flutter技术问题

1. Flutter是什么？为什么选择Flutter？

  答：Flutter 是Google推出的开源UI框架，用于开发跨平台应用，包括iOS、Android、Web和桌面应用

  选择Flutter的原因：

  (1)单一代码库：一次编写，随处运行。

  (2)热重载：提高开发效率。

  (3)丰富的Widget库：创建高性能、漂亮的用户界面。

  (4)强大的社区和文档支持。

2. 解释Dart语言的主要特点

  答：Dart是有google开发的编程语言，专为客户端设计。

  （1）强类型语言，支持类型判断

  （2）支持异步编程，拥有类似async/await的语法

  （3）丰富的标准库，适合用于web和移动开发

  （4）良好的性能，AOT（Ahead-of-time）和JIT（just-in-time）编译。

3. 什么事widget？flutter中的widget有哪些类型？

  答：widget事flutter的基本构成块，用于表示UI的一部分。

  类型：

  （1）有状态的Widget（StatefulWidget）：拥有内部状态，可以动态改变 =》

  生命周期：https://p3-juejin.byteimg.com/tos-cn-i-k3u1fbpfcp/06943d7b439f41fea821b7892650f01b~tplv-k3u1fbpfcp-zoom-in-crop-mark:1512:0:0:0.awebp

  initState()：Widget 初始化当前

  State，在当前方法中是不能获取到 Context 的，如想获取，可以试试 Future.delayed()

didChangeDependencies()：在 initState() 后调用，State对象依赖关系发生变化的时候也会调用。

deactivate()：当 State 被暂时从视图树中移除时会调用这个方法，页面切换时也会调用该方法，和Android里的

onPause 差不多。

dispose()：Widget 销毁时调用。

didUpdateWidget：Widget 状态发生变化的时候调用。

具体情况和调用顺序

1 . 初始化阶段

  当 StatefulWidget 第一次创建时，会按照以下顺序调用：

createState: 创建与此 StatefulWidget 关联的状态对象。

initState: 初始化状态。在这个方法中，您可以执行一次性的操作，例如初始化变量、订阅服务等。

didChangeDependencies: 在 widget 被插入到 widget 树中之后立即调用。在这个方法中，可以安全地访问 BuildContext 和依赖项

build: 构建 widget 的描述。在每次需要重建 widget 时，都会调用此方法。

2 . 更新阶段

  当 StatefulWidget 需要更新（例如父 widget 重新构建并传入新的参数）时，会调用以下方法：

didUpdateWidget: 当 widget 被重新构建时且新的 widget 是相同类型但具有不同配置时调用。可以在这里进行与旧配置比较的逻辑。

setState: 触发 widget 的重建。在这个方法中，你可以修改状态变量，Flutter 会在之后调用 build 方法。

build: 每次状态改变时，都会调用 build 方法来重新构建 widget。

3 . 销毁阶段

  当 StatefulWidget 从 widget 树中移除时，会调用以下方法：

deactivate: 当 State 对象从树中移除时调用。在这个方法中，你可以进行一些清理操作，但要避免过多的耗时操作，因为它可能在后续的重新插入过程中再次调用。

（2）无状态Widget（StatelessWidget）：没有内部状态，静态展示。

4. StatefulWidget和StatelessWidget的区别？

  答：

  StatefulWidget：

  （1）有内部状态，状态改变时可以重新构建。

  （2）build方法在状态改变时会多次调用

  StatelessWidget：

  （1）没有内部状态。

  （2）build只调用一次。

5. 解释下flutter的热重载（Hot Reload）功能

  答：热重载在不重启应用的情况下可以立即查看代码更改的效果，极大的提高了开发的效率，它保留了应用的状态，只重新加载了变更部分的代码。

6. 如何在flutter中处理状态？列出几种常见的状态管理方法

  答：处理状态的方法

  （1）使用StatefulWidget的SetState{

      setState方法在StatefulWidget中起着关键作用，它通知Flutter框架这个组件的状态 发生了变化，并触发重新构建UI。

   setState仅会影响当前组件和子树的重建，而不会影响到整个应用程序的重建，从而提升性能。

    }

  （2）使用InheritedWidget{

    1、InheritedWidget允许在Widget树中传递数据，而不需要通过构造函数逐层传递

    2、当数据发生变化时，只有依赖该数据的子Widget会重建，而其他部分不会受到影响，从而提升性能。

}

  （3）使用Provider的包{

         1、创建一个状态类，这个类通常是一个集成自“ChangeNotifier”的类，“ChangeNotifier”提供了管理状态的类，并通知监听器的状态变化。

        2、使用ChangeNotifierProvider在应用的根部或需要使用状态的地方提供状态。通常在main.dart中提供全局状态。

      3、在需要所用组件地方进行调用自定义change方法context.read<Counter>().increment()，然后  Consumer<Counter>( builder: (context, counter, child) { return Text( '${counter.count}', style: Theme.of(context).textTheme.headline4, ); }下的子容器会动态重新创建调用

