**反应原生优化的最终指南**



改善应用程序的用户体验、性能和稳定性。



由

2020

IMAGE

**表的内容**

**组织部分**

**反应原生优化的介绍**

**第一组**

1.注意UI重新呈现

2.为特定的布局使用专用组件

3.在选择外部库之前请三思

4.永远记住使用专用于移动平台的库

5.在原生和JavaScript之间找到平衡

6.以60FPS的速度动画

**第二组**

1.始终运行最新的React本机版本来访问新特性

2.如何用Flipper调试更快更好

3.使用“autolinking”自动化依赖管理

4.使用Hermes优化Android应用程序的启动时间

5.优化你的Android应用程序的大小与这些Gradle设置

**第三组**

1.为应用程序的关键部分运行测试

2.是否有一个工作的持续集成(CI)

3.不要害怕使用持续部署来快速交付

4.在紧急情况下使用OTA(无线)

**谢谢你！**

**作者**

**Callstack**

**反应原生优化的最终指南**

**组织部分**



在有限的开发预算下优化React原生应用程序可能很困难，但并非不可能。在这种情况下，你需要专注于你的应用程序的要点，并尽可能地从它们中挤出来，以保持你的业务连续性。

**这就是我们准备这份指南的原因。**

在接下来的章节中，我们将向你展示如何优化你的应用程序的性能和稳定性。多亏了指南中描述的实践，您将改善用户体验并加速应用程序的上市时间。

**该指南包含优化以下方面的最佳实践:-稳定性**

——性能

——资源使用情况

——用户体验

——维护成本

——上市时间

上述所有这些方面都对你的应用程序的创收效率有特殊的影响。稳定性、性能和资源使用等因素直接关系到提高产品的ROI，因为它们对更好的用户体验有影响。通过更快的上市时间，您可以保持领先于您的竞争对手，而更简单和更快的维护流程将帮助您减少在该特定流程上的花费。

**{callstack.com}**

3.

**反应原生优化的最终指南**

**指南将是什么样子的以及它将涵盖什么主题。**

**指南分为三组:**

**第一组是关于通过理解React本地实现细节和知道如何最大限度地利用它来提高性能。**以下是我们将讨论的主题:

1.注意UI重新呈现

2.为特定的布局使用专用组件

3.在选择外部库之前请三思

4.永远记住使用专用于移动平台的库

5.在原生和JavaScript之间找到平衡

6.以60FPS的速度动画

**第二组专注于通过使用最新的React本机特性或启用其中一些特性来提高性能。**本部分将介绍以下主题:

1.始终运行最新的React本机版本来访问新特性

2.如何用Flipper调试更快更好

3.使用“autolinking”自动化依赖管理

4.使用Hermes优化Android应用程序的启动时间

5.优化你的Android应用程序的大小与这些Gradle设置

**第三组是关于通过投资测试和持续部署来提高应用程序的稳定性。**这部分讲的是:

1.为应用程序的关键部分运行测试

2.是否有一个工作的持续集成(CI)

3.不要害怕使用持续部署来快速交付

4.在紧急情况下使用OTA(无线)

**每篇文章结构简单:**

**问题:第一部分描述了主要的问题以及你可能做错了什么。解决方案:第二部分讲述这个问题如何影响你的业务，以及解决这个问题的最佳做法是什么。**

**好处:第三部分关注我们提议的解决方案的业务好处。**

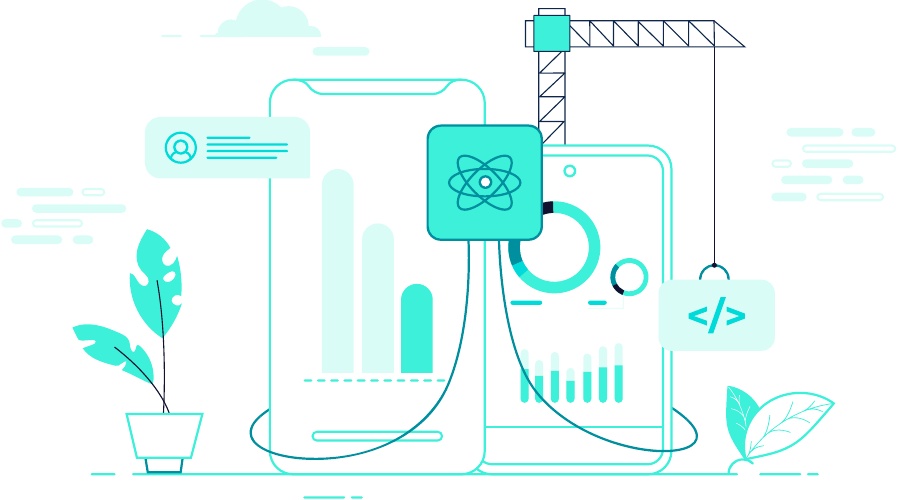
**{callstack.com}**

4

**反应原生优化的最终指南**

信息和组织的部分已经讲过了。现在，让我们看看优化应用程序性能的最佳实践。

我们走吧!



