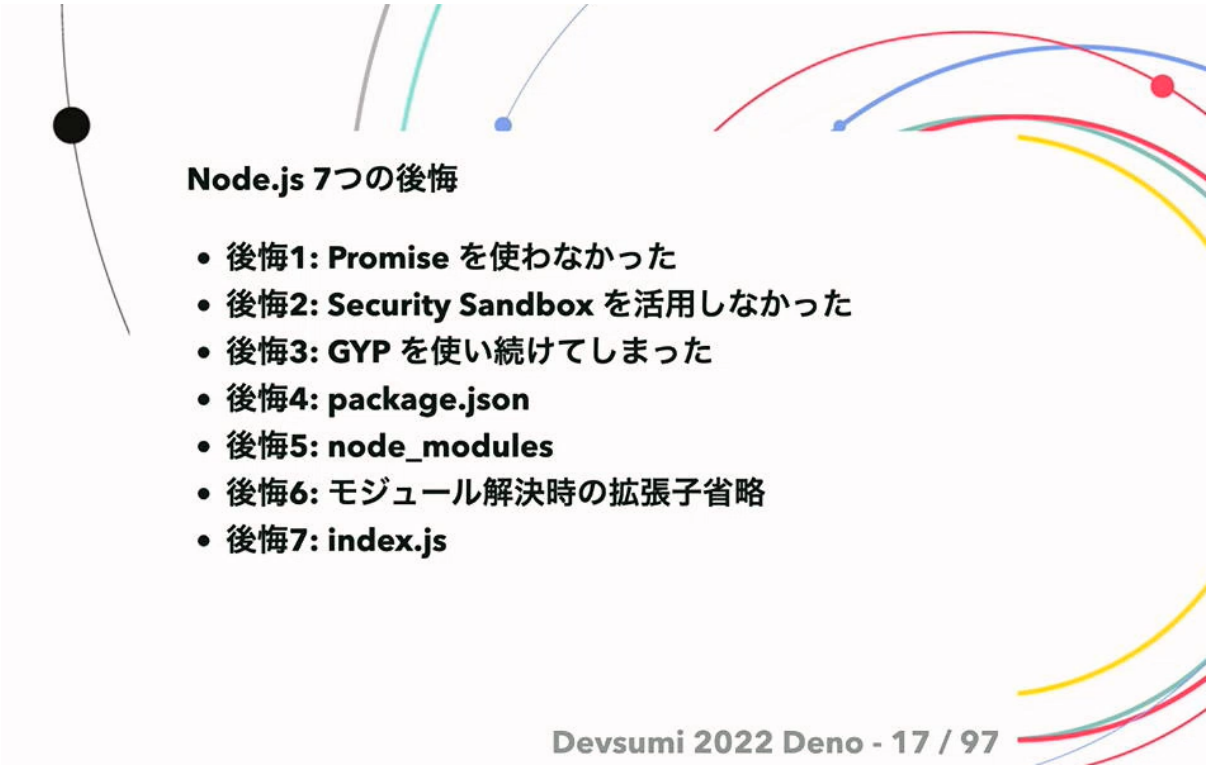
モダンなJavaScript/TypeScript実行環境「Deno」



Deno の特徴

- ブラウザ互換性
- TypeScript サポート
- サンドボックスセキュリティ
- ビルトイン開発ツール

ブラウザ互換性

- Deno には可能な限りブラウザ互換 API を取り入れるというデザイン方針がある
- Node の開発が始まった時に比べてかなり多くのブラウザ API が定義されていて、いろいろな事がブラウザと同じ API で出来るようになっている
- ただし、Node.js も可能な場合は後からブラウザ互換 API を取り入れるという流れがあり、ややこしい状態になっている
  - 例. url と URL、Buffer extends Uint8Array

Deno に実装されている ブラウザ互換API の例

fetch API

```
const resp = await fetch("https://example.com");
const html = await resp.text();
console.log(html);
```

- 簡単に HTTP リクエストが出来る
- http client library などが不要

Deno に実装されている ブラウザ互換API の例

バイナリ処理 - TypedArray API (Uint8Array, etc)

```
const data = Uint8Array.from([0x66, 0x6f, 0x6f]);
const text = new TextDecoder().decode(data);
// => foo
```

- Node の場合は Buffer (独自クラス)
- ただし今は Node は TypedArray も持っている

Deno に実装されている ブラウザ互換API の例

URL パーサー

```
const url = new URL("https://example.com/?foo=bar");
console.log(url.hostname); // => example.com
console.log(url.searchParams.get("foo")); // => bar
```

- Node の場合は require("url") が昔からあるが、後に URL も実装されて両方ある状態

Deno に実装されている ブラウザ互換API の例

Web Storage

```
localStorage.setItem("key", data);
...
// プログラム再起動後
console.log(localStorage.getItem("key"));
// => さっき保存したデータが残っている
```

- エントリポイント毎に独立したストレージを持てる
- 内部では SQLite を使って保存している
- Node には無い機能

Deno に実装されている ブラウザ互換API の例

HTTP imports

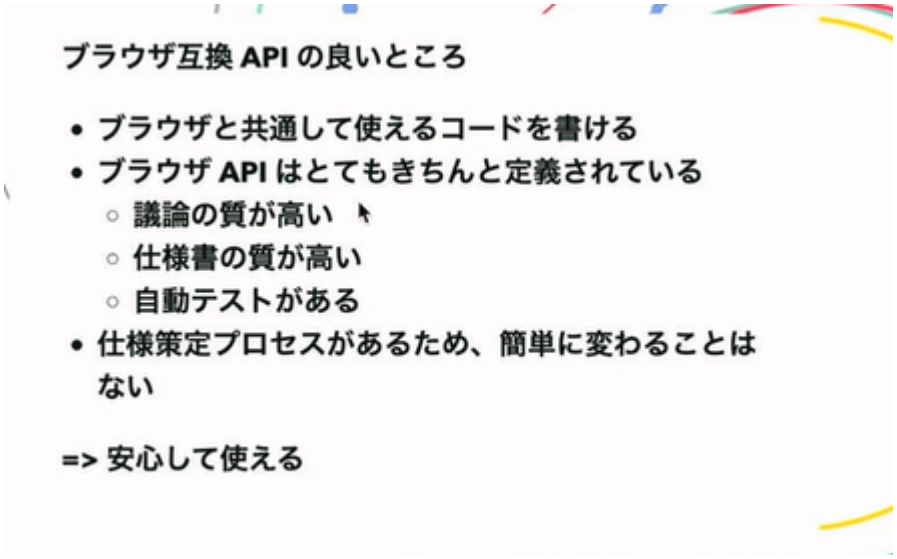
```
import { serve }
  from "https://deno.land/std@0.126.0/http/server.ts";

serve((_req) => new Response("Hello, world"));
```

