Effect, une solution efficace aux problèmes de software engineering

Antoine Coulon @ Paris.js #115 - 24/04/2024

Antoine Coulon

Lead Software Engineer @ **evryg**

Créateur **skott**

Auteur **effect-introduction**

Contributeur **Rush.js, NodeSecure**

antoine-coulon

c9antoine

dev.to/antoinecoulon

Effect : le pourquoi plutôt que le comment

- Comprendre quels sont les problèmes les plus communs auxquels nous faisons face en tant que software engineers
- Prendre connaissances des limites actuelles quand on utilise TypeScript et les runtimes JavaScript
- Apprentissage basique d'Effect

Effect ne résout pas uniquement des problèmes propres à TypeScript

- Explicitation
- Testing
- Résilience
- Concurrence
- Gestion des ressources

- Type-Safety
- Composabilité
- Maintenabilité
- Efficacité & Performance

- Monitoring

Prélude : qu'est-ce qu'un Effect

Un Effect se définit à l'aide d'un datatype **Effect<A, E, R>**

- **[A]** représente le résultat qui peut être produit en cas de réussite de l'exécution
- **[E]** représente l'ensemble des erreurs connues qui peuvent survenir lors de l'exécution
- **[R]** représente l'ensemble des dépendances requises pour que l'Effect puisse être exécuté

```
import { Effect } from "effect";

const randomString: Effect.Effect<string, never, never> = //

const now: Effect.Effect<Date, never, Clock> = //
```

Explicitation : mais où sont les erreurs et dépendances ?

> La capacité à rendre un programme transparent, facile à comprendre et sans laisser de place à une quelconque ambiguité vis-à-vis de son fonctionnement.

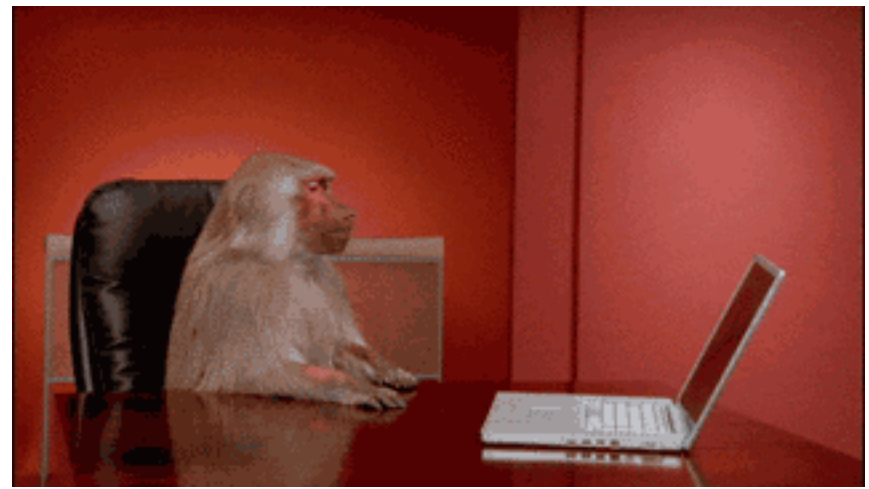
Explicitation (erreurs) : opérations synchrones

```ts
// random.ts
export function generateRandomNumber(): number {
  // some implementation ...
}
```

```ts
// main.ts
import { generateRandomNumber } from "./random";

function main() {
  return generateRandomNumber() * 10;
}
```

```
❯ node main.js
Error: Oops!
    at generateRandomNumber
```



Explicitation (erreurs) : à la recherche de l'info perdue

```ts
// main.ts

function isSomeErrorException(exception: unknown): exception is SomeError {
  return exception instanceof Error && exception.name === "SomeError";
}

function isSomeOtherErrorException(exception: unknown): exception is SomeOtherError {
  return exception instanceof Error && exception.name === "SomeOtherError";
}

try {
  generateRandomNumber();
} catch (exception: unknown) {
  // Worst case? We don't even know what to expect from "exception"

  // Best case:
  if (isSomeErrorException(exception)) {
    // do something
  } else if (isSomeOtherErrorException(exception)) {
    // do something else
  }
}
```

