Flutter チュートリアル

flutter & riverpod & flutter_hooks のチュートリアル

はじめに

本チュートリアルではFlutterの初心者がRiverpod・FlutterHooksを使用し、フロントエンドアプリケーションを開発できるようになることを目的とする。

また、以下の点を前提としていることに留意する。

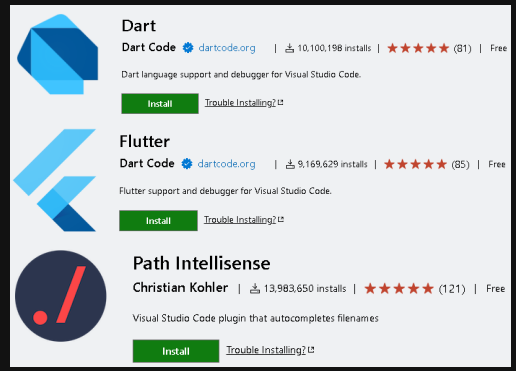
- Flutterの環境構築を行っている
  Flutterの公式などを見ながら環境構築を行っておく
- プログラミング言語の基礎的な文法を理解している
  Dart特有の文法などは知っている必要はない
  (Effective Dartを読むと特有の書き方を知れる)

また、（本チュートリアルに限らず）実際にコードを書いて、"どこに何を書くとどうなるか"という試行錯誤をしながら学ぶことを推奨する。

はじめに(VSCodeの場合)

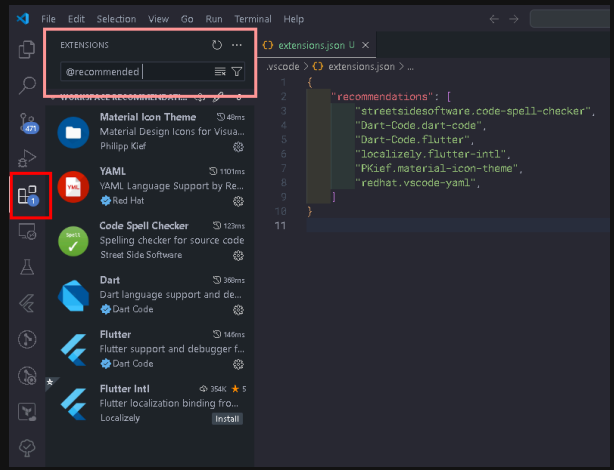
以下の拡張機能をインストールしておく

- 必須
  - Dart
  - Flutter
- おすすめ
  - パス入力の補完：Path Intellisense



また、複数人で管理されているコードであれば `.vscode/extensions.json` ファイルに従って推奨されている拡張機能を適宜インストールする。

Extensionsタブを開き(Ctrl+Shift+X)、「@recommended」で検索すると推奨されている拡張機能が一覧で見える。



Widgetの基本

An error occurred on this slide. Check the terminal for more information.

An error occurred on this slide. Check the terminal for more information.

An error occurred on this slide. Check the terminal for more information.

StatefulWidgetは使用しない

次に画面に動的な変化を与えるようにしたい。(例: 画面に表示する数字をボタンを押すことでカウントアップさせるなど)

`StatelessWidget` はその名の通り状態を持つことができないため、これだけでは画面を動的に更新することができない。
そこで標準で用意されているのが `StatefulWidget` である。
`StatefulWidget` は `StatelessWidget` と違い、継承することで変数を管理して画面を動的に更新することができるようになる。

しかし、これにはいくつも欠点があり、参考までに右記に記載しておく。

右記の問題を解決するためにあるのがRiverpodであり、
本チュートリアルではRiverpodを使用する。
そのため、`StatefulWidget` の存在の紹介をするだけにとどめておく。

欠点例

- 状態の再利用性が低い
  手動で親子間で変数の授受を行わなければならない
- 状態の永続化が困難
  状態がWidgetと直接紐づいているため永続化しづらい
- 再描画効率の悪さ
  setStateによって状態を更新すると、そのWidget全体が再描画される可能性がある
- 可読性の低下
  Widget内部にロジックが書かれやすいため、可読性が悪くなりがちである
- ステートの解放が手動
  disposeやinitStateを手動で行って初期化・解放する必要があり非効率
- テストが難しい
  ロジックとUIが絡み合ってしまい、ユニットテストしづらい

Riverpod

Riverpodとは

Riverpod: Flutter(dart)のライブラリで、プロバイダーというものを使って、どの `Widget` からもアクセスできる状態を管理する。

Riverpodのライブラリは以下の3つがあるが、今回はFlutterを使用しつつflutter_hooksも使用するためhooks_riverpodを選択する。

- riverpod: Dartのみ使用する場合
- flutter_riverpod: FlutterでRiverpodのみ使用する場合
- hooks_riverpod: Riverpodとflutter_hooksを使用する場合

flutter_hooksも状態管理のためのライブラリで、後に使い方も紹介する。
まずはFlutterプロジェクトで以下のコマンドを実行して、最新のriverpodを使用できるようにしておく。

```
flutter pub add hooks_riverpod hooks_riverpod
```

An error occurred on this slide. Check the terminal for more information.