      4、获取自定义的监听类final counter = Provider.of<Counter>(context);

}

  （4）使用Riverpod{

   1、属于Provider的加强版，解决了一些provider实现不了的局限性，更灵活更好的管理应用状态

  2、StateNotifier 是一种更强大的状态管理方式，适用于复杂的业务逻辑和状态管理。

  class CounterNotifier extends StateNotifier<int> { CounterNotifier() : super(0); void    increment() { state++; } }

 final counterNotifierProvider = StateNotifierProvider<CounterNotifier, int>((ref) {

  return CounterNotifier();

});

 3、通过ConsumerWidget进行创建widget{

 final counter = ref.watch(counterProvider);

 动态响应数据

 ref.read(counterProvider.notifier).state++;

}

}

  （5）使用GetX ==》 Widget是怎么绑定Controller的，又是怎么找到的？定义Controller：

  • 创建一个继承自GetxController的类，定义状态和业务逻辑。

  将Controller注入到依赖中：

  • 使用Get.put方法将Controller注入到依赖中，这样其他Widget可以找到并使用这个Controller。=》

  T put<T>(T dependency, {String tag, bool permanent = false}) { return GetInstance().put<T>(

  dependency, tag:

  tag, permanent: permanent); }

  class GetInstance { Map<String, dynamic> \_singleton = {}; T put<T>(T

  dependency, {String tag, bool permanent = false}) { final key = \_getKey(T, tag); \_

  singleton[key] =

  dependency; return dependency; }

  使用Controller管理状态：

  • 在Widget中使用GetBuilder或Obx来管理和响应状态变化。

（6）使用BloC（Business Logic Component）模式{

    在flutter中，Bloc（Business Logic、 Component）是一种状态管理模式，用于将业务逻辑与UI代码分离，它提供了一种结构化的方式来管理应用程序状态，使代码更具可维护和可测试性，BloC模式主要利用“Stream”和‘Sink‘来处理状态变化和事件。Business Logic将业务逻辑从UI代码中分离出来，几种处理业务逻辑，提高代码的可维护性和可测试性。

Streams：BloC模式使用Dart的Stream来处理异步事件和数据流。‘Stream’用于向UI提供数据和更新，而sink用于接收用户输入或事件。

event：事件是由用户或者系统触发的操作，如按钮点击或者数据加载。BloC接收事件并相应的更新状态。

States：状态是应用程序在任何事件点的表达形式，BloC处理事件后会产生新的状态，‘Stream’将新的状态推送给UI

}

7. 解释下flutter中的InheritedWidget

  答：InheritedWidget 是 Flutter 框架中用于在 widget 树中高效共享数据的一种机制。它允许数据在 widget

  树中向下传播，而无需通过构造函数手动传递数据。这对于状态管理、主题、配置等场景非常有用。

作用

InheritedWidget 主要用于以下几个方面：

数据共享：在 widget 树的某个位置定义一次数据，树中下层的所有 widget 都可以直接访问这些数据，而无需逐层传递。

状态管理：在某个地方集中管理状态，然后在需要的地方访问和响应这些状态的变化。

性能优化：当数据变化时，仅那些依赖这些数据的 widget 会被重建，而不是整个 widget 树。

工作原理

InheritedWidget 通过上下文 (context) 提供数据，子 widget 可以通过

context.dependOnInheritedWidgetOfExactType 方法来获取数据并建立依赖关系。当 InheritedWidget

中的数据发生变化时，所有依赖于它的子 widget 会被重新构建

8. 如何在flutter中实现导航？

  答：导航方法

  （1）使用Navigator类进行路由管理

  （2）使用Navigator.push和Navigator.pop进行页面跳转和返回

  （3）使用named routes定义和管理路由

  （4）使用第三方导航包（如Getx的导航功能）=》 Getx的路由和原生路由的区别 ？ 简化路由管理：

  • GetX使用GetMaterialApp代替MaterialApp，并通过getPages属性定义路由。无需显式声明Navigator，简化了路由管理。

  导航更简洁：

  • 使用Get.toNamed和Get.back等方法进行导航，无需上下文（context），代码更简洁。

  状态管理和依赖注入：

  • GetX集成了状态管理和依赖注入功能，可以在路由中方便地管理和传递状态。

  更多功能：

  • GetX提供了更多功能，如中间件、全局状态管理、响应式编程等，增强了Flutter应用的灵活性和可维护性。

9. 什么是flutter中BuildContext？

  答：BuildContext是flutter中的一个对象，它代表了一个Widget在Widget树中的位置。它提供了从当前Widget向上查找父级Widget的能力

