

# Flutter（十四）状态 State 管理

AlanGe

状态管理是**声明式编程**非常重要的一个概念，我们在前面介绍过Flutter是声明式编程的，也**区分声明式编程和命令式编程**的区别。

这里，我们就来系统的学习一下Flutter声明式编程中非常重要的状态管理

## 一. 为什么需要状态管理？

### 1.1. 认识状态管理

很多从命令式编程框架（Android或iOS原生开发者）转成声明式编程（Flutter、Vue、React等）刚开始并不适应，因为需要一个新的角度来考虑APP的开发模式。

Flutter作为一个现代的框架，是声明式编程的：

$$\text{UI} = f(\text{state})$$

The layout  
on the screen

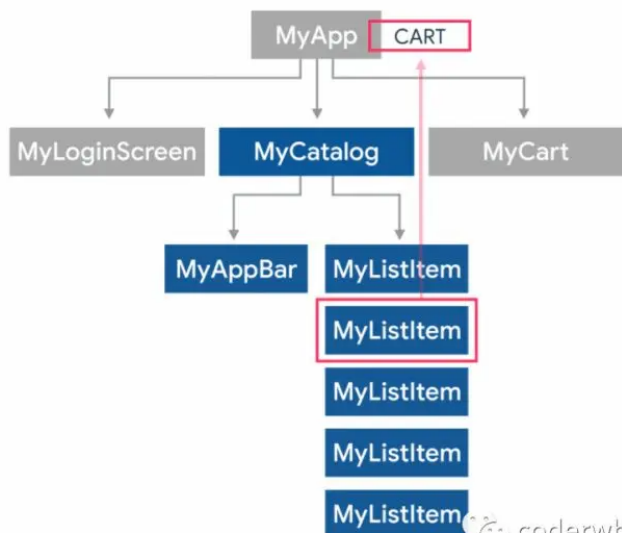
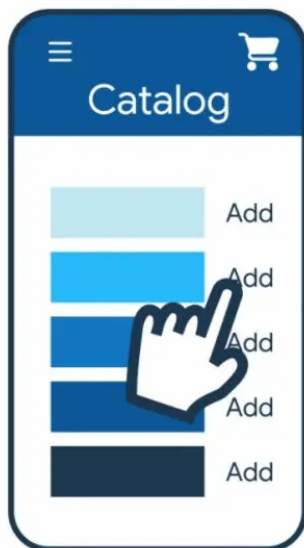
Your  
build  
methods

The application state

coderwhy

Flutter 构建应用过程

在编写一个应用的过程中，我们有大量的 State 需要来进行管理，而正是对这些 State 的改变，来更新界面的刷新：



coderwhy

## 1.2. 不同状态管理分类

### 1.2.1. 短时状态 Ephemeral state Ephemeral 短暂的，瞬间的

某些状态只需要在自己的 Widget 中使用即可

- 比如我们之前做的简单计数器 counter
- 比如一个 PageView 组件记录当前的页面
- 比如一个动画记录当前的进度
- 比如一个 BottomNavigationBar 中当前被选中的 tab

这种状态我们只需要使用 StatefulWidget 对应的 State 类自己管理即可，Widget 树中的其它部分并不需要访问这个状态。

这种方式在之前的学习中，我们已经应用过非常多次了。

### 1.2.2. 应用状态 App state

开发中也有非常多的状态需要在多个部分进行共享

- 比如用户一个个性化选项
- 比如用户的登录状态信息
- 比如一个电商应用的购物车
- 比如一个新闻应用的已读消息或者未读消息

这种状态我们如果在 Widget 之间传递来、传递去，那么是无穷尽的，并且代码的耦合度会变得非常高，牵一发而动全身，无论是代码编写质量、后期维护、可扩展性都非常差。

