点

- **互換性に関する記述が仕組み化されていない**
  - パッケージのリリースで後方互換性の有無の記述が強制されず,
    そのパッケージを使う側でのアップグレードが大変
    - semver がほしい！
  - 長期的に保守する文書に関して互換性は重要課題
    - 互換性の担保が難しいために TEX Live をアップグレードできない
      出版社の現場が実際にある

- **同一パッケージの非互換なバージョンが共存できない**
  - 共存できないことが不都合になった実例が既にある
    - `enumitem` パッケージ [monaqa 2019] の v2 と v3

Satyrographos を介して OPAM に依存しており
OPAM の歴史的経緯による不便な点を引き受けているのが要因

改善の方針

OPAM の定式化に依存しない SATγSF| のエコシステムをつくる

- OPAM の都合を加味せず，無条件に SATγSF| にとって
  望ましいエコシステムの在り方を考える

- Satyrographos も OPAM 以外をバックエンドとするように
  実装を替えること自体は十分可能とのこと

- いずれにせよどんな仕組みが良いのかの評価・検討のために
  叩き台となる設計案とプロトタイプ実装が必要

処理系本体に関する要件

“1 つのバージョンに由来するモジュールの集まり” が
処理系からどう見える定式化にするか？

- 同一パッケージの複数バージョン共存のためには,
  そもそも処理系本体が “リリース単位としてのパッケージ” を意識できないようになっていると見通しがよい
  - 処理系本体は，同じパッケージ由来であることを全く知らずに
    2 つの独立な “モジュールの集まり” を処理すればよい
  - ただし，その 2 つを異なる名前で参照できる仕組みが必要

- 一方で，異なるパッケージ由来のモジュール名が
  衝突しては困るので，**パッケージの規模に対応する名前空間の切り分け機能はあってほしい**

エコシステムの構成案

- 以下の 3 要素に分ける
  1. **SATγSFI 処理系本体**
     - リリースの単位としてのパッケージは認識しない
     - **“モジュールの集まりとしてのパッケージ” は認識する**
  2. **パッケージマネージャ・ビルドシステム**
     - 仮称： **Saphe** /seɪf/ (**SA**TγSFI **P**ackage-**H**andling **E**cosystem)
     - パッケージの依存解決・配置，および処理系本体の起動を担う
  3. **アップグレーダ**
     - 1 と 2 のバージョンの更新・切替えを行なうツール
     - 今回は扱いません（実はまだ全く定式化してない😁）

- Rustc・Cargo・Rustup の責務分担とよく似ている

目次

処理系本体からの “モジュールの集まり” の見え方

- “リリースの単位としてのパッケージ” は認識しないが,
  “モジュールの集まったひとかたまり” を単位として認識
  - 仮称： **エンベロープ**（***envelope***, 封筒の意）
  - 典型的には，バージョンを固定して配置されたパッケージ
  - ディレクトリ構造：
    - ルートにコンフィグ **satysfi-envelope.yaml** があり,
      どのサブディレクトリにソースファイルがあるかなどを記載

```
foo-envelope/
├── satysfi-envelope.yaml
└── src/
    ├── foo.satyh
    └── foo-sub.satyh
```

```
source_directories:
  - "src/"
main_module: "Foo"


...
```

エンベロープを構成するソースファイル

- 各エンベロープはただ 1 つの**メインモジュール**を
  エンベロープ外に公開する
  - 他のモジュールは入れ子の
    メンバーとしてのみ公開可能で,
    名前空間を汚染しない

```
foo-envelope/
├── satysfi-envelope.yaml
└── src/
    ├── foo.satyh
    └── foo-sub.satyh
```

```
source_directories:
  - "src"
main_module: "Foo"
```

foo.satyh（メインモジュール）

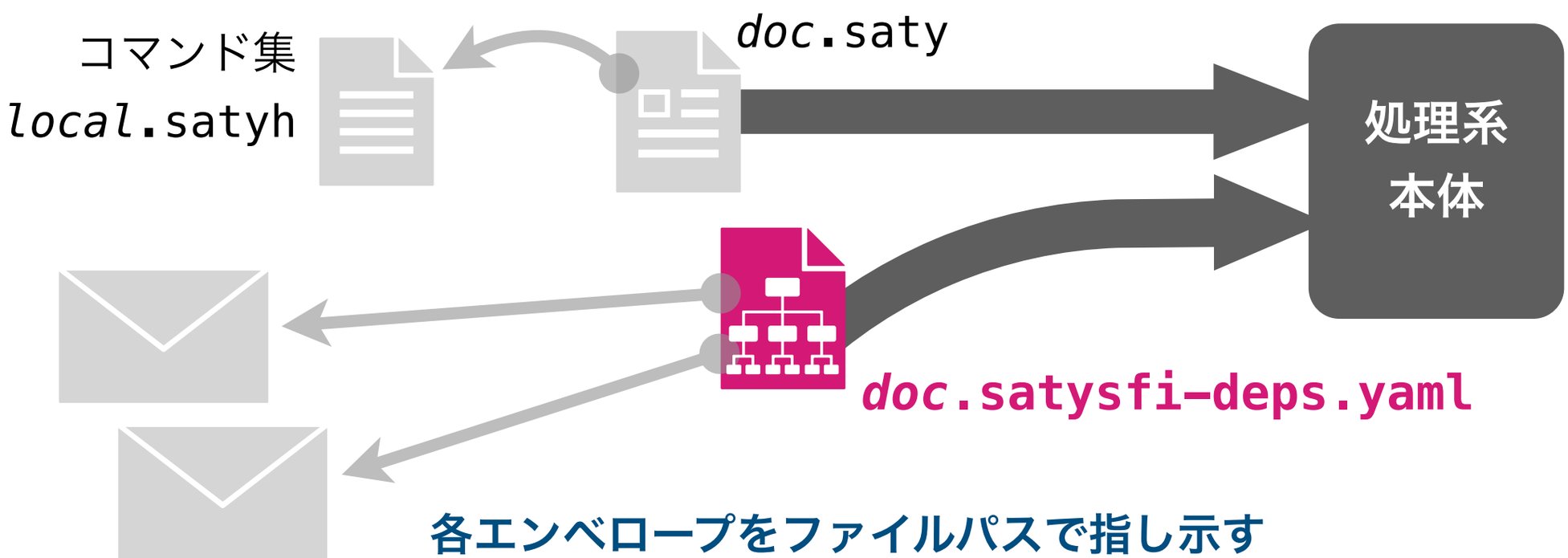
```
import FooSub

module Foo :> sig
  val succ : int -> int
  module Sub : sig
    val pred : int -> int
  end
end = struct
  val succ n = n + 1
  module Sub = FooSub
end
```

