Swift

岸川克己 **@k_katsumi**

Yuta Koshizawa **@koher**

岸川克己 **@k_katsumi**

Apple 好きが高じて iOS アプリケーションの開発を 14 年続けています。趣味で作っている OSS ライブラリはけっこう人気があります。

Yuta Koshizawa **@koher**

"Heart of Swift" を書きました。普段は主にリアルタイム画像処理を用いた iOS アプリを開発しています。

Swift の最新動向

Swift 5.5: 9 月リリース（？）

Concurrency （並行処理）

Swift 5.5+ の Concurrency

並行処理に起因するデータ競合やデッドロックが起こらないことをコンパイラが保証してくれる。

一見 Ruby や Python のようだけど・・・

```
// 0 以上 100 未満の整数の合計
var sum = 0
for i in 0 ..< 100 {
    sum += i
}
print(sum)
```

コンパイラがガチガチに静的検査する言語

- Null Safety （ 2014 年のリリース当初から）
- `throws/try` （検査例外のようなもの）

コンパイラがガチガチに静的検査する言語

- Null Safety （ 2014 年のリリース当初から）
- `throws/try` （検査例外のようなもの）
- データ競合とデッドロックの防止

Swift の並行処理

1. `async/await`
2. Structured Concurrency
3. `actor`

1️⃣ async/await

async/await の例

```swift
// Swift
func fetchUser(...) async -> User {
    let userData = await downloadData(...)
    ...
    return user
}
```

一見 JavaScript 等の async/await と似ている。

JS/TS の async/await との違い

```
// TypeScript
async function fetchUser(...): Promise<User> {
    const userData = await downloadData(...);
    ...
    return user;
}
```

JS/TS では async 関数は Promise を返す。

JS/TS の async/await との違い

```swift
// Swift
func fetchUser(...) async -> User {
    let userData = await downloadData(...)
    ...
    return user
}
```

Swift の async 関数は Promise を返さない。

JS/TS の async/await との違い

```
// TypeScript
async function fetchUser(...): Promise<User> {
    const promise = downloadData(...);
    ...
    return user;
}
```

JS/TS では await しないと Promise を得る。

JS/TS の async/await との違い

```swift
// Swift
func fetchUser(...) async -> User {
    let userData = downloadData(...) // ⛔ コンパイルエラー
    ...
    return user
}
```

Swift では await しないとコンパイルエラー。必ず await されるので Promise に相当するものはない。

await 必須なので await 忘れが起こらない

```
func updateUser(user: User) async {
    // ⛔ コンパイルエラー
    db.collection("users")
        .document(user.id)
        .setData(from: user)
}
```

（特に戻り値がない場合でも）意図せず await を付け忘れる事故が起こらない。

JS/TS では Promise を活用して並行に実行

```
// TypeScript
async function fetchUser(...): Promise<User> {
    const userPromise = downloadData(...);
    const avatarPromise = downloadData(...);
    const [userData, avatarData]
        = await Promise.all([userPromise, avatarPromise]);
    ...
    return user;
}
```

await しないことが可能だと複数の非同期処理を並行に実行できる。

await 必須で Promise がないなら複数の非同期処理をどうやって並行に実行する？🤔

2️⃣ Structured Concurrency

async let で並行処理

```
// Swift
func fetchUser(...) async -> User {
    async let userData = downloadData(...)
    async let avatarData = downloadData(...)
    let user = await User(data: userData,
        avatar: avatarData)
    return user
}
```

JS/TS の場合

```
// TypeScript
async function fetchUser(...): Promise<User> {
    const userPromise = downloadData(...);
    const avatarPromise = downloadData(...);
    const [userData, avatarData]
        = await Promise.all([userPromise, avatarPromise]);
    ...
    return user;
}
```

async let で並行処理

```
// Swift
func fetchUser(...) async -> User {
    async let userData = downloadData(...)
    async let avatarData = downloadData(...)
    let user = await User(data: userData,
        avatar: avatarData)
    return user
}
```

async let は Promise と違ってスコープの外に持ち出せない。

Structured Concurrency

構造化プログラミング（ Structured Programming ）では `goto` でスコープから脱出できないように、 `async let` は必ずスコープの中で解決される。

Structured Concurrency の何がうれしいのか?

