Flutter状态管理之 Provider 使用详解


云层之上 LV.3
2022年02月22日 10:31 · 阅读 3276

关注

> Flutter中状态管理是永恒的话题，本篇就Provider使用做出最简单、最全面的介绍（只涉及使用，暂不剖析原理）。因为每个知识点可能都会涉及一个示例代码（都是些重复简单的代码），所以篇幅较长。希望对你有所启发。

「介绍」

Provider是社区构建的状态管理工具，也是**Flutter Favorite**一员。基于 `InheritedWidget` 组件进行封装，使其更简单易用。它拥有着一套完整的解决方案，能够解决开发者们遇到的绝大多数情况，它能够让你开发出简单、高性能、层次清晰的应用，可见它是真的优秀啊！

因为是基于 `InheritedWidget` 组件进行封装，开发者应需对其有所了解，不了解的可以查看上一篇关于 InheritedWidget 的介绍。

» 优势

引用官方描述：

- 简化的资源分配与处置
- 懒加载
- 创建新类时减少大量的模板代码
- 支持 DevTools
- 更通用的调用 InheritedWidget 的方式（参考 Provider.of/Consumer/Selector）
- 提升类的可扩展性，整体的监听架构时间复杂度以指数级增长（如 ChangeNotifier， 其复杂度为O(N)）

注：笔者也是很推荐刚刚接Flutter的开发者使用Provider。

Provider 中提供了几种不同类型的provider（暂且称为提供者）和几种使用模式。本文分别通过提供者和使用者两个方面，多种模式进行说明。

「提供者」

» 1、Provider

最基础的 provider 组成，接收一个任意值并暴露它，但是并不会更新UI。

源码：

swift 复制代码

```swift
Provider({
  Key? key,
  required Create<T> create,
  Dispose<T>? dispose,
  bool? lazy,
  TransitionBuilder? builder,
  Widget? child,
})...
```

示例：

- Model

ini 复制代码

```ini
class Person{
  String name = "Provider";
}
```

- 程序入口设置

css 复制代码

```css
return Provider<Person>(
  create: (ctx)=> Person(),
  child: const MaterialApp(
    home: ProviderDemo(),
  ),
);
```

> 说明： 所有示例若不说明，默认状态管理放在顶层 MaterialApp 之上，开发者可自行定义。

- View

scala 复制代码

```scala
class ProviderDemo extends StatelessWidget {
  const ProviderDemo({Key? key}) : super(key: key);
  @override
  Widget build(BuildContext context) {
    return Scaffold(
```

```
        body: Center(
          child: Consumer<Person>( /// 在程序任何地方都可以拿到person对象，读取数据
              builder: (_,person,child){
                return Text(person.name);
              }
          ),
        ),
      );
    }
  }
```

这里的 `Consumer` 后续会讲到。

除了使用 `create` 来创建新的对象模型时，也可以使用一个已经存在的对象实例。这里是模型不是狭义上的model，也可以是viewModel、工具类等。

如：

less 复制代码

```
Provider.value(
  value: value, /// value是已经存在的模型对象
  child: ...
)
```

或

less 复制代码

```
ChangeNotifierProvider.value(
  value: value, /// value是已经存在的模型对象
  child: ...
)
```

注：使用 `.value` 是将已经存在的对象实例暴露出来，如果是新的模型对象，务必使用 `create` 。

在使用provider 的 `create` 和 `update` 回调时，回调函数默认是延迟调用的。也就是说，变量被读取时，`create` 和 `update` 函数才会被调用。开发者也可以使用 `lazy` 参数来禁用这一行为：

scss 复制代码

```
Provider(
  create: (_) => Something(),
  lazy: false,
)
```

我们还注意到有个 dispose 的回调：

dart 复制代码

```
typedef Dispose<T> = void Function(BuildContext context, T value);
```

当 Provider 所在节点被移除的时候，它就会启动 `Disposer<T>`，然后我们便可以在这里释放对应的资源。

» 2、ChangeNotifierProvider

监听模型对象的变化，而且当数据改变时，它也会重建 `Consumer`，更新UI。

- 继承、混入ChangeNotifier
- 调用了 `notifyListeners()`

它不会重复实例化模型，除非在个别场景下。如果该实例已经不会再被调用，ChangeNotifierProvider 也会自动调用模型的dispose()方法。

- ChangeNotifier

它类似于一个Observable，继承自Listenable，内部维护了一个 ObserverList _listeners，提供添加和删除listener的方法，有一个notify方法，用来通知所有的观察者（在Provider或者称为消费者Consumer）。