这个时候我们可以选择全局状态管理的方式，来对状态进行

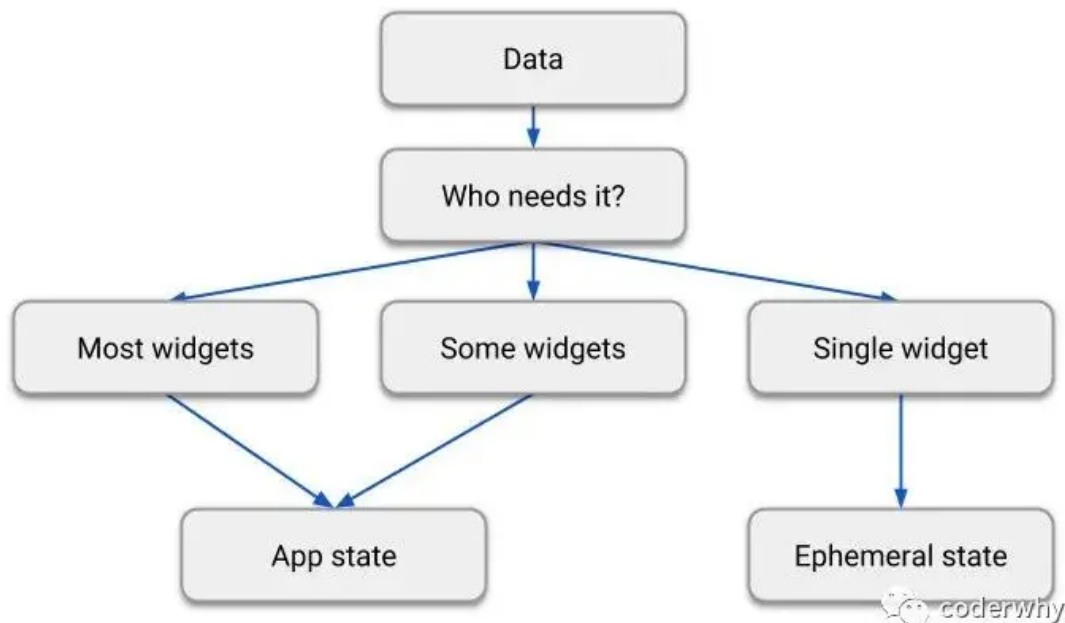
统一的管理和应用。

### 1.2.3. 如何选择不同的管理方式

开发中，没有明确的规则去区分哪些状态是**短时状态**，哪些状态是**应用状态**。

- 某些**短时状态**可能在之后的开发维护中需要**升级为应用状态**。

但是我们可以简单遵守下面这幅流程图的规则：



状态管理选择

针对 React 使用 **setState** 还是 Redux 中的 **Store** 来管理状态哪

个更好的问题，Redux的issue上，Redux的作者Dan Abramov，它这样回答的：

The rule of thumb is: Do whatever is less awkward

经验原则就是：选择能够减少麻烦的方式。



选择能够减少麻烦的方式

## 二. 共享状态管理

### 2.1. InheritedWidget Inherited 继承的

InheritedWidget 和 React 中的 context 功能类似，可以实现跨组件数据的传递。

定义一个共享数据的 InheritedWidget，需要继承自 InheritedWidget

- 这里定义了一个 of 方法，该方法通过 context 开始去查找祖先的 HYDataWidget（可以查看源码查找过程）
- updateShouldNotify 方法是对比新旧 HYDataWidget，是否需要更新相关依赖的 Widget

```
class HYDataWidget extends InheritedWidget {  
  final int counter;  
  
  HYDataWidget({this.counter, Widget child}): super(child:  
child);  
  
  static HYDataWidget of(BuildContext context) {  
    return context.dependOnInheritedWidgetOfExactType();  
  }  
  
  @override  
  bool updateShouldNotify(HYDataWidget oldWidget) {  
    return this.counter != oldWidget.counter;  
  }  
}
```