プロバイダーの種類

プロバイダーには様々な種類があり、一覧で記述しておく。
次頁以降で各プロバイダーの詳細を見ていく。

Provider	最も単純なプロバイダー
NotifierProvider	クラスで提供する値を制御するプロバイダー
FutureProvider	AsyncValue<T>を提供するプロバイダー
StreamProvider	Streamで変化する値をAsyncValue<T>として提供するプロバイダー
AsyncNotifierProvider	クラスで提供する値を制御するFutureProvider
StreamNotifierProvider	クラスで提供する値を制御するStreamProvider
ChangeNotifier	mutableで複雑な状態管理を行うプロバイダー (※基本的に使用することがない)

※AsyncValueは後述する

An error occurred on this slide. Check the terminal for more information.

プロバイダーの種類：NotifierProvider

`NotifierProvider` は `Provider` とは違い、提供する値を更新するメソッドも提供する。
また、`ref.read(**.notifier)` で提供されるメソッドにアクセスできる。

NotifierProviderの基本のサンプルコード: カウンターを表すint型を提供している

```
final counterProvider = NotifierProvider<CounterNotifier, int>(CounterNotifier.new);
class CounterNotifier extends Notifier<int> {
  @override
  int build() {
    return 0;
  }
  void increment() {
    state += 1;
  }
}
// widget
TextButton(
  child: Text(
    'カウンター: ${ref.watch(counterProvider)}'
  ),
  onPressed: () {
    ref.read(counterProvider.notifier).increment(); // increment呼び出し
  },
)
```

An error occurred on this slide. Check the terminal for more information.

An error occurred on this slide. Check the terminal for more information.

プロバイダーの種類：FutureProvider

`FutureProvider` は `Provider` のFuture版で、
非同期的に値を取得するため、`AsyncValue<T>` として値を提供するプロバイダーである。

```
final sampleProvider = FutureProvider<int>((ref) async {
  await Future.delayed(const Duration(seconds: 1)); // 非同期処理を待ってから初期値を返す
  return 10;
});

class SampleWidget extends ConsumerWidget {
  const SampleWidget({super.key});

  @override
  Widget build(BuildContext context, WidgetRef ref) {
    final asyncVal = ref.watch(sampleProvider);
    return switch (asyncVal) {
      AsyncData<int>(:final value) => Text('$value'),
      AsyncError<int>(:final error) => Text('$error'),
      _ => CircularProgressIndicator(),
    };
  }
}
```

An error occurred on this slide. Check the terminal for more information.

プロバイダーの種類：FutureProvider (AsyncValueについて)

`AsyncValue` は抽象クラスで、3つの具象クラスにわけられる。

- `AsyncData` : データを保持しているとき
- `AsyncError` : 取得中にエラーが発生したとき
- `AsyncLoading` : データ取得中のとき

また、それぞれが3つのパラメータを持っている。

- `value` (`hasValue` でチエックできる)
  `AsyncData` の場合は常に `null` ではない
- `error` (`hasError` でチエックできる)
  `AsyncError` の場合は常に `null` ではなく、 `AsyncData` の場合は常に `null`
- `isLoading`
  `AsyncLoading` の場合は常に `true`

これらを組み合わせて前頁の図のような様々な状態を表現している。
`AsyncData` でも `isLoading` が `true` 、 `AsyncError` でも `isLoading` が `true` などが存在し得る。

An error occurred on this slide. Check the terminal for more information.

プロバイダーの種類：FutureProvider (AsyncValueについて)

`FutureProvider`を初期化すると前回の値を保持しつつも`isLoading`を`true`とすることで非同期処理中であることを表す。
これは、前回取得できていた値や発生していたエラーを表示しつつ、新しい値を取得中であることを表すためである。

`Widget`側で受け取った`AsyncValue`を先述の6状態で分岐させて書くには以下のように記述する。

switch式による分岐の例：毎回全て書く必要はなく、適宜省略して書く

```
Widget build(BuildContext context, WidgetRef ref) {
  final asyncVal = ref.watch(sampleProvider);
  return switch(asyncVal) {
    AsyncData<String>(:final value, isLoading: false) => Text(value),
    AsyncData<String>(:final value, isLoading: true) => Text('has $value, but isLoading'),
    AsyncError<String>(:final error, isLoading: false) => Text('$error'),
    AsyncError<String>(:final error, isLoading: true, hasValue: false) => Text('has $error, but isLoading'),
    AsyncError<String>(:final error, hasValue: true, :final value?, isLoading: true) => Text('has $error & $value, but isLoading'),
    _ => const CircularProgressIndicator(), // default
  };
}
```

dartのswitch式についてはこちらを参照する。多少癖がある書き方なので、慣れるまで時間がかかるかもしれない。
`AsyncValue`を扱う上では頻繁に使用するものなので習得できることが望ましい。