示例：

- Model

```scala
class Person with ChangeNotifier{
  String name = "ChangeNotifierProvider";
  /// 提供一个改变name的方法
  void changName({required String newName}){
    name = newName;
    notifyListeners();
  }
}
```

- 程序入口设置

```css
return ChangeNotifierProvider<Person>(
  create: (ctx)=> Person(),
  child: const MaterialApp(
    home: ChangeNotifierProviderDemo(),
  ),
);
```

- View

```scala
class ChangeNotifierProviderDemo extends StatelessWidget {
  const ChangeNotifierProviderDemo({Key? key}) : super(key: key);
  @override
  Widget build(BuildContext context) {
    return Scaffold(
```

```
      appBar: AppBar(title: const Text("ChangeNotifierProvider"),),
      body: Center(
        child: Column(
          mainAxisAlignment: MainAxisAlignment.spaceEvenly,
          children: [
            /// 拿到person对象，读取数据
            Consumer<Person>(builder: (ctx,person,child) =>  Text(person.name),),
            /// 拿到person对象，调用方法
            Consumer<Person>(builder: (ctx,person,child){
              return ElevatedButton( /// 点击按钮，调用方法更新
                onPressed: () => person.changName(newName: "ChangeNotifierProvider更新了"),
                child: const Text("点击更新"),
              );
            },),
          ],
        ),
      ),
    );
  }
}
```

- 结果 显示数据；点击按钮数据成功修改。

» 3、FutureProvider

`FutureProvider` 有一个初始值，接收一个 `Future` ，并在其进入 complete 状态时更新依赖它的组件。

- `FutureProvider` 只会重建一次
- 默认显示初始值， `Future` 进入 complete 状态时，会更新UI
- `FutureProvider` 只会重建一次

示例：

- Model

dart 复制代码

```
class Person {
  String? name;
  Person({required this.name});
}
```

- 程序入口设置

csharp 复制代码

```
return FutureProvider<Person>(
  initialData: Person(name: "初始值"),
  create: (ctx){
    /// 延迟2s后更新
```

```
    return Future.delayed(const Duration(seconds:2), () => Person(name: "更新FutureProvider"
  },
  child: const MaterialApp(
    home: FutureProviderDemo(),
  ),
);
```

- View

scala 复制代码

```
class FutureProviderDemo extends StatelessWidget {
  const FutureProviderDemo({Key? key}) : super(key: key);
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: const Text("FutureProvider"),),
      body: Center(
        child:Consumer<Person>(builder: (ctx,person,child) =>Text(person.name!),),,
      ),
    );
  }
}
```

- 结果 先显示默认值，2s后更新数据

» 4、StreamProvider

监听流，并暴露出当前的最新值，并多次触发重新构建UI。

示例：

- Model

dart 复制代码

```
class Person {
  String? name;
  Person({required this.name});
}
```

- 程序入口设置

kotlin 复制代码

```
return StreamProvider<Person>(
  initialData: Person(name: "初始值"),
  create: (ctx) {
    /// 传入一个Stream，每间隔1s数据更新一次
    return Stream<Person>.periodic(const Duration(seconds: 1),(value){
      return Person(name: "StreamProvider ---  $value");
    });
```

```
  },
  child: const MaterialApp(
    home: StreamProviderDemo(),
  ),
);
```

- View

scala 复制代码

```
class StreamProviderDemo extends StatelessWidget {
  const StreamProviderDemo({Key? key}) : super(key: key);
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: const Text("FutureProvider"),),
      body: Center(
        child: Consumer<Person>(builder: (ctx,person,child) =>  Text(person.name!),),
      ),
    );
  }
}
```

建议开发者先行了解**StreamBuilder**。

- 结果 先显示默认值，每1s更新一次数据，刷新一次UI。

» 5、MultiProvder

上述中我们只使用了一种Provder，但在实际开发中，程序肯定存在多种Provder，如果我们还是用嵌套的方式来解决，但是这样无疑是混乱的，可读性级差。

如：

php 复制代码

```
return ProvderA(
  child: ProvderB(
    child: ProvderC(
      child: ProvderD(
        ...
        child: MaterialApp()
      )
    )
  )
)
```

