Provider 使用介绍

1. **Provider是什么?**

2. **InheritedWidget 介绍**

3. **Provider API介绍，什么场景下使用哪种API**

4. **基于Provider 封装MVVM场景扩展**

5. **注意事项**

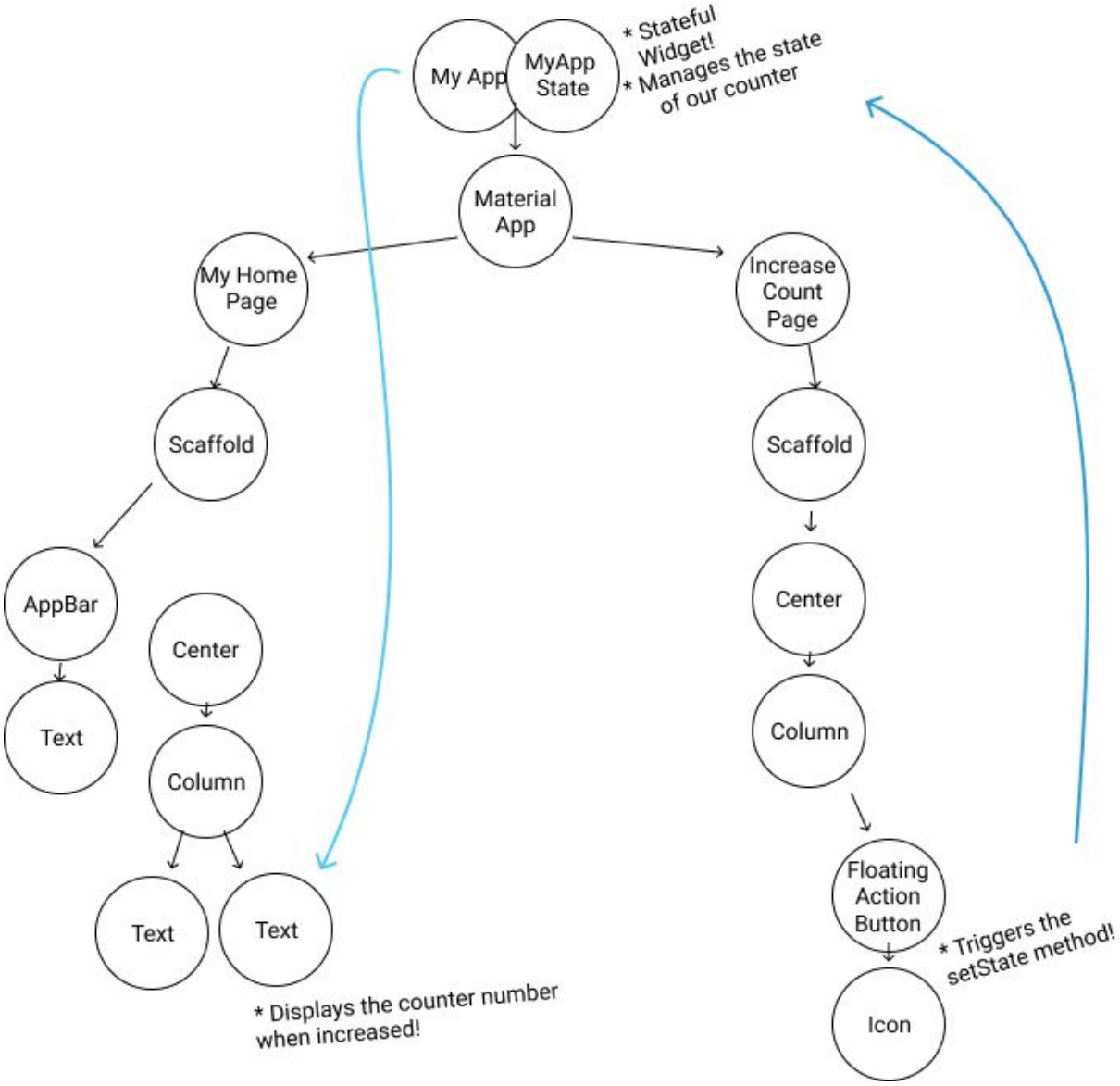
6. **QA**

正文

Flutter团队建议初学者使用**Provider**来管理state。但是Provider到底是什么，该如何使用?