プロバイダーの種類：StreamProvider

基本的なProviderのStream版
Futureと同じで非同期な値を提供するため、AsyncValueとして値を提供するProvider

しかし、これまでのProvider・FutureProvider・StreamProviderだけでは提供する値に後から変更を加えることが難しい
そこで使用するのがNotifierProviderである

プロバイダーの種類：AsyncNotifierProvider

非同期処理における状態を管理するための `AsyncValue` を提供する `NotifierProvider`
`FutureProvider` と違い、取得後に任意のメソッドで変更を加えることができる

```
final sampleProvider = AsyncNotifierProvider<SampleNotifier, int>(
  SampleNotifier.new,
);
class SampleNotifier extends AsyncNotifier<int> {
  @override
  Future<int> build() async {
    await Future.delayed(const Duration(seconds: 1)); // 非同期処理を待ってから初期値を返す
    return 0;
  }
  void increment() {
    state = state.whenData((data) => data + 1)
  }
}
// widget側
return switch (ref.watch(sampleProvider)) {
  AsyncData<int>(:final value) => Text('$value'),
  AsyncError<int>(:final error) => Text('$error'),
  _ => CircularProgressIndicator(),
};
ref.read(sampleProvider.notifier).increment();
```

プロバイダーの種類：AsyncNotifierProvider

`state`を自由に変更できてしまうため、変更処理には気を付けないといけない。
例えば、`FutureProvider`では初回の非同期処理中にだけ`AsyncLoading`になっていたが、
`state`を直接`AsyncLoading`にすることで、任意のタイミングで`AsyncLoading`になり得るというように注意が必要になってくる。

`AsyncValue`を後から変更するための便利なメソッドがいくつか用意されており、それらを駆使して実装することが望ましい。

guard: 発生したエラーをAsyncErrorに変換、エラーがなければAsyncDataに変換

```
state = await AsyncValue.guard<String>(() async {
  await Future<dynamic>.delayed(const Duration(seconds: 1));
  return 'Hello World';
});
```

whenData: stateがAsyncDataの時だけdataに変更を加える

```
state = state.whenData((data) {
  final newString = '$data!';
  return newString;
});
```

copyWithPrevious: 前回の状態を保持して更新

```
// 再取得前にローディングにする
state = const AsyncLoading<String>().copyWithPrevious(state);
// 条件を変えて取得する前にローディングにする
state = const AsyncLoading<String>().copyWithPrevious(
  state,
  isRefresh: false,
);
```

unwrapPrevious: 前回までの状態を消す

```
state = state.unwrapPrevious();
```

プロバイダーの種類：AsyncNotifierProvider (AsyncValueについて)

余談になるが、内部的に `AsyncValue` を使用しているプロバイダー(`FutureProvider`・`AsyncNotifierProvider` など)については、 `invalidate()` によって自動で `isLoading` を `true` にしてくれるなど行ってくれるため、極力直接 `AsyncValue` を使用しないようにする。

推奨：AsyncNotifierProviderの例

```dart
final sampleProvider = AsyncNotifierProvider<
  SampleNotifier,
  int
>(SampleNotifier.new);

class SampleNotifier extends AsyncNotifier<int> {
  @override
  Future<int> build() async {
    await Future.delayed(const Duration(seconds: 1));
    return 0;
  }
}
```

非推奨：NotifierProviderでAsyncValueを直接使用する例

```dart
final sampleProvider = NotifierProvider<
  SampleNotifier,
  AsyncValue<int>
>(SampleNotifier.new);

class SampleNotifier extends Notifier<AsyncValue<int>> {
  @override
  AsyncValue<int> build() async {
    return AsyncData(0);
  }
}
```

プロバイダーの種類：StreamNotifierProvider

Stream処理における状態を管理するための `AsyncValue` を提供する `NotifierProvider`
`StreamProvider` の `NotifierProvider` 版

```
final sampleProvider = StreamNotifierProvider<SampleNotifier, int>(SampleNotifier.new);

class SampleNotifier extends StreamNotifier<int> {
  @override
  Stream<int> build() async* {
    final streamList = Stream<int>.fromIterable(List<int>.generate(10, (i) => i + 1));
    await for (final val in streamList) {
      await Future.delayed(const Duration(seconds: 1));
      yield val;
    }
  }
}
// widget側
return switch (ref.watch(sampleProvider)) {
  AsyncData<int>(:final value) => Text('$value'),
  AsyncError<int>(:final error) => Text('$error'),
  _ => CircularProgressIndicator(),
};
```