为了解决这样的嵌套地狱，MultiProvder应运而生。它实际上是多种Provder的集合，且仅仅是改变了代码的书写方式。

示例：

- Model

```scala
class Person1 with ChangeNotifier{
  String name = "MultiProvider ---  1";
  void changName(){
    name = "更新MultiProvider ---  1";
    notifyListeners();
  }
}
```

```scala
class Person2 with ChangeNotifier{
  String name = "MultiProvider ---  2";
  void changName(){
    name = "更新MultiProvider ---  2";
    notifyListeners();
  }
}
```

- 程序入口设置

```scss
return MultiProvider(
  providers: [
    ChangeNotifierProvider<Person1>(
      create: (ctx) => Person1(),
    ),
    ChangeNotifierProvider<Person2>(
      create: (ctx) => Person2(),
    )
  ],
  child: const MaterialApp(
    home: MultiProviderDemo(),
  ),
);
```

providers多个Provider的集合

- View

```scss
class MultiProviderDemo extends StatelessWidget {
  const MultiProviderDemo({Key? key}) : super(key: key);
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      body: Center(
```

```
          child: Column(
            mainAxisAlignment: MainAxisAlignment.spaceEvenly,
            children: [
              Consumer<Person1>(builder: (ctx,person1,child) => Text(person1.name)),
              Consumer<Person2>(builder: (_,person2,child) => Text(person2.name)),
              Consumer<Person1>( /// 拿到person1对象，调用方法，也可以修改person2
                builder: (ctx,person1,child){
                  return ElevatedButton(
                    onPressed: (){
                      person1.changName();
                    },
                    child: const Text("点击修改"),
                  );
                },
              ),
            ],
          ),
        ),
      );
    }
}
```

» 6、ProxyProvider

我们日常开发中会遇到一种模型嵌套另一种模型、或一种模型的参数用到另一种模型的值、或是需要几种模型的值组合成一个新的模型的情况，在这种情况下，就可以使用 `ProxyProvider` 。它能够将多个 provider 的值聚合为一个新对象，将结果传递给 `Provider` （注意是 `Provider` 而不是 `ChangeNotifierProvider` ），这个新对象会在其依赖的任意一个 provider 更新后同步更新值。

r 复制代码

```
ProxyProvider<T, R> /// R依赖T或用到T的值，T发生改变会通知R
```

> 注：同步更新不代表同步更新UI，也可能只是值更新了。是否同步更新UI取决了使用了哪一种依赖的provider，比如使用最基础的 `Provider` 值已经改变了（通过热更新或debug可知），但是不会更新UI；若使用 `ChangeNotifierProvider` 更新值的同时会同步更新UI。

示例：

- Model

scala 复制代码

```
class Person extends ChangeNotifier{
  String name = "小虎牙";

  void changName(){
```

```
    name = "更新的小虎牙";
    notifyListeners();
  }
}
```

EatModel 需要用到 Person 的 name 值，才知道到底是谁在吃饭：

dart 复制代码

```
class EatModel{
  EatModel({required this.name});

  final String name;

  String get whoEat => "$name正在吃饭";
}
```

- 程序入口设置

scss 复制代码

```
return MultiProvider(
  providers: [
    ChangeNotifierProvider<Person>(
      create: (ctx) => Person(),
    ),
    ProxyProvider<Person, EatModel>(
      update: (ctx, person, eatModel) => EatModel(name: person.name),
    )
  ],
  child: const MaterialApp(
    home: ProxyProviderDemo(),
  ),
);
```

- View

scala 复制代码

```
class ProxyProviderDemo extends StatelessWidget {
  const ProxyProviderDemo({Key? key}) : super(key: key);
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      body: Center(
        child: Column(
          mainAxisAlignment: MainAxisAlignment.spaceEvenly,
          children: [
            Consumer<EatModel>(builder: (ctx,eatModel,child) => Text(eatModel.whoEat)),
            Consumer<Person>( // 拿到person对象，调用方法
              builder: (ctx,person,child){
                return ElevatedButton( /// 点击按钮更新Person的name，eatModel.whoEat会同步更新
```

```
                onPressed: () => person.changName(),
                child: const Text("点击修改"),
              );
            },
          ),
        ],
      ),
    ),
  );
 }
}
```

