Riverpod

什么是RiverPod

它是由推出Provider框架的作者推出的一款新的反应式缓存框架。它通过使用声明式和反应式编程，riverpod能够为应用程序处理大部分的逻辑，它可以执行带有内置错误处理和缓存的网络请求，并在必要时自动重新获取数据。

为什么使用RiverPod

RiverPod最明显的特点是外部不依赖BuildContext，所有提供者都是全局声明的，可以在任何地方访问，这意味着我们可以将应用程序的状态和业务逻辑通过这种方式保存起来，而不必放在小部件树内。

使用方法

1. 引入依赖

```
dependencies: flutter: sdk: flutter flutter_riverpod: ^2.4.9 riverpod_annotation: ^2.3.3
```

应用类型	包名称	描述
仅限Flutter	flutter_riverpod	使用Riverpod的基本方法与Flutter结合使用。
Flutter + flutter_hooks	hooks_riverpod	同时使用flutter_hooks和Riverpod的方式。
仅限Dart（无Flutter）	riverpod	去除与Flutter相关的所有类的Riverpod版本。

2. 初步使用
    1. 嵌套一个顶级的***ProviderScope***
    2. 创建一个StateProvider
    3. 使用 ***ref*** 对state进行读取和修改

```dart
void main() {
  runApp(const ProviderScope(child: MaterialApp(home: MyApp())));
}

final counterProvider = StateProvider((ref) => 0);

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

  @override
  Widget build(BuildContext context, WidgetRef ref) {
    return Scaffold(
      appBar: AppBar(...),  // AppBar
      body: Center(
        child: Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: <Widget>[
            const Text(...),  // Text
            Consumer(builder: (context, ref, _) {
              return Text(
                '${ref.watch(counterProvider)}',
                style: Theme.of(context).textTheme.headlineMedium,
              );  // Text
            }),  // Consumer
          ],  // <Widget>[]
        ),  // Column
      ),  // Center
      floatingActionButton: FloatingActionButton(
        onPressed: () => ref.read(counterProvider.notifier).state++,
        tooltip: 'Increment',
        child: const Icon(Icons.add),
      ),  // FloatingActionButton
    );  // Scaffold
  }
}
```

这个demo是对flutter的初始项目使用riverpod进行重新编写，可以看到它明显减少了代码量。
注意：在riverPod 3.0.0以前onPressed的版本还可以使用 `ref.read(provider.state).state` 去对状态进行修改，但是在3.0.0以后，这个方法会被废弃掉。

provider类型

1. Provider
   - 将一个值暴露给其他Provider
   - 仅返回一个value

- `final helloWorldProvider = Provider<String>((ref) => 'hello world');`

2. StateProvider
   - 是一个公开了状态的Provider
   - 所持有的状态通常是int，bool，String，emun
   - 适用于存储可以发生变化的简单状态对象，比如计数器的值之类的。
   - `final countProvider = StateProvider((ref) => 0);`

3. StateNotifierProvider
   - 具体功能和StateProvider类似，但适用于复杂的，具有复杂逻辑的场景
   - 需要使用StateNotifier类来管理状态和处理状态变化
   - 提供的状态是只读的，无法直接修改，需要调用notifier上的方法。
   -
```dart
class CounterNotifier extends StateNotifier<int> {
  CounterNotifier() : super(0);
  void increment() => state = state + 1;
  void decrement() => state = state - 1;
}
final counterProvider =
    StateNotifierProvider<CounterNotifier, int>((ref) => CounterNotifier());
class MyApp extends ConsumerWidget {
  const MyApp({super.key});
  @override
  Widget build(BuildContext context, WidgetRef ref) {
    final counterValue = ref.watch(counterProvider);
    return Scaffold(
      appBar: AppBar(
        backgroundColor: Theme.of(context).colorScheme.inversePrimary,
        title: const Text('test'),
      ),  // AppBar
      body: Center(
        child: Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: <Widget>[
            const Text(...),  // Text
            Consumer(builder: (context, ref, _) {
              return Text(
                '$counterValue',
                style: Theme.of(context).textTheme.headlineMedium,
              );  // Text
            }),  // Consumer
          ],  // <Widget>[]
        ),  // Column
      ),  // Center
      floatingActionButton: FloatingActionButton(
        onPressed: () => ref.read(counterProvider.notifier).increment(),
        tooltip: 'Increment',
        child: const Icon(Icons.add)
```

4. FutureProvider
   - 通过feature可知，可以拿来获取异步数据
   - 一次性获取
   - 常用来执行和缓存异步操作，处理异步操作的错误和加载状态，将多个异步值组合成另一个值，重新获取和刷新数据等。

```dart
class Weather {
  Weather({required this.isRain});

  bool isRain;
}

class WeatherRepository {
  Future<Weather> getWeather(String city) async {
    return Weather(isRain: true);
  }
}

final weatherRepositoryProvider = Provider((ref) => WeatherRepository());
final weatherFutureProvider = FutureProvider((ref) {
  final repository = ref.read(weatherRepositoryProvider);
  return repository.getWeather("city");
});
```