10. 如何在flutter中处理异步操作？解释下flutter和Stream的使用

   答：异步操作使用Dart中的‘Future’和Stream

   （1）Future：表示一个异步的计算结果，使用async和await处理

   （2）Stream：用于处理多个异步事件，类似Dart中的事件流

11. 如何优化flutter应用的性能？

   （1）使用const构造函数创建不可变得widget

   （2）使用listView.builder和GridView.builder创建长列表，设置 cacheExtent 提前加载部分项目，避免滚动时卡顿，使用

   FadeInImage 或 cached\_network\_image 库优化图片加载

   （3）使用惰性加载(Lazy loading)

   （4）避免重复构建，使用缓存和局部更新

   （5）使用Profiler工具进行性能分析

   （6） 使用 RepaintBoundary：将需要频繁重绘的部分包裹在 RepaintBoundary 中，减少重绘范围。

   （7） 布局缓存：使用 AutomaticKeepAliveClientMixin 保持列表项的状态，避免重建

   （8）使用 compute 函数：将耗时操作移到后台 Isolate 执行，避免阻塞主线程。

12. 解释下flutter中的RenderObject

   答：RenderObject是渲染树中的基本元素，用于描述布局Widget，它负责计算Widget的尺寸和位置，以及绘制到屏幕上

13. 什么是flutter中的keys，什么时候需要使用它们？

   （1）作用和重要性

   标识唯一性：Key 提供了一种方式来唯一标识 widget。当 widget 在每次构建过程中需要保持状态时，使用 Key

   可以帮助 Flutter 确定哪些 widget 是新的，哪些 widget 是重用的。

状态保持：在使用状态管理的 widget（如 StatefulWidget）时，Key 能帮助 Flutter 正确地关联当前 widget

和其状态对象。这在动态列表或拥有复杂交互的 UI 中尤其重要。

控制 widget 更新：在有些情况下，即使两个 widget 从外观上看是相同的，我们也可能希望它们被视为完全独立的，以避免由于错误的

widget 重用导致的问题，如列表中错误的数据显示。在这种情况下，可以通过为每个 widget 实例分配一个唯一的

Key 来确保它们的独立性。

（2）Key 的类型

Flutter 提供了几种不同类型的 Key，以适应不同的场景：

ValueKey：一个简单的 Key，基于 Dart 中的 == 运算符。如果你有一个简单的标识符，如字符串或数字，可以使用

ValueKey。

UniqueKey：每次创建时都会生成一个唯一的 Key。当你需要确保 widget 每次都不被框架重用时使用。

GlobalKey：不仅可以唯一标识一个 widget，还允许在 widget 树的任何位置访问该 widget 的状态。这对于需要跨

widget 访问状态或全局控制某些数据的情况非常有用。

14. 如何在flutter中进行本地化（Localization）？

   （1）使用‘intl’包

   （2）创建‘arb’文件存储翻译字符串

   （3）MaterialApp的localizationsDelegates和supportedlocales。

   （4）使用“localizations”类和“Intl.message”获取翻译的字符串

15. 如何在Flutter中使用平台通道（Platform Channels）与原生代码进行交互？有哪些channels 方法？

   答：Method Channels、Event Channels和Basic Message Channels。

   ◦ Method Channels（方法通道）：最常用的通信方式，允许Flutter和原生代码之间进行同步方法调用。用途：

   ◦ 用于同步的请求-响应模式的通信。

   ◦ 适用于一次性调用方法并获取结果的场景。

   ◦ 工作原理：

   ◦ Flutter端调用原生方法，并等待返回结果。

   ◦ 通信是同步的，意味着Flutter端会等待原生方法执行完毕并返回结果后再继续执行后续代码。

   ◦ 性能特点：

   ◦ 延迟：每次调用都需要等待原生方法执行完毕，可能会有一定的延迟。

   ◦ 资源消耗：每次调用都是一次完整的请求-响应过程，资源消耗相对较高。

   ◦ 适用场景：适用于不需要频繁调用的操作，如获取设备信息、调用系统功能等。

   ◦

   ◦ Event Channels（事件通道）：用于从原生端向Flutter发送异步事件流。用途：

   ◦ 用于异步的事件流模式的通信。

   ◦ 适用于持续发送数据的场景，如传感器数据、实时位置更新等。

   ◦ 工作原理：

   ◦ 原生平台持续向Flutter发送事件，Flutter端订阅这些事件并处理。

   ◦ 通信是异步的，Flutter端通过流（Stream）来处理事件数据。

   ◦ 性能特点：

   ◦ 延迟：事件流模式，延迟较低，因为事件是持续发送的，Flutter端可以立即处理。

   ◦ 资源消耗：一次性建立连接后，持续发送事件，资源消耗较低，适合高频率的数据传输。

   ◦ 适用场景：适用于需要频繁更新数据的操作，如实时数据更新、传感器数据流等

   ◦

   ◦ Basic Message Channels（基本消息通道）：用途：

   ◦ 适用于需要双向通信的场景。

   ◦ 可以传递任意的二进制数据或字符串数据。

   ◦ 工作原理：

   ◦ BasicMessageChannel是通过消息传递的机制，Flutter和原生平台可以相互发送消息并接收响应。

   ◦ 支持任意类型的数据，只要能序列化为二进制格式或字符串格式。

   ◦

16. 介绍一下flutter boost：

   答：Flutter\_boost是一个用于Flutter与原生页面混合开发的解决方案，它允许Flutter与原生页面之间的无缝跳转，并支持在一个应用中同时使用Flutter和原生页面。核心特点

