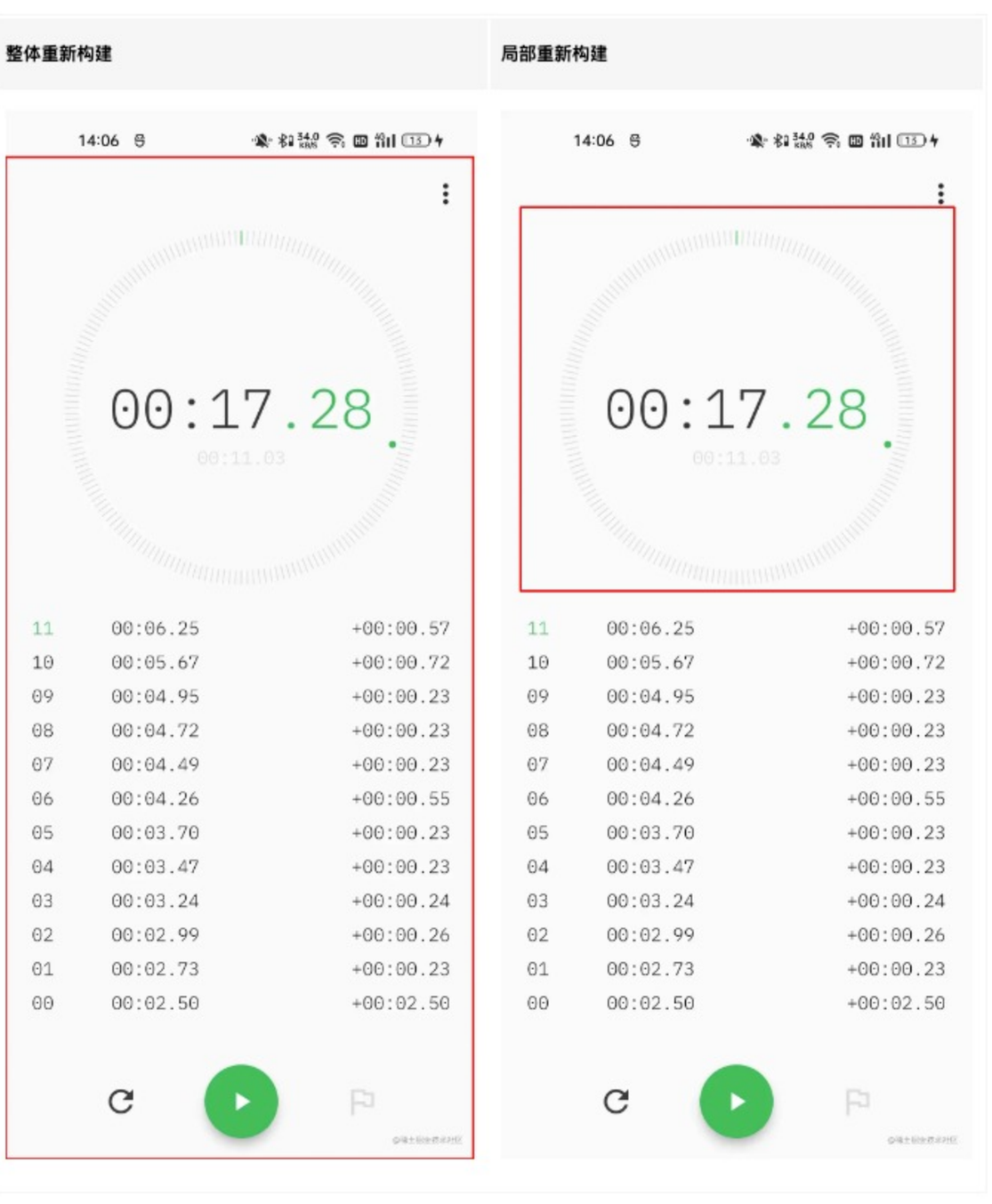
一、局部重新构建

1. 什么是局部重新构建

状态类在触发 `setState` 方法后，状态类中的 `build` 方法会被触发，重新构建组件。比如在秒表项目中，秒表值的变化会伴随重新构建和界面更新。而 `_HomePageState` 的 `build` 方法构建的是整个 `Scaffold` ，也就是左图所示的区域被重新构建。