プロバイダー全般の注意点: 定義場所

Providerは必ずグローバルに定義するように注意する。
ローカルに定義するとRiverpodの公式ドキュメントでも指摘されているように、メモリリークなどの原因になる。

x ローカルに定義した例

```dart
class SampleWidget extends ConsumerWidget {
  const SampleWidget({super.key});

  Widget build(BuildContext context, WidgetRef ref) {
    final sampleProvider = Provider<int>((ref) => 0);
    return Text(ref.watch(sampleProvider));
  }
}
```

o グローバルに定義した例

```dart
final sampleProvider = Provider<int>((ref) => 0);

class SampleWidget extends ConsumerWidget {
  const SampleWidget({super.key});

  Widget build(BuildContext context, WidgetRef ref) {
    return Text(ref.watch(sampleProvider));
  }
}
```

プロバイダー全般の注意点: 定義場所

プロバイダーがグローバルに定義されることによって、依存関係が煩雑になりがちな問題への対処法のひとつを紹介する。
それが、`mixin`を使用することである。

どのプロバイダーがどこの`Widget`からアクセスされているかといった依存関係を整理するために
`mixin`を使用して「このクラスにこのプロバイダーが関係している」といったことを、コード上に記述することができるようになる。

記述するかどうかは実装者に委ねられるので明確なものではないが、
プロジェクトで使用を強制してみるかどうかの検討を行う価値はあると思われる。

```
mixin class SampleState {
  int sampleValue(WidgetRef ref) => ref.watch(sampleProvider);
}

class SampleWidget extends ConsumerWidget with SampleState {
  const SampleWidget({super.key});

  @override
  Widget build(BuildContext context, WidgetRef ref) {
    return Text('${sampleValue(ref)}');
  }
}
```

プロバイダー全般の注意点: 参照の更新

`state` がクラスのような参照を表すものの場合、参照が変更されなければ値が更新されたとして判定されないことに注意する。

x 間違った例

```
class SampleState {
  SampleState({required this.x});
  int x;
}

final sampleProvider = NotifierProvider<
  SampleStateNotifier,
  SampleState
>(SampleStateNotifier.new);

class SampleStateNotifier extends Notifier<SampleState> {
  @override
  SampleState build() {
    return SampleState(x: 0);
  }

  void increment() {
    state.x += 1; // 参照が更新されないため、watchしても更新されない
  }
}
```

o 正しい例

```
class SampleState {
  const SampleState({required this.x});
  final int x;
}

final sampleProvider = NotifierProvider<
  SampleStateNotifier,
  SampleState
>(SampleStateNotifier.new);

class SampleStateNotifier extends Notifier<SampleState> {
  @override
  SampleState build() {
    return SampleState(x: 0);
  }

  void increment() {
    state = SampleState(x: state.x + 1);
  }
}
```

プロバイダー全般の注意点: 参照の更新

この参照の更新問題のために、`copyWith` メソッドをクラスに定義しておくことがよくある。
(freezedというライブラリを使用するとcopyWithを含む便利なメソッドが自動生成できる。)

```
@immutable
class SampleState {
  const SampleState({required this.x, required this.y});
  final int x;
  final int y;
  SampleState copyWith({int? x, int? y}) {
    return SampleState(x: x ?? this.x, y: y ?? this.y);
  }
}

final sampleProvider = NotifierProvider<SampleStateNotifier, SampleState>(SampleStateNotifier.new);
class SampleStateNotifier extends Notifier<SampleState> {
  @override
  SampleState build() {
    return SampleState(x: 0, y: 0);
  }
  void increment() {
    state = state.copyWith(x: state.x + 1); // 参照が更新されるため、UIも更新される
  }
}
```

各プロバイダーに付与できる修飾子について

プロバイダーには `autoDispose` と `family` という修飾子を付与できる。

- `autoDispose`
  `ref.watch` するものがいなくなると保持している状態を破棄し、メモリを解放する
- `family`
  外部のパラメータで別々のプロバイダーを作成できるようになる (例：IDによって別のプロバイダーとして使用するなど)

autoDispose例

```
final sample1Provider = Provider.autoDispose<int>(
  (ref) => 10,
);

final sample2Provider = NotifierProvider.autoDispose<
  Sample2Notifier,
  int
>(Sample2Notifier.new);

class Sample2Notifier extends AutoDisposeNotifier<int> {
  @override
  int build() => 0;
}
```

family例

```
final sample3Provider = Provider.family<int, String>(
  (ref, arg) => 10, // arg(String)によって別の値を返すなど
);

final sample4Provider = NotifierProvider.family<
  Sample4Notifier,
  int,
  String
>(Sample4Notifier.new);

class Sample4Notifier extends FamilyNotifier<int, String> {
  @override
  int build(String arg) => 0; // argによって別の値を返すなど
}
```

各プロバイダーに付与できる修飾子について: autoDispose

プロバイダーはグローバルに定義するため、普通に定義すると常にメモリを使い続ける。
そのため、基本的にプロバイダーには `autoDispose` を付与することが推奨される。