例: 非同期処理のキャンセル

async 関数の呼び出しの制約

```
func foo() async -> Int { ... }

func bar() async -> Int {
    await foo() * 2 // ✅ OK
}
```

async 関数は async 関数の中でしか呼べない。

async 関数の呼び出しの制約

```
func foo() async -> Int { ... }

func bar() -> Int {
    await foo() * 2 // ⛔ コンパイルエラー
}
```

async 関数は async 関数の中でしか呼べない。

Task

```
Task({
    await foo()
})
```

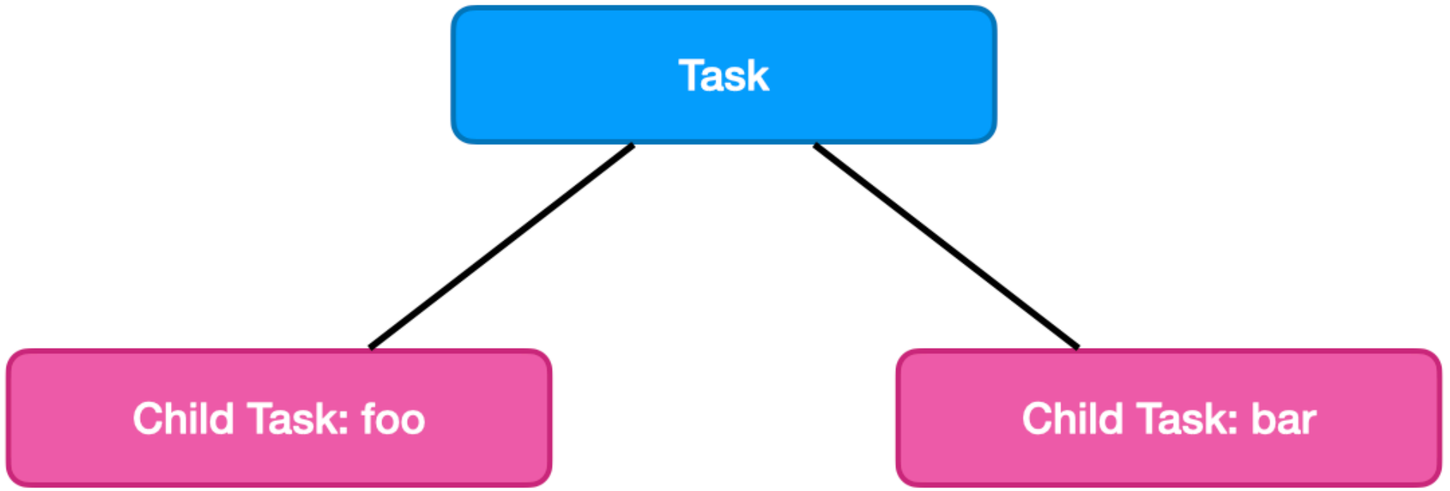
async 関数の呼び出しの大元は Task のコンストラクタ（イニシャライザ）。すべての async 関数は Task の上で実行される。