- URL 指定でモジュールを取得できる。
- この機能があるため Deno では `package.json` や `node_modules` が不要になっている
- Node でも最近実験的な実装が始まったが物議を醸している

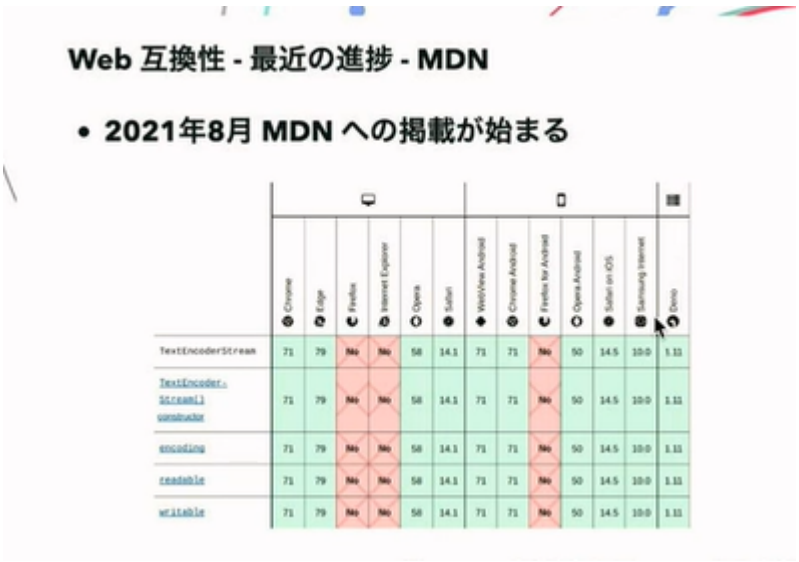
Deno に実装されているブラウザAPI の例

- PubSub - EventTarget API
- ストリーミング処理 - Web Stream API
- 暗号 - Web Crypto API
- GPU - WebGPU
- http server - Request, Response API

