1.didChangeDependencies 被调用的场合

```
@override
void _firstBuild() {
  try { // 触发 initState 回调
    final dynamic debugCheckForReturnedFuture = state.initState() as dynamic;
  } finally { ..... }
  state.didChangeDependencies(); // 触发 didChangeDependencies 回调
  super._firstBuild();
}
```

didChandependencies 被调用的scnenario ①：mount里面调用的firstBuild

```
  @override // StatefulElement
  void performRebuild() {
    if (_didChangeDependencies) { // 通常在代码清单 8-13 中设置为 true，详见 8.2 节
      state.didChangeDependencies(); // 当该字段为 true 时再次触发 didChangeDependencies
      _didChangeDependencies = false;
    }
    super.performRebuild();
  }
```

_didChangeDependencies 标志该 Element 的依赖节点发生了改变，此时 didChangeDependencies 方法会被再次调用，所以该回调比较适合响应一些依赖的更新。performRebuild 最终还会触发

didChandependencies 被调用的scnenario ②：rebuild里面调用的performRebuild

To summarize, createElement以及setState的时候都会调用didChangeDependencies.

2.didUpdateWidget 被调用的场合

对 StatefulElement 来说，其逻辑如代码清单 8-5 所示。

代码清单 8-5　flutter/packages/flutter/lib/src/widgets/framework.dart

/// Override this method to respond when the [widget] changes (e.g., to start implicit animations).

child.update()　call

```
void update(StatefulWidget newWidget) {  // StatefulElement，见代码清单 5-47
  super.update(newWidget);
  assert(widget == newWidget);
  final StatefulWidget oldWidget = state._widget!;
  _dirty = true;
  state._widget = widget as StatefulWidget;
  try {
    _debugSetAllowIgnoredCallsToMarkNeedsBuild(true);
    final dynamic debugCheckForReturnedFuture = state.didUpdateWidget(oldWidget)
        as dynamic;
  } finally {
    _debugSetAllowIgnoredCallsToMarkNeedsBuild(false);
  }
  rebuild(); // 触发代码清单 8-4 中的逻辑
}
```

rebuild里面会调用到_child = updateChild(child, newWidget, slot), updateChild里面会调用child.update, 此时先调用didUpdateWidget方法再调用rebuild方法

以上逻辑在通过 rebuild 触发 State 的 build 方法之前，会触发其 didUpdateWidget 方法。对于移除对 oldWidget 的一些引用和依赖，以及更新一些依赖 Widget 属性的资源，通过该方法进行操作是一个合适的时机。

when to override

3.Element.deactivate方法被调用的场合

```
@pragma('vm:prefer-inline')
Element? updateChild(Element? child, Widget? newWidget, Object? newSlot) {
  if (newWidget == null) {
    if (child != null)
      deactivateChild(child);
    return null;
  }
  final Element newChild;
  if (child != null) {
    bool hasSameSuperclass = true;
```

以上逻辑会触发 State 的 deactivate 方法，并导致当前节点进入 inactive 阶段。该回调触发说明当前 Element 节点被移出 Element Tree，但仍有可用被再次加入，该时机适合释放一些和当前状态强相关的资源，而对于那些和状态无关的资源，考虑到该 Element 节点仍有可能进入 Element Tree，并不适合在此时释放（可以类比 Android 中 Activity 的 onPause 回调）。

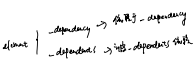
when to override

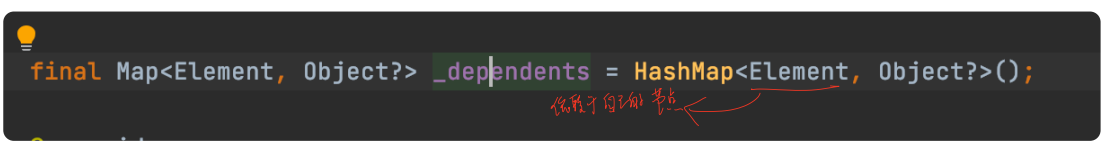
依赖的ancestor列表

```
@mustCallSuper
void deactivate() { // Element
  if (_dependencies != null && _dependencies!.isNotEmpty) { // 依赖清理
    for (final InheritedElement dependency in _dependencies!)
      dependency._dependents.remove(this);
  }
  _inheritedWidgets = null;
  _lifecycleState = _ElementLifecycle.inactive; // 更新状态
}
```

deactivate方法的source code

InheritedWidget 浅析






```
@mustCallSuper // Element
void mount(Element? parent, dynamic newSlot) {
  _parent = parent; // 对根节点而言，parent 为 null
  _slot = newSlot;
  _lifecycleState = _ElementLifecycle.active; // 更新状态
  _depth = _parent != null ? _parent!.depth + 1 : 1; // 树的深度
  if (parent != null) _owner = parent.owner; // 绑定 BuildOwner
  final Key? key = widget.key;  // Global Key 注册，详见第 8 章
  if (key is GlobalKey) { key._register(this); } // 见代码清单 8-15
  _updateInheritance(); // 见代码清单 8-14
}
```