逆に `autoDispose` を付与したプロバイダーは `ref.watch` しているものがいなくなったときに値を破棄してしまう。
`Widget` 内の必要なところで `ref.watch` することはもちろんのこと、条件分岐で `ref.watch` が該当しなくなっても監視していないと判定されて破棄されることに注意する。

sampleProviderが破棄される可能性があるコード

```
class SampleWidget extends ConsumerWidget {
  const SampleWidget({super.key, required this.isWatch});
  final bool isWatch;

  @override
  Widget build(BuildContext context, WidgetRef ref) {
    if (isWatch) {
      return Text(ref.watch(sampleProvider));
    } else {
      return Text('sample');
    }
  }
}
```

sampleProviderが破棄されないように書いた例

```
class SampleWidget extends ConsumerWidget {
  const SampleWidget({super.key, required this.isWatch});
  final bool isWatch;

  @override
  Widget build(BuildContext context, WidgetRef ref) {
    final val = ref.watch(sampleProvider);
    if (isWatch) {
      return Text(val.toString());
    } else {
      return Text('sample');
    }
  }
}
```

各プロバイダーに付与できる修飾子について: autoDispose(onDispose)

`autoDispose` を付与すると自動でメモリを解放するが、このようなプロバイダーが解放される時にメソッドを呼び出すことができる。
初期化時に `ref.onDispose` を使用して記述する。

Provider.autoDispose

```dart
final sampleProvider = Provider.autoDispose<int>(
  (ref) {
    ref.onDispose(() {
      print('dispose!');
    });
    return 10;
  }
);
```

NotifierProvider.autoDispose

```dart
class SampleNotifier extends AutoDisposeNotifier<int> {
  @override
  int build() {
    ref.onDispose(() {
      print('dispose!');
    });
    return 10;
  }
}
```

(例: 通信のキャンセルやCloseなどを行うことが多い。TextEditingControllerなどのController系のdisposeには使用せず、後述するFlutterHooksを使用すること。理由も後述している)

各プロバイダーに付与できる修飾子について: family

familyを使用することで、外部のパラメータで別々のプロバイダーを作成できるようになる。
(例：IDによって別の `Provider` として使用するなど)

読み取るときは `ref.watch(sampleProvider(id))` のように、パラメータを渡す。

family例

```dart
final sampleProvider = FutureProvider.family<int, String>(
  (ref, id) async {
    final res = await dio.get('sample_api/$id'); // idによって別の値を返すなど
    return res.data.toString();
  }
);

class SampleWidget extends ConsumerWidget {
  const SampleWidget({super.key, required this.id});
  final String id;

  @override
  Widget build(BuildContext context, WidgetRef ref) {
    return Text(ref.watch(sampleProvider(id))); // `ref.watch(sampleProvider)`としないように注意する
  }
}
```

各プロバイダーに付与できる修飾子について: family

familyに渡すことができるパラメータはひとつだけなので、複数渡したいときはクラスとして定義して渡すか、recordを使用する。
名前付きでないパラメータも渡せるが、名前付きで渡したほうが可読性が高まる。

Recordを使用する例

```
final sampleProvider = Provider.family<String, ({String id, String name})>((ref, ({String id, String name}) arg) {
  return 'sample_${arg.id}_${arg.name}';
});

final sampleNotifierProvider = NotifierProvider.family<SampleNotifier, String, ({String id, String name})>(SampleNotifier.new);
class SampleNotifier extends FamilyNotifier<String, ({String id, String name})> {
  @override
  String build(({String id, String name}) arg) {
    return 'sample_${arg.id}_${arg.name}';
  }
}

// widget
Text(ref.watch(sampleProvider((id: '1', name: '2'))))
Text(ref.watch(sampleNotifierProvider((id: '1', name: '2'))))
```

Recordについては公式を参照する。

各プロバイダーに付与できる修飾子について: autoDisposeとfamilyの併用

familyとautoDisposeは併用することが可能。
IDによって異なるプロバイダーを作成し、`ref.watch`しているものがいなくなると保持している状態を破棄する、といったことができる。

FutureProviderの例

```
final sampleProvider = FutureProvider.autoDispose.family<int, String>( // 連結して記述するだけ
  (ref, id) async {
    return await sampleCalc(id);
  }
);
```

NotifierProviderの例

```
final sampleProvider = NotifierProvider.autoDispose.family<
  SampleNotifier,
  int,
  String
>(SampleNotifier.new);

class SampleNotifier extends AutoDisposeFamilyNotifier<int, String> { // 親クラス名はAutoDisposeが先になっている
  @override
  int build(String arg) => 0;
}
```

refの使い方

プロバイダーにアクセスするための `ref` の使い方で、これまでread・watch・invalidateだけを紹介してきた。
ほとんどの場合でそれだけで困ることはないと思われるが、他にもあるのでここで一覧で示しておく。