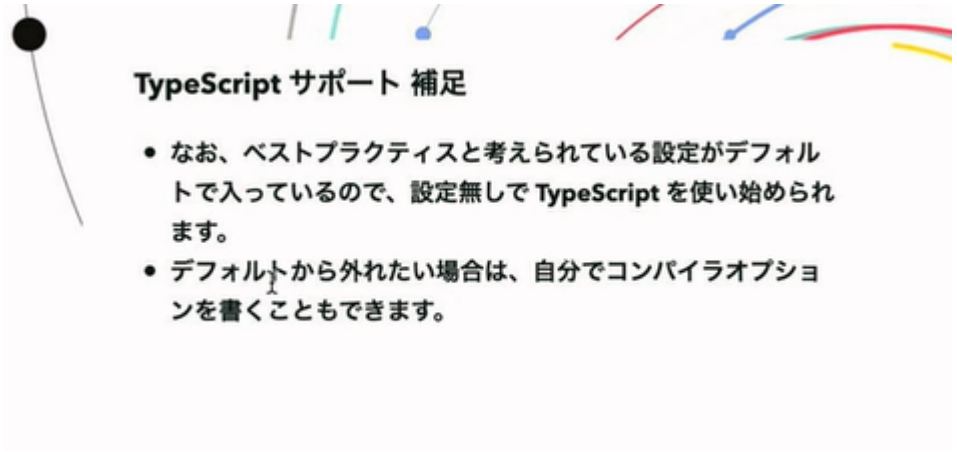
参考: A list of every web API in Deno

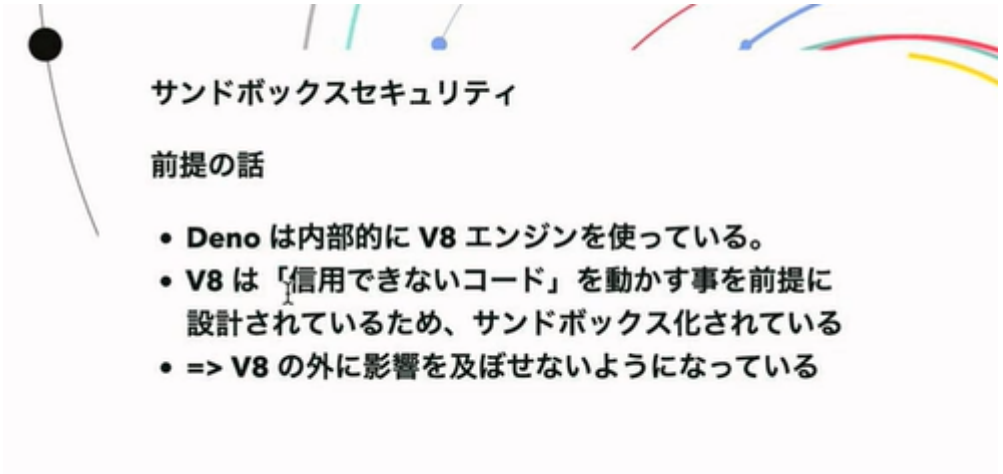


GoogleやAppleの人が議論するほど



デフォルトで型チェックが可能

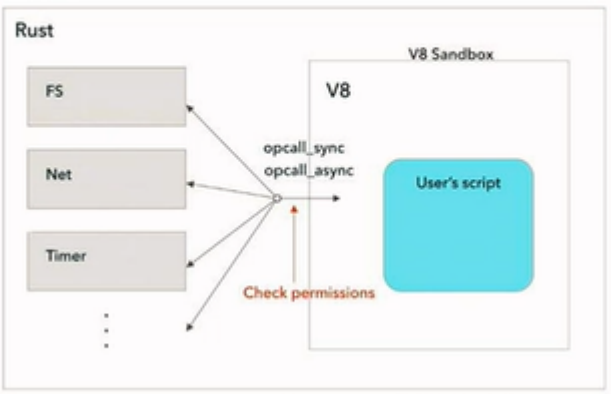




サンドボックスセキュリティ：ユーザ作成の信用できないコードを動かす前提でつくられている

サンドボックスセキュリティ

Deno



サンドボックスセキュリティ

ファイルの読み取りを許可する場合 (全部許可)

```
deno run --allow-read program.ts
```

カレントディレクトリのみ読み込み許可

```
deno run --allow-read=. program.ts
```

サンドボックスセキュリティ

ファイルの書き込みを許可する場合 (全部許可)

```
deno run --allow-write program.ts
```

dist/ ディレクトリのみ書き込み許可

```
deno run --allow-write=dist/ program.ts
```

サンドボックスセキュリティ

ネットワークアクセスを許可する場合 (全部許可)

```
deno run --allow-net program.ts
```

特定のドメイン・ポートのみネットワークアクセス許可

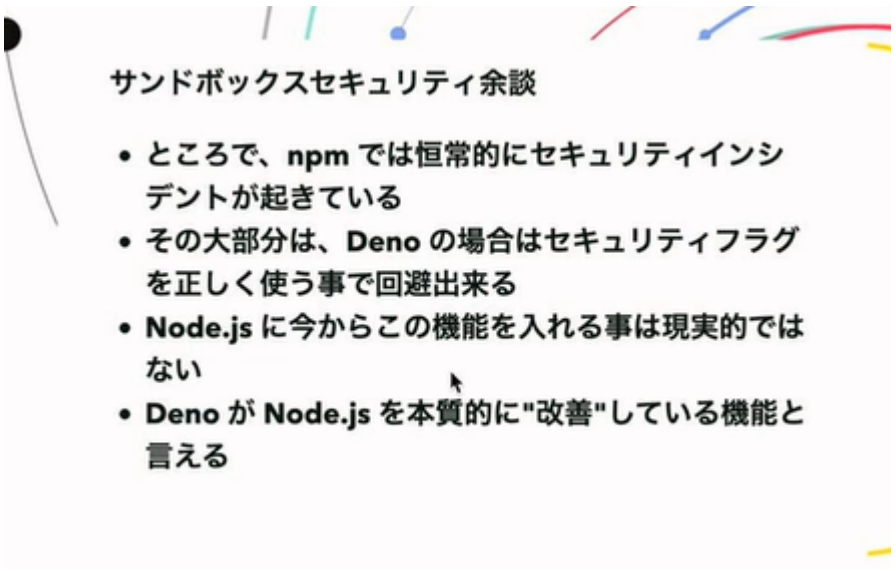
```
deno run --allow-net=example.com:80 program.ts
```

=> 意図しない攻撃コード混入時などに被害を防ぐ事が出来る

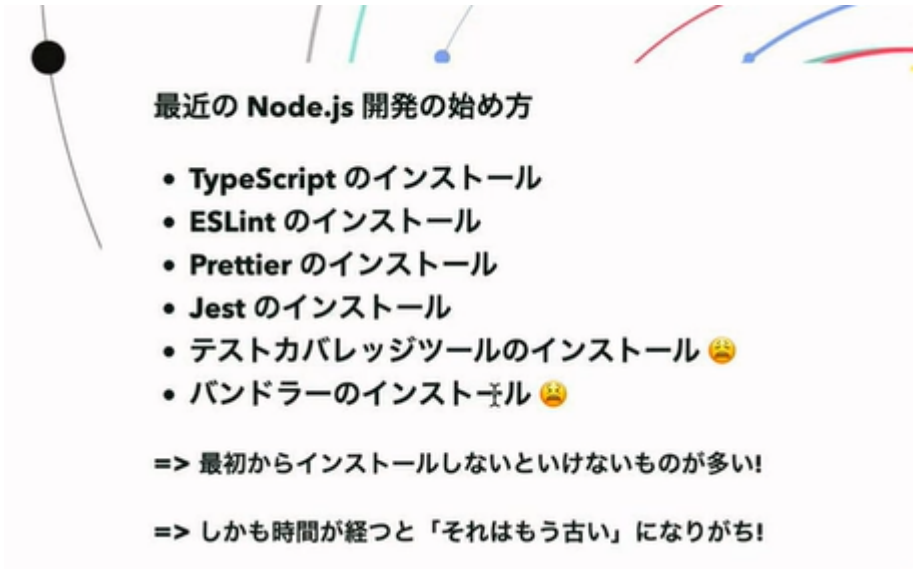
サンドボックスセキュリティ

その他のパーミッション

- `--allow-env` 環境変数の使用許可
- `--allow-run` 別プロセス実行の使用許可
- `--allow-ffi` 外部ネイティブ拡張使用許可
  (Deno のセキュリティモデルを無視したコードが実行されるため、使用注意)
- `--allow-hrtimes` 高精度タイマー使用許可・スペクター対策 (基本許可しない)
- `--allow-all`, `-A` 全部許可、開発時などに利用



開発する前にやることが多すぎる



Denoなら開発を始めるコストが低い

Deno のビルトイン開発ツール

- 例
  - コードのフォーマット => `deno fmt`
  - コードのリント => `deno lint`
  - ユニットテストの実行 => `deno test`
  - テストカバレッジ => `deno coverage`
  - スクリプトのバンドル => `deno bundle`
  - TypeScript => 本体に内包

=> Deno 本体さえあれば、開発に必要なツールが一通り揃っている!

Deno の採用例 - GitHub

次世代 Data Access API

Deno の採用例 - Slack

Slack Introduces New Platform With Help From Deno

November 16, 2021 Jr.
Aaron O'Mullan, Ryan Dahl

Today Slack has announced their next generation development platform. Slack chose Deno for its "secure by default" principles, its web standard APIs, and its first-class TypeScript support. Read more about it at https://api.slack.com/future.