Explicitation (erreurs) : opérations asynchrones

Utilisation des Promises, une des primitives asynchrones

```
interface Promise<T> {}

async function generateRandomNumber(): Promise<number> {
  //
}

generateRandomNumber().catch((_: any) => {});

try {
  await generateRandomNumber();
} catch (exception: unknown) {}
```

- On fait face aux mêmes problèmes
- Divergences/différences au niveau des APIs

Explicitation (erreurs) : Effect à la rescousse

```ts
import { Effect, pipe } from "effect";

class NumberIsTooBigError {
  readonly _tag = "NumberIsTooBigError";
}

class NumberIsTooSmallError {
  readonly _tag = "NumberIsTooSmallError";
}

export const generateRandomNumber: Effect.Effect<number, NumberIsTooBigError | NumberIsTooSmallError, never> = //

const main = pipe(
  generateRandomNumber,
  // exhaustive pattern matching
  Effect.catchTags({
    NumberIsTooBigError: () => Effect.succeed(0),
    NumberIsTooSmallError: () => Effect.succeed(1),
  })
);

type main = Effect.Effect<number, never, never>;
```

Explicitation : et les dépendances ?

Gestion explicite et puissante des erreurs typées

- filtering
- recovery
- mapping
- pattern matching

```
interface Effect<
  A,  // success channel
  E, // error channel
  R, // context channel
> {}
```

Explicitation (dépendances) : inférence automatique des dépendances

```ts
import { Context, Effect } from "effect";

class UserAlreadyExistsError {
  readonly _tag = "UserAlreadyExistsError";
}

class CreatedUser {}

interface UserRepository {
  createUser: () => Effect.Effect<CreatedUser, UserAlreadyExistsError, never>;
}

const UserRepository = Context.GenericTag<UserRepository>("UserRepository");

const registerUser: Effect.Effect<
  CreatedUser,
  UserAlreadyExistsError,
  UserRepository,
> = Effect.flatMap(UserRepository, (userRepository) => userRepository.createUser());
```

Explicitation (dépendances) : type-safety autour du contexte

```ts
import { Effect, pipe } from "effect";

const registerUser: Effect.Effect<
  CreatedUser,
  UserAlreadyExistsError,
  UserRepository
> = // whatever Effect there

// ts(2345): Type 'UserRepository' is not assignable to type 'never'
//              ^^^^^^^^^^^^
Effect.runSync(registerUser);

const registerUserWithSatisfiedDependencies: Effect<CreatedUser, UserAlreadyExistsError, never> = pipe(
  registerUser,
  Effect.provideService(UserRepository, {
    createUser: () => Effect.succeed(new CreatedUser()),
  })
);

// compiles and works
Effect.runSync(registerUserWithSatisfiedDependencies);
```

Testing

> Testing c'est l'art de garantir que le système produit les résultats attendus.

Effect, au service du **Dependency Inversion Principle**

```
class InMemoryUserRepository implements UserRepository {
  createUser() {
    //
  }
}

test("Should blabla", async () => {
  const fakeRepository = new InMemoryUserRepository();

  const user = await pipe(
    createUser(),
    Effect.provideService(UserRepository, fakeRepository),
    Effect.runPromise
  );

  expect(user).toEqual("whatever");
});
```

Résilience

> L'art de designer et d'implémenter des systèmes qui peuvent réagir à des erreurs attendues et de les gérer correctement.

- Recovery
- Retries
- Interruptions
- Resource management
- Circuit breakers
- etc