foo-sub.satyh

```
module FooSub = struct
  val pred n = n - 1
end
```

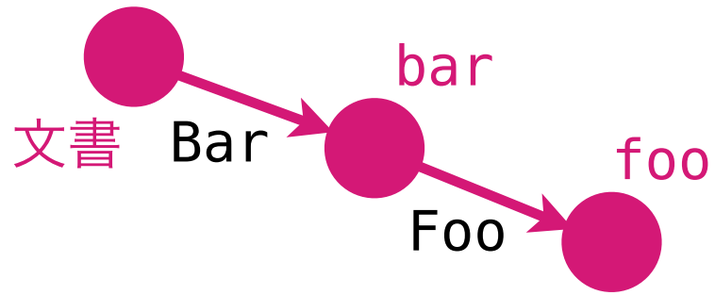
処理系本体へのエンベロープの与え方

- 組版に際して，処理系は *doc*.saty に加えて
  ファイル **doc.satysfi-deps.yaml** を受け取る
  - **使用するエンベロープ一覧とその間の依存関係を記載**
    - 人間が手で書くのではなく，**エコシステムが生成することを想定**
  - このファイルと各依存エンベロープのコンフィグを見るだけで
    各ソースファイルをどう読み込めばよいかの情報が全て得られる



`doc.satysfi-deps.yaml` の具体的内容

- 各エンベロープに対し，ID とファイルパスを記載
- エンベロープ間の依存関係は `dependencies` に記述され，DAG をなす
  - 処理系はこれをもとに**トポロジカルソートして順次エンベロープを読込む**
- 依存するエンベロープのメインモジュールを参照する名前を `used_as` で指定
  - **メインモジュールを参照するための名前は使う側が決められる**
  - **同一パッケージの複数バージョンの共存に必要（詳細後述）**




```
envelopes:
  - id: "foo"
    path: "path/to/foo-envelope/"
    dependencies: []
  - id: "bar"
    path: "path/to/bar-envelope/"
    dependencies:
      - id: "foo"
        used_as: "Foo"
dependencies:
  - id: "bar"
    used_as: "Bar"
```