Deno の採用例 - Slack

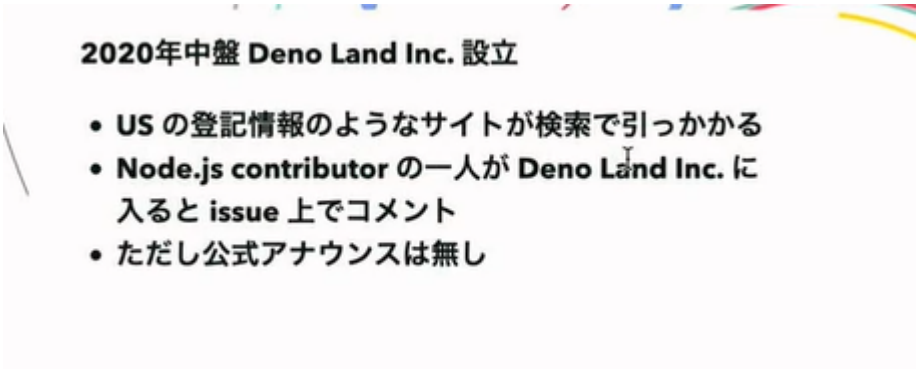
- Slack の新しい SDK は Deno ベース
- この件以降、各種スタートアップから Deno 社への問い合わせがかなり増えている

まとめ

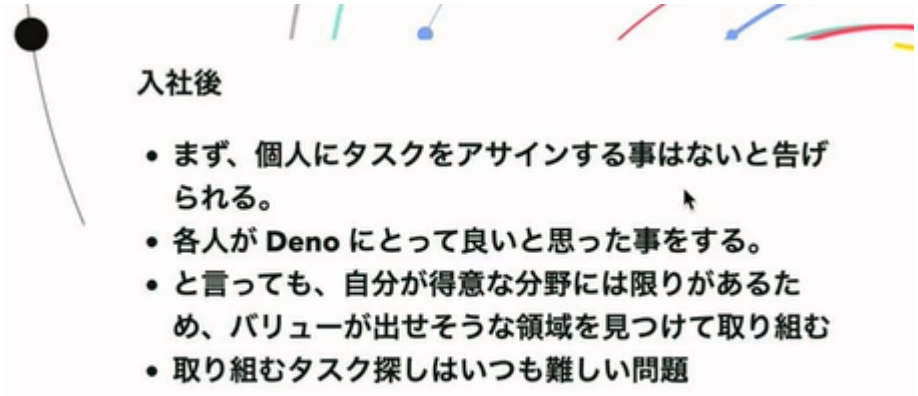
- Deno は "改良版" Node.js を目指すプロジェクト
- Deno は Web 互換性、セキュリティ、TypeScript サポート、ビルトイン開発ツールなどが特徴的でそれらの機能はかなり充実・安定してきている
- Deno は GitHub、Slack などから採用が始まっている

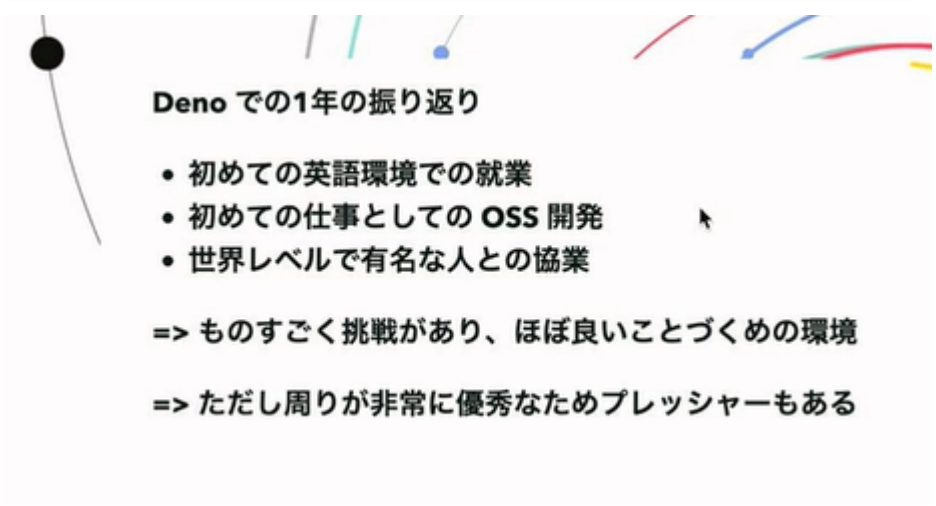
入社するまでのコントリビューション 2018 - 2020

- 純粋に技術的に面白そうという理由で開発に参加
- この時点では会社は存在していなかった
- 外部コントリビュータとしては 3 番目ぐらいに contribute していた