**{callstack.com}**

5

**反应原生优化的最终指南**

**反应原生优化的介绍**



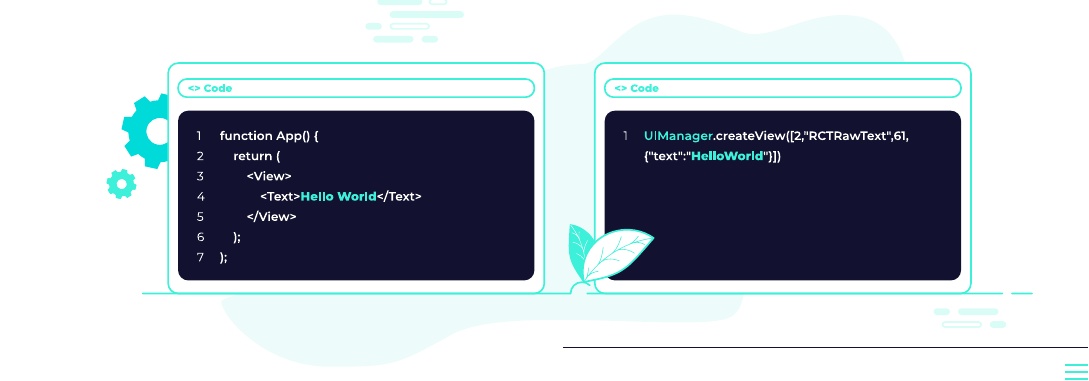
React Native负责渲染。但性能仍是如此。

使用React Native，您可以创建描述接口外观的组件。在运行时，React Native将它们转换为特定于平台的本机组件。不是直接与底层api对话，而是将重点放在应用程序的用户体验上。

然而，这并不意味着所有使用React Native完成的应用程序都一样快，并提供相同水平的用户体验。

**每个声明式方法(包括React Native)都是用命令式api构建的。当你命令式地做事情时，你必须小心。**

当您以命令式的方式构建您的应用程序时，您要仔细分析外部api的每个调用站点。例如，当您在多线程环境中工作时，您可以以线程安全的方式编写代码，并且要知道代码要查找的上下文和资源。



**{callstack.com}**

6

**反应原生优化的最终指南**

尽管声明式和命令式的做事方式有很多不同，但它们有很多共同之处。每个声明式抽象都可以分解为许多命令式调用。例如，React Native使用与本地开发人员使用相同的api在iOS上呈现应用程序。

**反应本机统一性能，但不能使其快速开箱!**

虽然您不必担心底层iOS和Android api调用的性能，但如何将组件组合在一起会产生很大的影响。所有组件都将提供相同级别的性能和响应能力。

**但这是“最好”的同义词吗?它不是。**

这时我们的清单就开始发挥作用了。使用对其潜力产生自然反应。

如前所述，React Native是一个声明性框架，负责为您呈现应用程序。换句话说，并不是由您来决定如何呈现应用程序。

您的工作是定义UI组件，然后忘记其他的事情。但是，这并不意味着您应该理所当然地考虑应用程序的性能。为了创建快速响应的应用程序，您必须考虑以本机方式进行响应。您必须了解它如何与底层平台api交互。

**如果您需要帮助的性能，稳定性，用户体验或其他复杂的问题-联系我们!作为反应本机的核心贡献者和**

**社区的领导们，我们会很乐意帮忙的。**

**{callstack.com}**

7



**第一组**

通过理解React本机实现细节来提高性能。

**{callstack.com}**

**最终引导反应原生优化|第一组**

**介绍**

在本组中，我们将深入研究最常见的性能瓶颈，并对导致这些瓶颈的本机实现细节作出反应。这不仅是对一些高级的React本机概念的平滑介绍，而且还可以通过执行一些小的调整和更改来显著提高应用程序的稳定性和性能。

下面的文章主要关注性能优化策略的整个检查表中的第一点:UI重新呈现。这是React原生优化过程中非常重要的一部分，因为它可以减少设备的电池使用量，从而为应用程序提供更好的用户体验。