Provider是一个UI工具，它是**状态管理的helper**，它是一个widget。通过这个widget可以把model对象传递给它的子widget。

1. **InheritedWidget 介绍**



`InheritedWidget`是Flutter中非常重要的一个功能型组件，它提供了一种数据在widget树中从上到下传递、共享的方式，比如我们在应用的根widget中通过`InheritedWidget`共享了一个数据，那么我们便可以在任意子widget中来获取该共享的数据！这个特性在一些需要在widget树中共享数据的场景中非常方便！如Flutter SDK中正是通过InheritedWidget来共享应用主题（`Theme`）和Locale (当前语言环境)信息的。

```
class ShareDataWidget extends InheritedWidget {
  ShareDataWidget({
    @required this.data,
    Widget child
  }) :super(child: child);

  final int data; //需要在子树中共享的数据，保存点击次数

  //定义一个便捷方法，方便子树中的widget获取共享数据
  static ShareDataWidget of(BuildContext context) {
```

```
    return context.dependOnInheritedWidgetOfExactType<ShareDataWidget>();
    //return context.getElementForInheritedWidgetOfExactType<ShareDataWidget>().widget;
  }

  //该回调决定当data发生变化时，是否通知子树中依赖data的Widget
  @override
  bool updateShouldNotify(ShareDataWidget old) {
    //如果返回true，则子树中依赖(build函数中有调用)本widget
    //的子widget的`state.didChangeDependencies`会被调用
    return old.data != data;
  }
}
```

1. Provider 介绍

Provier 包装的是**InheritedWidget**， 是我们的代码更容易使用和复用；如果我们使用**InheritedWidget** 每次都创建一个他的子类共享和管理数据；Provider 能够解决跨页面的数据问题，同时可以控制页面的刷新的粒度；

Provier 几个不错的好处（官方）

1. 简单 创建 / 销毁
2. 延迟加载
3. 大大减少每次创建新类模版的时间
4. 非常友好配套调试工具（暂时未研究）
5. 封装了比较简单通用好理解方法去消费**InheritedWidget** 中的数据；
    1. `Provider.of`
    2. `Consumer`
    3. `Selecor`
    4. `context.watch`
    5. `context.read`

> Provider提供了很多不同类型的Provider

name	description
Provider	最基础的Provider，可以提供Value类型数据共享
ListenableProvider	特定的Listenable 监听对象. ListenableProvider 将要监听对象的改变，并且询问小组件依赖他进行重建，当他的调用者在被调用时候
ChangeNotifierProvider	与ListenableProvider区别在于，当需要的时候，会自动调用dispose.
ValueListenableProvider	与ListenableProvider区别在于，仅仅支持value方式获取状态。
StreamProvider	与ListenableProvider区别在于，仅仅支持value方式获取状态。
FutureProvider	与ListenableProvider区别在于，仅仅支持value方式获取状态。

管理数据之Provider.of

```
var style = TextStyle(color: Colors.white);

class ChildWidget1 extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    debugPrint('ChildWidget1 build');
    var model = Provider.of<TestModel>(context);
    return Container(
      color: Colors.redAccent,
      height: 48,
      child: Row(
        mainAxisAlignment: MainAxisAlignment.spaceEvenly,
        children: <Widget>[
          Text('Child1', style: style),
          Text('Model data: ${model.value}', style: style),
          RaisedButton(
            onPressed: () => model.add(),
            child: Text('add'),
          ),
        ],
      ),
```

```
    );
  }
}
```

- `Provider.of<T>(context)`：用于需要根据数据的变化而自动刷新的场景
- `Provider.of<T>(context, listen: false)`：用于只需要触发Model中的操作而不关心刷新的场景