個人にタスクをアサインすることはない





JavaScriptは.jsもTypeScriptは.tsとすればよい

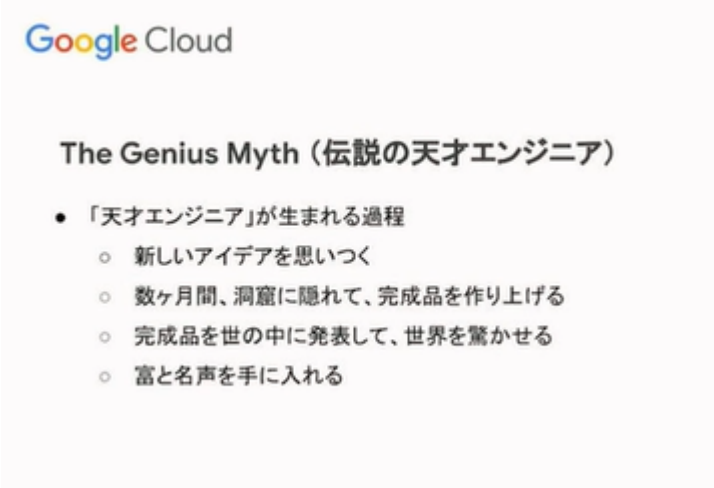
Deno2.0からNode.js完全上位互換になる予定

Denoを学ぶには？
https://examples.deno.land/

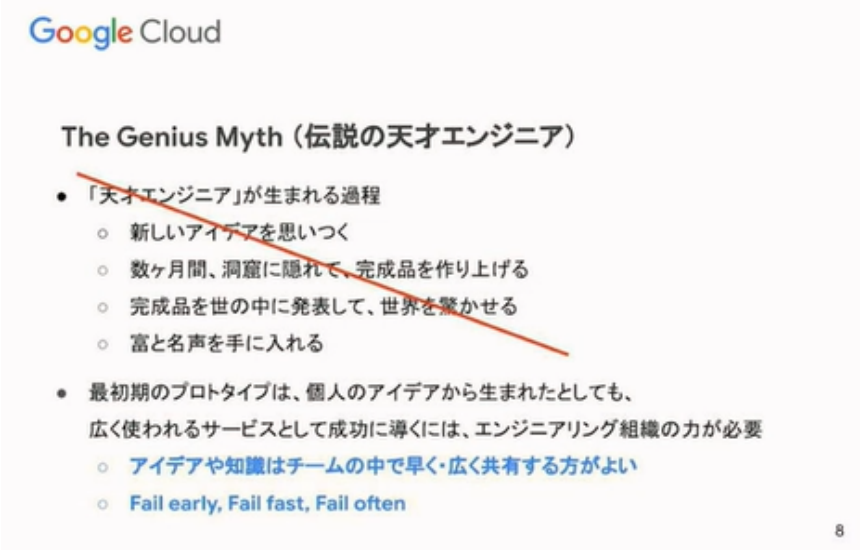
Google Cloud Japanメンバーに聞いてみよう！**IT**エンジニアの人材育成と知識共有 ～ **Google**のスケーラブルなアプローチ

書籍「Software Engineering at Google」は気になる

現実的ではないイメージ



実際



心理的安全性について認識しているあたり、Googleという組織がいかに理知的であるかがよくわかる気がする

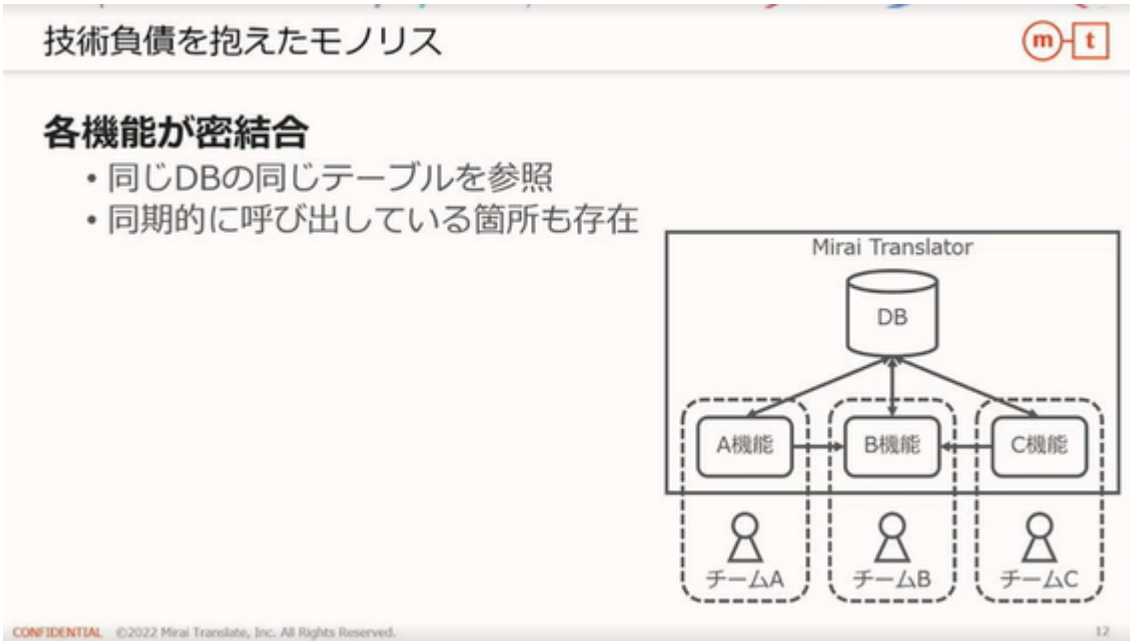
Google Cloud

「データドリブンな判断」と心理的安全性

- データに基づいて客観的に判断することが大切
  - ただし、判断に必要なデータがすべて手に入るとは限らない
  - 現状で手に入るデータで、まずは、判断することが大切
- プロジェクトを進める中で、新しいデータが手に入る
  - 途中で「失敗」に気づく事もある
- 失敗によって得られたデータから、再度、判断をやり直す
  - 「失敗」=「新しいデータを得るためのプロセス」
  - Fail fast and Iterate（早く失敗して、改善を繰り返す）