   无缝导航：

支持 Flutter 页面与原生页面之间的无缝跳转，用户体验流畅。

支持原生页面打开 Flutter 页面，Flutter 页面打开原生页面。

单引擎架构：

采用单引擎架构，整个应用中共享一个 Flutter 引擎，避免了多引擎带来的资源消耗和性能问题。

路由管理：

提供统一的路由管理机制，可以在原生和 Flutter 之间共享路由信息。

高效的页面切换：

利用 Flutter 的特性，提升页面切换的速度和流畅度，减少白屏和闪烁问题。

优缺点：

优点

无缝导航：支持原生和 Flutter 页面之间无缝跳转，用户体验流畅。

单引擎架构：节省资源，提高性能，避免了多引擎带来的额外开销。

灵活性高：支持复杂场景下的页面导航和状态管理。

缺点

复杂性：相对于纯 Flutter 或纯原生项目，配置和管理更为复杂。

学习曲线：需要一定的学习成本来掌握 Flutter Boost 的使用和配置方法

常见问题

（1）. 页面跳转动画问题

问题：

从原生页面跳转到 Flutter 页面时，过渡动画不如预期。

解决方法：

确保在 Android 端使用 overridePendingTransition 方法，并在 iOS 端使用自定义过渡动画。参见之前的示例代码。

（2）. 状态管理问题

问题：

Flutter 页面和原生页面之间的数据传递和状态同步可能会变得复杂。

解决方法：

使用 Flutter Boost 的页面参数传递功能，或者使用全局状态管理工具（如 GetX、Provider）来管理状态。

（3）. 页面生命周期问题

问题：

在 Flutter 页面与原生页面之间跳转时，可能会出现页面生命周期（如 initState、dispose）未按预期触发的问题。

解决方法：

确保在原生代码中正确处理页面的生命周期方法。使用 Flutter Boost 提供的生命周期回调接口来管理页面的生命周期。

（4）. 路由管理问题

问题：

在使用 Flutter Boost 时，原生路由和 Flutter 路由的管理可能会变得混乱。

解决方法：

定义清晰的路由规则，并确保在原生端和 Flutter 端都使用统一的路由管理方式。可以使用 Flutter Boost 的

BoostNavigator 进行统一的路由管理。

（5）. 引擎初始化问题

问题：

在某些情况下，Flutter 引擎初始化可能会失败，导致页面无法正常显示。

解决方法：

确保在应用启动时正确初始化 Flutter 引擎，并在需要时重新初始化。检查 Flutter Boost 的初始化代码是否正确配置。

（6）. 性能问题

问题：

在高频次页面切换或复杂页面渲染时，可能会遇到性能问题。

解决方法：

优化页面渲染逻辑，避免不必要的重绘。使用 Flutter DevTools 进行性能调优。确保 Flutter Boost

的版本是最新的，并且已经解决了已知的性能问题。

（7）. 平台通道通信问题

问题：

在 Flutter 和原生之间使用平台通道（如 MethodChannel、EventChannel）进行通信时，可能会遇到数据丢失或通信失败的问题。

解决方法：