创建 HYDataWidget，并且传入数据（这里点击按钮会修改数据，并且重新 build）

```
class HYHomePage extends StatefulWidget {  
  @override  
  _HYHomePageState createState() => _HYHomePageState();  
}  
  
class _HYHomePageState extends State<HYHomePage> {  
  int data = 100;
```

```

@override
Widget build(BuildContext context) {
  return Scaffold(
    appBar: AppBar(
      title: Text("InheritedWidget"),
    ),
    body: HYDataWidget(
      counter: data,
      child: Center(
        child: Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: <Widget>[
            HYShowData()
          ],
        ),
      ),
    ),
    floatingActionButton: FloatingActionButton(
      child: Icon(Icons.add),
      onPressed: () {
        setState(() {
          data++;
        });
      },
    ),
  );
}
}

```

在某个Widget中使用共享的数据，并且监听

- 1.创建类InheritedProvide，继承于InheritedWidget，新增成员变量：父控件的state；
- 2.在类InheritedProvide中，重写updateShouldNotify方法，并返回true，通知子widget更新其相关的依赖；
- 3.在父组件中新增Widget类型的成员变量child
- 4.在父组件中新定义of方法，该方法通过类InheritedProvide返回其新增的成员变量state，让子组件通过这个of方法访问state数据；
- 5.父组件的state类的build方法中返回新建的类InheritedProvide通过构造函数创建的对象，传入父组件新增成员变量child，重新build和InheritedProvide想关联的子组件；
- 6.子组件通过父组件的of方法访问父组件state里面的数据。

## 2.2. Provider

**Provider**是目前官方推荐的**全局状态管理工具**，由社区作者 Remi Rousselet 和 Flutter Team 共同编写。

使用之前，我们需要先引入对它的依赖，截止这篇文章，Provider 的最新版本为 **4.0.4**：

```
dependencies:  
  provider: ^4.0.4
```

### 2.2.1. Provider 的基本使用

在使用 Provider 的时候，我们主要关心三个概念：

- ChangeNotifier：真正**数据（状态）存放的地方**
- ChangeNotifierProvider：**Widget 树中提供数据（状态）的地方**，会在其中**创建对应的 ChangeNotifier**
- Consumer：Widget 树中**需要使用数据（状态）的地方**

我们先来完成一个简单的案例，将官方计数器案例使用 Provider 来实现：**Consumer 消费者，顾客**

#### 第一步：创建自己的 ChangeNotifier

我们需要一个 ChangeNotifier 来**保存我们的状态**，所以创建



它

- 这里我们可以使用继承自 ChangeNotifier，也可以使用混入，这取决于概率是否需要继承自其它的类
- 我们使用一个私有的 \_counter，并且提供了 getter 和 setter
- 在 setter 中我们监听到 \_counter 的改变，就调用 notifyListeners 方法，通知所有的 Consumer 进行更新

```
class CounterProvider extends ChangeNotifier {  
  int _counter = 100;  
  int get counter {  
    return _counter;  
  }  
  set counter(int value) {  
    _counter = value;  
    notifyListeners();  
  }  
}
```

## 第二步：在 Widget Tree 中插入 ChangeNotifierProvider

我们需要在 Widget Tree 中插入 ChangeNotifierProvider，以便 Consumer 可以获取到数据：

- 将 ChangeNotifierProvider 放到了顶层，这样方便在整个应用的任何地方可以使用 CounterProvider

```
void main() {
```

```

runApp(ChangeNotifierProvider(
  create: (context) => CounterProvider(),
  child: MyApp(),
));
}

```

### 第三步：在首页中使用 Consumer 引入和修改状态

- 引入位置一：在 body 中使用 Consumer，Consumer 需要传入一个 builder 回调函数，当数据发生变化时，就会通知依赖数据的 Consumer 重新调用 builder 方法来构建；
- 引入位置二：在 floatingActionButton 中使用 Consumer，当点击按钮时，修改 CounterNotifier 中的 counter 数据；

```

class HYHomePage extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text("列表测试"),
      ),
      body: Center(
        child: Consumer<CounterProvider>(
          builder: (ctx, counterPro, child) {
            return Text("当前计数:${counterPro.counter}",
style: TextStyle(fontSize: 20, color: Colors.red),);
          }
        ),
      ),
    );
  }
}