**{callstack.com}**

**最终引导反应原生优化|第一组**

**1.注意UI重新呈现**

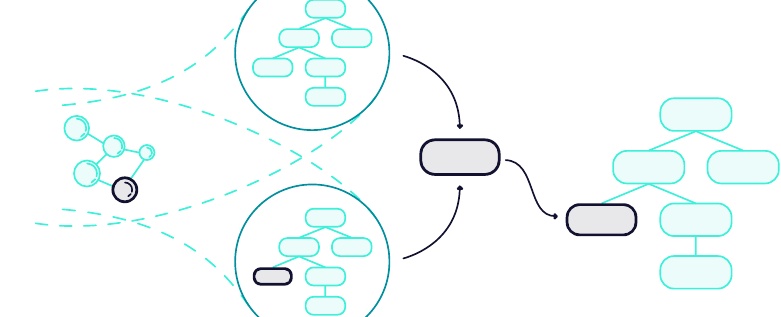


优化状态操作的数量，记住纯组件和记忆组件，使您的应用程序工作得更快，所需的资源更少。

**问题:不正确的状态更新导致不必要的渲染周期/或者设备太慢**

如前所述，React Native负责为您呈现应用程序。您的工作是定义所需的所有组件，并从这些较小的构建块中组成最终的接口。在这种方法中，您不能控制应用程序的呈现生命周期。

换句话说——何时以及如何在屏幕上重新绘制东西完全是React Native的责任。React会查看您对组件所做的更改，比较它们，并且根据设计，只执行所需的、最小数量的实际更新。



Diff

**模型**

补丁

Diff

**虚拟DOM**

**DOM**

**{callstack.com}**

10

**最终引导反应原生优化|第一组**

这里的规则很简单——默认情况下，如果父组件被重新渲染或者道具不同，组件可以重新渲染。这意味着你的组件的“render”方法有时可以运行，即使它们的道具没有改变。在大多数场景中，这是一个可以接受的折衷，因为比较两个对象(以前的和当前的道具)将花费更长的时间。

**对性能、UI闪烁和FPS降低的负面影响**

虽然上述启发式方法在大多数情况下是正确的，但执行太多操作可能会导致性能问题，特别是在低端移动设备上。

因此，您可能会观察到您的UI闪烁(当更新正在执行时)或帧下降(当动画正在发生并且更新正在进行时)。

**注意:您不应该执行任何过早的优化。**这样做可能会产生反作用。当你发现在你的应用程序中丢失帧或不期望的性能时，试着看看这个。

一旦看到这些症状，就应该更深入地了解应用程序的生命周期，并注意可能不会发生的无关操作。

**解决方案:优化状态操作的数量，并记住在需要时使用纯组件和记忆组件**

有很多方法可以使应用程序变成不必要的渲染循环，这一点值得单独写一篇文章。在本节中，我们将关注两种常见的场景—使用受控组件，比如“TextInput”和全局状态。

**受控组件与非受控组件**

让我们从第一个开始。几乎每个React本机应用程序都包含至少一个由组件状态控制的“TextInput”，如下面的代码片段所示。

**{callstack.com}**

11

**最终引导反应原生优化|第一组**

从“React”导入React， {Component};

从' response -native '导入{TextInput};

导出默认函数UselessTextInput() {

const [value, onChangeText] = response . usestate (' Text ');

回报(

< TextInput

style={{height: 40, borderColor: ' gray '， borderWidth: 1}}

onChangeText ={文本= > onChangeText(文本)}

值= {value}

/>

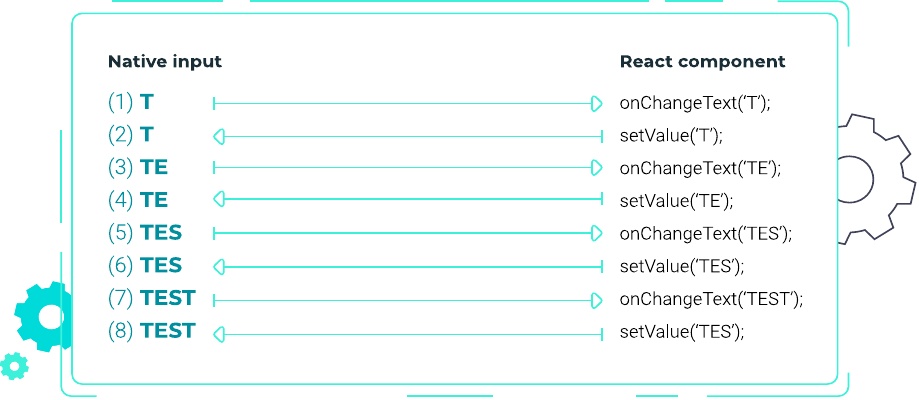
);

