プロバイダーパッケージについてすべて学習したので、今度はそれをアプリで実際に実行して、保存されているタスクを読み取ったり、新しいタスクを追加したり、タスクをチェックオフしたり、タスクを削除したりできます。ユーザーがそれを望んだとき。

したがって、プロバイダーを実践するこの最初の部分では、タスクを取得し、タスク画面とタスクリストの両方でそれを使用できるようにすることを最初の目標に取り組みます。

基本的に、タスクプロバイダーの状態を高く保ち、ツリー全体で使用できるようにします。また、そのデータをTaskScreenとTasks Listの両方に提供します。タスクデータの変更をリッスンします。タスク画面で必要な理由は、12個のタスクと表示されているこの小さな部分があるためです。

この数は動的で、現在To Doリスト内にあるすべてのタスクを反映している必要があります。そして、タスクリストについても、それは一目瞭然です。

このタスクリストを実際にレンダリングできるように、そこで利用可能なタスクデータが必要です。

ここで、プロバイダーを実装する次のレッスンで、最初に必要な動作を正確に説明します。次に、それをチャレンジとして完了するか、単に私と一緒に進むかを選択できます。

しかし、自分で実際に試してみることをお勧めします。行き詰まって戻ってきた場合でも、ただ単に従うだけでなく、もっと多くのことを学びます。

したがって、この場合は、変更通知プロバイダーを使用して、ウィジェットツリーのルートの一番上のアプリの最上部にタスクデータを提供する必要があります。

そして、それをタスク画面内でリッスンしてその番号を更新することも、タスクリスト内でタスクタイルのリストを更新することもできます。

そのため、既にしたら、現在のTodoeyアプリプロジェクトを開いて、この目標を達成してください。

だから私はあなたがそれを試してみて、あなたが解決策をチェックするためだけにここにいることを願っています

同じことを行う方法はたくさんあることを覚えておいてください。さまざまな方法を使用してまったく同じ動作を実現できますが、私の解決策はそのための1つの方法にすぎません。

したがって、最初に行う必要があるのは、もちろんpubspec.yamlに移動して、プロバイダーを追加することです。

依存関係として。

これをここで実行し、プロバイダー関連のすべてのファイルを取得して、使用できるようにします。

最初にやるのは、モデルフォルダー内に新しいDartファイルを作成することです。

これをtask\_dataと呼びます。

これが、ツリー全体に提供されるオブジェクトとなり、タスクのデータを利用できるようになります。

このクラスは単にTaskDataと呼ばれ、変更通知クラスを拡張します。

現時点でエラーが発生しているのは、foundation.dartパッケージをインポートできる必要があるか、またはmaterial.dartパッケージをインポートできるためです。

ただし、material.dartパッケージには実際にはFoundation.dartパッケージが含まれているため、軽量にしたい場合は、Foundation.dartを使用するだけで済みます。

これで、foundation.dart内で変更通知を見つけることができるようになり、それを使用してタスクデータクラスを拡張できるため、このクラスを他のオブジェクトが変更をリッスンできるようにしたり、指示したときに更新したりできるようになります。 。

したがって、最初に行うのは、他のモデルクラスであるtask.dartをインポートすることです。これは、基本的には、タスクオブジェクトの構造です。そして今、自分のタスクscreen.dartに移動すると、現在自分のタスクデータが存在する場所を見つけることができます。それは私のTasksScreenState内のこのタスクリスト内にあります。

これをすぐに解除するため、ここから切り取り、タスクデータオブジェクトに移動します。

ここに貼り付けます。

まったく同じ方法で作成されます。タスクには名前があり、デフォルトでは「未完了」に設定されています。これで、アプリ全体でこのタスクプロパティを提供する準備ができました。そして、それをウィジェットツリーの最上部で実行します。

それで、MyAppクラス内でそれを行い、この変更通知プロバイダクラスでマテリアルアプリ内にあるすべてのものをラップします。そしてもちろん、それを使用するには、provider.dartパッケージをインポートする必要があります。

そのため、プロバイダーは、後で分岐する残りのウィジェットから見えるように十分に高いツリー内の場所にある必要があることを覚えておいてください。

したがって、私たちの場合、それはMyAppクラスの一番上に配置することを意味します。

ここで、タスクタイルの内側だけでデータが必要な場合は、1つ上のレベルの親に移動して、その親をラップできます。このタスクリストはもちろんここにあり、ラップできます。そのすべての子に提供できるようにするためのプロバイダーウィジェットがあります。

しかし、これを間違ってツリーの低すぎる位置に配置すると、子供がアクセスできなくなる可能性があり、アプリを実行してエラーが発生し、正しい位置に配置されていないことを通知しようとします。ウィジェットツリー内。

これがどこに行くかに注意してください。

私たちの場合は、マテリアルアプリの上にある必要があります。

必要な変更通知プロバイダのビルダーと呼ばれるパラメータがあるため、ここで少し警告が出ます。

だからこれは前に見た。ビルダーは現在のコンテキスト、つまりウィジェットツリー内の現在の位置を取得し、ツリー内のすべての子に提供する必要があるオブジェクトを返します。そして、そのオブジェクトはもちろん、新しく作成されたタスクデータオブジェクト上にあります。

それでは、タスクデータオブジェクトを提供し、modelsフォルダーからのこのタスクデータオブジェクトとtask\_data.dartについて説明します。

これで、ウィジェットツリーの最上部に変更通知プロバイダがあり、そこにあるデータをリッスンするツリーの下のすべての子にタスクデータオブジェクトを提供しています。これで、実際にここからデータ、つまりタスクリストを読み取ることができます。