确保通信接口的实现是正确的，并且在通信时处理好所有可能的异常情况。使用异步方法来处理通信，以避免阻塞 UI

线程。

（8）. 混合开发调试问题

问题：

在进行 Flutter 和原生混合开发时，调试可能会变得复杂。

解决方法：

熟悉 Flutter 和原生开发的调试工具。使用 Android Studio 或 Xcode

进行联合调试。对于复杂的通信和状态同步问题，可以使用日志和调试断点来排查问题。

17. 解释一下flutter线程模型

   答：Flutter 使用单线程模型，主要由事件队列和微任务队列组成，

   Dart 事件循环的基本原理

   Dart 使用单线程模型，通过事件循环来处理所有任务。事件循环的工作方式如下：

微任务队列 (Microtask queue)

：微任务队列是另一种任务队列，用于处理微任务（microtask）。微任务通常是那些需要快速执行的小任务，它们的执行优先级高于事件队列中的任务。微任务包括：

Future 完成时触发的 .then、.catchError、.whenComplete 回调

通过 scheduleMicrotask() 显式加入微任务队列的任务

当当前执行堆栈为空时，Dart 首先会检查微任务队列。如果微任务队列中有任务，Dart

会执行这些任务直到微任务队列为空。

事件队列 (Event queue)：事件队列是用于存放异步任务的队列。这些任务包括但不限于：

I/O 事件（如文件读写操作、网络请求响应）

定时器事件（如 Timer.run）

用户交互事件（如鼠标点击、键盘事件）只有当微任务队列为空时，Dart 才会从事件队列中取出事件进行处理

Future 与微任务

当你创建一个 Future 并使用 then、catchError 或 whenComplete 添加回调时，这些回调被放入微任务队列中。

如果 Future 已经完成，它的回调会在当前事件循环迭代或者下一个事件循环迭代中尽快被执行（取决于 Future

完成的时机）。

解释 async 和 await

async: 当你将 async 添加到函数声明时，这意味着该函数将返回一个 Future。这是 Dart

的一种处理异步操作的方式。函数体内的代码依然按顺序执行，但整个函数的执行不会阻塞其他操作。

await: await 关键字只能在 async 函数内使用。它暂停当前异步函数的进一步执行，直到等待的 Future

完成，然后继续执行 await 后面的代码。这种等待是非阻塞的，即它允许 Dart 的事件循环继续处理其他事件，如 UI

更新或数据响应。

非阻塞性

当你在 async 函数中使用 await 时，如果 Future 尚未解决，函数的执行会在那点暂停（这里的“暂停”是非阻塞的），允许

CPU 去执行其他任务，比如绘制 UI、响应用户输入等。一旦 Future 完成，函数会恢复执行，从 await

表达式停下的地方继续。这样的处理确保了主 UI 线程不会因为等待 Future 完成而阻塞，从而避免了应用 UI

的冻结或卡顿。

18. Widget 树、Element 树和 Render 树

   答：1. Widget 树 (Widget Tree)