bar は foo に依存し，
Foo という名前で
foo のメインモジュールを使用
- foo が提供する名前と違ってもよい

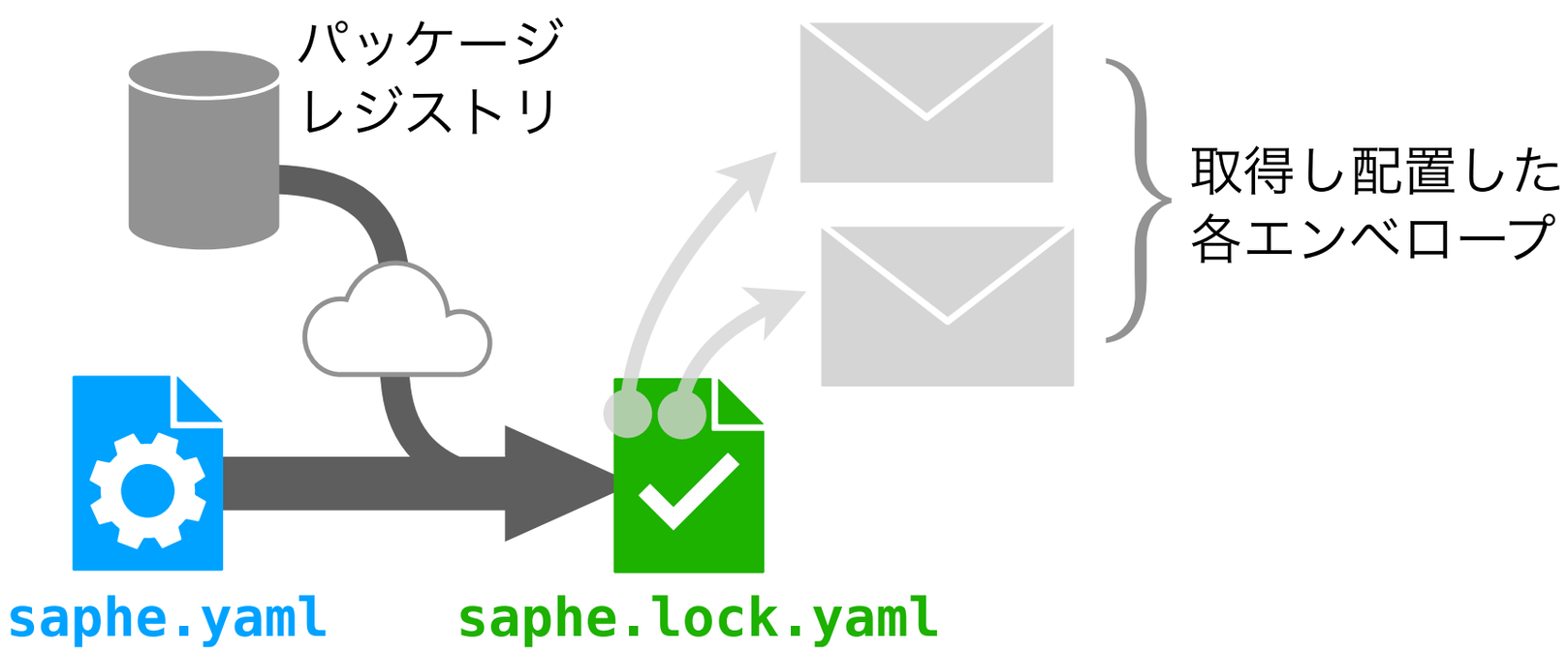
文書本体は bar のみに依存

目次

エコシステム Saphe（仮）の責務

ユーザが書いたコンフィグ **`saphe.yaml`** をもとに以下を実施：

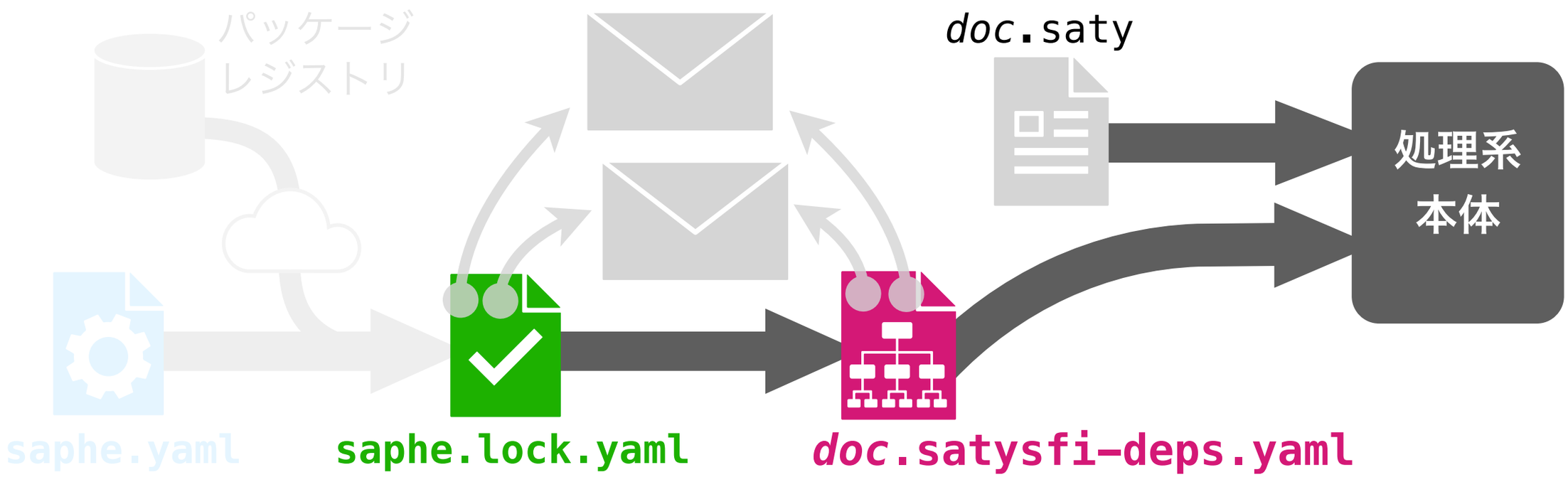
- **`$ saphe solve`**
  - コンフィグに指定されたパッケージの依存制約をもとに，
    パッケージレジストリを参照して制約解消を試み，解消できたら
    各パッケージの所望バージョンをエンベロープとして取得・配置し，
    結果をロックファイル **`saphe.lock.yaml`** に出力