因此对应的，在新版本的Provider中，作者还提供了两个Context的拓展函数，来进一步简化调用。

- `T watch<T>()`
- `T read<T>()`

他们就分别对应了上面的两个使用场景，所以在上面的示例中，Text获取数据的方式，和在Button中点击的方式还可以写成下面这张形式。

```
Text('watch: ${context.watch<TestModel>().value}', style: style)

RaisedButton(
  onPressed: () => context.read<TestModel>().add(),
  child: Text('add'),
),
```

ChangeNotifierProvider

```
class ProviderState1Widget extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return ChangeNotifierProvider(
      create: (_) => TestModel(modelValue: 1),
      child: Padding(
        padding: const EdgeInsets.all(8.0),
        child: Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: <Widget>[
            ChildWidget1(),
            SizedBox(height: 24),
            ChildWidget2(),
          ],
```

```
      ),
    ),
  );
 }
}
```

通过ChangeNotifierProvider的create函数，创建初始化的Model。同时创建其Child，这个风格和InheritedWidget是不是有异曲同工之妙。

管理数据之Consumer

获取Provider管理的数据Model，有两种方式，一种是通过Provider.of(context)来获取，另一种，是通过Consumer来获取，在设计Consumer时，作者给它赋予了两个功能。

- 当传入的BuildContext中，不存在指定的Provider时，Consumer允许我们从Provider中的获取数据（其原因就是Provider使用的是InheritedWidget，所以只能遍历父Widget，当指定的Context对应的Widget与Provider处于同一个Context时，就无法找到指定的InheritedWidget了）
- 提供更加精细的数据刷新范围，避免无谓的刷新

控制更加精细的刷新范围

管理数据之Selector

Selector同样是获取数据的一种方式，从理论上来说，Selector等于Consumer等于Provider.of，但是它们对数据的控制粒度，才是它们之间根本的区别。

获取数据的方式，从Provider.of，到Consumer，再到Selector，实际上经历了这样一种进化。

- Provider.of：Context内容进行Rebuild
- Consumer：Model内容变化进行Rebuild
- Selector：Model中的指定内容变化进行Rebuild

```
class ChildWidgetA extends StatelessWidget {
```

```
  @override
  Widget build(BuildContext context) {
    debugPrint('ChildWidgetA build');
    return Selector<TestModel, int>(
      selector: (context, value) => value.modelValueA,
      builder: (BuildContext context, value, Widget child) {
        return Container(
          color: Colors.redAccent,
          height: 48,
          child: Row(
            mainAxisAlignment: MainAxisAlignment.spaceEvenly,
            children: <Widget>[
              Text('ChildA', style: style),
              Text('Model data: $value', style: style),
              RaisedButton(
                onPressed: () => context.read<TestModel>().addA(),
                child: Text('add'),
              ),
            ],
          ),
        );
      },
    );
  }
}
```

MultiProvider

```
MultiProvider(
  providers: [
    Provider<Something>(create: (_) => Something()),
    Provider<SomethingElse>(create: (_) => SomethingElse()),
    Provider<AnotherThing>(create: (_) => AnotherThing()),
  ],
  child: someWidget,
)
```

总计消费数据

非常简单的方式读取 `Provider` 中的数据, 通过BuildContext 中的扩展方法: `Consumer` `watch`, `read`, `selector`

1. `Consumer` `Consumer2<A, B>` `Consumer3<A, B>`

```
Widget build(BuildContext context) {
  return ChangeNotifierProvider(
    create: (_) => Foo(),
  child: Text(Provider.of<Foo>(context).value),
  );
}

Widget build(BuildContext context) {
      return ChangeNotifierProvider(
        create: (_) => Foo(),
        child: Consumer<Foo>(
          builder: (_, foo, __) => Text(foo.value),
        },
      );
   }
```

2. `context.watch<T>()`，监听小部件上T数据的改变

```
extension WatchContext on BuildContext {
  T watch<T>() {
    /// 隐藏了部分代码
    return Provider.of<T>(this);
  }
}
```

3. `context.read<T>()` 不会监听数据的改变

```
/// Exposes the [read] method.
extension ReadContext on BuildContext {
  T read<T>() {
      /// 隐藏了部分代码
    return Provider.of<T>(this, listen: false);
  }
}
```

4. `context.select<T, R>(R cb(T value))` 容许组件监听一小部分数据T的改变

   或者使用静态方法 `Provider.of<T>(context)`，使用的方式和 `watch` / `read` 相同

```
class Home extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Text(
      // Don't forget to pass the type of the object you want to obtain to `watch`!
      context.watch<String>(),
    );
  }
}
```

什么情况下应该使用哪种Provider?