Child Task

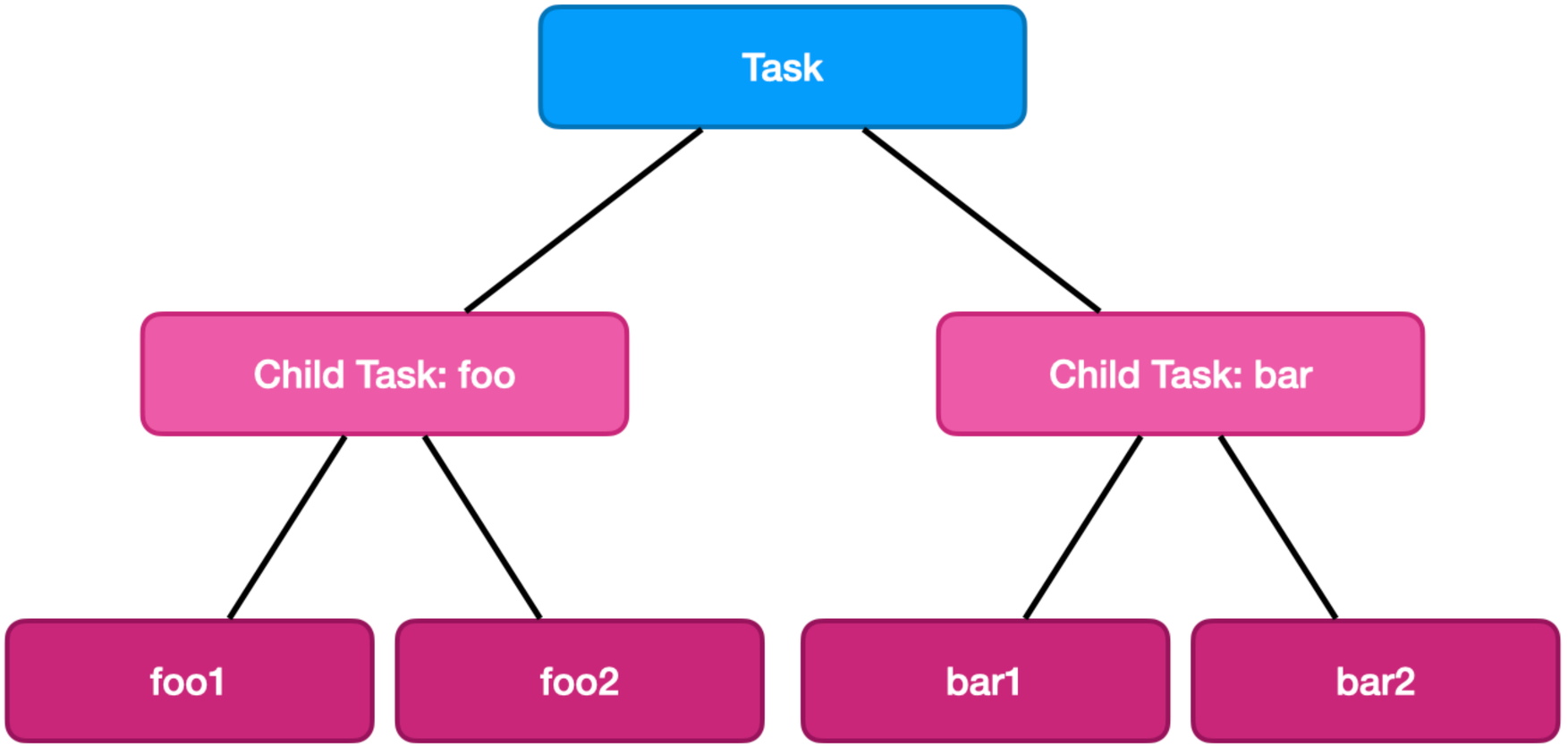
```
Task({
    async let a = foo() // Child Task: foo
    async let b = bar() // Child Task: bar
    print(await a + b)
})
```

async let では並行に実行される Child Task が作られる。構造化されているので Task はツリーを作る。

Task Tree



Task Tree



非同期処理のキャンセル

```
let task = Task({
    async let a = foo()
    async let b = bar()
    print(try await a + b)
})

task.cancel()
```

Task を cancel すると Task ツリー全体が一括キャンセルされる。非同期処理のキャンセルを扱いやすい。

3️⃣ actor

データ競合とデッドロックの防止

データ競合の例

```
class Counter {
    var count: Int = 0
    func increment() -> Int {
        count += 1
        return count
    }
}
```

カウンタに並行にアクセスする場合

```
let counter = Counter()

Task {
    print(counter.increment()) // ?
}
Task {
    print(counter.increment()) // ?
}
```

カウンタに並行にアクセスする場合

```
let counter = Counter()

Task {
    print(counter.increment()) // 1
}
Task {
    print(counter.increment()) // 2
}
```

カウンタに並行にアクセスする場合

```
let counter = Counter()

Task {
    print(counter.increment()) // 2
}
Task {
    print(counter.increment()) // 1
}
```

カウンタに並行にアクセスする場合

```
let counter = Counter()

Task {
    print(counter.increment()) // 2
}
Task {
    print(counter.increment()) // 2
}
```