エコシステム Saphe（仮）の責務

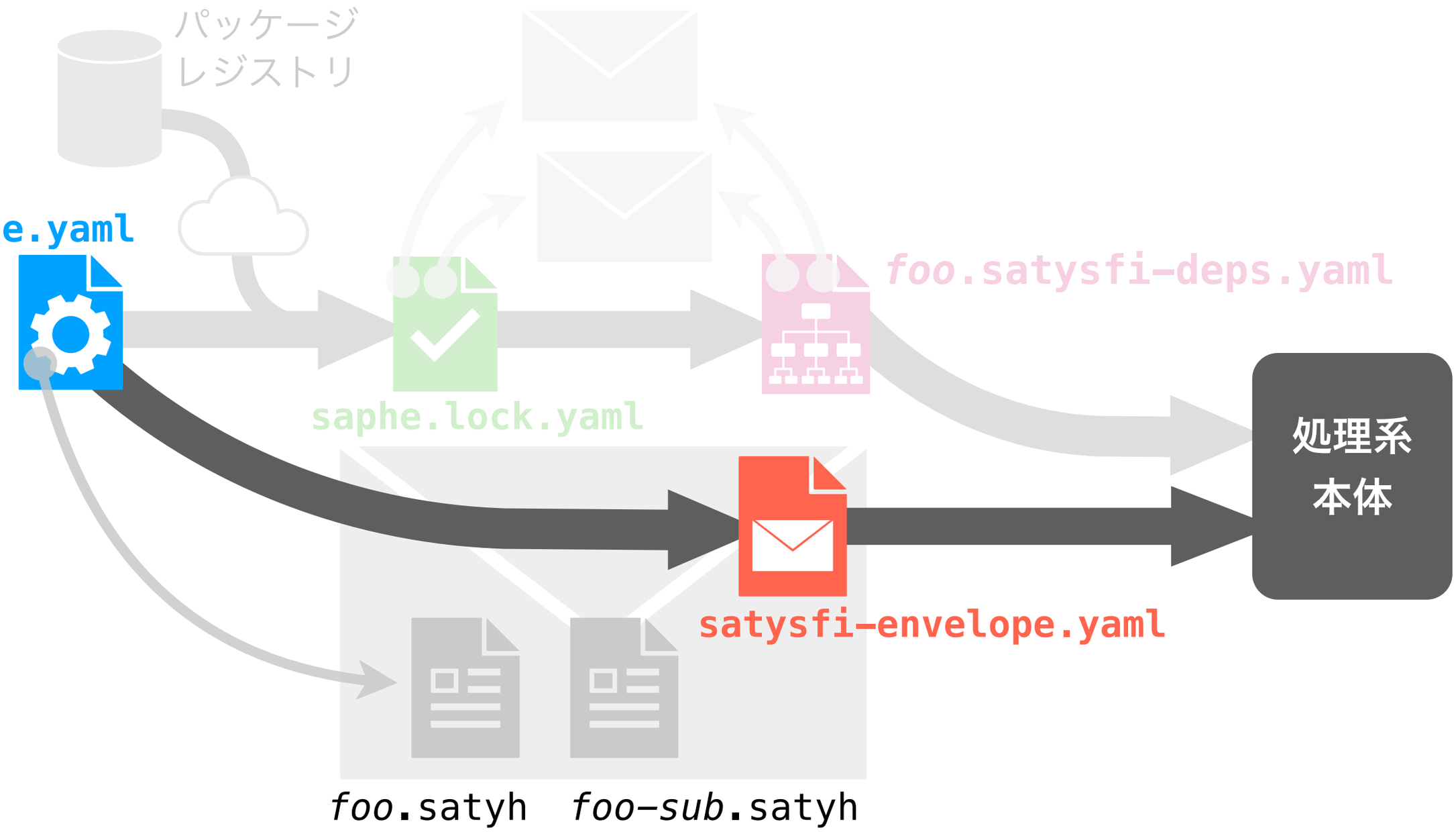
ユーザが書いたコンフィグ **saphe.yaml** をもとに以下を実施：

- `$ saphe build`
  - ロックファイル **saphe.lock.yaml** をもとに
    ***doc*.satysfi-deps.yaml** を生成し，処理系本体に渡して起動



エコシステム Saphe（仮）の責務

ライブラリのビルドでは，さらに `satysfi-envelope.yaml` も `saphe.yaml` から生成される



コンフィグの具体的形式

saphe.yaml

```yaml
language: "^0.1.0"
…

dependencies:
  - used_as: "Table"
    registered:
      name: "easytable"
      constraint: "^2.1.0"
```

コンフィグの具体的形式

saphe.yaml

```
language: "^0.1.0"
…

dependencies:
  - used_as: "Table"
    registered:
      name: "easytable"
      constraint: "^2.1.0"
```

依存解決

saphe.lock.yaml

```
language: "0.1.1"
…

locks:
  - id: "default.easytable.2.1.3"
    registered:
      name: "easytable"
      version: "2.1.3"
    dependencies:
      - id: "default.base.1.0.2"
        used_as: "Base"
  - id: "default.base.1.0.2"
    registered:
      name: "base"
      version: "1.0.2"
    dependencies: []
dependencies:
  - id: "default.easytable.2.1.3"
    used_as: "Table"
```

- 依存解決の結果は “出る辺を識別する名前のついた DAG”

非互換バージョン共存 その1

saphe.yaml

```
language: "^0.1.0"
…

dependencies:
  - used_as: "Table"
    registered:
      name: "easytable"
      constraint: "^2.1.0"
  - used_as: "Base"
    registered:
      name: "base"
      constraint: "^2.0.0"
```