   - 然后在build方法中观察，并将生成的***AsyncValue***（数据，加载，错误）映射到用户界面

```dart
Widget _widget(BuildContext context, WidgetRef ref) {
  final weatherAsync = ref.watch(weatherFutureProvider);
  return weatherAsync.when(
    data: (weather) => Text("isRain:$weather"),
    error: (err, stack) => Text('Error: $err'),
    loading: () => const CircularProgressIndicator(),
  );
}
```

5. StreamProvider
   - 与FutureProvider类似，都是用来处理异步数据
   - 但最大区别在于FutureProvider是一次性获取，而StreamProvider是实时更新

```dart
Stream<int> countStream() async* {
  int count = 0;
  while (true) {
    await Future.delayed(const Duration(seconds: 1));
    yield count++;
  }
}

final streamProvider = StreamProvider((ref) => countStream());

class MyApp extends ConsumerWidget {
  const MyApp({super.key});
  @override
  Widget build(BuildContext context, WidgetRef ref) {
    final AsyncValue<int> streamData = ref.watch(streamProvider);
    return Scaffold(
      appBar: AppBar(
        backgroundColor: Theme.of(context).colorScheme.inversePrimary,
        title: const Text('test'),
      ),  // AppBar
      body: streamData.when(
        data: (count) => Text(
          'Count: $count',
          style: const TextStyle(fontSize: 24),
        ),  // Text
        error: (error, stackTrace) => Text('Error: $error'),
        loading: () => const CircularProgressIndicator(),
      ),
```

6. ChangeNotifierProvider
   - 主要是用于从Provider迁移到RiverPod使用

ref

1. ref是什么
   > 在riverpod中， `ref` 是“reference“的缩写，代表一个状态提供者(provider)的引用，我们可以通过这个ref获取到任何提供者，因为所有的provider都是全局的。

2. 怎么拿到ref
   - StatelessWidget替换成了ConsumerWidget
   - StatefulWidget 和 State 替换为 ConsumerStatefulWidget 和 ConsumerState
   - Consumer可以直接在widget tree中使用

   在这些widget中我们可以直接拿到ref

通过ref获取状态

1. ref.watch
   - 获取一个provider的值进行监听，当值发生变化时，将重建订阅该值的Widget或者Provider
   - 可在Widget的构建方法中使用，或者在Provider中去监听另外的Provider
   - 应避免在异步方法中使用该方法，可能会导致不必要的重建，影响性能

2. ref.read
   - 获取一个provider的值，但是忽略它的变化，不进行监听，通常使用在与用户交互中使用
   - 但由于它不是响应式的，因此应尽可能避免适应ref.read，除非ref.watch或者ref.listen会出现一些问题
   - 不应在build方法中使用
   - 官方建议ref.read应只被用在不能使用ref.watch的地方

3. ref.listen
   - 在provider上添加一个监听器，以执行一个action，比如导航到一个新的页面或在该rovider发生变化时执行一些操作
   - 与watch类似，但是如果监听的provider发生变化，不会重建widget或者provider，而是会调用传入的自定义函数

使用notifier进行值的修改

使用ref.read和ref.watch都可以实现，比如上面对count的值进行修改(StateProvider)，我们可以这样
- `ref.read(counterProvider.notifier).update((state) => state + 1)`
- `ref.read(counterProvider.notifier).state++`
- `StateController<int> count = ref.watch(counterProvider.notifier);` 然后在onPress中 `count.state++` 就可以了

在使用中发现一个奇怪的现象，在update方法中如果写成 `state++` 就不能更新状态。

provider的扩展方法

1. select
   可在ref.read和ref.watch中使用，监听某一个值，从而减少build的次数
   比如有一个user类，我们只想监听其中的某一个字段的变化，那么我们就可以使用select

```
Widget _test(BuildContext context, WidgetRef ref) {
  String name = ref.watch(userProvider.select((user) => user.name));
  return Text(name);
}

final userProvider = StateProvider((ref) => User());
```

   这样只有当name变化时才会去重新build
2. 组合provider
   在provide中，我们可以通过watch对另一个provider进行操作

```dart
final passwordProvider = StateProvider((ref) => '');

Widget _test(BuildContext context, WidgetRef ref) {
  int age = ref.watch(userProvider.select((user) {
    if (ref.read(passwordProvider.notifier).state != '') {
      return user.age;
    } else {
      return 0;
    }
  }));
  return Text('$age');
}
```

3. family
   传入一个额外的参数来创建state

```dart
final idFamilyProvider = StateProvider.family<int,String>((ref,id){
  if(id == 'true') {
    return 1;
  } else {
    return 0;
  }
});
```

   其中family要求传入两个类型，第一个为返回类型，第二个为传入参数类型。在使用时

```dart
int id = ref.watch(idFamilyProvider('true'));
```

4. autoDispose
   provider不使用时则销毁，解决内存泄漏问题

当然，生成provider也可通过注解的方式进行生成，在依赖中添加 `riverpod_annotation: ^2.3.3` 就可以通过注解的方式生产provider
例如：

```dart
@riverpod
class Name extends _$Name {

}

@Riverpod(keepAlive: true)
class Age extends _$Age{

}
```

这里写了两种生成方式，调用时都可通过对应类名+provider便可获取到该provider，注意其中加参数的注解，与上面的autoDispose相区别，如果设置 `keepAlive = false` 表示不保存状态，但是不会释放资源，若设置 `keepAlive = true` ,表示会保存状态，即使没有消费者使用也会继续保存。