}

阅读更多:https://snack.expo.io/q75wcVYnE

上面的代码示例在大多数情况下都可以工作。但是在慢的设备上，在用户打字速度非常快的情况下，视图更新可能会造成问题。

原因很简单——React Native的异步特性。为了更好地理解这里发生了什么，让我们首先看看当用户输入<TextInput />并使用新字符填充时发生的标准操作的顺序。



显示键入测试时发生的情况的图表

**{callstack.com}**

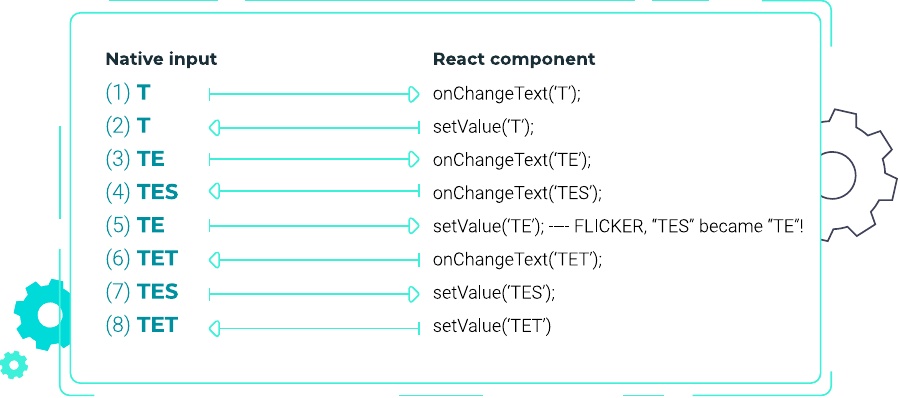
12

**最终引导反应原生优化|第一组**

一旦用户开始向本机输入输入一个新字符，就会通过onChangeText prop(图上的操作1)发送一个更新来响应本机输入。通过调用setState响应信息处理并相应地更新其状态。接下来，典型的受控组件将其JavaScript值与本机组件值同步(图中的操作2)。

这种方法的好处很简单。React是一个真理的来源，它决定了你输入的价值。这项技术允许你在用户输入发生的时候改变它，例如执行验证，屏蔽它或者完全修改它。

不幸的是，上述方法虽然最终更干净，更符合React的工作方式，但也有一个缺点。当可用资源有限和/或用户打字速度非常快时，这一点最为明显。



图表显示当输入TEST太快时发生了什么

当通过onChangeText实现的更新在响应本机同步之前到达时，接口将开始闪烁。当用户开始键入T时，第一次更新(操作1和操作2)执行时没有问题。

接下来,操作3到来,紧随其后的是另一个更新操作(4)。用户输入E & S而反应本地正忙着做其他的事情,拖延的同步E字母(操作5)。因此,本机的输入将会改变其价值暂时从TE TE。

**{callstack.com}**

13

**最终引导反应原生优化|第一组**

现在，当文本输入的值被设置为TE时，用户输入的速度已经快到可以实际输入另一个字符了。结果，另一个更新到达(操作6)，值为TET。这不是有意的—用户不希望其输入的值从TES更改为TE。

最后，操作7将输入同步到用户之前接收到的正确输入(操作4告知我们关于TES的信息)。不幸的是，它很快被另一个更新(操作8)覆盖，该更新将该值同步到TET(输入的最终值)。

造成这种情况的根本原因在于操作的顺序。如果操作5在操作4之前执行，事情就会顺利运行。此外，如果用户在值为TE而不是TES时没有键入T，则界面将闪烁，但输入值将保持正确。

同步问题的解决方案之一是从TextInput中完全移除value prop。因此，数据将只以一种方式流动，即从本机流到JavaScript端，从而消除了前面描述的同步问题。

从' React '导入React， {Component, useState};

从' response -native '中导入{Text, TextInput, View};

导出默认函数PizzaTranslator() {

const [text, setText] = useState(");

回报(

<视图风格={{填充:10}}>

< TextInput

风格={{高度:40}}

占位符= "输入这里来翻译! "

onChangeText ={文本= > setText(文本)}

defaultValue ={文本}

/>

<Text style={{padding: 10, fontSize: 42}

**{callstack.com}**

14

**最终引导反应原生优化|第一组**



”)。加入(' ')}

{文本。split(' ').map((word) => word && '

< /文本>

< /视图>

);

}

阅读更多:https://snack.expo.io/DYMECpVPQ

然而，正如@nparashuram在他的YouTube视频(这是学习React本机性能的一个很好的资源)中指出的，在某些情况下，仅靠解决方案是不够的。例如，在执行输入验证或屏蔽时，您仍然需要控制用户输入的数据，并更改最终在TextInput中显示的内容。React Native团队很清楚这个限制，目前正在进行新的重新架构，以解决这个问题。