――――――――――――――――

異なるAPIがひとつのDBを参照



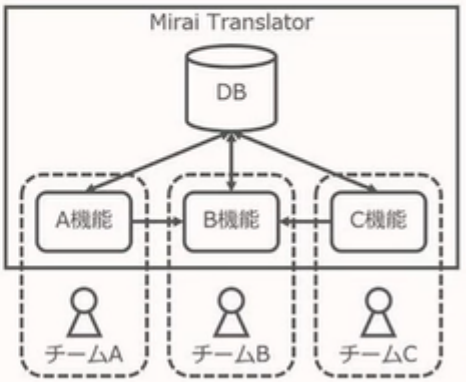
技術負債を抱えたモノリス

各機能が密結合

- 同じDBの同じテーブルを参照
- 同期的に呼び出している箇所も存在

デプロイ単位がシステム全体

- “A機能だけリリース”ができない
- 複数案件をまとめてリリースしがち
- リードタイムが一番遅い案件に引っ張られる



理想的な状態

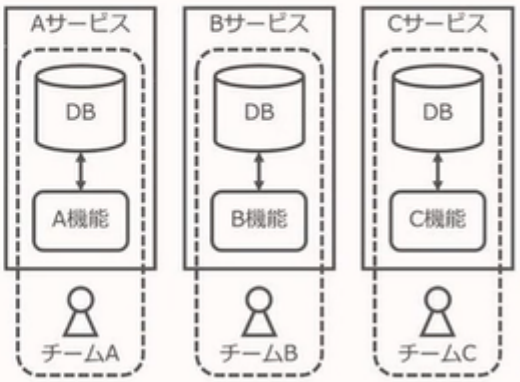
- 各機能が疎結合
- あるチームのリリース（デプロイ）を独立して実施できる

→その為に**マイクロサービス**を目指す

マイクロサービス

複数のサービスでシステムを構成する

利点
- サービス単位にデプロイできる
- サービス間の依存が発生しにくい

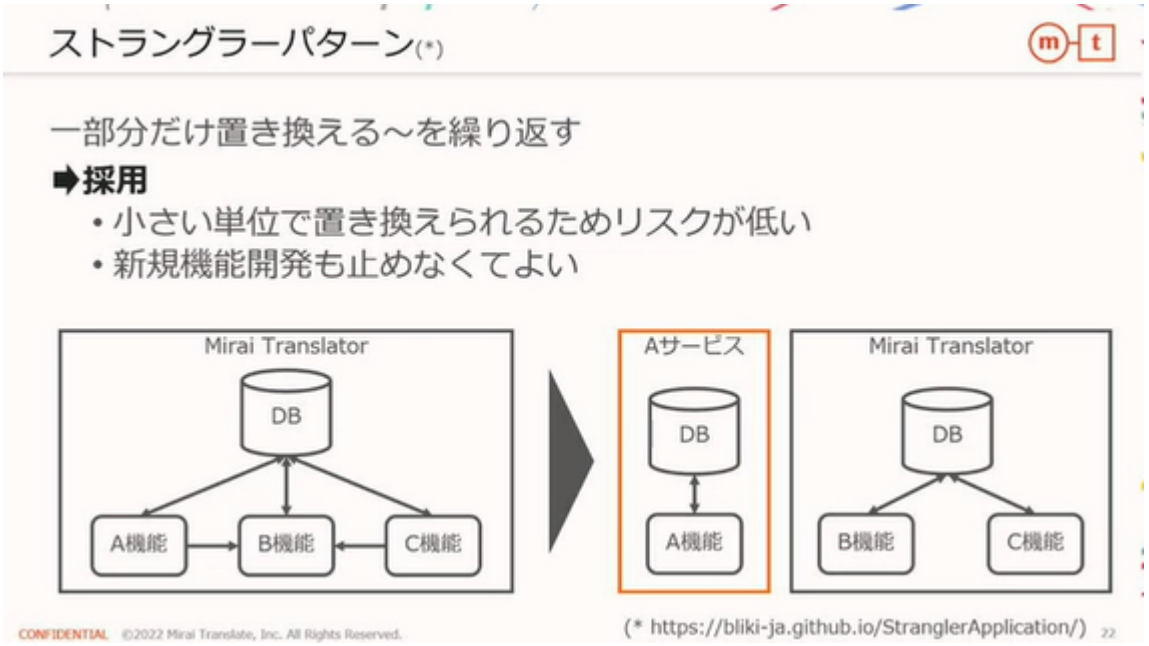


マイクロサービスを目指す理由

理由①：サービス単位のデプロイを可能にしたい
- サービス間で競合開発が発生しない状態を目指す

理由②：モノリスのリファクタリングによる解決は困難
- 不適切なデータモデリングに起因する複雑さが存在
  - テーブル定義の改修は難易度が高い

結果整合性
https://ja.wikipedia.org/wiki/%E7%B5%90%E6%9E%9C%E6%95%B4%E5%90%88%E6%80%A7#:~:text=%E7%B5%90%E6%9E%9C%E6%95%B4%E5%90%88%E6%80%A7%EF%BC%88%E8%8B%B1%3A%20Eventual,%E4%BF%9D%E8%A8%BC%E3%81%99%E3%82%8B%E3%82%82%E3%81%AE%E3%81%A7%E3%81%82%E3%82%8B



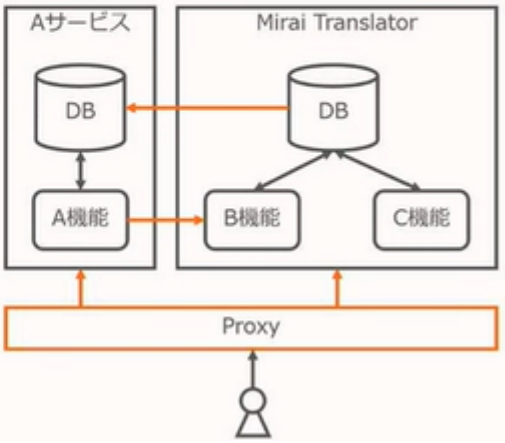
マイクロサービス化では通信は非同期性が大切。でないと、結局密結合になる

移行に向けた課題

モノリスでは不要だった技術基盤が必要

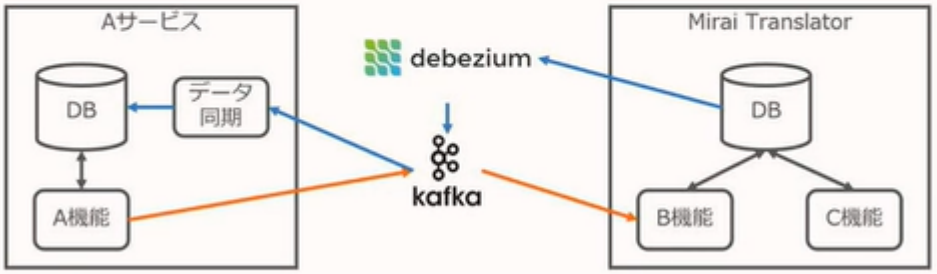
- サービス間の非同期なメッセージング基盤
- モノリスからサービスへのデータ同期基盤
- ユーザーリクエストをサービスとモノリスに振り分ける基盤