```

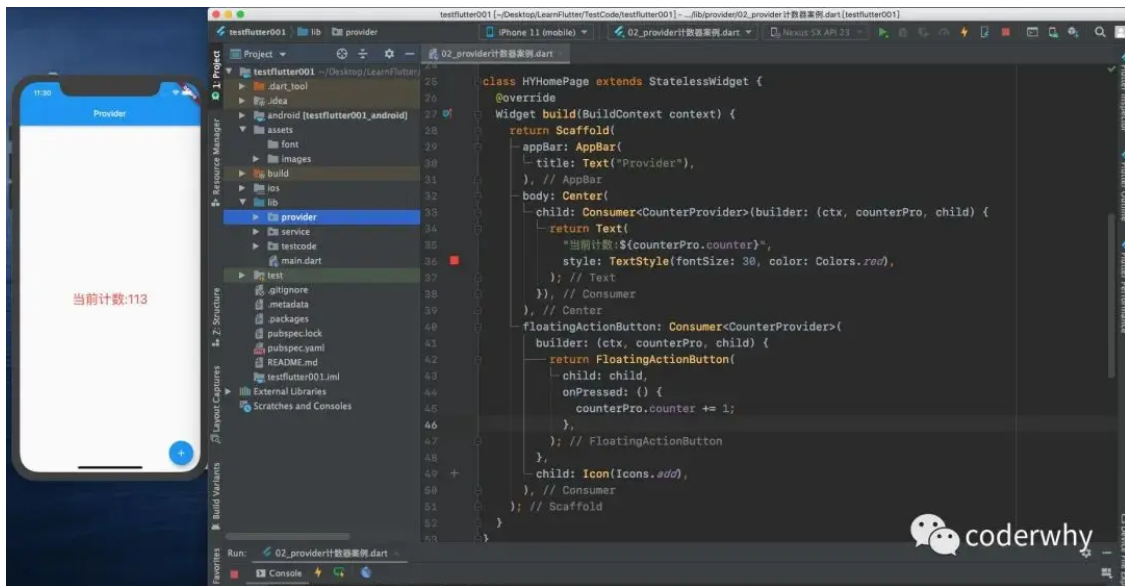
```

    ),
    floatingActionButton: Consumer<CounterProvider>(
      builder: (ctx, counterPro, child) {
        return FloatingActionButton(
          child: child,
          onPressed: () {
            counterPro.counter += 1;
          },
        );
      },
      child: Icon(Icons.add),
    ),
  );
}
}

```

Consumer 的 builder 方法解析：

- 参数一：context，每个 build 方法都会有上下文，目的是知道当前树的位置
- 参数二：ChangeNotifier 对应的实例，也是我们在 builder 函数中主要使用的对象
- 参数三：child，目的是进行优化，如果 builder 下面有一颗庞大的子树，当模型发生改变的时候，我们并不希望重新 build 这颗子树，那么就可以将这颗子树放到 Consumer 的 child 中，在这里直接引入即可（注意我案例中的 Icon 所放的位置）