   作用：

Widget 树是 Flutter 应用中最高层的抽象，它描述了界面的结构。Widget 树中的每一个节点都是一个 Widget

对象，代表了界面的一部分。Widgets 是不可变的，意味着它们一旦被创建就不能改变。

Widgets 定义了界面应该如何看起来（它的配置信息），但并不直接参与渲染过程。

示例：

Text、Row、Column、Container 等。

2. Element 树 (Element Tree)

  作用：

Element 树是 Widget 树的实例化，可以理解为 Widget 树的具体实现。当 Widget 树发生变化时，Flutter 框架会更新

Element 树以反映这些变化。

每一个 Widget 在 Element 树中有一个对应的 Element。Element 对象是持久的，它们代表了 Widget

在界面上的位置。Element 负责生命周期管理，包括从 Widget 树获取配置并使用这些配置来构建和更新 Render 树。

示例：

当你创建一个 Container widget 时，相应的 Element 负责管理这个 Container 的生命周期和状态。

3. Render 树 (Render Tree)

  作用：

Render 树负责具体的渲染工作。每个 Render 对象都有一个关联的 Element，但并不是每个 Element 都有一个

Render 对象。

Render 对象知道如何将自己绘制到屏幕上，它们处理实际的像素渲染、布局计算和大小调整等。

示例：

对于 Container Widget，对应的 Render 对象会处理实际的布局和绘制工作，例如渲染边框、填充、背景色等。

对应关系

每一个 Widget 都能创建一个或多个 Element。例如，状态无关的 Widget（如 Text）通常对应一个 Element，而状态相关的

Widget（如 StatefulWidget）则可能对应更复杂的 Element 结构。

每个 Element 对象可以关联一个 Render 对象（如果这个 Widget 是可视化的）。Element 对象负责根据 Widget

的配置创建和更新 Render 对象。

更新流程

当 Widget 树的结构或 Widget 的配置发生变化时，Flutter 会重建 Widget 树，并相应地更新 Element 树和

Render 树。

Element 树的更新会遵循 Widget 树的指引，创建、更新或销毁 Element。

Element 树的变更会导致 Render 树的相应更新，这包括重新布局和重新绘制界面。

19. Widget 重建，Element会怎么更新？

   答：Element 更新过程

   Widget 重建：

开发者在应用中更改了状态或配置，导致 Flutter 需要重建 Widget。这通常发生在调用 setState、触发事件处理器或父

Widget 传入新的构建参数时。

框架比较 Widget 和 Element：

当 Widget 树中的某部分由于数据变化而需要重建时，Flutter 框架会对新的 Widget 树和现有的 Element 树进行比较。

框架会为新的 Widget 树中的每个 Widget 查找相应的 Element。

更新 Element：

如果新 Widget 的类型和旧 Widget 的类型相同，并且 Key 相同（如果指定了的话），则 Flutter 会更新现有的

Element，而不是创建一个新的 Element。这个过程称为 更新 (update)。

在更新过程中，Element 会从新的 Widget 中获取配置，并适当调整其内部状态和属性。

重用 Element：

如果找到与新 Widget 类型匹配且 Key 也匹配的现有 Element，该 Element 将被重用。

这种重用避免了不必要的 UI 重构和渲染过程，提高了性能。

移除和添加 Element：

如果新的 Widget 在当前的 Element 树中没有对应的 Element（即它是新添加的或之前的 Element

已经不匹配），则会创建一个新的 Element。

如果某些现有的 Element 在新的 Widget 树中没有对应的 Widget，这些 Element 将会被从树中移除。

调用 Element 生命周期方法：

在 Element 更新过程中，会调用相关的生命周期方法，如 initState(), didUpdateWidget(), build(), 和

dispose() 等。

这些方法允许 Element 在关键阶段执行必要的逻辑，比如在构建前后做一些准备或清理工作。

20.

Flutter手势冲突有了解吗？Gesture包裹container，container包裹包含文字的text，container给的区域大于text，点击哪里会响应事件。回答：container假如有背景颜色就是container响应。假如没有，就是text。为什么?

答：在 Flutter 中，手势冲突和响应是由框架的命中测试（hit testing）逻辑决定的。这个逻辑定义了当用户触摸屏幕时，哪个

widget 应该接收和处理该触摸事件。理解这一机制有助于解释在不同的布局和样式配置下，为什么某些部分的点击会或不会触发预期的响应。

命中测试（Hit Testing）

在 Flutter 中，当一个触摸事件（如点击）发生时，Flutter 会进行命中测试来确定触摸点位于哪个 widget

上。这个测试是从 widget 树的根节点开始，沿着每个节点向下进行，直到找到最具体的 widget。哪个 widget

响应触摸事件取决于几个因素，包括：

Widget 的可见性：如果一个 widget 是不可见的（如透明），它通常不会参与命中测试，除非特别指定。

Widget 的大小和位置：触摸点必须在 widget 的边界内，widget 才能被命中。

hitTestBehavior 属性：特别是对于具有子 widget 的容器类 widget（如 GestureDetector、Stack 等），它们的

hitTestBehavior 属性决定了命中测试的行为。

您的场景

在您描述的情况中，有一个 Container 包裹了一个包含文字的 Text widget，并且 Container 有一个

GestureDetector。关于哪里会响应事件，依赖于以下因素：

背景色：在 Flutter 中，如果 Container 设置了背景色，即使这个颜色是透明的 (Colors.transparent)