Résilience : recovery et retries

```ts
// RECOVERY

import { Effect, pipe } from "effect";

const program = () => pipe(
  Effect.fail(new Error()),
  // Recover from all errors
  Effect.catchAll((_) => Effect.succeed(0)),
  // Recover from all errors and provide spe
  Effect.catchTags({
    _1: () => Effect.succeed(1),
    _2: () => Effect.succeed(2)
  }),
  // Recover from specific errors only
  Effect.catchTag({
    _1: () => Effect.succeed(1)
  })
)
```

```ts
// RETRY

import { Effect, pipe, Schedule, Duration } from "effect";

const schedulePolicy = pipe(
  Schedule.recurs(5),
  Schedule.addDelay(() => Duration.millis(500)),
  Schedule.compose(Schedule.elapsed),
  Schedule.whileOutput(Duration.lessThanOrEqualTo(Duration.seconds(3))),
  Schedule.whileInput(
    (error) => error instanceof Error && error.message !== "_"
  )
);

const programWithRetryPolicy = pipe(
  Effect.failSync(() => new Error("Some_error")),
  Effect.retry(schedulePolicy),
  Effect.catchAll(() => Effect.sync(() => {
    console.log("Program ended")
  }))
);
```

Résilience : la nécessité d'interruption et de cleanup

```
import { setTimeout } from "node:timers/promises";

const leakingRace = () => Promise.race([
  setTimeout(1000),
  setTimeout(10000)
]);

function raceWithInterruptions() {
  const abortController1 = new AbortController();
  const abortController2 = new AbortController();

  async function cancellableTimeout1() {
    await setTimeout(1000, undefined, { signal: abortContr
    abortController2.abort();
  }

  async function cancellableTimeout2() {
    await setTimeout(10000, undefined, { signal: abortCont
    abortController1.abort();
  }

  return Promise.race([cancellableTimeout1(), cancellableT
}
```

```
import { Effect } from "effect";

const race = [
  Effect.sleep(1000),
  Effect.sleep(10_000).pipe(
    Effect.onInterrupt(() => Effect.log("interrupted"))
  ),
];
```

```
const backgroundJob = Effect.async(() => {
  const timer = setInterval(() => {
    console.log("processing job ... ");
  }, 500);

  return Effect.sync(() => {
    console.log("releasing resources ... ");
    clearInterval(timer);
  });
});
```

Concurrence

> L'art de gérer des opérations en simultanée ou de manière coopérative avec un contrôle sur leur exécution et des garanties fortes sur l'intégrité de l'état du programme, afin d'améliorer son efficacité.

"concurrency is about dealing with lots of things at once", Rob Pike

Gérer la concurrence correctement c'est compliqué :

- Difficile d'avoir un modèle d'exécution déterministe
- Problème des ressources partagées + gestion des ressources
- Deadlocks, race conditions
- Contrôle et efficacité Mémoire et CPU
- …etc

Concurrence : bounded vs unbounded

```ts
// Unbounded
const userIds = Array.from(
  { length: 1000 }, (_, idx) => idx
);

function fetchUser(id: number): Promise<User> {}

function retrieveAllUsers() {
  return Promise.all(userIds.map(fetchUser));
}
```

- Pas de gestion des interruptions
- Pas de garantie sur la libération des ressources
- Pas de contrôle sur l'exécution concurrente
- `Promise.allSettled` permet un contrôle plus fin sur le résultat produit mais souffre des mêmes problèmes

```ts
const users = pipe(
  userIds,
  Effect.forEach(
    (id) => Effect.promise(() => fetchUser(id)),
    { concurrency: 30 },
    // OR
    { concurrency: "unbounded" },
    // OR
    { concurrency: "inherit" }
  )
);
```

Resource management

> L’art de gestion du cycle de vie des ressources allouées lors de l’exécution du programme

- Proposal "Explicit Resource Management", mais généralisé et plus composable
- Introduction de Scopes, dès lors qu’on a plus besoin des ressources, des finalizers sont appelés
- Finalizers appelés dès lors qu’une interruption/erreur d’un Effect est produite
- Contexte qui contrôle la propagation des scopes, on évite le "props drilling" des Abort Signals
- Libération Sync/Async, peut être elle-même rendue interruptible/non-interruptible

Et aussi d’autres modules…

- Batching/Caching
- Tracing/Monitoring
- Layer
- Stream
- Queue
- Semaphore
- Pub/Sub
- Config

- @effect/schema
- @effect/cli
- @effect/fastify
- @effect/http
- @effect/rpc
- @effect/opentelemetry

Merci d’avoir écouté !

- **Effect website**: https://effect.website
- **Effect introduction**: https://github.com/antoine-coulon/effect-introduction

Questions ?