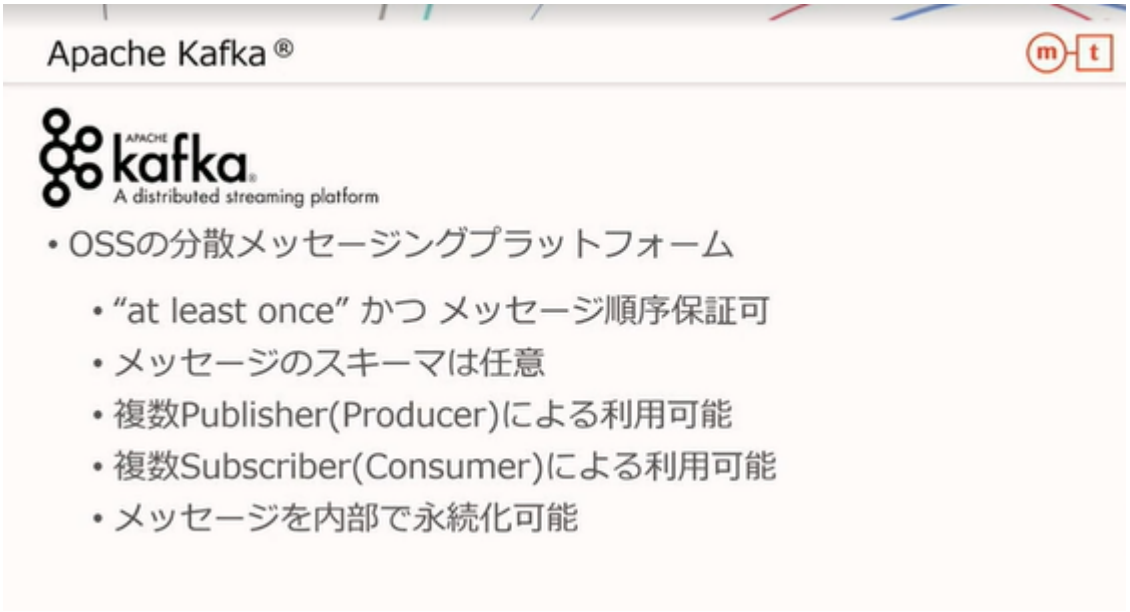
…etc



今作っている技術基盤

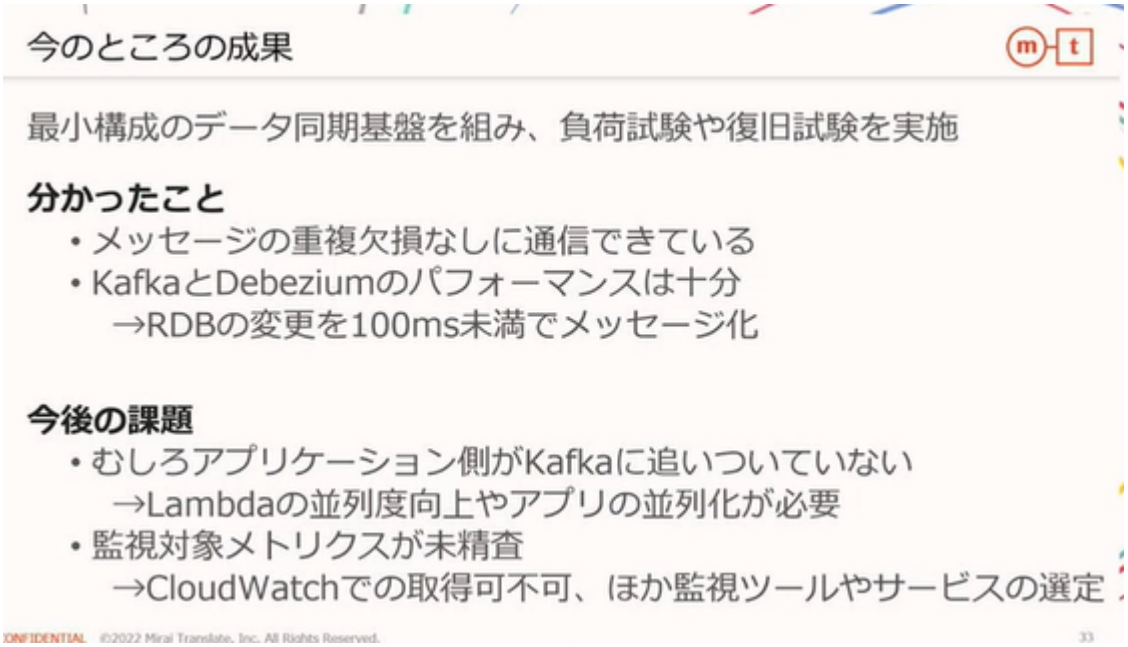
- Kafkaを用いた非同期メッセージング基盤
- Debeziumを用いたデータ同期基盤