- 结果 页面显示 **小虎牙正在吃饭**；点击按钮后：内容更新为**更新的小虎牙正在吃饭**。

`ProxyProvider` 还有其他不同的形式： `ProxyProvider` 、 `ProxyProvider2` 、 `ProxyProvider3` ... `ProxyProvider6` 。类名后的数字是 `ProxyProvider` 依赖的 provider 的数量。一般很难用到6个或以上的模型糅合一个新的模型，如果有，那么应该需要思考工程架构是否出了问题。

» 7、ChangeNotifierProxyProvider

与 `ProxyProvider` 的相似，不同的是 `ChangeNotifierProxyProvider` 会将它的值传递给 `ChangeNotifierProvider` 而非 `Provider` 。

官方的示例看起来有点繁复，我们简化下写个商品展示和收藏列表的示例。当然我们应该学习官发示例：尽量在Model中使用私有变量 `_` ，减少耦合。这里为了简单明了就随意一点。

示例：

- Model 一共有3个模型对象，分别是商品模型 `ShopModel` 、商品列表模型 `ListModel` 、收藏列表 `CollectionListModel` 。

分别为：

arduino 复制代码

```
class Shop{
  final int id;
  final String name;
  Shop(this.id, this.name);
}
```

arduino 复制代码

```
class ListModel {
  // 商品列表
  final List <Shop> shops = [
    Shop(1, "Apple/苹果 14 英寸 MacBook"),
    Shop(2, "HUAWEI/华为Mate 40 RS "),
    Shop(3, "Apple/苹果 11 英寸 iPad Pro"),
    Shop(4, "Xiaomi 12Pro5g骁龙8"),
```

```
    Shop(5, "Apple/苹果 iPhone 13 Pro"),
    Shop(6, "华为/HUAWEI Mate X2"),
    Shop(7, "小米11 Ultra至尊5g手机"),
    Shop(8, "HUAWEI/华为P40 Pro+ 5G 徕卡"),
  ];
}
```

scala 复制代码

```
class CollectionListModel extends ChangeNotifier{

  // 依赖ListModel
  final ListModel _listModel;

  CollectionListModel(this._listModel);

  // 所有收藏的商品
  List<Shop> shops = [];

  // 添加收藏
  void add(Shop shop){
    shops.add(shop);
    notifyListeners();
  }
  // 移除收藏
  void remove(Shop shop){
    shops.remove(shop);
    notifyListeners();
  }
}
```

- 程序入口设置

yaml 复制代码

```
return MultiProvider(
  providers: [
    Provider<ListModel>(
      create: (ctx) => ListModel(),
    ),
    ChangeNotifierProxyProvider<ListModel, CollectionListModel>(
      create: (ctx) => CollectionListModel(ListModel()),
      update: (ctx, listModel, collectionModel) => CollectionListModel(listModel),
    )
  ],
  child: MaterialApp(
    theme: ThemeData(
      primarySwatch: Colors.orange,
    ),
    debugShowCheckedModeBanner: false,
    home: const ChangeNotifierProxyProviderDemo(),
```

```
  ),
);
```

- Widget

scala 复制代码

```
class ChangeNotifierProxyProviderDemo extends StatefulWidget {
  const ChangeNotifierProxyProviderDemo({Key? key}) : super(key: key);
  @override
  _ChangeNotifierProxyProviderDemoState createState() => _ChangeNotifierProxyProviderDemoSta
}

class _ChangeNotifierProxyProviderDemoState extends State<ChangeNotifierProxyProviderDemo> {
  int _selectedIndex = 0;
  final _pages = [const ListPage(), const CollectionPage()];
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      body: _pages[_selectedIndex],
      bottomNavigationBar: BottomNavigationBar(
        currentIndex: _selectedIndex,
        onTap: (index) {
          setState(() {
            _selectedIndex = index;
          });
        },
        items: const [
          BottomNavigationBarItem(
              icon: Icon(Icons.list_alt_outlined),
              label: "商品列表"
          ),
          BottomNavigationBarItem(
              icon: Icon(Icons.favorite),
              label: "收藏列表"
          )
        ],
      ),
    );
  }
}
```

商品列表：

scala 复制代码

```
class ListPage extends StatelessWidget {
  const ListPage({Key? key}) : super(key: key);

  @override
  Widget build(BuildContext context) {
```

```
    ListModel listModel = Provider.of<ListModel>(context);
    List<Shop> shops = listModel.shops;
    return Scaffold(
      appBar: AppBar(title: const Text("商品列表"),),
      body: ListView.builder(
        itemCount: listModel.shops.length,
        itemBuilder: (ctx,index) => ShopItem(shop: shops[index],),
      ),
    );
  }
}
```