而像秒表这样触发非常频繁的场景，每 `16 ms` 就会触发一次更新。其中记录列列表面板、按钮工具、头部栏和秒表值的变化并没直接的关系，也就是说它们没有必要随着秒表的运动而被重新构建。秒表数值的变化时最好如右图，只让表盘组件重新构建。这种当状态变化时，只让相关区域重构的手段就是局部构建。

整体重新构建	局部重新构建



2.实现局部更新 - ValueListenableBuilder 组件

`ValueListenableBuilder` 组件通过对 `ValueListenable` 可监听对象的监听，通过回调来触发组件重新构建，来达到局部更新的目的。如下所示，在构建表盘方法中，通过 `builder` 回调来构造 `StopwatchWidget` 组件。

```dart
---->[timer/11/home_page/home_page.dart]----
ValueNotifier<Duration> _duration = ValueNotifier(Duration.zero);

Widget buildStopwatchPanel() {
  double radius = MediaQuery.of(context).size.shortestSide / 2 * 0.75;
  return ValueListenableBuilder<Duration>(
      valueListenable: _duration,
      builder: (_, value, __) => StopwatchWidget(
            radius: radius,
            duration: value,
            themeColor: Theme.of(context).primaryColor,
            secondDuration: _secondDuration,
          ));
}
```

这样，在 `_onTick` 回调中，就无须使用 `setState` 重新构建整个 `Sacffold` 。如下 `tag1` 处，当 `_duration` 的值发生变化时，会触发 `builder` 回调，只会重构 `StopwatchWidget` 组件，从而达到局部更新的目的。

```dart
void _onTick(Duration elapsed) {
  dt = elapsed - lastDuration;
  _duration.value += dt; // tag1
  if (_record.isNotEmpty) {
    _secondDuration = _duration.value - _record.last.record;
  }
  lastDuration = elapsed;
}
```

3. 不要过度地局部构建

局部构建的作用不像大家想象中的那么大。在一帧的渲染过程中，构建逻辑只是一小部分，而且 `Flutter` 框架本身在更新时，会对未变动的组件进行优化处理。所以没有必要将每个状态的变化都用局部构建来完成，这样的话，额外代码量增加，导致可读性变差，可能会适得其反。

如下是通过 `setState` 重新构建整个 `Sacffold` 组件的调试效果，可以看出秒表运行过程中界面可以保持在 `60 FPS` 之内。浅蓝色的柱状表示构建时间，在不是非常复杂的界面中，整体的界面的更新并不会造成太大的性能问题。

如下是通过 `ValueListenableBuilder` 局部构建的效果，总体来说构建时间要短一些。

像 `秒表` 、 `倒计时` 、 `动画` 、 `拖动回调` 这类更新频率非常快的场景，最好可以控制重新构建的区域范围。来避免一些不必要的逻辑处理频繁触发。或者构建过程非常复杂，一次构建耗时较多，局部构建组件也能有些收益。但对于触发不是很频繁，构建逻辑也不是非常复杂的，也没有必要非要 `局部构建` 。万事都有利弊，凡事把握度非常重要。

在状态管理中介绍 `局部构建` 的原因是：状态管理也可以实现局部构建。如下所示， `BlocBuilder` 中通过 `builder` 回调根据状态对象构建组件，这样就能在状态类变化时，仅触发 `builder` 回调进行局部构建。不过上一章介绍的，对状态量的维护和触发更新是 `状态管理` 存在的意义，而 `局部构建` 只是触发更新过程中的一个小特点，大家要分清主次，千万不要认为状态管理的目的是为了 `局部构建` 。

二、通过 bloc 封装逻辑

我们知道，状态 (`State`) 类需要实现 `build` 方法来完成组件的 `构建逻辑` ；但同时还要维护数据的变化、触发界面更新的 `业务逻辑` 。这样当数据较多，或逻辑比较复杂时， `State` 的派生类就会承担非常多的职能，导致代码量的累加，变得庞大和复杂。所以逻辑的分离在一些复杂的场景中是很必要的，比如 `10` 文件夹中的 `_HomePageState` 中需要维护五个数据，逻辑比较复杂。我们将借由这套管理来对 `业务逻辑` 的处理进行封装，简化 `_HomePageState` 中的代码。