したがって、最初に実行したい場所は、実際にはタスク画面です。ここには、リストに残っているタスクの数をユーザーに示すすべてのtasks.lengthがあります。これは、タスクリストに基づく必要があります。

ここに何個あるかを数え、補間された文字列としてここに提供する必要があります。

また、タスク画面の状態からタスクを削除したため、現在エラーが発生しています。

先に進み、tasks.lengthを削除し、以前に学習したものを使用してプロバイダーにアクセスし、.ofメソッドを呼び出します。

次に、Android Studioにコードの入力を手伝ってもらいます。

ただし、最初に、provider.dartをインポートする必要があります。

そして今、プロバイダーの.ofメソッドを利用して、もちろん常にメソッドの日付タイプを指定することができます。これは、再びインポートする必要がある私のタスクデータオブジェクトになります。

モデルを利用して、task\_data.dartを利用します。

最終的には、task.dartの必要性を取り除き、関連するすべてのものをタスクデータ内で管理します。

しかし、とりあえず先に進んで、このタスクデータオブジェクトに相当するこのオブジェクトをタップして、そこに移動してtasksプロパティを取得しましょう。したがって、.tasksを記述してから、現在3に等しいオブジェクトの数または長さを取得します。

したがって、最初の部分は、前のタスクリストにタスクを追加するこの部分ですが、現在は存在しないため、未定義です。

それでは、実際に先に進んで、設定状態をコメントアウトし、次のレッスンでタスクデータプロバイダーに新しいタスクを追加する方法に取り組むときに、この部分を理解します。

したがって、他の部分はもちろん、タスクリストにタスクが渡される場所です。

これはもちろん、タスクのdata.taskプロパティから読み取る必要がある2番目の場所ですが、それを渡してツリーをドリルする代わりに、タスクリストに移動して、代わりに、ここでそのタスクデータオブジェクトへの参照。

したがって、実際にはもう何も渡す必要はないので、これら2つは必要ありません。また、このtask.dartへの参照も必要ありません。

ただし、代わりに、provider.dartとモデルのフォルダーtask\_data.dartもインポートします。タスクリスト内で使用して、リストビューを作成します。

ここで、ウィジェットに渡されたプロパティを利用するwidget.tasksを使用する代わりに、

タスクリストのステートフルウィジェットではなく、単にprovider.ofメソッドを利用します。したがって、widget.tasks。[index] .nameの代わりに、widget.tasksをprovider.ofに置き換え、タスクデータであるデータ型を指定します。そして、それはprovider.of.tasks [index] .nameになります。

これをコピーして、これらすべての場所でウィジェットを置き換えることができます。

そして今のところは、実際に設定されている状態をコメントアウトして、チェックボックスがオンになったときに何が起こるかを処理します。

これについては、後のレッスンで扱います。

しかし、現在必要なのは、プロバイダーパッケージを使用して提供されているタスクデータ内のタスク内のデータを読み取ることができることだけです。

では、ステートフルウィジェットからのウィジェットをもう使用していないことに注目してください。

これが、削除したプロパティにアクセスする方法です。

したがって、実際には、これをステートフルにする必要はありません。

実際には、それをより単純なステートレスウィジェットに変えることができます。コンストラクターを介してタスク画面から渡さなくなったため、プロバイダーに追いつき、必要なデータを取得できます。

タスク画面内で状態を必要としないことにも注目してください。これらの行をすべて削除し、これをステートレスウィジェットに変更することで、これをステートレスウィジェットに変換し、未使用のimortをここで削除できます。上手。

これらのメモリレスステートレスウィジェットは、必要なときに構築され、更新されたときに破棄されて新しいウィジェットに置き換えられるため、アプリは実際にははるかに単純になります。

タスクのデータを提供するために、ここでタスクデータのプロバイダーを利用してそのリストを取得し、タスク画面内でそれを使用して、残っているタスクの数を示します。

そして、これは単にタスクを読み取るだけです。

保存して、アプリをゼロから実行してみましょう。そして、うまくいけば、すべてがうまくいき、すべてのコードが適切な場所にあることがわかります。以前とまったく同じユーザーインターフェイスを使用する必要があります。しかし今回は、このデータはもはや渡されず、ツリーをドリルダウンされ、代わりにプロバイダーから提供されています。

それで、あなたが自分でそれに取り組むことを試みたとき、あなたはなんとかこの時点に到達しましたか？

まあ、変更通知を拡張するタスクデータクラスの作成や、そのタスクデータオブジェクトを提供する変更通知プロバイダ内でのマテリアルアプリのラップ、最後にprovider.ofタスクデータを使用したタップによるデータデータの使用など、いくつかのことを行いました。タスク画面とタスクリストの両方に入れます。それは今はうまくいきますが、実際にはもっと簡単にすることができます。また、provider.dartから提供されるコンシューマウィジェットと呼ばれるものを使用してこれを行うこともできます。ここで、3つの場所でprovider.ofタスクdata.tasksをどのように使用しているかに注目してください。

少し反復的になっていますか？

したがって、provider.ofを3か所で呼び出す代わりに、このタスクプロパティがコンシューマーウィジェットと呼ばれるものの内部で変更されたときに更新する必要があるすべてのダウンストリームウィジェットをラップするだけで済みます。

タスクタイル内でprovider.ofをどのように使用しているのか、アイテムカウント内でも使用していることに注意してください。このリストで、すでにセット状態に遭遇していることを考えれば、それを使用する方法がわかります。ローカル状態を非常に適切に制御および管理するために、そしてもちろんそれをステートフルウィジェットと組み合わせて使用する必要があります。ただし、ステートフルウィジェットはステートレスウィジェットよりもはるかに複雑であるため、1つのステートフルウィジェットには実際にはメモリが必要です。