收藏列表：

scala 复制代码

```
class CollectionPage extends StatelessWidget {
  const CollectionPage({Key? key}) : super(key: key);
  @override
  Widget build(BuildContext context) {
    CollectionListModel collectionModel = Provider.of<CollectionListModel>(context);
    List<Shop> shops = collectionModel.shops;
    return Scaffold(
      appBar: AppBar(title: const Text("收藏列表"),),
      body: ListView.builder(
        itemCount: shops.length,
        itemBuilder: (ctx,index) => ShopItem(shop: shops[index],),
      ),
    );
  }
}
```

商品Item：

scala 复制代码

```
class ShopItem extends StatelessWidget {
  const ShopItem({Key? key,required this.shop}) : super(key: key);
  final Shop shop;
  @override
  Widget build(BuildContext context) {
    return ListTile(
      leading: CircleAvatar(
        child: Text("${shop.id}"),
      ),
      title: Text(shop.name,style: const TextStyle(fontSize: 17),),
      trailing: ShopCollectionButton(shop: shop,),
    );
  }
}
```

商品收藏按钮：

```scala
class ShopCollectionButton extends StatelessWidget {
  const ShopCollectionButton({Key? key,required this.shop}) : super(key: key);
  final Shop shop;
  @override
  Widget build(BuildContext context) {
   CollectionListModel collectionModel = Provider.of<CollectionListModel>(context);
   bool contains = collectionModel.shops.contains(shop);
   return InkWell(
     onTap: contains ? ()=> collectionModel.remove(shop) : ()=> collectionModel.add(shop),
     child:  SizedBox(
       width: 60,height: 60,
       child: contains ? const Icon(Icons.favorite,color: Colors.redAccent,) : const Icon(I
     ),
   );
  }
}
```

> 这里使用了 `Provider.of` 来获取数据，官方示例中使用了 `context.read` 、 `context.read` ，文章后续会有详细的介绍。

- 结果：

» 8、ListenableProvider

供可监听对象使用的特殊 provider。ListenableProvider 会监听对象，并在监听器被调用时更新依赖此对象的 widgets。类似于 ChangeNotifierProvider，是其父类。适用于任何的 Listenable。一般情况下使用 ChangeNotifierProvider，但如果是自己实现 Listenable 或使用动画的话，就可以使用 ListenableProvider。

且 ListenableProvider 提供（provide）的对象是继承了 Listenable 抽象类的子类。由于无法混入，所以通过继承来实现，同时必须实现其 addListener / removeListener 方法，手动管理收听者。显然这样太过复杂，所有通常情况下都不需要这样做。

java 复制代码

```
class ChangeNotifier implements Listenable
```

混入了 `ChangeNotifier` 的类会自动帮我们实现了管理，所以 ListenableProvider 同样也可以接收混入了 ChangeNotifier 的类。

» 9、ListenableProxyProvider

`ListenableProxyProvider` 是 `ListenableProvider` 的一个变体，但是在使用上和 `ChangeNotifierProvider` 效果惊人的一致。即存在2种或多种模型相互依赖的情况。

这里就不写示例了，相信开发者通过 ListenableProvider 和 ChangeNotifierProvider 的讲解应该能够很好的运用。

» 10、ValueListenableProvider

监听 ValueListenable，并且只暴露出 `ValueListenable.value` 。它实际上是专门用于处理只有一个单一变化数据的 ChangeNotifier。

```scala
class ValueNotifier<T> extends ChangeNotifier implements ValueListenable<T>
```

scala 复制代码

它也有两种方式： `ValueListenableProvider()` 和 `ValueListenableProvider.value()` 。两种方式几乎是相同的。

「消费者」

» 1、Provider.of

这个我们在上面已经用到了，还记得我们在上一篇关于 InheritedWidget 的介绍中提到了有个默认的约定：如果状态是希望暴露出的，应当提供一个 “of” 静态方法来获取其对象，开发者便可直接通过该方法来获取。

```arduino
static T of<T>(BuildContext context, {bool listen = true})
```

arduino 复制代码

其中 `listen` ：默认true监听状态变化，false为不监听状态改变。

`Provider.of<T>(context)` 是 `Provider` 为我们提供的静态方法，当我们使用该方法去获取值的时候会返回查找到的最近的 `T` 类型的 `provider` 给我们，且也不会遍历整个组件树。