文書で base.2.0.0 を使いたいが,
既に使っている easytable が base.1
しかサポートしていない場合どうなる？

非互換バージョン共存 その1

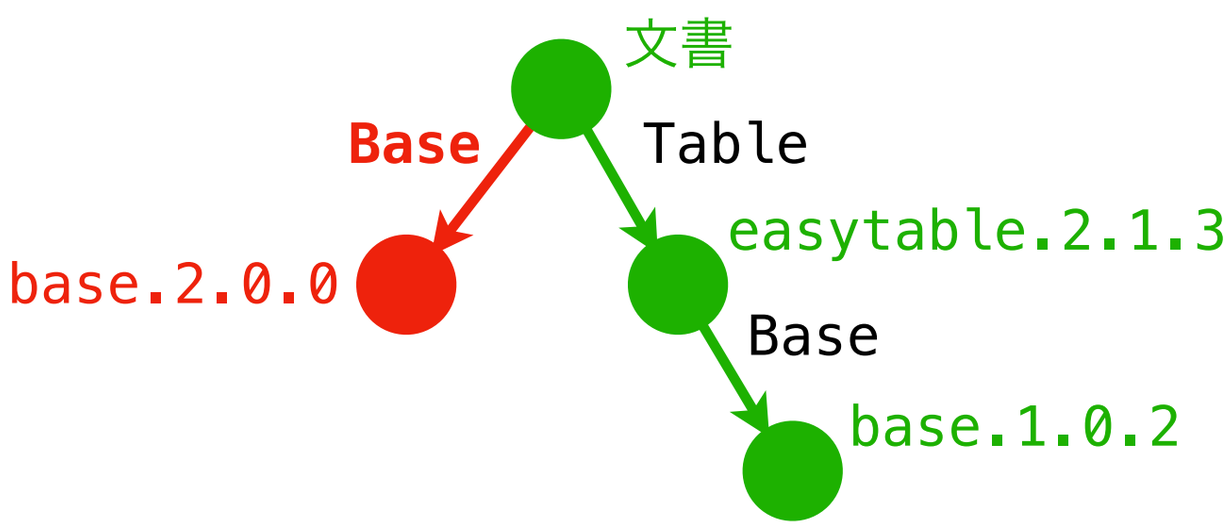
saphe.yaml

```
language: "^0.1.0"
…

dependencies:
  - used_as: "Table"
    registered:
      name: "easytable"
      constraint: "^2.1.0"
  - used_as: "Base"
    registered:
      name: "base"
      constraint: "^2.0.0"
```

文書で base.2.0.0 を使いたいが,
既に使っている easytable が base.1 しかサポートしていない場合どうなる？

問題なく共存できる

- 文書からは Base という名前で base.2.0.0 のみが参照できる

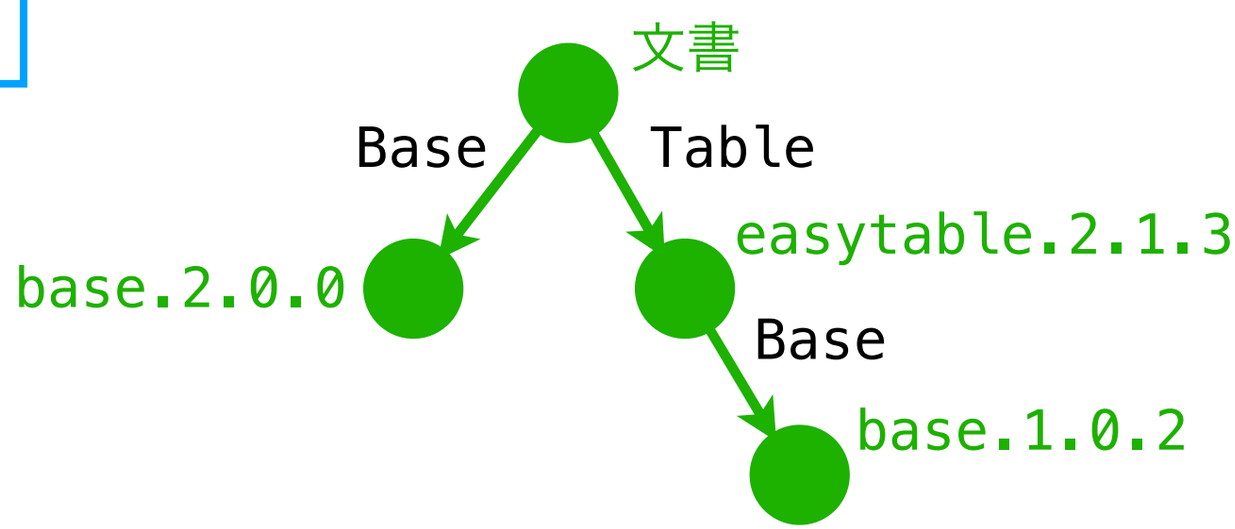


非互換バージョン共存 その2

saphe.yaml

```
language: "^0.1.0"
…

dependencies:
  - used_as: "Table"
    registered:
      name: "easytable"
      constraint: "^2.1.0"
  - used_as: "Base"
    registered:
      name: "base"
      constraint: "^2.0.0"
  - used_as: ???
    registered:
      name: "base"
      constraint: "^1.0.0"
```