ステートレスウィジェットは操作がはるかに簡単であり、作成、破棄、再構築できるだけなので、状態を変更できるように、その状態を覚えておく必要があります。

これで、以前にコールバックの組み合わせを使用して状態を上げることと、ウィジェットツリーで状態を上げることもわかりました。状態を管理することができました。しかしもちろん、アプリがより複雑になり、ツリー全体をトラバースするコールバックが必要な場合や、状態を複数のレベルに渡す必要がある場合は、いくつかの問題が発生する可能性があります。

そのため、現在、状態を管理する最も柔軟な方法は、プロバイダーパッケージを使用することです。

これは現在、Google Flutterチームからの推奨事項です。

しかし、私が言いたいのは、これがFlutterアプリで状態を管理するための公式に推奨されている方法であっても、それが唯一の方法ではないことを覚えておいてください。

多くの場合、別の方法が必要な場合や、状態を管理するこれらの異なる方法の組み合わせが必要な場合があります。

そしてそれは間違いではありません。

それは実際にはユースケースに依存します。つまり、要件、コードの編成方法に対する特定のスタイルの感覚、そしてFlutterのフレームワークの成熟度にも依存します。

しかし、プロバイダーは本当に素晴らしいツールなので、話しましょう。

これは、コミュニティによって実際に作成されたパッケージです。

グーグルにも同様のパッケージを提供するプロビッドという似たようなパッケージがありましたが、それに伴い、この人のレミが実際に優れていると思われるバージョンを作成することにしました。そのため、「Google Flutterチーム」への全面的な貢献により、彼らはこのパッケージの潜在能力に気づき、実際に彼らの全力を注いでいます。

そして今、それは公式に推奨されているだけでなく、レミ自身とGoogle Flutterチームの両方によってもサポートされています。

したがって、その潜在能力を最大限に発揮するために必要なすべてのリソースを備えています。

ここで、プロバイダーがなぜそれほど有用であるかを十分に理解するために、まず、何が問題であるかを検討する必要があります。

ですから、最初に私たちのコードから、あなたはすでにいくつかのことが毛深いことを理解しているかもしれません。

データをツリーに戻すためにこれらすべてのコールバックがあり、一連のステートフルウィジェットがあり、sets dateを使用して子ウィジェットのbuildメソッドを呼び出す必要があり、多くのものが常に再構築されています。そして、これはおそらく最も効率的な方法ではありません。

しかし、アプリが非常に大きくなり、状態を管理しているときに何が起こっているかを一目で確認するのが非常に難しいため、問題を少し詳しく説明できるように、本当にシンプルなアプリを作成しました。

したがって、この場合、最上位のクラスはMyAppで、2つの子があり、そのうちの1つはレベル1と呼ばれるものです。次に、レベル1はレベル2と呼ばれる別のクラスを作成し、レベル2はレベル3を作成します。

したがって、そのコードは次のようになります。これはかなり単純です。重要な部分は、レベル1クラスを作成するMyAppクラスがあり、次にレベル2およびレベル3クラスをこの順序で作成することです。

これがレベル1を作成し、次にレベル1がレベル2、レベル2がレベル3を作成するマテリアルアプリです。

私たちは基本的にこの構造を正確に持っています。

今ここに問題があります。

こことここで使用する必要があるプロパティがあるとしましょう。そのためには、状態を持ち上げて、アプリ内に置く必要があります。

これで問題はなくなりましたが、そのプロパティまたはそのプロパティの状態をレベル3に渡すと、レベル1、レベル2、およびレベル3を通過する必要があります。つまり、そのデータは実際にそれを必要としないこれらの2つのクラス内で更新されます。

したがって、下位レベルに到達できるようにするために、基本的にツリーをドリルダウンする必要があります。そして、リアクティブプログラミングでは、これはプロップドリルと呼ばれるものであり、プロパティを渡すためにツリーをドリルします。これを実際のコードで見てみましょう。

単にデータと呼ばれる文字列があり、それが「トップシークレットデータ」であるとします。

これは両方のAppBar内で使用する必要があるため、タイトル内でテキストウィジェットとして使用され、そのデータが含まれます。ただし、ここではレベル3でも使用する必要があるため、このレベル3のテキストウィジェットにもデータが必要です。しかし、それをレベル3に下げるには、すべてのレベルをドリルスルーする必要があります。

つまり、レベル1の内部では、データを呼び出すだけのプロパティが必要であり、レベル1を初期化するときは、このデータプロパティを渡して初期化します。次に、コンストラクターを介してレベル1に渡し、データはレベル1内にあります。

しかし、それをレベル3に下げるためには、レベル2に渡さなければなりません。

それをレベル2に渡して、データと呼ぶ最後のStringプロパティを作成しましょう。次に、レベル2が初期化されるときに、このプロパティもthis.dataに渡します。

そして最後に、レベル3に渡すだけです。

もう一度、最終的な文字列データを作成し、レベル3を初期化すると、必要なデータの値が渡されます。

そして最後に、レベル3で必要なところにデータを用意し、テキストウィジェット内に実際に配置できるようにします。

では、保存をクリックしてアプリを実行すると、正しく機能していることがわかりますか？

最上位のウィジェットであるMyApp内にあるデータは、このデータプロパティをAppBarだけでなく、テキストウィジェットを持つレベル3ウィジェットにも渡すことができます。しかし、そのためには、データをまったく関係のない、まったく関係のないウィジェット全体に渡さなければなりませんでした。これは、レベル2で確認できるので、レベル1ではそのデータは必要ないためです。そのデータは必要ありません。レベル3にするために、このデータを通過させています。