- ref.read: プロバイダーの値を取得する
- ref.watch: プロバイダーの提供する値を監視して、変更があれば再構築する
- ref.invalidate: プロバイダーを再初期化する
- ref.invalidateSelf: 自分自身のプロバイダーをinvalidateする
- ref.listen: プロバイダーの提供する値を監視して、変更があれば任意の関数を実行する
- ref.listenSelf: 自分自身のプロバイダーの提供する値を監視して、変更があれば任意の関数を実行する

```
ref.read(sampleProvider); // or ref.read(sampleProvider.notifier).method();
ref.watch(sampleProvider);
ref.invalidate(sampleProvider);
ref.invalidateSelf(); // プロバイダー内で記述する
ref.listen(sampleProvider, (previous, next) {
  print('previous: $previous, next: $next');
});
ref.listenSelf((previous, next) { // プロバイダー内で記述する
  print('previous: $previous, next: $next');
});
```

ref.listenManualとref.refreshは非推奨

Flutter Hooks

FlutterHooksによる状態管理

これまで紹介したRiverpodのプロバイダーではグローバルに定義されているため、基本的に `Widget` 全体で共有される。
FlutterHooksでは特定の `Widget` 内部でのみで管理できる変数の定義を行うことができる。
`StatelessWidget` ではなく、`HookWidget` を継承することで使用できるようになる。

Hooksの使用例

```
class SampleWidget extends HookWidget {
  const SampleWidget({super.key});
  Widget build(BuildContext context) {
    final isEnabledField = useState<bool>(false);
    return Column(children: [
      TextButton(
        child: Text('toggle'),
        onPressed: () {
          isEnabledField.value = !isEnabledField.value // 更新
        },
      ),
      TextField(
        enabled: isEnabledField.value, // 参照
      ),
    ])
  }
}
```

refと一緒に使うこともでき、HookConsumerWidgetを継承する

```
class SampleWidget extends HookConsumerWidget {
  const SampleWidget({super.key});

  Widget build(BuildContext context, WidgetRef ref) {
    ref.watch(sampleProvider);
    final isSample = useState<bool>(true);
    return Text(isSample.value ? 'sample' : 'foobar');
  }
}
```

FlutterHooksによるController管理

`TextEditingController` などのFlutterで出てくるController系は
`StatefulWidget` で使用することを前提としているため、`StatelessWidget` では基本的に使用できない。
そこで、`HookWidget` を使用することでController系を `StatefulWidget` なしで定義・使用できるようにする。

また、**Controller系は `Widget` と同じライフサイクルであるべきであるため、`HookWidget` を使用することが強く推奨される。**
Controller系をプロバイダーで管理して `onDispose` でControllerを `dispose` するような実装だと、一見同じライフサイクルに思えるが、異なるライフサイクルで管理されていることに注意する。
加えてRiverpodは複数の `Widget` で共有する状態を管理することに適しているため、1画面で確実に完結するController系はRiverpodで管理しないこと。

```
class SampleWidget extends HookWidget {
  const SampleWidget({super.key});

  Widget build(BuildContext context) {
    final editingController = useTextEditingController();
    return TextField(
      controller: editingController,
    );
  }
}
```

FlutterHooksでuse**Controllerを自作する

他パッケージなどで定義されているController系についても自分で `use` を定義して使用することができる。

```
SampleController useSampleController() {
  return use(const _SampleController());
}

class _SampleController extends Hook<SampleController> {
  const _SampleController();
  @override
  _SampleController createState() => _SampleControllerState();
}
class _SampleControllerState extends HookState<SampleController, _SampleController> {
  final SampleController _controller = SampleController();
  @override
  SampleController build(BuildContext context) {
    return _controller;
  }
  @override
  void dispose() {
    super.dispose();
  }
}
```

パフォーマンス考慮

constによる再描画判定の阻止

`Widget` のコンストラクタでLinterに `const` を付けるように注意されている場合、`const` をつけておく。
`Widget` インスタンスが使いまわされることで、ある程度のパフォーマンス改善が期待できるため必ず実施する。
Linterによって注意される部分なので、有効活用する。

```
class SampleWidget extends StatelessWidget {
  const SampleWidget({super.key});

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: const Text('Sample Title'),
      ),
      body: const Center(
        child: Text('sample text'),
      ),
    );
  }
}
```

(※その他にもLinterによって注意されている部分については、可能な限り対応しておくことを強く推奨する)

Riverpodのパフォーマンス向上: refの範囲を絞る

小さな範囲で `ref` を使うことで、再構築を行う範囲を狭めることができる。
例えば、`ConsumerWidget` を継承するclassを分割して定義することで範囲を絞ることができる。
しかし、classを細かく定義しすぎると可読性が下がるため、`Consumer` を使用することも検討する。

ConsumerWidget継承例

```
class Sample1Widget extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Column(
      children: [
        const Text('sample'),
        const Sample2Widget(),
      ],
    );
  }
}
class Sample2Widget extends ConsumerWidget {
  const SampleWidget({super.key});
  @override
  Widget build(BuildContext context, WidgetRef ref) {
    return Text(ref.watch(sampleProvider));
  }
}
```

Consumer使用例

```
class SampleWidget extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Column(
      children: [
        Consumer(
          builder: (context, ref, child) {
            return Text(ref.watch(sampleProvider));
          },
        ),
        const Text('sample'),
      ],
    );
  }
}
```