文書で base.2.0.0 を使いたいが
easytable が base.1 の型を
シグネチャに公開しているなどして
base.1 も使わねばならない場合
どうなる？



非互換バージョン共存 その2

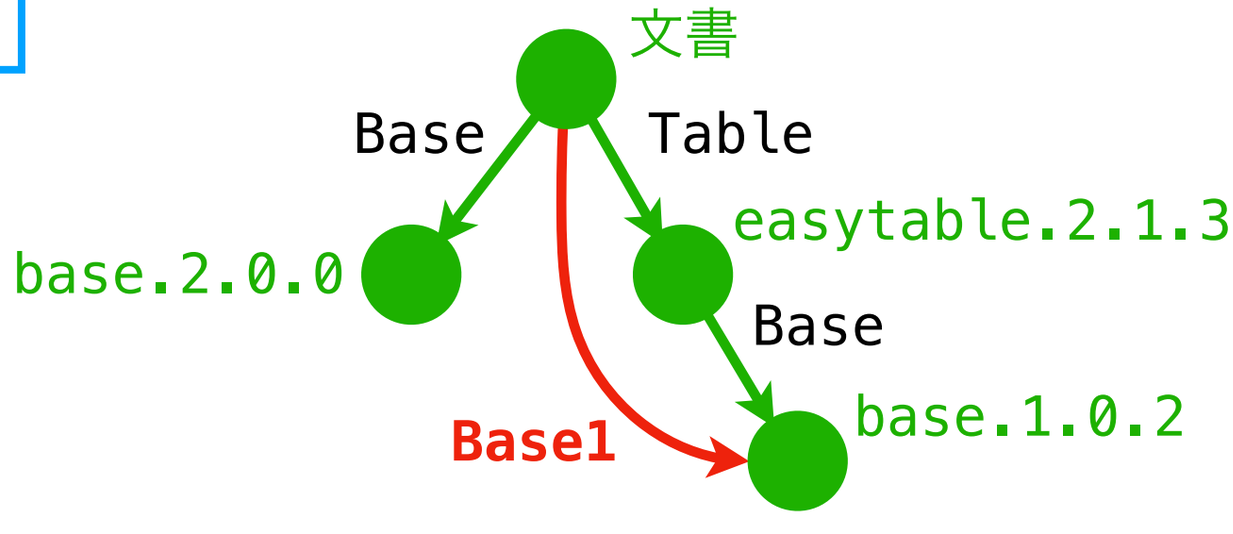
saphe.yaml

```
language: "^0.1.0"
…

dependencies:
  - used_as: "Table"
    registered:
      name: "easytable"
      constraint: "^2.1.0"
  - used_as: "Base"
    registered:
      name: "base"
      constraint: "^2.0.0"
- used_as: "Base1"
  registered:
    name: "base"
    constraint: "^1.0.0"
```

文書で base.2.0.0 を使いたいが
easytable が base.1 の型を
シグネチャに公開しているなどして
base.1 も使わねばならない場合
どうなる？

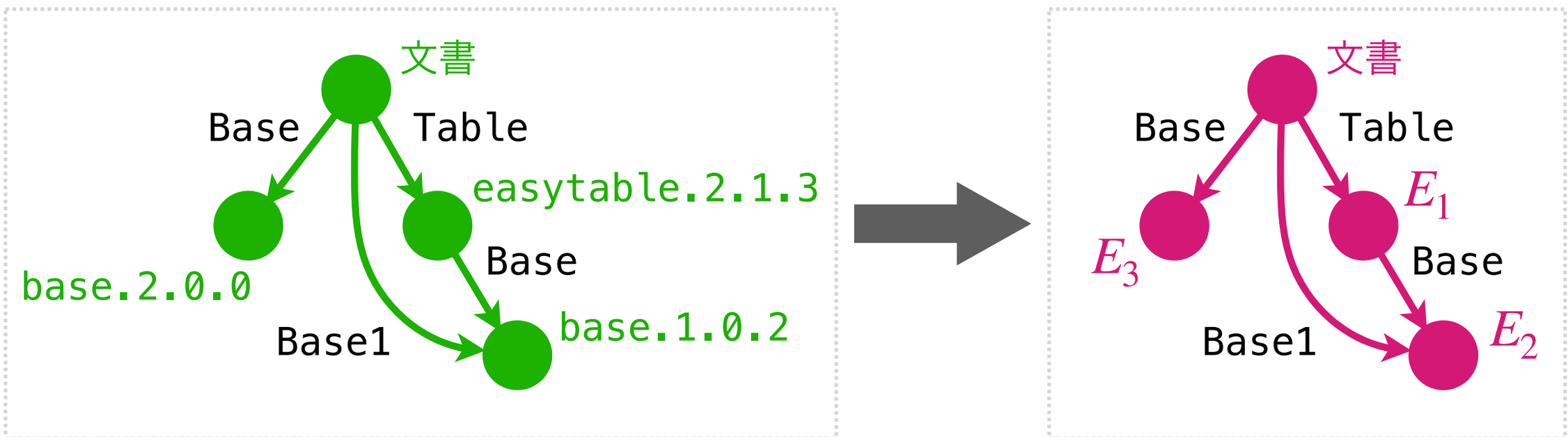
これも問題なく共存できる

- 文書から見て Base と Base1 という
  メインモジュール名で呼び分ける



実際の流れ： *doc*.satysfi-deps.yaml 経由

- より正確には，ビルド時に saphe.lock.yaml をもとにして
  ほぼ同じ DAG を記載した *doc*.satysfi-deps.yaml が生成され，
  処理系本体がそれを読む
  - この時点で，同一パッケージの異なる 2 つのバージョンは
    “互いに無関係” な 2 つのエンベロープになる
  - 処理系本体は 2 つのエンベロープ E_2, E_3 が
    同一パッケージ起源であることをそもそも知る必要がない