だから、そこにいる怠惰なプログラマーにとって、これを見るのはかなり苦痛であり、複雑さも大幅に増大します。

わからない、ここにタイプミスがある、またはレベル1内に他のある種のデータがあったと言い、渡されたデータとそのローカルバージョンとの間で混乱します本当にきれいではありません。

では、代わりに何ができるでしょうか？

まあ、Providerパッケージがあれば、状態を一番上に移動できますが、状態をサブスクライブするだけのウィジェットを作成できます。したがって、レベル3のウィジェットに手を上げてもよいでしょう。つまり、状態の変化をリッスンするつもりです。そして、それが変化すると、通知され、レベル3に到達するためにレベル1とレベル2をたどる代わりに、レベル3ウィジェットにデータを渡すだけで、再構築レベルではなく、この更新された状態を実際に処理します。 1、レベル2、レベル3。

そのツリーを再構築して状態を渡し、実際にレベル3にするのではなく、レベル3に更新をリッスンさせるだけです。

では、プロバイダーを使用してこれらの優れた機能をすべて取得するにはどうすればよいでしょうか。

まず、アプリにプロバイダーパッケージを含める必要があります。

それでは、インストールに取り掛かって、依存関係としてプロバイダーを追加しましょう。

したがって、pubspec.yamlを開き、すべての依存関係が存在するクパチーノアイコンのすぐ下に、プロバイダーパッケージを追加し、パッケージを呼び出して、そのパッケージからコードを取得します。

これで、provider.dartファイルをインポートできるはずです。

コードにプロバイダーが追加されたので、それを使用できます。

最初に行う必要があるのは、データを公開することです。つまり、提供するデータは何でしょうか。そのためには、ご想像のとおり、フラッターランドにいるので、ウィジェットを使用して実現します。そして、AppBarとレベル3の両方にデータが必要な場合に備えて、プロバイダーを必要なだけ高く設定します。

それでは、データをマテリアルアプリ全体に提供できるように、それを最上部に配置します。

したがって、ここでは新しいウィジェットでラップし、そのウィジェットはプロバイダーウィジェットになります。

次にプロバイダーウィジェットを使用して、このプロバイダーが提供するデータの種類のデータ型も指定する必要があります。このデータ型は文字列になります。データは本当に単純な文字列だからですよね？

プロバイダーウィジェットを作成しました。子があり、後続のすべてのレベルで、提供するデータをリッスンできるようになっています。プロバイダービルダーであるプロパティをもう1つ追加する必要もあります。

ビルダーは、現在のビルドコンテキストを取得して、ウィジェットツリー内のどこにあるかを認識し、すべての子が使用できるようにする必要があるデータを返す必要があります。ここにデータがあります。

ツリーの最上位レベルでデータが提供されたので、必要な場所にデータを提供するようプロバイダーに依頼できます。

それではまず、このコードを単純化しましょう。

単純な1行のリターンがあるため、この中括弧などすべてとreturnキーワードは必要ありません。

単純に太い矢印を使用するだけで、まったく同じことができます。

これで、マテリアルアプリの上にあるアプリの最上位レベルにデータを提供できるようになりました。これにより、必要な場所にアクセスして使用できるようになります。

必要な最初の場所はテキスト内です。

したがって、このデータを実際に必要としないレベル1レベル2ウィジェットを介してデータを渡す代わりに、実際にこのコードのすべてを削除することはできません。

代わりに、レベル3で使用します。

そして、そのデータにアクセスするために必要なのは、provider.ofと言うことだけです。また、「of」メソッドは、取得しようとしているデータの種類を指定することを期待しています。

これが、ツリー内のすべての子にここで提供する文字列になります。

これで、[保存]をクリックしてアプリを確認すると、トップシークレットデータがレベル3のステートレスウィジェットに渡されていることがわかります。

そして、それはここから来ています。ご覧のとおり、このデータを次のように変更することを決定したとしましょう。わかりませんが、極秘コードとしましょう。

アプリを再実行すると、レベル1とレベル2をドリルスルーする必要なしに、データが実際にレベル3に渡されていることがわかります。これで、問題はなくなりました。

最上位レベルでデータを提供することができ、ツリー内の中間部分を経由することなく、ここまでデータを読み取ることができました。

したがって、ここでそのプロパティの状態が変化すると、レベル3がそれらの変更をリッスンしているので、レベル1またはレベル2を再構築する必要なく、すぐにそれを受信して​​自分自身を再構築できるため、より高速で効率的なコードになります。

今、これをもう少し複雑にしてみましょう。現時点でプロバイダーに対して行っているのは、私たちが提供し、読んでいることです。ツリー内のどこかから状態を更新する必要がある場合はどうなりますか？

シンプルなアプリをレベル1、2、3に更新してみましょう。ただし、ここに別のクラスのMyTextというテキストウィジェットもあり、レベル2の子であるMyTextFieldウィジェットもここにあります。

つまり、ここでは、ツリーの上位にあるMyTextまで、さらには電源プロバイダーを使用してレベル3に至るまで、ここで発生する更新をどのように取得できるのでしょうか。

したがって、ここにMyTextとMyTextFieldという別のクラスがあり、それらをツリーに挿入します。

したがって、私のテキストは、ここでAppBarのテキストウィジェットとして表示されます。

それをMyTextに変更すると、MyTextFieldがレベル2ウィジェットの子になります。

この空のコンテナの代わりに、ここにMyTextFieldを追加します。

これで、基本的にはウィジェットツリーの正確な構造ができました。介在するウィジェットを再構築する必要なく、このデータをここだけでなく、ここからも取得できるようにする必要があります。