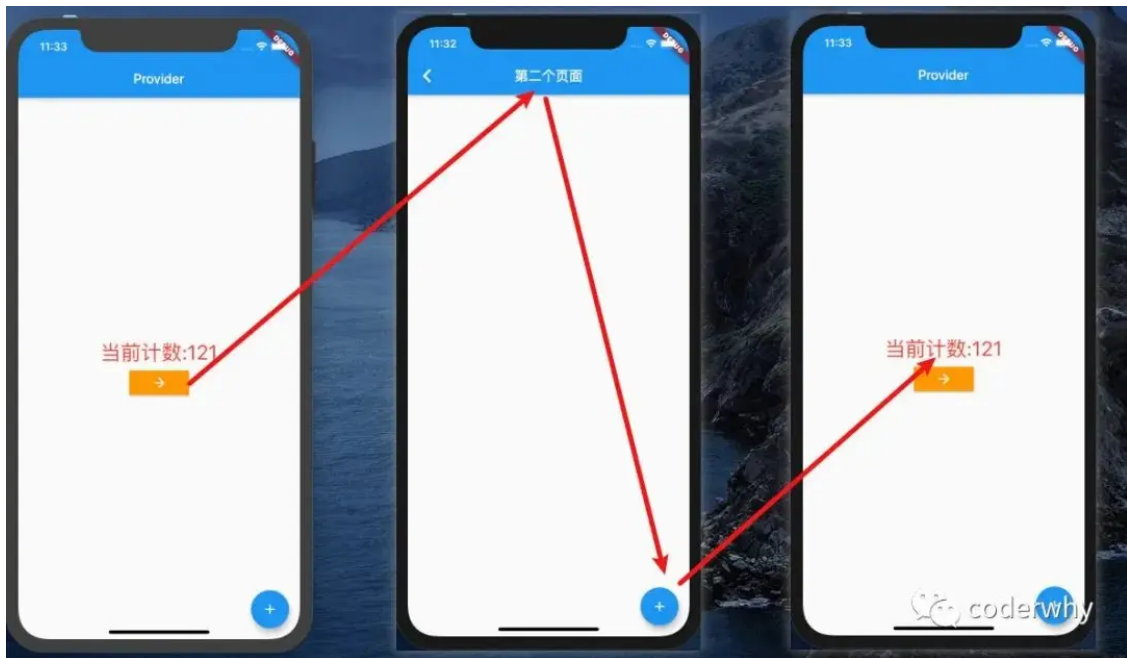


图片

步骤四：创建一个新的页面，在新的页面中修改数据

```
class SecondPage extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text("第二个页面"),
      ),
      floatingActionButton: Consumer<CounterProvider>(
        builder: (ctx, counterPro, child) {
          return FloatingActionButton(
            child: child,
            onPressed: () {
              counterPro.counter += 1;
            },
          );
        },
        child: Icon(Icons.add),
      ),
    );
  }
}
```

```
}  
}
```



图片

## 2.2.2. Provider.of的弊端

事实上，因为 **Provider** 是基于 **InheritedWidget**，所以我们在 **使用 ChangeNotifier 中的数据时**，我们可以通过 **Provider.of** 的方式来使用，比如下面的代码：

```
Text("当前计数:$  
{Provider.of<CounterProvider>(context).counter}",  
  style: TextStyle(fontSize: 30, color: Colors.purple),  
)
```

我们会发现很明显上面的代码会更加简洁，那么开发中是否要选择上面这种方式呢？

- 答案是**否定的**，更多时候我们还是要选择 **Consumer** 的**方式**。

为什么呢？因为 Consumer 在刷新整个 Widget 树时，会**尽可能少的 rebuild Widget**。

方式一：Provider.of 的方式完整的代码：

- 当我们点击了 floatingActionButton 时，HYHomePage 的 build 方法会被重新调用。
- 这意味着**整个 HYHomePage 的 Widget 都需要重新 build**

```
class HYHomePage extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    print("调用了HYHomePage的build方法");
    return Scaffold(
      appBar: AppBar(
        title: Text("Provider"),
      ),
      body: Center(
        child: Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: <Widget>[
            Text("当前计数:$
{Provider.of<CounterProvider>(context).counter}",
style: TextStyle(fontSize: 30, color:
Colors.purple),
          ],
        ),
      ),
    );
  }
}
```

```

    ),
    floatingActionButton: Consumer<CounterProvider>(
      builder: (ctx, counterPro, child) {
        return FloatingActionButton(
          child: child,
          onPressed: () {
            counterPro.counter += 1;
          },
        );
      },
      child: Icon(Icons.add),
    ),
  );
}
}