1. 使用 bloc 封装业务逻辑 - 状态类

如下，定义 `StopWatchState` 状态类来维护秒表界面所需的数据。 `type` 表示秒表运行的状态； `duration` 表示当前秒表运行的时长； `durationRecord` 表示秒表记录数据。 `secondDuration` 可以根据当前时长和记录的最后一值计算得出。这样就对状态数据进行 `封装` ，便于维护。

```dart
---->[timer/12/home_page/bloc/state.dart]----
class StopWatchState {
  final StopWatchType type;
  final List<TimeRecord> durationRecord;
  final Duration duration;

  Duration get secondDuration {
    if (durationRecord.isNotEmpty) {
      return duration - durationRecord.last.record;
    }
    return duration;
  }

  const StopWatchState({
    this.type = StopWatchType.none,
    this.durationRecord = const [],
    this.duration = Duration.zero,
  });

  StopWatchState copyWith({
    StopWatchType? type,
    List<TimeRecord>? durationRecord,
    Duration? duration,
  }) {
    return StopWatchState(
      type: type ?? this.type,
      durationRecord: durationRecord ?? this.durationRecord,
      duration: duration ?? this.duration,
    );
  }
}
```

2. 使用 bloc 封装业务逻辑 - 事件类

Bloc 通过事件来触发状态的变化，之前切换主题色和语言，由于状态变化的逻辑比较简单，在 `AppConfigBloc` 中通过成员方法来表述 `事件` 。这里状态变化稍微复杂一点，可以通过定义 `事件类` 来触发状态变化，当然通过成员方法来表述事件也未尝不可。对于事件而言， `方法` 和 `类型` 本质上都是一种事件触发的标志，不用过于纠结。

```dart
---->[timer/12/home_page/bloc/event.dart]----
abstract class StopWatchEvent {
  const StopWatchEvent();
}

class ResetStopWatch extends StopWatchEvent{
  const ResetStopWatch();
}

class ToggleStopWatch extends StopWatchEvent {
  const ToggleStopWatch();
}

class _UpdateDuration extends StopWatchEvent {
  final Duration duration;

  _UpdateDuration(this.duration);
}

class RecordeStopWatch extends StopWatchEvent {
  const RecordeStopWatch();
}
```

3. 使用 bloc 封装业务逻辑 - Bloc 类

在界面按键点击时，通过 `Bloc` 触发事件即可，数据的变化逻辑由 `Bloc` 派生类进行维护。如下，在 `StopWatchBloc` 构造中监听事件，当发送事件时，会触发对应的方法，对状态进行维护。

```dart
class StopWatchBloc extends Bloc<StopWatchEvent,StopWatchState>{
  Ticker? _ticker;

  StopWatchBloc():super(const StopWatchState()){
    on<ToggleStopWatch>(_onToggleStopWatch);
    on<ResetStopWatch>(_onResetStopWatch);
    on<RecordeStopWatch>(_onRecordeStopWatch);
    on<_UpdateDuration>(_onUpdateDuration);
  }
```

维护 `Ticker` 对象的逻辑也放在 `StopWatchBloc` 中，在 `_onTick` 中触发 `_UpdateDuration` 事件来更新时长数据。可以覆写 `close` 方法，触发 `_ticker` 的 `dispose` 进行销毁工作。

```dart
void _initTickerWhenNull() {
  if (_ticker != null) return;
  _ticker = Ticker(_onTick);
}

Duration _dt = Duration.zero;
Duration _lastDuration = Duration.zero;

void _onTick(Duration elapsed) {
  _dt = elapsed - _lastDuration;
  add(_UpdateDuration(state.duration + _dt));
  _lastDuration = elapsed;
}

@override
Future<void> close() async {
  _ticker?.dispose();
  _ticker = null;
  return super.close();
}
```

在 `_onUpdateDuration` 监听中，通过 `emit` 产出新状态：

```dart
void _onUpdateDuration(_UpdateDuration event, Emitter<StopWatchState> emit) {
  emit(state.copyWith(duration: event.duration));
}
```

在点击中间按钮时 `开启/暂停` 秒表，其中的逻辑处理和之前类似，只不过将逻辑封装到 `StopWatchBloc` 中而已。

```dart
void _onToggleStopWatch(ToggleStopWatch event, Emitter<StopWatchState> emit) {
  _initTickerWhenNull();
  if (_ticker!.isTicking) {
    _ticker!.stop();
    _lastDuration = Duration.zero;
    emit(state.copyWith(type: StopWatchType.stopped));
  } else {
    _ticker!.start();
    emit(state.copyWith(type: StopWatchType.running));
  }
}
```