目次

想定される疑問 1

Q. どうして **saphe.lock.yaml** と ***doc.satysfi-deps.yaml*** とに分かれているのか？（後者だけではダメか？）

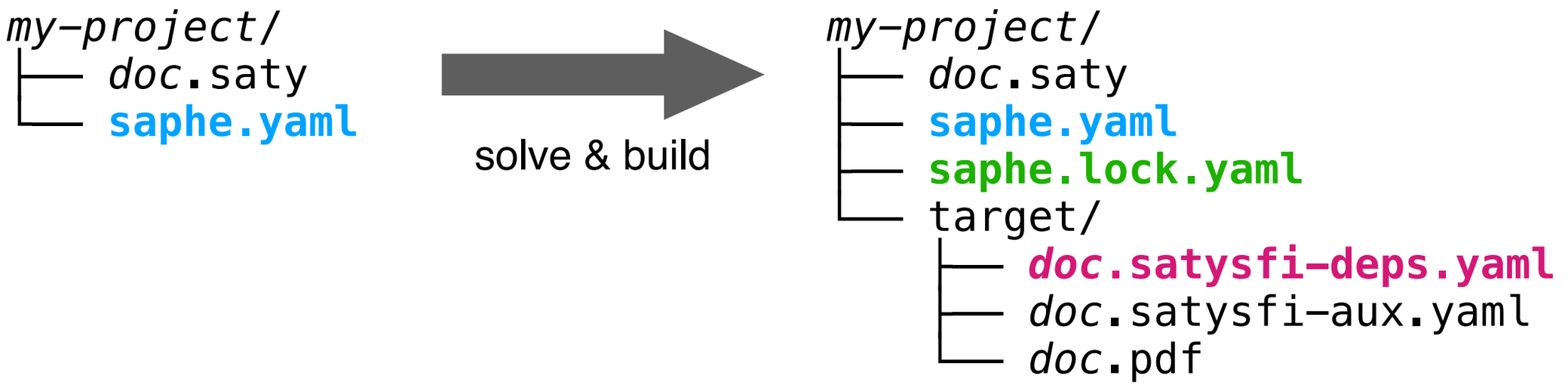
- **saphe.lock.yaml** のフォーマットは「どのパッケージのどのバージョン」という概念を記述する必要があるため
  - 書き出した Saphe 自身が読んで，ビルドの再現やアップグレードに使えるとうれしい
- ***doc.satysfi-deps.yaml*** は処理系本体のための形式であり，パッケージという概念を知らない
  - Saphe がこれを読んでパッケージの情報を復元するのは“逆アセンブル”であり，不自然なのでやりたくない

想定される疑問 2

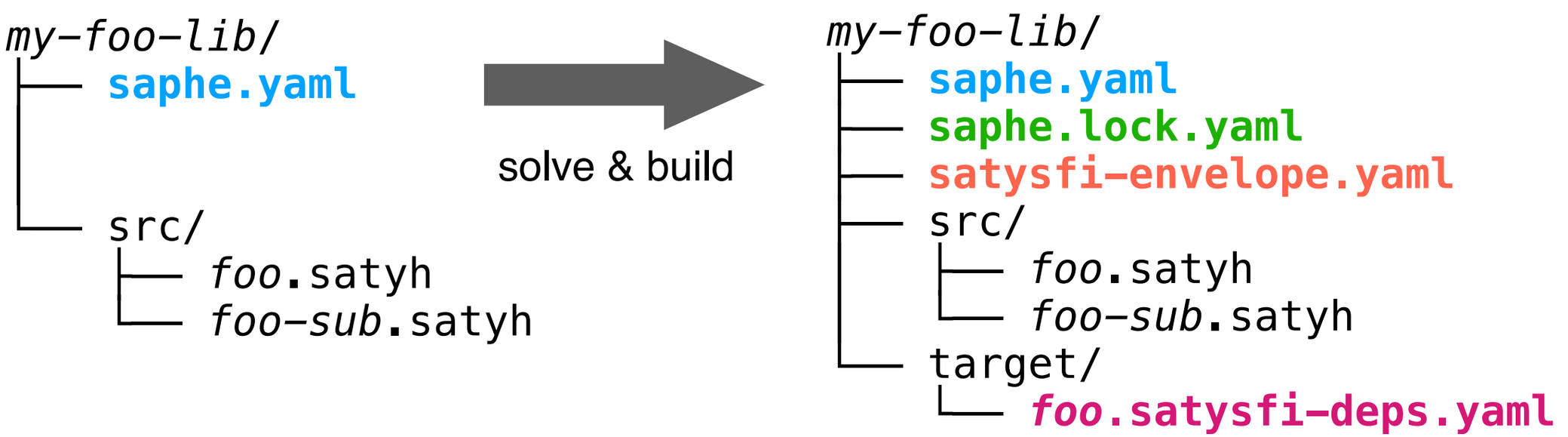
Q. 中間生成物が多くて邪魔にならないか？

- バージョン管理に含めない生成物は `target/` に入ればよさそう

文書



ライブラリ



目次

ユニットテストのインターフェイス

- ライブラリのユニットテストを以下のように書いて test/ に置き，`$ saphe test` で実行できるとうれしい

```
import Testing
import Foo

module FooTest = struct
  #[test]
  val succ-test () =
    Testing.assert-equal 43 (Foo.succ 42)
end
```