両方のインクリメントが行われた後で return されると両者とも

2 を表示する。 → **データ競合**

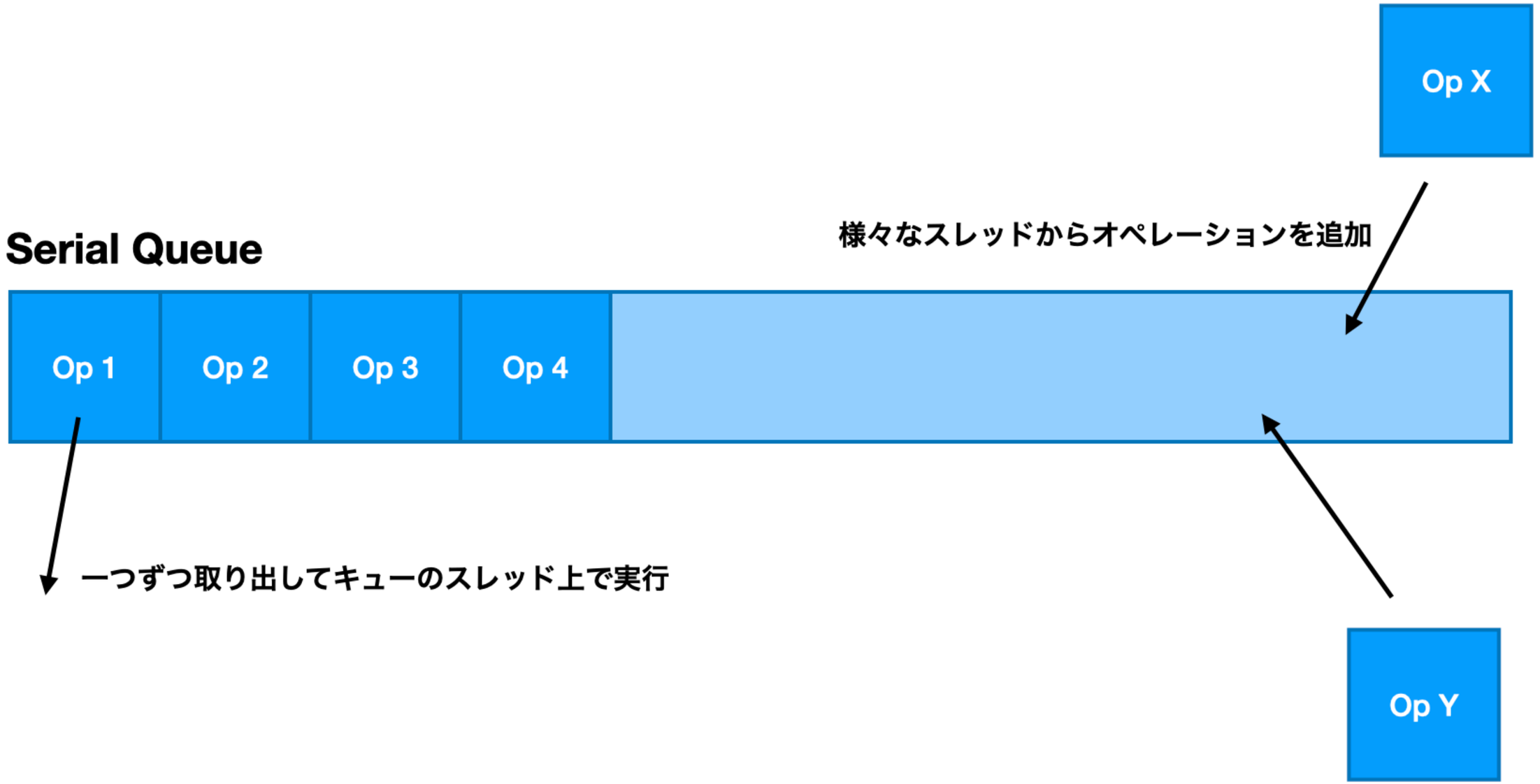
従来の解決法: ロック

```
class Counter {
    var count: Int = 0
    func increment() -> Int {
        lock()
        defer { unlock() }
        count += 1
        return count
    }
}
```

ロックを使うとスレッドをブロックしてしまうパフォーマンス上の不利益に加えて、デッドロックの原因となる恐れがある。

従来の解決法: シリアルキュー

オペレーションをシリアルキューに入れて、キューが一つずつ順番に取り出して処理をする。データ競合もデッドロックも起こらない。


Op X
Serial Queue
様々なスレッドからオペレーションを追加
Op 1
Op 2
Op 3
Op 4
一つずつ取り出してキューのスレッド上で実行
Op Y

従来の解決法: シリアルキュー

```
class Counter {
    let queue = ...
    var count: Int = 0
    func increment(completion: (Int) -> Void) {
        queue.async {
            count += 1
            completion(count)
        }
    }
}
```