それでは、どうすればよいでしょうか。

さて、ChangeNotifierと呼ばれるクラスを使用する必要があります。

ここで、データクラスとなる別のクラスを作成し、このデータクラスにChangeNotifierクラスを拡張させます。

現在、このChangeNotifierクラスは実際にはFlutter自体に由来しています。

これはプロバイダーに付属するものではありません。

これはすでに存在し、この特定のクラスにクラスの更新をリスナーに通知する目的で存在します。

それでは、データをデータクラスに移動して、最終的なものにする代わりに、正しく変更できるようにする必要がありますか。

データを更新する必要があります。

それでは、ダミーデータから始めましょう。「Some data」と呼びましょう。そして、以前の単純な文字列ではなく、このデータオブジェクトの提供を開始しましょう。

したがって、現在は存在しないデータを返す代わりに、データクラスのオブジェクトを返します。

そして、このオブジェクトは、文字列データ型を提供するだけの通常のプロバイダーを使用しなくなります。代わりに、ChangeNotifierプロバイダーになり、Dataデータ型を提供します。

したがって、ここでデータにアクセスするために、ここでもprovider.ofを使用し、文字列データ型を使用する代わりに、実際に提供したいデータ型であるデータを提供する必要があります。そして、このオブジェクト全体（クラスから作成されたデータオブジェクト）を提供する代わりに、データオブジェクトのデータプロパティを提供します。これは文字列であるためです。

そして、その文字列はテキストウィジェットに入れられて表示されます。

これで、それをコピーしてMyTextウィジェットにも追加できるようになりました。これにより、アプリバーとレベル3のテキストウィジェットにレンダリングされます。

それでは、まずアプリを実行して、実際にデータを提供できることと、アプリバーのMyTextウィジェットとレベル3のテキストウィジェットの両方でデータを読み取ることができることを確認してみましょう。

これでかなりうまくいきます。

したがって、次の目標であるデータの更新を達成する必要があります。MyTextField内でデータを更新します。

そのため、MyTextFieldにはonChangedというプロパティがあり、テキストフィールドに入力される新しい値または新しいテキストにアクセスできるコールバックを取ることがわかっています。

次に、このテキストをデータクラスに移動して、この文字列を更新できるようにする必要があります。

それでは、データクラス内にメソッドを作成しましょう。 何も返さないようにvoidメソッドを作成しましょう。

そして、それは私たちの文字列を変更します。

つまり、changeStringメソッドです。 そして、データの代わりとなる新しい文字列を渡し、データを新しい文字列と等しくなるように設定します。 そして、最も重要なのは、ChangeNotifierから継承しているため、利用可能なメソッドを使用することです。

そして、そのメソッドはnotifyListenersメソッドです。

つまり、データを新しい値に更新するとすぐに、通知リスナーを呼び出します。これにより、プロバイダーをリッスンしているすべての人が、状態の変化について通知を受けるため、ウィジェットも再構築できます。

このリストで考えてみると、すでにセット状態に遭遇しているので、それを使用してローカル状態を非常によく制御および管理する方法を知っています。もちろん、ステートフルウィジェットと組み合わせて使用​​する必要があります。ただし、ステートフルウィジェットはステートレスウィジェットよりもはるかに複雑であるため、1つのステートフルウィジェットには実際にメモリが必要です。

ステートレスウィジェットは操作がはるかに簡単であり、作成、破棄、および再構築することができるのに対し、変更できるように状態を覚えておく必要があります。

また、コールバックを組み合わせて使用​​し、ウィジェットツリーで状態を解除することで、以前に状態を解除することも確認しました。状態を管理することができました。しかし、もちろん、アプリがより複雑になり、ツリー全体を横断するコールバックが必要な場合、または状態を複数レベル下に渡す必要がある場合は、いくつかの問題が発生する可能性があります。

そのため、現在、状態を管理する最も柔軟な方法は、プロバイダーパッケージを使用することです。

これは現在、Google Flutterチームからの推奨事項です。

しかし、私が言おうとしているのは、これが公式に推奨されている方法であるにもかかわらず、Flutterアプリで状態を管理する方法であることに留意してください。

また、多くの場合、別の方法が必要な場合や、状態を管理するこれらの異なる方法の組み合わせが必要な場合があります。

それは間違っていません。

それは本当にあなたのユースケース、つまりあなたの要求、あなたのコードをどのように整理したいか、そしてFlutterのフレームワークの成熟度に特有のスタイルの感覚に依存します。

しかし、それは本当に素晴らしいツールだからです。

これは、実際にコミュニティによって作成されたパッケージです。

グーグルはProvidと呼ばれる似たようなパッケージを持っていましたが、これは何か似たようなことをしましたが、彼が来たので、彼はレミと彼は実際により良いと思われるバージョンを構築することにしましたそのため、「Google Flutterチーム」への全面的な功績は、このパッケージの可能性を実感し、実際に全力を尽くしました。

そして今では公式に推奨されているだけでなく、レミ自身とGoogle Flutterチームの両方からもサポートされています。

そのため、その潜在能力を最大限に引き出すために必要なすべてのリソースを備えています。

ここで、プロバイダが非常に有用である理由を完全に理解するために、最初に実際に解決しようとしている問題を検討する必要があります。

最初に私たちのコードから、あなたはすでに少し毛むくじゃらになっているものがいくつかあることにすでに気付いているかもしれません。

ツリーにデータを渡すためにこれらのコールバックがすべてあり、set dateを使用して子ウィジェットのビルドメソッドを呼び出す必要があるステートフルウィジェットの束があり、多くのものが常に再構築されています。そして、おそらくこれは最も効率的な方法ではありません。