1. 不需要监听状态的改变，子组件不需要获取父组件的状态，使用 `Provider` 就足够
2. 需要共享数据，并且监听数据的变化，可以使用 `ListenableProvider` 或者 `ChangeNotifierProvider`，如果需要自主管理数据的 `dispose`，建议使用 `ListenableProvider`
3. 如果有两种数据类型，或者有多种数据依赖关系，使用ProxyProvider系列0

```
/// 创建单一的[Provider]
  /// 使用这个方法时，需要在main.dart中添加 Provider.debugCheckInvalidValueType = null;
  static Provider createProvider<T>(T t) {
    return Provider<T>(
      create: (BuildContext c) => t,
    );
  }

 /// 创建单一的[ListenableProvider]
  static ListenableProvider createListenableProvider<T extends Listenable>(
      T t) {
    return ListenableProvider<T>(
      create: (BuildContext c) => t,
    );
  }
```

1. Provider 在MVVM中的实践

```
import 'package:flutter/foundation.dart';
import 'package:flutter_provider_mvvm/viewmodels/view_state.dart';

class BaseViewModel extends ChangeNotifier {
  bool _disposed = false;

  ViewState _viewState = ViewState.idle;

  ViewState get viewState {
    return _viewState;
  }

  void setState(ViewState state) {
    _viewState = state;
    notifyListeners();
  }

  @override
  void dispose() {
    super.dispose();
    _disposed = true;
  }

  @override
  void notifyListeners() {
    if (!_disposed) {
      super.notifyListeners();
    }
  }
}
```

```
import 'package:flutter/widgets.dart';
import 'package:provider/provider.dart';

import 'base_view_model.dart';

class BaseViewModelWidget<T extends BaseViewModel> extends StatefulWidget {
  final Widget Function(BuildContext context, T model, Widget child) builder;

  final T model;
```

```
  final Widget child;
  final Function(T) onModelReady;

  const BaseViewModelWidget(
      {Key key, this.builder, this.model, this.onModelReady, this.child})
      : super(key: key);

  @override
  _BaseViewModelWidgetState<T> createState() => _BaseViewModelWidgetState<T>();
}

class _BaseViewModelWidgetState<T extends BaseViewModel>
    extends State<BaseViewModelWidget<T>> {
  T _model;

  @override
  void initState() {
    // TODO: implement initState
    _model = widget.model;
    if (widget.onModelReady != null) {
      widget.onModelReady(_model);
    }
    super.initState();
  }

  @override
  Widget build(BuildContext context) {
    return ChangeNotifierProvider<T>(
      create: (BuildContext context) {
        return _model;
      },
      child: Consumer<T>(
        builder: widget.builder,
        child: widget.child,
      ),
    );
  }
}
```

```
BaseViewModelWidget(
              model: HomeViewModel(),
              onModelReady: (HomeViewModel model) {
                model.loadData();
              },
              builder:
                  (BuildContext context, HomeViewModel model, Widget widget) {
                print("builder:===><");
                return Builder(
```

```
            builder: (context) => Center(
              child: model.viewState == ViewState.loading
                  ? CircularProgressIndicator()
                  : Column(
                      children: <Widget>[
                        // Text('APP首页 ${widget.arguments['mobile']} '),
                        Text('APP首页 ${model?.count} '),
                        FlatButton(
                          onPressed: () {
                            model.fav();
                          },
                          child: Text('点击'),
                        ),
                      ],
                    )));
        },
      ),
```

参考