» 2、Consumber

Provider 中使用比较频繁的消费者，查看源码：

```scss
Consumer({
  Key? key,
  required this.builder,
  Widget? child,
}) : super(key: key, child: child);
```

scss 复制代码

```
...

@override
Widget buildWithChild(BuildContext context, Widget? child) {
  return builder(
    context,
    Provider.of<T>(context),
    child,
  );
}
```

发现它就是通过 `Provider.of<T>(context)` 来实现的。而且实际开发中使用 `Provider.of<T>(context)` 比 `Consumer` 简单好用太多，那 `Consumer` 有什么优势吗?

对比一下，我们发现 `Consumer` 有个 `Widget? child`，它非常重要，能够在复杂项目中，极大地缩小你的控件刷新范围。

我们通过一个简单的计数器示例来说明：

css 复制代码

```
return Scaffold(
  body: Consumer(
    builder: (BuildContext context,CounterModel counterModel,Widget? child){
      return Column(
        children: [
          Text("${counterModel.count}"),
          ElevatedButton(
            onPressed: ()=> counterModel.increment(),
            child: const Text("点击加1"),
          ),
          Text("其他更多组件"),
          Text("其他更多组件"),
          Text("其他更多组件"),
          Text("其他更多组件"),
          Text("其他更多组件"),
        ],
      );
    },
  ),
);
```

在上述示例中，我们后面很多的 `Text` 组件没有用到模型数据且不需要更新状态的，但是因为被 `Consumer` 包裹，导致每次数据改变都会重新构建！严重影响性能且不优雅！

解决以上问题，一方面我们可以尽可能调整 Consumer 的位置，在需要使用数据的组件包裹 Consumer，但这也会存在一个问题，单独用大量 `Consumer` 包裹，也跟 `Provider` 诞生的理念背道而驰。这时候我们就可以使用到 `Widget? child` 了。我们针对上述示例优化一下：

```css 复制代码
return Scaffold(
  body: Consumer(
    builder: (BuildContext context,CounterModel counterModel,Widget? child){
      return Column(
        children: [
          Text("${counterModel.count}"),
          ElevatedButton(
            onPressed: ()=> counterModel.increment(),
            child: const Text("点击加1"),
          ),
          child!
        ],
      );
    },
    child: Column(
      children: [
        Text("其他更多组件"),
        Text("其他更多组件"),
        Text("其他更多组件"),
        Text("其他更多组件"),
        Text("其他更多组件"),
      ],
    ),
  ),
);
```

我们使用 `Widget? child` 将不需要更新状态的组件包裹起来，大大提升了性能。

» 3、Selector

Selector 也是一个消费者。与 Consumer 类似，只是对 `build` 调用 `Widget` 方法时提供更精细的控制。Consumer 是监听一个 Provider 中所有数据的变化，Selector 则是监听某一个/多个值的变化。

比如用户模型 Person：有姓名、性别、年龄、身高、体重等信息，但是我们可能只会更新下年龄，其他的信息我们不希望重建，就可以使用 `Selector` 实现这个功能。

示例：

- Model

```scala 复制代码
class Person with ChangeNotifier {
  String name = "小虎牙";
  int age = 18;
  double height = 180.0;

  // 年龄改变
  void increaseAge() {
```

```
    age ++;
    notifyListeners();
  }
}
```

- 程序入口设置：

```scss 复制代码
return ChangeNotifierProvider(
  create: (ctx) => Person(),
  child: const MaterialApp(
    home: SelectorDemo(),
  ),
);
```

- View

```php 复制代码
class SelectorDemo extends StatelessWidget {
  const SelectorDemo({Key? key}) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: const Text("Selector"),),
      body: Center(
        child: Selector<Person,int>(
          selector: (ctx,person) => person.age,
          builder: (ctx,age,child) {
            return Column(
              children: [
                Text("年龄为：$age"),
                child!
              ],
            );
          },
          child: Padding(
            padding: const EdgeInsets.only(top: 50),
            child: ElevatedButton(
              onPressed: () => Provider.of<Person>(context,listen: false).increaseAge(),
              child: const Text("点击改变年龄"),
            ),
          ),
        )
      )
    );
  }
}
```