しかし、私たちのアプリは非常に大きくなり、状態を管理しているときに何が起こっているのかを一目で見るのは非常に難しいため、問題を少しよく説明できるように本当にシンプルなアプリを作成しました。

したがって、この場合、MyAppのトップレベルクラスがあり、2つの子があり、そのうちの1つはレベル1と呼ばれます。次に、レベル1はレベル2とレベル2を作成し、レベル3を作成します。

そのため、そのコードは次のようになります。それは非常に簡単です。重要な部分は、レベル1クラスを作成し、次にレベル2およびレベル3クラスをこの順序で作成するMyAppクラスがあることです。

レベル1を作成し、レベル1がレベル2、レベル2がレベルLevel 3を作成するマテリアルアプリです。

基本的にまさにこの構造です。

ここに問題があります。

ここだけでなくここでも使用する必要があるプロパティがあったとしましょう。そのためには、状態を解除してアプリ内で動作させる必要があります。

これで非常にうまくいきましたが、そのプロパティまたはそのプロパティの状態をレベル3に渡すと、そこに到達するためにレベル1とレベル2とレベル3を通過する必要があります。つまり、そのデータはこれらの2つのクラス内で更新されますが、実際には必要ありません。

したがって、下位レベルに到達するには、基本的にツリーをドリルダウンする必要があります。そして、リアクティブプログラミングでは、これはプロップドリルと呼ばれることもあります。そのため、プロパティを継承するためにツリーをドリルします。これを実際のコードで見てみましょう。

単にデータと呼ばれる文字列があり、それが「トップシークレットデータ」だとしましょう。

そして、これは両方のAppBar内で使用する必要があります。したがって、タイトル内でテキストウィジェットとして使用され、そのデータが含まれます。ただし、ここではレベル3でも使用する必要があるため、このレベル3のテキストウィジェットにもデータが必要です。しかし、レベル3に移行するには、すべてのレベルをドリルスルーする必要があります。

したがって、レベル1の内部では、データを呼び出すだけのプロパティが必要になり、レベル1を初期化するときに、このデータプロパティを渡すことで初期化することになります。そして、コンストラクタを介してレベル1に渡し、データがレベル1内にあるようにします。

しかし、レベル3に下げるには、レベル2に渡し続ける必要がありますか？

それをレベル2に渡して、dataという名前の最終的なStringプロパティを作成し、レベル2が初期化されたら、このプロパティをthis.dataに渡します。

そして最後に、レベル3に渡すだけです。

したがって、最終的な文字列データを作成し、レベル3を初期化するときに、必要なデータの値を渡します。

そして、最終的に、レベル3で必要なデータを取得し、テキストウィジェット内に実際に配置できるようにしました。

それで、保存をクリックしてアプリを実行すると、正しく動作することがわかりますか？

私のトップレベルのウィジェットであるMyApp内にある私のデータは、このデータプロパティを私のAppBarと、テキストウィジェットを持つこのレベル3ウィジェットの両方に渡すことができます。しかし、それを行うために、このデータをまったく関係のない、本当に関係のないウィジェットの束全体に渡さなければなりませんでした。レベル2では、レベル1のデータは必要ないことがわかります。そのデータは必要ありません。レベル3に下げるために、このデータをそのまま渡します。

怠け者のプログラマーにとっては、これは見るのがかなり苦痛であり、また複雑さを大幅に増加させます。

ここにタイプミスがあったり、レベル1内に他のデータがあると言ったり、「渡されるデータとローカルバージョンのデータとが混同される」と言ってください。本当にきれいではありません。

では、代わりに何ができますか？

Providerパッケージがある場合は、状態を最上部に移動できますが、状態にサブスクライブするだけのウィジェットを作成できます。そのため、レベル3のウィジェットに手を上げて、状態の変化をリッスンするように言ってもよいでしょう。そして、それが変更されると、私は通知されます-レベル3に到達するためにレベル1とレベル2を通過する代わりに、レベルを再構築する代わりに、この更新された状態を実際に処理するレベル3ウィジェットにデータを渡すだけです1、レベル2およびレベル3。

そのツリーを再構築して状態を渡し、実際にレベル3に到達させる代わりに、単にレベル3に更新をリッスンさせるだけです。

では、プロバイダーを使用してこれらすべてのクールな機能を取得するにはどうすればよいでしょうか？

まず、アプリにプロバイダーパッケージを含める必要があります。

それでは、インストールに戻り、プロバイダーを依存関係として追加しましょう。

pubspec.yamlを開き、Cupertinoアイコンのすぐ下にすべての依存関係があります。プロバイダーパッケージを追加し、パッケージを呼び出してそのパッケージからコードを取得します。

そして今、provider.dartファイルをインポートできるはずです。

これで、コードにプロバイダーができたので、それを使用できます。

最初にやらなければならないのは、データを公開することです。つまり、提供するデータは何かということです。それを行うために想像できるとおりで、私たちはフラッターランドにいるので、ウィジェットを使用して実現します。そして、AppBarのデータとレベル3の両方のデータが必要なため、必要に応じてそのプロバイダーを上位に配置します。

それでは、マテリアルアプリ全体にデータが提供されるように、最上部に配置します。

そこで、ここで新しいウィジェットでラップし、そのウィジェットをプロバイダーウィジェットにします。

プロバイダーウィジェットでは、このプロバイダーが提供するデータの種類のデータ型も指定する必要があります。このデータ型は文字列になります。データは本当に単純な文字列だからです。