`重置` 和 `记录` 的逻辑如下所示，也是把之前的逻辑封装到 `StopWatchBloc` 中。

```dart
void _onResetStopWatch(ResetStopWatch event, Emitter<StopWatchState> emit) {
  _lastDuration = Duration.zero;
  _dt = Duration.zero;
  emit(const StopWatchState());
}

void _onRecordeStopWatch(RecordeStopWatch event, Emitter<StopWatchState> emit) {
  List<TimeRecord> currentList = state.durationRecord.map((e) => e).toList();
  Duration current = state.duration;
  Duration addition = state.duration;
  if (currentList.isNotEmpty) {
    addition = current - currentList.last.record;
  }
  currentList.add(TimeRecord(record: current, addition: addition));
  emit(state.copyWith(durationRecord: currentList));
}
```

三、业务逻辑和构建逻辑分离

将 `业务逻辑` 封装到 `StopWatchBloc` 中后，就可以将 `_HomePageState` 中的相关代码移除。让状态类专注于构建逻辑的实现，达到业务逻辑和构建逻辑分离。如下所示，将文件进行分层：数据的维护逻辑分离之后，也有利于复用，别人也可以基于这些数据来构建不同的界面，这是分离之后带来的额外好处。

通过 `BlocBuilder` 组件可以基于状态数据构建界面，其中 `buildWhen` 参数可以控制是否触发更新，能在局部构建的基础上，进一步避免无用的构建。下面是表盘界面的构建逻辑：

```dart
Widget buildStopwatchPanel() {
  double radius = MediaQuery.of(context).size.shortestSide / 2 * 0.75;
  return BlocBuilder<StopWatchBloc, StopWatchState>(
    buildWhen: (p, n) => p.duration != n.duration,
    builder: (_, state) => StopwatchWidget(
      duration: state.duration,
      radius: radius,
      themeColor: Theme.of(context).primaryColor,
      secondDuration: state.secondDuration,
    ),
  );
}
```

工具按钮的构建逻辑如下，其中按钮的事件原本要维护数据，现在只需要通过 `StopWatchBloc` 对象触发事件即可。

```dart
Widget buildTools() {
 return BlocBuilder<StopWatchBloc, StopWatchState>(
    buildWhen: (p, n) => p.type != n.type,
    builder: (_, state) => ButtonTools(
      state: state.type,
      onRecoder: onRecoder,
      onReset: onReset,
      toggle: toggle,
    ),
  );
}

StopWatchBloc get stopWatchBloc => BlocProvider.of<StopWatchBloc>(context);

void onReset() => stopWatchBloc.add(const ResetStopWatch());
void onRecoder() => stopWatchBloc.add(const RecordeStopWatch());
void toggle() => stopWatchBloc.add(const ToggleStopWatch());
```

记录面板构建逻辑如下：

```dart
Widget buildRecordPanel() {
  return Expanded(
    child: BlocBuilder<StopWatchBloc, StopWatchState>(
      buildWhen: (p, n) => p.durationRecord != n.durationRecord,
      builder: (_, state) => RecordPanel(
        record: state.durationRecord,
      ),
    ),
  );
}
```

本章通过 `逻辑分类` 的角度介绍了 `状态管理` 的作用，通过分离可以简化原本在状态类中的复杂逻辑。通过分层，不仅有利于代码的维护，还可以便于复用。在 `Flutter` 开发中，对 `数据维护` 和 `界面构建` 的理解是非常重要的。思想是共通的，这在其他的和界面相关的领域中也是通用的，希望大家可以好好体会。到这里本册的主要内容就介绍完了，但这些只是一个起点，需要熟练掌握 `Flutter` 还有很长的路要走。

全部评论（3）

season_zhu LV.5 JY.5 | 大前端API调用工程师 ... | 17小时前
我特别不喜欢BLoC，基本没写过，要么用Provider，要么用GetX
点赞 回复

一路漫漫 LV.3 JY.4 | 安卓开发工程师 | 4月前
这篇相比前面的有点看不懂了，代码，主要是bloc状态管理貌似没讲的太清楚
点赞 回复

一路漫漫 LV.3 JY.4 | 安卓开发工程师 | 4月前
我感觉bloc是不是用起来门槛比较高，getX好像更容易些
点赞 回复