Riverpodのパフォーマンス向上: selectを使用する

`select` を使用して必要な値だけを監視することで、必要最低限の時だけ再構築するようにする。
以下コード例では `x` だけを監視しており、`SampleState` の参照が変わるだけでは更新されないようになる。

```
class SampleState {
  const SampleState({
    required this.x,
    required this.y,
  });
  final int x;
  final int y;
}
// sampleProviderはSampleStateを提供するProvider
// 詳細は省略する
class SampleWidget extends ConsumerWidget {
  const SampleWidget({super.key});
  @override
  Widget build(BuildContext context, WidgetRef ref) {
    return Text(
      ref.watch(sampleProvider.select((val) => val.x.toString())),
    );
  }
}
```

Riverpodのパフォーマンス向上: Providerを挟む

考え方は `select` と同じで、UI側で必要とする部分だけを監視するために `Provider` を挟む。
これは `select` と違って、複雑な計算が必要になる場合などで使用することが多い。

`select` だと `Widget` 内で記述することになるため、ロジックを書かなければならない場合にはこの方法が適している。

```
final calcResultProvider = Provider<String>((ref) {
  var tmp = ref.watch(sampleProvider);
  // ここで複雑な計算やフィルタ処理などを行う
  return tmp.x.toString();
});

class SampleWidget extends ConsumerWidget {
  const SampleWidget({super.key});

  @override
  Widget build(BuildContext context, WidgetRef ref) {
    return Text(ref.watch(calcResultProvider));
  }
}
```

Riverpodのパフォーマンス向上: selectAsyncを使用する

`select` では `AsyncValue` を監視するときに記述量が多くなり面倒である。
そこで、`selectAsync` を使用することで `Future` を取得することができるのでこれも活用する。

```dart
final sampleProvider = FutureProvider<int>((ref) async {
  await Future.delayed(const Duration(seconds: 1));
  return 10;
});

final sample2Provider = FutureProvider<String>((ref) async {
  final sample = await ref.watch(sampleProvider.selectAsync((val) => val));
  return (sample * 2).toString();
});

class SampleWidget extends ConsumerWidget {
  const SampleWidget({super.key});

  @override
  Widget build(BuildContext context, WidgetRef ref) {
    return Text(ref.watch(sample2Provider));
  }
}
```

FlutterHooksのパフォーマンス向上

考え方はプロバイダーの `select` などと同様で、再描画判定の範囲を狭めるために可能な限り小さな範囲で `use**` を使用する。
細かくしすぎると読みづらいコードになるため注意すること。

```
class SampleHookWidget extends HookWidget {
  const SampleHookWidget({super.key});

  @override
  Widget build(BuildContext context) {
    final isEnabled = useState<bool>(false);
    final controller = useTextEditingController();
    return TextField(
      enabled: isEnabled.value,
      controller: controller,
    );
  }
}
```

画面遷移(Navigation)

An error occurred on this slide. Check the terminal for more information.

Navigator 2.0

Navigator 2.0は、Navigator 1.0の改良版です。

- ルート（Route）：画面を表す
- スタック：画面の履歴を管理
- プッシュ（Push）：新しい画面をスタックに追加
- ポップ（Pop）：現在の画面をスタックから削除

以下は簡単な例です：

```
class MyRouterDelegate extends RouterDelegate<List<Page<dynamic>>> {
  @override
  Widget build(BuildContext context) {
    return Navigator(
      pages: [Page(child: HomePage())],
      onPopPage: (route, result) ⇒ route.didPop(result),
    );
  }

  @override
  Future<void> setNewRoutePath(List<Page<dynamic>> path) async {
    // 新しいルートを設定
  }
```

Flutterにおけるテスト

依存性注入でテストなどを実施する

プロバイダーを使って依存性注入できる。

Repositoryを注入するコード例

```
final repositoryProvider = Provider<Repository>(
  (ref) => UnimplementedError(), // 実装を与えないようにしておく
);
abstract class Repository {
  Future<int> fetch();
}

final sampleProvider = AsyncNotifierProvider<SampleStateNotifier, int>(SampleStateNotifier.new);
class SampleStateNotifier extends AsyncNotifier<int> {
  late Repository _repository;
  @override
  Future<int> build() async {
    _repository = ref.watch(repositoryProvider); // 参照はref.watchで行う
    return _repository.fetch();
  }
  void increment() {
    state = state.whenData((data) => data + 1);
  }
}
```

ProviderScopeで局所的にProviderをoverrideする

`ProviderScope` を用いることで特定の `Widget` 内でのみ使用するプロバイダーを定義することができる。
`family` のためのIDなどを毎回書く必要をなくしたり、外側と内側で別々のプロバイダーを使用したり、テスト用にoverrideしたりなど様々な使用方法がある。

```
ProviderScope(
  overrides: [
    sampleProvider.overrideWith(
      (ref) ⇒ ref.watch(sampleFamilyProvider('10')),
    ),
  ],
  child: const SampleWidget(),
)
```