プロバイダーウィジェットを作成し、後続のすべてのレベルが提供するデータをリッスンできるようにする子があるため、プロバイダービルダーであるもう1つのプロパティを追加する必要があります。

ビルダーは、現在のビルドコンテキストを使用して、ウィジェットツリー内のどこにあるかを認識し、すべての子が利用できる必要があるデータを返す必要があります。それがここのデータになります。

ツリーの最上位でデータが提供されるようになったので、プロバイダーに必要な場所にデータを提供するように依頼できます。

それでは、まずこのコードを単純化しましょう。

単純な1行のリターンがあるので、この中括弧などとreturnキーワードは必要ありません。

太い矢印を使用するだけで、まったく同じことができます。

そのため、マテリアルアプリの上にアプリの最上位レベルでデータを提供しているので、必要に応じてデータにアクセスして使用できます。

最初に必要なのはテキスト内です。

そのため、このデータを実際に必要としないレベル1レベル2ウィジェットにデータを渡す代わりに、実際にこのコードのすべてをこことこことここですべて削除することはできません。

代わりに、レベル3内で使用するだけです。

そして、そのデータにアクセスするために必要なことは、provider.ofだけです。また、「of」メソッドは、取得しようとしているデータの種類を指定することを期待しています。

これは、ツリー内のすべての子にここで提供する文字列になります。

そのため、保存ボタンを押してアプリを確認すると、トップシークレットデータがレベル3のステートレスウィジェットに渡されていることがわかります。

そして、それはここから来ます。ご覧のとおり、このデータを変更することに決めたとしましょう。私は知りません。極秘コードを言ってみましょう。

そして今、アプリを再実行すると、レベル1とレベル2をドリルスルーする必要なしに、データが実際にレベル3に渡されていることがわかります。これですべてうまくいきました。

最上位レベルでデータを提供できたので、ツリー内の中間部分を経由せずに、ここからずっとデータを読み取ることができました。

そのため、ここでそのプロパティの状態が変更されると、レベル3はそれらの変更をリッスンしているため、レベル1またはレベル2を再構築する必要なく、すぐにそれを受け取り、それ自体を再構築できます。

さて、これをもう少し複雑にしましょう。現時点では、プロバイダーで行っていることはすべて提供し、読んでいるからです。ツリー内のどこかから状態を更新する必要がある場合はどうなりますか？

単純なアプリを更新して、レベル1、2、および3を保持しますが、ここに別のクラスのMyTextというテキストウィジェットがあり、レベル2の子であるMyTextFieldウィジェットもあります。

それで、アイデアは、ここで下の方にある更新をMyTextに到達させる方法です。MyTextは、ツリーの上位にあり、電源プロバイダーを使用してレベル3にあります。

そのため、ここにはMyTextクラスとMyTextFieldである別のクラスがあり、それらをツリーに挿入します。

したがって、私のテキストはAppBarのテキストウィジェットとして使用されます。

それをMyTextに変更すると、MyTextFieldは私のレベル2ウィジェットの子になります。

この空のコンテナの代わりに、ここにMyTextFieldを追加します。

したがって、基本的にウィジェットツリーのこの正確な構造が得られ、介在するウィジェットを再構築することなく、このデータをここだけでなくここまで取得できる必要があります。

それでは、どうすればいいのでしょうか？

さて、ChangeNotifierと呼ばれるクラスを使用する必要があります。

そこで、ここでデータクラスとなる別のクラスを作成し、このデータクラスでChangeNotifierクラスを拡張します。

現在、このChangeNotifierクラスは実際にはFlutter自体から来ています。

これはプロバイダーに付属しているものではありません。

これは既に存在し、この特定のクラスのリスナーにクラスの更新を通知するために存在します。

それでは、データをデータクラスに移動し、最終的なものにするのではなく、適切に変更できるようにする必要がありますか。

データを更新する必要があります。

それでは、いくつかのダミーデータから始めましょう。「いくつかのデータ」と呼びましょう。そして、以前から持っていた単純な文字列だけでなく、このデータオブジェクトの提供を始めましょう。

そのため、現在存在しないデータを返す代わりに、データクラスのオブジェクトを返します。

このオブジェクトは、文字列データ型を提供するだけの通常のプロバイダーを使用しなくなり、代わりにChangeNotifierプロバイダーになり、データデータ型を提供します。

したがって、ここでデータにアクセスするには、provider.ofを使用し、文字列データ型を使用する代わりに、提供するデータ型を実際に提供する必要があります。そして、このオブジェクト全体、つまりクラスから作成されるこのデータオブジェクトを提供する代わりに、データオブジェクトのデータプロパティを提供します。それは文字列だからです。

そして、その文字列はテキストウィジェットに挿入されて表示されます。

そのため、これをコピーしてMyTextウィジェットにも配置し、レベル3のテキストウィジェットだけでなくアプリケーションバーにもレンダリングできるようにします。

最初にアプリを実行して、アプリバーのMyTextウィジェットとレベル3のテキストウィジェットの両方でデータを実際に提供し、読み取ることができることを確認します。

だからそれはかなりうまく機能しています。

そのため、データを更新するという次の目標を達成する必要があり、MyTextField内で更新します。

そのため、MyTextFieldには、新しい値またはテキストフィールドに入力される新しいテキストにアクセスするコールバックを受け取るonChangedというプロパティがあることがわかっています。

そして、この文字列を更新するには、このテキストをデータクラスに移動できる必要があります。

それでは、データクラス内にメソッドを作成しましょう。何も返さないようにvoidメソッドを作成しましょう。

そして、文字列が変更されます。

それがchangeStringメソッドです。そして、データの代わりに使用する新しい文字列を渡し、新しい文字列に等しくなるようにデータを設定します。そして最も重要なのは、ChangeNotifierを継承しているため、利用可能なメソッドを使用することです。