，它也会在命中测试中被视为一个具体的绘制区域。因此，整个 Container 的区域，包括超出 Text widget

的部分，都能响应触摸事件。

无背景色：如果 Container 没有设置背景色，且没有其他显示效果（如边框），那么在命中测试中它通常会被视为不存在。这种情况下，只有

Text widget 才会接收和响应触摸事件，因为它是该区域唯一可视的内容。

为什么这样设计

这种设计允许开发者更精确地控制哪些部分的界面应该是可交互的。通过简单地调整背景色或使用 Material

widget（它自动响应命中测试），开发者可以确保触摸事件被正确地捕捉和处理。

21. const,final,var 的区别

   答：const：

编译时常量： const 用于声明编译时常量，即在编译时就已经确定了值的变量。

不可变对象： 用 const 声明的对象在程序运行期间是不可变的，这意味着它们的值在运行时不能改变。

适用场景： 适用于值在编译时就已知且不会改变的变量。

常量构造函数： const 可以用来创建常量构造函数实例，常量构造函数创建的对象在内存中是共享的。

标识符： const 可以用于修饰顶级变量、类变量和局部变量

final ：

运行时常量： final 用于声明运行时常量，即在运行时第一次被赋值后值就不能再改变。

单次赋值： 用 final 声明的变量必须在声明时或者构造函数中被初始化，并且只能被赋值一次。

适用场景： 适用于在运行时确定且值不会改变的变量。

标识符： final 也可以用于修饰顶级变量、类变量和局部变量。

主要区别

赋值时机：

const 的值在编译时就确定，必须是编译时常量。

final 的值在运行时第一次赋值后确定，可以是运行时常量。

使用场景：

const 适用于那些在编译时就已知且不会改变的值。

final 适用于那些在运行时确定后不会再改变的值。

对象的不可变性：

const 声明的对象是完全不可变的，整个对象及其内容都不可改变。

final 声明的对象引用不可变，但对象的内容如果是可变类型，可以改变其内部状态。

例如：

final List<int> finalList = [1, 2, 3];

finalList[0] = 10; // 合法，修改列表内容

// finalList = [4, 5, 6]; // 非法，重新赋值列表引用

const List<int> constList = [1, 2, 3];

// constList[0] = 10; // 非法，无法修改常量列表内容

22. dart 内存管理机制

   答：Dart 使用垃圾收集（Garbage Collection, GC）机制自动管理内存。GC 会自动回收不再使用的内存，但有些情况还是可能导致内存泄漏：

长时间存活的对象引用：

如果某个对象被一个长期存在的对象引用，它就不会被 GC 回收。常见的情况是静态变量、全局变量或生命周期较长的对象持有对其他对象的引用。

闭包（Closures）：

闭包会捕获其作用域内的变量。如果这些变量占用大量内存，并且闭包长时间存活或被错误地保留，就可能导致内存泄漏。

订阅（Subscriptions）和监听器（Listeners）：

如果 StreamSubscription 或事件监听器没有正确取消，可能会导致内存泄漏，因为这些订阅或监听器会保持对回调函数的引用。

复杂数据结构：

复杂的数据结构，如链表、树等，如果存在循环引用或没有正确处理，可能会导致内存泄漏。

常见的内存泄漏示例及避免方法

1.避免未取消的订阅和监听器

2.避免长时间存活的对象持有不必要的引用

3.正确处理闭包中的变量 // 错误示例：闭包捕获大量内存变量

void loadData() {

List<int> largeData = [/\* ... 大量数据 ... \*/];

someStream.listen((event) {

print(largeData); // 闭包持有 largeData 的引用，可能导致内存泄漏

});

}

// 解决方案：避免在闭包中捕获大对象，或确保及时取消闭包

Dart 的内存管理机制

垃圾收集（Garbage Collection, GC）

Dart 使用垃圾收集器来自动管理内存。Dart

的垃圾收集器主要采用了标记-清除（Mark-and-Sweep）和标记-压缩（Mark-and-Compact）算法。其主要特点如下：

自动管理内存：Dart 的 GC 会自动跟踪对象的生命周期，回收不再使用的内存，从而避免了内存泄漏和手动内存管理的复杂性。

标记-清除：GC 在发现不再使用的对象时，会标记这些对象，然后清除它们以释放内存。

标记-压缩：在清除对象后，GC 还会进行内存压缩，以减少内存碎片，提高内存使用效率。

23. Flutter 渲染流程是什么？（GPU）

Flutter只关心向 GPU提供视图数据，GPU的 VSync信号同步到 UI线程，UI线程使用 Dart来构建抽象的视图结构，这份数据结构在

GPU线程进行图层合成，视图数据提供给 Skia引擎渲染为 GPU数据，这些数据通过 OpenGL或者 Vulkan提供给 GPU。

https://p3-juejin.byteimg.com/tos-cn-i-k3u1fbpfcp/2999bc85511645e89478ec74bb32948c~tplv-k3u1fbpfcp-jj-mark:3024:0:0:0:q75.awebp#?w=1200&h=675&s=98737&e=png&b=fcfafa