```

方式二：将Text中的内容采用 Consumer的方式修改如下：

- 你会发现 HYHomePage 的 build 方法不会被重新调用；
- 设置如果我们有 对应的 child widget，可以采用上面案例中的方式来组织，性能更高；

```

Consumer<CounterProvider>(builder: (ctx, counterPro, child)
{
  print("调用Consumer的builder");
  return Text(
    "当前计数:${counterPro.counter}",
    style: TextStyle(fontSize: 30, color: Colors.red),
  );
}),


```

### 2.2.3. Selector 的选择

Consumer 是否是最好的选择呢？并不是，它也会存在弊端

- 比如当点击了 floatingActionButton 时，我们在代码的两处分别打印它们的 builder 是否会重新调用；
- 我们会发现只要点击了 floatingActionButton，两个位置都会被重新 builder；
- 但是 floatingActionButton 的位置有重新 build 的必要吗？没有，因为它是否在操作数据，并没有展示；
- 如何可以做到让它不要重新 build 了？使用 Selector 来代替 Consumer

```
body: Center(  
  child: Consumer<CounterProvider>(builder: (ctx, counterPro, child) {  
    print("Text展示的位置builder被调用");  
    return Text(  
      "当前计数:${counterPro.counter}",  
      style: TextStyle(fontSize: 30, color: Colors.red),  
    ); // Text  
  }), // Consumer  
, // Center  
floatingActionButton: Consumer<CounterProvider>(builder: (ctx, counterPro, child) {  
  print("floatingActionButton展示的位置builder被调用");  
  return FloatingActionButton(  
    child: child,  
    onPressed: () {  
      counterPro.counter += 1;  
    },  
  ),  
); // FloatingActionButton  
),  
child: Icon(Icons.add),  
), // Consumer
```

 coderwhy

图片

我们先直接实现代码，在解释其中的含义：



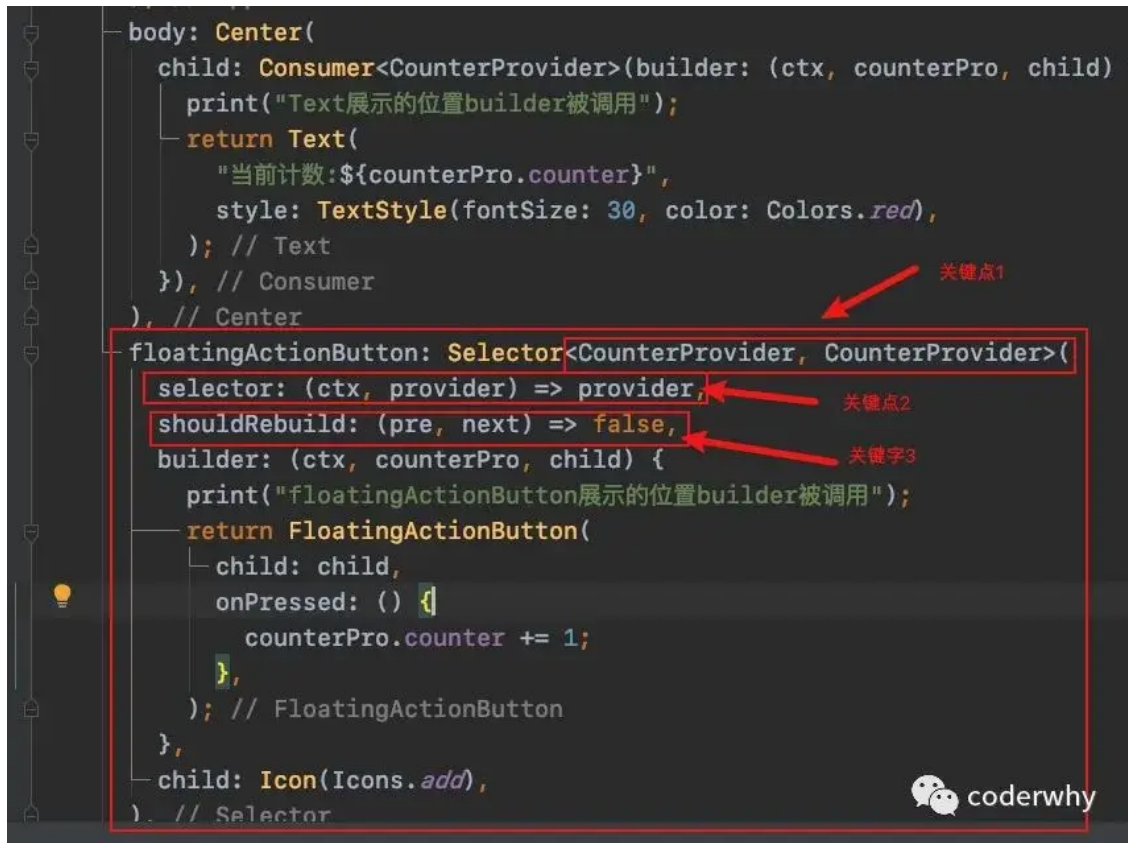
```
floatingActionButton: Selector<CounterProvider,
CounterProvider>(
  selector: (ctx, provider) => provider,
  shouldRebuild: (pre, next) => false,
  builder: (ctx, counterPro, child) {
    print("floatingActionButton展示的位置builder被调用");
    return FloatingActionButton(
      child: child,
      onPressed: () {
        counterPro.counter += 1;
      },
    );
  },
  child: Icon(Icons.add),
),
```