そして、そのメソッドはnotifyListenersメソッドです。

そのため、データを新しい値に更新したらすぐに、通知リスナーを呼び出して、プロバイダーの状態を通知しているため、プロバイダーをリッスンしているすべてのユーザーがウィジェットを再構築できるようにします。

したがって、onChanged内でprovider.ofを利用し、関心のあるデータのタイプを指定します。これはデータクラスです。

それでは、changeStringというメソッドを利用して、このデータプロパティに挿入するnewStringとしてnewTextを渡します。

したがって、アプリを再実行してここに入り、入力を開始すると、このデータのすべてのリスナー、つまりレベル3のmyTextウィジェットとテキストウィジェットが再構築されていることがわかります。これで、テキストフィールドのテキストを変更するたびに状態が更新されます。

それでは、もう一度コードを確認しましょう。

そこで、このクラス内のデータの一部がデータの一部、つまりデータというプロパティを追跡しようとしているリスナーに通知できるようにするChangeNotifierと呼ばれるものを拡張するDataというこのクラスを作成しました。そして、そのプロパティは、AppBarのこのテキストウィジェット内とレベル3内のこのテキストウィジェットの両方で読み取られています。

したがって、これら2つのテキストウィジェットの外観は、このデータプロパティの状態に関連付けられています。

そして、そのデータプロパティが更新されると、このテキストウィジェットはその変更を反映するために再構築されます。

ここでデータプロパティがここから来て、change Stringメソッド内でそれを更新し、新しい文字列を渡してデータプロパティ内の古い文字列を置き換えます。そして、決定的に重要なのは、notifyListenersを呼び出して、そのデータプロパティをリッスンしているすべてのユーザーが通知を受け、自分自身を再構築して最新の状態を更新できるようにすることです。

これで、このnotifyListenersを呼び出す必要があるため、基本的にこのデータオブジェクトであることがわかっているデータのプロバイダーを単にタップして、単純にそのデータプロパティをタップして、新しいテキストに等しく設定することはできません。

この場合、外部からこの文字列を更新するだけであるため、実際にはnotifyListenersを呼び出していないためです。

このプロパティの値を更新しているだけです。したがって、実際にnotifyListenersを呼び出すように、メソッドを使用して文字列を変更する必要があります。

覚えておくべきもう1つの重要なことは、このprovider.ofメソッドには実際にはlistenというオプションのパラメーターがあることです。

AppBar内のテキストウィジェットであるMyTextが、MyTextFieldのすべての更新で実際に更新されないようにしたいとしましょう。

格納されているデータの元の値を取得したいが、更新をリッスンしたくない、このデータが変更されるたびに再構築したくないとしましょう。

それでは、listenプロパティを追加するだけで、falseに設定できます。

現在、このプロパティはデフォルトでtrueに設定されているため、provider.ofを使用するたびに、リスニングしているものの変更に基づいて自身を更新するリスナーを取得します。

しかし、falseをリッスンすると言ったので、アプリを再実行してみましょう。両方のテキストプロパティは、データの値となるデータが初期化されると、ここから来るいくつかのデータで始まることがわかります。

しかし、テキストフィールドに入力を開始すると、ここのテキストウィジェットは変更をリッスンしていないため、更新も再描画もされません。

したがって、これにより、データの更新をリッスンして再描画したいものと、データの初期値のみを必要とし、それが再び変更されないようにするものを非常に細かく制御できます変更。プロバイダーパッケージの動作を確認したので、「まあ、それはどのように機能しますか？」

それを理解するために、ボンネットの中にポップアップして、舞台裏で何が起こっているのかを見てみましょう。状態を管理するこれらのさまざまな方法のうち、継承されたウィジェットを使用する方法があります。継承されたウィジェットは、基本的には、ツリーの介在部分を再構築することなく、ツリーにデータを渡すことができるウィジェットです。

それはちょっとおなじみのように聞こえますか？

これが、プロバイダーの仕組みです。

基本的にプロバイダーパッケージは、継承されたウィジェットの単なるラッパーです。ただし、継承ウィジェットクラスには、それを構築するための特定の要件があり、それを特定の方法で構築して、依然として効率的であり、期待どおりに実行されるようにする必要があります。

したがって、非常に多くの定型コードがあり、それを使用しているときにどのように機能するかを十分に意識する必要があります。

そのため、継承されたウィジェットクラスを見て、非常にクールなGoogle自転車シャツでフィリップのこのウィジェットの説明を聞いて、舞台裏で実際にどのように機能するかについてもう少し学ぶことをお勧めします。本質的に私たちのプロバイダーパッケージはこのクラスのパワーを活用しているだけですが、それを操作しやすくし、安全で効果的な方法でそれを使用することを強制する方法でそれを行っているからです。

そして、それが完了したら、このウィジェットツリーの独自のバージョンを構築することをお勧めします。 したがって、この場合は別のクラスがうまくあります。このように構成された約6つの異なるクラスがあり、プロバイダーパッケージを使用して、最上位レベルでデータを提供する方法、下位でデータを読み取る方法を確認します レベルおよびChangeNotifierプロバイダーを使用してデータを更新します。

それで試してみてください。このレッスンで使用したコードを確認したい場合は、コースリソースでダウンロードすることもできます。

しかし、次のレッスンでは、実際にプロバイダーを実際のTodoeyアプリにプラグインし、より複雑で現実的なシナリオでそれを確認します。

それで、それ以上については、次のレッスンで説明します。