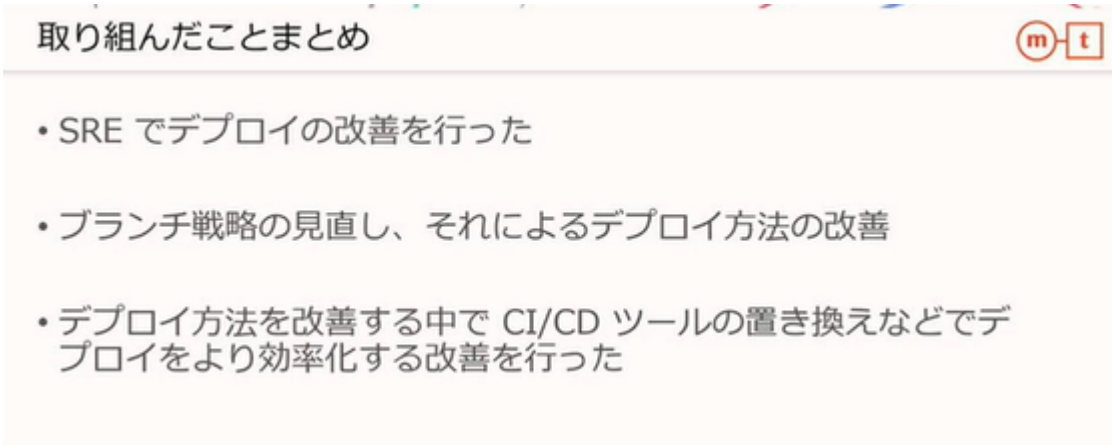


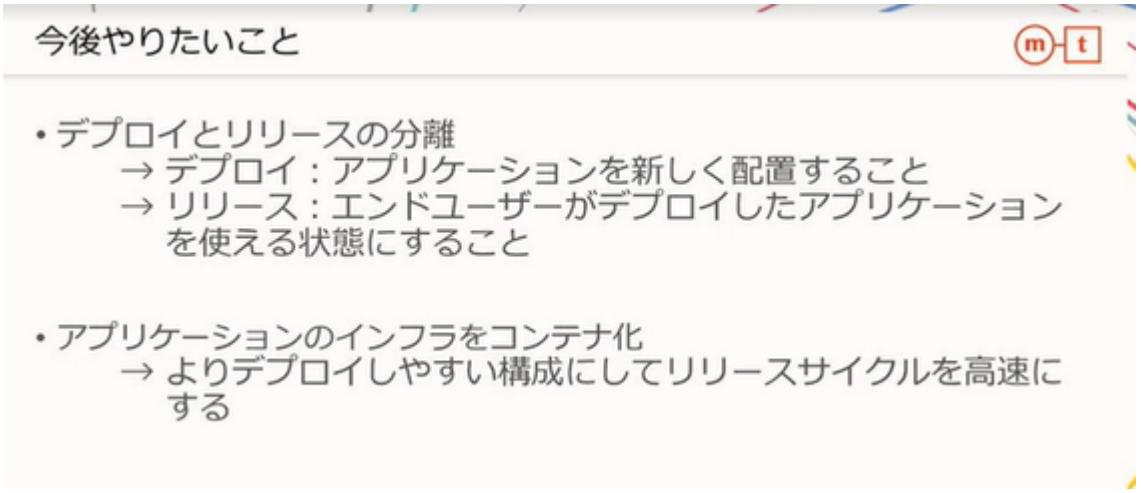
GCPでいうところのCloud PubSub
マイクロサービス化には非同期メッセージングが重要

Debezium
DebeziumはDBに対するデータ操作をキャプチャしてイベントストリームに変換してくれる分散プラットフォーム。
https://debezium.io/
https://rheb.hatenablog.com/entry/debezium-intro
ユーザの声：debezium便利です。特に既存のシステムをあまりいじらなくて済むのがありがたいです



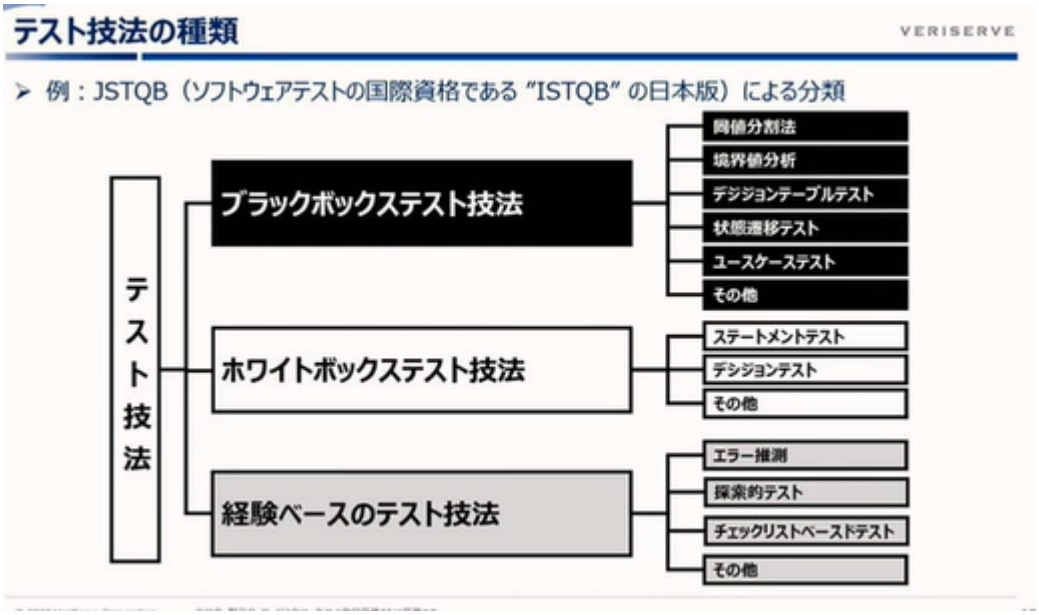
OSSでマイクロサービス化を目指すならこういった技術が有用か。
OSSにこだわらないなら、クラウドプラットフォームのリソースを利用してもよいのだろう。

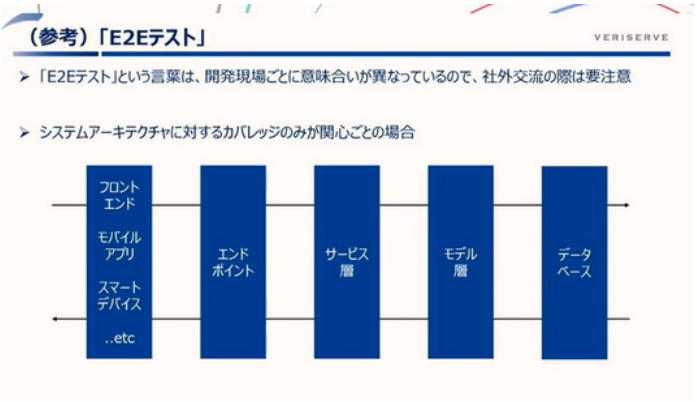




マイクロサービス化は年単位で実施していく

“GIHOZ”を活用したWeb APIテスト設計の勘所





E2E = End to End (頭からお尻まで)

エンジニアキャリアと組織のつくり方、これまでの10年と今後の10年

ベテランの声
すべてを理解するには、人知を超えているので、専門領域をつくるのがふつう。