実装が複雑だったり、キューを使うのを忘れたりしがち。

```
// シリアルキューを使う現実のコードはより複雑
let queue = DispatchQueue(...)

func load(_ url: URL) {
  queue.sync {
    // メモリキャッシュを探す
    ...
  }

  queue.sync {
    // ディスクキャッシュを探す
    ...
  }

  queue.async {
    // 画像をダウンロード
    fetch(url) {
      if {
        // 成功: キャッシュに保存
        // コールバックを呼ぶ
        ...
      } else if {
        // セッション切れ:
        queue.async {
          // リフレッシュトークンでリトライ
        }
        ...
      } else {
        // 失敗: エラーハンドラを呼ぶ
        ...
      }
    }
  }
}
```

actor による解決

```
actor Counter {
    var count: Int = 0
    func increment() -> Int {
        count += 1
        return count
    }
}
```

キューに入れるのと同じようなことをコンパイラが行ってくれる。パフォーマンスもより良い。

actor のメソッドを外から見ると

```
actor Counter {
    ...
    func increment() async -> Int {
    ...
    }
}
```

actor 外部からは async メソッドに見える。

actor のメソッドを外から見ると

```
let counter = Counter()
print(await counter.increment())
```

actor のメソッドを中から見ると

```
actor Counter {
    var count: Int = 0
    func countUp(amount: Int) -> Int {
        count += amount
        return count
    }
    func increment() -> Int {
        countUp(amount: 1) // 同期呼び出し
    }
}
```

外から見たら async だけど、中から見たら同期的。

actor

- オペレーションをキューに入れて順番に行うのでデータ競合が起こらない
- actor が自動的にオペレーションをキューに入れるのでキューに入れ忘れない
- 外部からは async に見え、ブロッキングが存在しないのでデッドロックが起こらない

これだけではデータ競合を完全には防げない

actor に渡したインスタンスを外部から変更

```
actor Foo {
    func method(x: X) { ... }
}

let foo = Foo()
let x = X()
foo.method(x: x)
x.value += 1 // 守られていない
```

actor に渡したインスタンスを外部から変更

```
actor Foo {
    func method(x: X) { ... }
}

let foo = Foo()
let x = X()
foo.method(x: x) // ⛔ コンパイルエラー
x.value += 1
```

ミュータブルクラスのインスタンスを渡すとコンパイルエラー。

値型やイミュータブルクラスのインスタンスは渡せる。

actor から受け取ったインスタンスを外部から変更

```
actor Foo {
    let x = X()
    func method() -> X { return x }
}

let foo = Foo()
let x = await foo.baz() // ⛔ コンパイルエラー
x.count += 1
```

ミュータブルクラスのインスタンスを返すとコンパイルエラー。

値型やイミュータブルクラスのインスタンスは取り出せる。

その他に、デッドロックを完全に防ぐためには `actor` の Reentrancy が・・・

時間が足りないので詳細は Proposal [SE-0304] を御覧ください。

[SE-0304] **Structured concurrency** https://github.com/apple/swift-evolution/blob/main/proposals/0304-structured-concurrency.md

まとめ

1️⃣ `async/await`
2️⃣ Structured Concurrency
3️⃣ `actor`

これらの仕組みを用いて、並行処理でデータ競合とデッドロックが起こらないことをコンパイラが保証。

参考文献

Swift Evolution Proposals

SE-0296 **Async/await** https://github.com/apple/swift-evolution/blob/main/proposals/0296-async-await.md

SE-0302 **`Sendable` and `@Sendable` closures** https://github.com/apple/swift-evolution/blob/main/proposals/0302-concurrent-value-and-concurrent-closures.md

SE-0304 **Structured concurrency** https://github.com/apple/swift-evolution/blob/main/proposals/0304-structured-concurrency.md

SE-0306 **Actors** https://github.com/apple/swift-evolution/blob/main/proposals/0306-actors.md

SE-0316 **Global actors** https://github.com/apple/swift-evolution/blob/main/proposals/0316-global-actors.md

SE-0317 **`async let` bindings** https://github.com/apple/swift-evolution/blob/main/proposals/0317-async-let.md

WWDC 2021 Sessions

10132 **Meet async/await in Swift** https://developer.apple.com/videos/play/wwdc2021/10132/

10133 **Protect mutable state with Swift actors** https://developer.apple.com/videos/play/wwdc2021/10133/

10134 **Explore structured concurrency in Swift** https://developer.apple.com/videos/play/wwdc2021/10134/

10254 **Swift concurrency: Behind the scenes** https://developer.apple.com/videos/play/wwdc2021/10254/