24. flutter生命周期，setstate会执行哪些生命周期？

   setState()执行后，会执行build()

app的状态：AppLifecycleState

inactive：活跃可见

paused：关闭或者切换到后台时，不可见的状态

hidden：后台运行状态

resumed：切回到前台可见状态

detached：关闭状态

Flutter SDK 3.13 之前的方式: with WidgetsBindingObserver

在initState()中注册 WidgetsBinding.instance.addObserver(this);

在dispose()移除 WidgetsBinding.instance.removeObserver(this);

在didChangeAppLifecycleState()回调中，检测app的状态

Flutter SDK 3.13 之后的方式：AppLifecycleListener

25. 如何获取控件的大小和位置？

   创建 GlobalKey：

创建一个 GlobalKey 实例，并将其赋给一个变量（如 \_key）。

将 GlobalKey 赋给 Widget：

在 build 方法中，将 GlobalKey 赋给你想要获取信息的 widget 的 key 属性。在这个例子中，我们将 \_key 赋给了一个

Container。

获取 widget 信息：

通过 GlobalKey 的 currentContext 属性，可以获取到 BuildContext。

使用 findRenderObject 方法，可以从 BuildContext 获取到 RenderBox 对象。

RenderBox 提供了 size 属性，可以获取 widget 的大小。

RenderBox 提供了 localToGlobal 方法，可以将本地坐标转换为全局坐标，从而获取 widget 在屏幕上的位置。

26. 说一下 mixin

   首先mixin是一个定义类的关键字。直译出来是混入，混合的意思 Dart为了支持多重继承，

   引入了mixin关键字，它最大的特殊处在于： mixin定义的类不能有构造方法，这样可以避

27. flutter开发中遇到了哪些比较棘手的问题，你是怎么解决的?

   性能问题：Flutter应用可能会出现性能瓶颈，例如卡顿、动画不流畅等。解决方法包括使用Flutter的性能工具分析性能问题、减少UI重建的次数、优化布局和渲染等。

设备兼容性问题：由于Flutter跨平台的特性，不同设备上的兼容性问题是常见的。解决方法包括使用平台特定的代码、适配屏幕尺寸和分辨率、处理不同平台的API差异等。

第三方库的问题：Flutter生态系统非常丰富，但有时候可能会遇到不稳定或不兼容的第三方库。解决方法包括查找替代库、修复或改进第三方库的问题、自己实现功能等。

调试问题：在开发过程中，可能会遇到难以调试的问题，例如UI显示异常、逻辑错误等。解决方法包括使用调试工具、打印日志、逐步调试等。

动态UI的复杂性：Flutter的动态UI能力非常强大，但也带来了一些复杂性。解决方法包括使用状态管理库（如Provider、GetX、Bloc）来管理UI状态、封装可复用的小部件、遵循单一职责原则等

28. 介绍下FFlutter的FrameWork层和Engine层，以及它们的作用

   Flutter的FrameWork层是用Drat编写的框架（SDK），它实现了一套基础库，包含Material（Android风格UI）和Cupertino（iOS风格）的UI界面，下面是通用的Widgets（组件），之后是一些动画、绘制、渲染、手势库等。这个纯

   Dart实现的 SDK被封装为了一个叫作 dart:ui的 Dart库。我们在使用 Flutter写 App的时候，直接导入这个库即可使用组件等功能。

   Flutter的Engine层是Skia 2D的绘图引擎库，其前身是一个向量绘图软件，Chrome和 Android均采用

   Skia作为绘图引擎。Skia提供了非常友好的 API，并且在图形转换、文字渲染、位图渲染方面都提供了友好、高效的表现。Skia是跨平台的，所以可以被嵌入到

   Flutter的 iOS SDK中，而不用去研究 iOS闭源的 Core Graphics / Core Animation。Android自带了 Skia，所以

   Flutter Android SDK要比 iOS SDK小很多。

29. flutter 如何监听widget build 完成

（1）.使用 addPostFrameCallback

通过 WidgetsBinding.instance.addPostFrameCallback 方法，可以在当前帧绘制完成后执行某个回调。这种方法适用于需要在 build 方法完成后立即执行某些操作的情况。

(2)使用 Future.microtask 将操作调度到当前事件循环的末尾

 @override Widget build(BuildContext context) { // 在 build 方法中调度 microtask 回调 Future.microtask(() { \_onBuildCompleted(); });}

30 **Flutter 混合架构方案探索**

https://juejin.cn/post/7262616799219482681

面试题：

https://juejin.cn/post/7371786399442419738?searchId=20240712164911C79FCAB550CBB6494C6F