`runApp` 直下の `ProviderScope` でRepositoryやServiceクラスなどをoverrideして適当なモッククラスを渡せば
UI側(`Widget` とプロバイダー)だけで実行できるようになる。

```
ProviderScope(
  overrides: [
    repositoryProvider.overrideWith((ref) ⇒ MockRepository()),
  ],
  child: const MyApp(),
)
```

困った時は

公式サイト・検索・AIChatでWeb検索をかけて聞くなどする。

AIChatでは基本的に古い情報が扱われるため、新しい情報が手に入らないことが多い。
そのため、誤情報・非推奨の情報などが出てくることに注意する。

Flutter・Riverpod・FlutterHooks

おまけ

以降は開発において必須ではない発展的な内容だが、理解を深めるための一助になれば

描画の仕組み

An error occurred on this slide. Check the terminal for more information.

具体的な描画の仕組み

前述した通り `Widget` は所詮設定を持つ程度であるため、実は `Widget` そのものの再構築はたいしたコストにならない。
一番コストになるのは `Element` の再構築である。

これは `Element` が実際に状態を管理していたり、`Widget` との関係も管理していたり、`RenderObject` との関係も管理していたりと、様々な処理を行っているためである。

つまり、パフォーマンスの最適化のためには `Element` の再構築を最小限に抑えることが重要となる。

以下の更新時の挙動を念頭に置いて、次頁でパフォーマンス改善方法の意味を考える。

1. `Element` は `StateObject` に変化があるとdirtyとしてマークする
2. マークされた箇所は次フレームで `Widget` の `build` を呼び出し、`Widget` ツリーを再構築する
3. 最新の `Widget` ツリーのそれぞれの `Widget` で、**位置・runtimeType・keyを比較**して、同一のものかどうかを判断する
4. 同一であれば対応する `Element` は使いまわし、異なれば新しい `Element` を生成する
   (`Element` が新しくなったとき、`RenderObject` も新しく生成される)

具体的な描画の仕組み: パフォーマンスについて考える

Elementツリーの再構築を抑えるためには、以下のことを行う。

- そもそもdirtyとしてマークされないようにする
  Consumerやクラス分割で `ref` の範囲を狭めることで、必要最低限のdirtyマークで済む
  selectを使って監視する値を絞り、dirtyマークされる条件を減らす
- マークされたときに、比較対象を減らす
  constをつけるとWidgetの比較がそれより下のWidgetでは行われなくなる
- 頻繁な `Widget` ツリーの再構築を減らす
  ifによる条件分岐でWidgetを構築すると、条件に当てはまるときと当てはまらないときで `Element` も再生成される
  これを防ぐために `Visibility` などを使用してツリー構造の変更を減らす
  非表示な状態がほとんどの場合は、 `Visibility` を使用するよりも `if` を使用して、そもそも `Element` を構築しないほうが良いこともある
- `Key` を適切に使用して、 `Widget` の同一性を明確にする
  `Key` が違えば別の `Widget` として扱われて `Element` を再構築する必要が出てくるため、基本は `Key` をつけない

`Element` の再構築を抑えるという視点で実装すると、他にも方法があるかもしれない。
より良いパフォーマンスのためには `Element` (と `RenderObject`)を意識してみると良い。

Macroについて

Dart言語のMacro

Dart言語では、Macroが2025年初頭に正式に導入されると予告されている。(執筆は2024/09)
マクロというのは、Dart言語においては以下のようになるとされている。

アノテーションをつけることで、toJsonとfromJsonが使用できるようになる例

```
@JsonCodable
class Example {
  final String name;
  final int age;
}

void main() {
  final tmp = Example.fromJson({'name': '名前', 'age': 10});
  final json = tmp.toJson();
}
```

Macroとはこのようにアノテーションによるコード自動生成機能である。

これと同じ機能は現在`build_runner`というパッケージを使用し、事前にコマンドを実行してコードを生成することで実現している。

```
dart run build_runner build
```

Dart言語のMacro

実はRiverpodにも `build_runner` による生成機能をriverpod_generatorによって使用できる。

にも関わらずgeneratorバージョンを紹介しなかった理由は2つある。

- コード生成を使用するとほとんどが隠蔽されてしまい、具体的にどうなっているかが分かりづらくなるためチュートリアルに適していない
- build_runnerは不評で、Macro版になってから使えば良いと考えているから

2点目について、Riverpodは公式もコード生成について以下のように言及している。

> 現在、コード生成はがオプションなのはbuild_runnerが多くの人に好まれないためです。
> しかし、Static Metaprogrammingが Dart で利用可能になると、build_runnerはもはや問題ではありません。
> その時点で、コード生成を使用することが Riverpod で唯一の方法になるでしょう。

Dart言語のMacroは既に実験的に導入されており、気になる人は触れてみても良いだろう。

プロバイダーを使用したアーキテクチャ考察

An error occurred on this slide. Check the terminal for more information.