- 结果： 显示 `年龄为18` ， 点击后年龄加1。

> 这里也使用到了 `Widget? child` ， 同 Consumer 一样极大地缩小控件刷新范围。

» 4、InheritedContext

InheritedContext 实际上是在 Provider 内置扩展了 BuildContext。 有三种方式：

BuildContext.read

使用与 `Provider.of` 类似。用来获取数据的，不会触发刷新。

示例：

- Model

```ini 复制代码
class Person {
  String name = "小虎牙";
}
```

- 程序入口类设置：

```javascript 复制代码
return Provider(
  create: (ctx) => Person(),
  child: const MaterialApp(
    home: ```
ReadDemo
```(),
  ),
);
```

- View

```scala 复制代码
class ReadDemo extends StatelessWidget {
  const ReadDemo({Key? key}) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: const Text("ReadDemo"),),
      body: Center(
        child: Text("姓名：${context.read<Person>().name.toString()}"),
      )
    );
```
```

```
  }
}
```

- 结果： 显示 **姓名：小虎牙**，使用方式同 `Provider.of()` 相同。

BuildContext.watch

顾名思义，观察，可读可写，改变依赖的数据会触发更新。效果与 `Consumer` 极度类似，使用更简单明了。区别是没有 `Consumer` 的 `Widget? child` 的 优化控件刷新的功能。

通过 `context.watch<Model>()` 获取模型对象，使用和刷新数据。

- Model

scala 复制代码

```
class Person with ChangeNotifier{
  String name = "小虎牙";

  changeName(){
    name = "刷新小虎牙";
    notifyListeners();
  }
}
```

- 程序入口类设置：

scss 复制代码

```
return ChangeNotifierProvider(
  create: (ctx) => Person(),
  child: const MaterialApp(
    home: WatchDemo(),
  ),
);
```

- View

php 复制代码

```
class WatchDemo extends StatelessWidget {
  const WatchDemo({Key? key}) : super(key: key);

  @override
  Widget build(BuildContext context) {

    /// 通过context获取模型对象
    final person = context.watch<Person>();

    return Scaffold(
      appBar: AppBar(title: const Text("Watch"),),
      body: Center(
```

```
        child: Column(
          mainAxisAlignment: MainAxisAlignment.spaceEvenly,
          children: [
            Text("姓名: ${person.name}"),
            Padding(
              padding: const EdgeInsets.only(top: 50),
              child: ElevatedButton(
                onPressed: () => person.changeName(),
                child: const Text("点击改变姓名"),
              ),
            ),
          ],
        )
      )
    );
  }
}
```

- 结果： 显示：**姓名：小虎牙** ，点击按钮：显示更新为 **姓名：刷新小虎牙**。

BuildContext.select

于前面提到的 Selector 类似，指定监听对象的部分属性，使用更简单明了。区别是没有 `Selector` 的 `Widget? child` 的 优化控件刷新的功能。

- Model

scala 复制代码

```
class Person with ChangeNotifier {
  String name = "小虎牙";
  int age = 18;
  double height = 180.0;

  void increaseAge() {
    age ++;
    notifyListeners();
  }
}
```

- 程序入口类：

scss 复制代码

```
return ChangeNotifierProvider(
  create: (ctx) => Person(),
  child: const MaterialApp(
    home: SelectDemo(),
  ),
);
```

- View

```php
class SelectDemo extends StatelessWidget {
  const SelectDemo({Key? key}) : super(key: key);

  @override
  Widget build(BuildContext context) {

    /// 通过context获取模型对象
    final age = context.select((Person person) => person.age);

    return Scaffold(
      appBar: AppBar(title: const Text("Watch"),),
      body: Center(
        child: Column(
          mainAxisAlignment: MainAxisAlignment.spaceEvenly,
          children: [
            Text("年龄: $age"),
            Padding(
              padding: const EdgeInsets.only(top: 50),
              child: ElevatedButton(
                onPressed: () => Provider.of<Person>(context, listen: false)..increaseAge(),
                child: const Text("点击改变姓名"),
              ),
            ),
          ],
        )
      )
    );
  }
}
```

- 结果： 显示 **年龄为18**，点击后年龄加1。

以上就是关于 **Provider** 常用的几种使用方式，开发者可以根据自身需求自行使用。

分类： 前端 标签： Flutter

文章被收录于专栏：

Flutter Home

关于Flutter的学习

关注专栏

安装掘金浏览器插件

多内容聚合浏览、多引擎快捷搜索、多工具便捷提效、多模式随心畅享，你想要的，这里都有！

前往安装