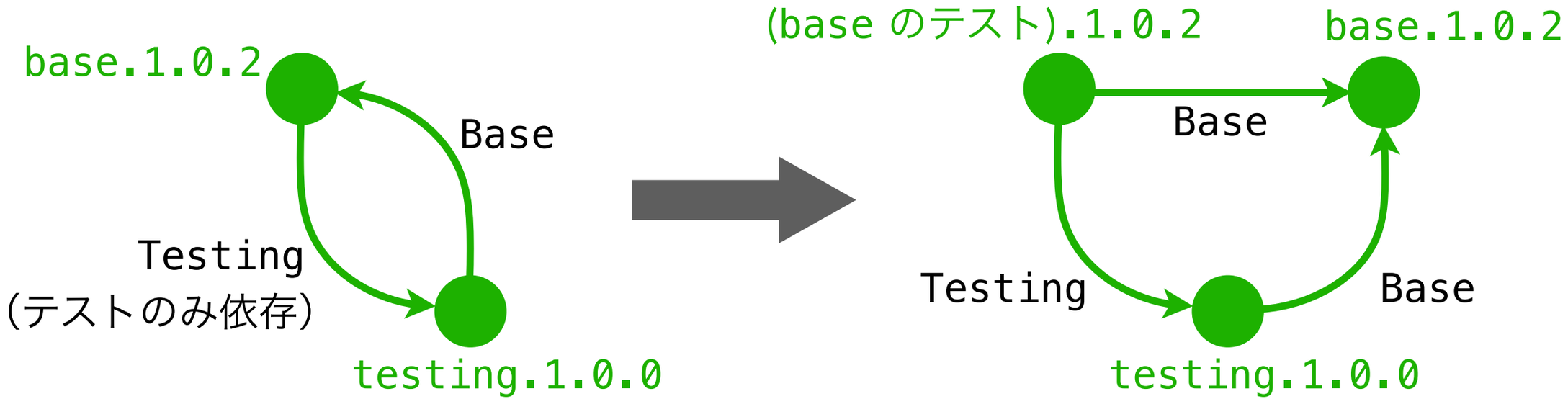
- テスト用ライブラリのために，依存パッケージの指定方法に「テストの場合だけ使う」というフラグを用意

```
dependencies:
  - used_as: "Testing"
    registered:
      name: "testing"
      constraint: "^1.0.0"
    for_test: true
…
```

テストは別パッケージ扱いすべきかも？

（Sakamoto さんとの会話中でのご指摘）[Sakamoto 2023]

- **テストがパッケージの一部だと循環依存をもたらすことがある**
  - 例えば「base のユニットテストが testing を使っているが，testing の実装も base に依存している場合」が相互依存
    - **考え方 1**： 既に信頼できる実装であるべきテスト用ライブラリがまだ信頼できない実装であるテスト対象に依存しているのは望ましくないので，本質的に弾くべき
    - **考え方 2**： **base のテストと実装とを別パッケージに分ければ循環依存ではなくなる**



目次

複数レジストリ

- 実際にはレジストリは
  複数個扱いたいことがある
  - 出版に使う公開できない
    パッケージなども扱いたい

- **saphe.yaml** 中でレジストリを
  指定できるようにする
  - dependencies の記述で
    registry: "default" は
    省略可能とする
  - 標準のレジストリ URL は
    毎回書くと面倒なので,
    `$ saphe new` で生成したい

```
language: "^0.1.0"
…
registries:
  - name: "default"
    git:
      url: "https://…"
      branch: "format-1"
  - name: "your-company"
    git:
      url: "https://…"
      branch: "master"
dependencies:
  - used_as: "Table"
    registered:
      name: "easytable"
      constraint: "^2.1.0"
  - used_as: "YourPrivatePkg"
    registered:
      registry: "your-company"
      name: "your-private-pkg"
      constraint: "^0.1.1"
```

目次

まとめ

- **エンベロープ**（仮）という単位を導入して以下を実現できそう
  - SATYSFI 処理系本体とエコシステム **Saphe**（仮）との間での見通しのよい責務の切り分け
    - 処理系本体は “モジュールの集まり” であるエンベロープを認識するが，リリース単位としてのパッケージは認識しない
  - パッケージ間でのモジュール名の衝突を防ぐ名前空間
  - 非互換な複数バージョンの共存
- ユニットテスト機構など，まだ少し考慮すべきことが残存

- ちなみに，Saphe と SATYSFI 処理系本体とが分離していないが，今回紹介した定式化に近い機能は仮実装済み
  - `dev-0-1-0-package-system` ブランチで

```
$ satysfi solve
$ satysfi build
$ satysfi test
```

が動く