Selector 和 Consumer 对比，不同之处主要是三个关键点：

- 关键点 1：泛型参数是两个
- 泛型参数一：我们这次要使用的 Provider
- 泛型参数二：转换之后的数据类型，比如我这里转换之后依然是使用 CounterProvider，那么他们两个就是一样的类型

- 关键点 2: selector 回调函数
- 转换的回调函数, 你希望如何进行转换
- S Function(BuildContext, A) selector
- 我这里没有进行转换, 所以直接将 A 实例返回即可
- 关键点 3: 是否希望重新 rebuild
- 这里也是一个回调函数, 我们可以拿到转换前后的两个实例;
- bool Function(T previous, T next);
- 因为这里我不希望它重新 rebuild, 无论数据如何变化,

所以这里我直接 return false;



图片

这个时候，我们重新测试点击 `floatingActionButton`，  
`floatingActionButton` 中的代码并不会进行 `rebuild` 操作。

所以在某些情况下，我们可以使用 `Selector` 来代替  
`Consumer`，性能会更高。

## 2.2.4. MultiProvider

在开发中，我们需要共享的数据肯定不止一个，并且数据之

间我们需要组织到一起，所以一个 Provider 必然是不够的。

我们在增加一个新的 ChangeNotifier

```
import 'package:flutter/material.dart';

class UserInfo {
  String nickname;
  int level;

  UserInfo(this.nickname, this.level);
}

class UserProvider extends ChangeNotifier {
  UserInfo _userInfo = UserInfo("why", 18);

  set userInfo(UserInfo info) {
    _userInfo = info;
    notifyListeners();
  }

  get userInfo {
    return _userInfo;
  }
}
```

如果在开发中我们有多个 Provider 需要提供应该怎么做呢？

方式一：多个 Provider 之间嵌套

- 这样做有很大的弊端，如果嵌套层级过多不方便维护，

扩展性也比较差

```
runApp(ChangeNotifierProvider(  
  create: (context) => CounterProvider(),  
  child: ChangeNotifierProvider(  
    create: (context) => UserProvider(),  
    child: MyApp()  
  ),  
));
```

方式二：使用 MultiProvider

```
runApp(MultiProvider(  
  providers: [  
    ChangeNotifierProvider(create: (ctx) =>  
CounterProvider()),  
    ChangeNotifierProvider(create: (ctx) =>  
UserProvider()),  
  ],  
  child: MyApp(),  
));
```