**全局状态**

性能问题的其他常见原因是组件如何依赖于应用程序的全局状态。最坏的情况是当单个控件的状态改变，如TextInput或复选框传播整个应用程序的渲染。造成这种情况的原因是糟糕的全局状态管理设计。

我们建议使用Redux或Overmind.js等专门的库来以更优化的方式处理状态管理。

首先，状态管理库应该只在定义的数据子集发生更改时更新组件——这是redux connect函数的默认行为。

其次，如果组件使用的数据与存储在状态中的数据的形状不同，它可能会重新呈现，即使没有真正的数据更改。为了避免这种情况，您可以实现一个选择器，它将记住派生的结果，直到传递的依赖项集发生更改为止。

**{callstack.com}**

15

**最终引导反应原生优化|第一组**

从“reselect”导入{createSelector}

const getVisibilityFilter = (state) => state. visibilityfilter

const getTodos = (state) => state.todos

const getVisibleTodos = createSelector(

getVisibilityFilter, getTodos]，

(visibilityFilter, todos) => {

开关(visibilityFilter) {

例“SHOW\_ALL”:

返回待办事项

例“SHOW\_COMPLETED”:

返回待办事项。过滤器(t = > t.completed)

例“SHOW\_ACTIVE”:

返回待办事项。过滤器(t = > ! t.completed)

}

}

)

const mapStateToProps = (state) = bbb {

返回{

待办事项:getVisibleTodos(状态)

}

}

=连接(

mapStateToProps,

)(基于网络的)

出口默认VisibleTodoList

带有redux状态管理库的典型选择器示例

**{callstack.com}**

16

**最终引导反应原生优化|第一组**

常见的糟糕性能实践是认为状态管理库可以用基于React上下文的自定义实现代替。一开始它可能很方便，因为它减少了状态管理库引入的样板代码。但是使用这种机制而没有适当的记忆将导致巨大的性能缺陷。您可能会最终将状态管理重构为redux，因为对您当前的解决方案来说，实现自定义选择器机制会更容易。

还可以在单个组件级别上优化应用程序。简单地使用纯组件而不是常规组件，并为函数组件使用备忘录包装器将节省大量的重新呈现。乍一看可能没有影响，但在显示大数据集的列表中使用非记忆组件时，您将看到差异。对于组件优化来说，通常已经足够了。

不要尝试提前实现这些技术，因为这样的优化很少使用，而且只在非常特定的情况下使用。

**优点:所需资源更少，应用程序更快**

您应该始终把应用程序的性能放在脑后，但不要试图提前优化所有内容，因为通常不需要这样做。你最终会把时间浪费在解决根本不存在的问题上。

大多数难以解决的性能问题都是由围绕状态管理的糟糕体系结构决策造成的，因此请确保其设计良好。只要您使用纯组件或备忘录包装器，特定组件不应该引入问题。

毕竟，记住所有这些步骤后，应用程序应该执行更少的操作，需要更少的资源来完成其工作。因此，这将导致更低的电池使用和总体上，更多的满意度与界面的互动。o更低的电池使用量，总体来说，界面交互带来更多的满意度。

**{callstack.com}**

17

**最终引导反应原生优化|第一组**

**2.使用专用的**



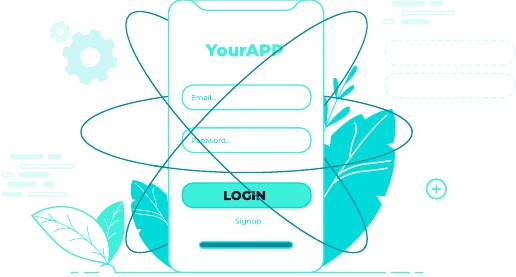
**特定布局的组件**

了解如何使用专用的高阶React本机组件来改善用户体验和应用程序的性能。

**问题:您不知道React Native提供的高阶组件**

在React Native应用程序中，一切都是组件。在组件层次结构的末尾，有所谓的原始组件，如文本、视图或TextInput。这些组件由React Native实现，由您的目标平台提供，以支持最基本的用户交互。

在构建应用程序时，我们用较小的构建块将其组合起来。为此，我们使用原始组件。例如，为了创建登录屏幕，我们将使用一系列TextInput组件来注册用户详细信息，并使用一个可触摸组件来处理用户交互。这种方法从我们在应用程序中创建的第一个组件开始就适用，在开发的最后阶段也适用。



**{callstack.com}**

18

**最终引导反应原生优化|第一组**

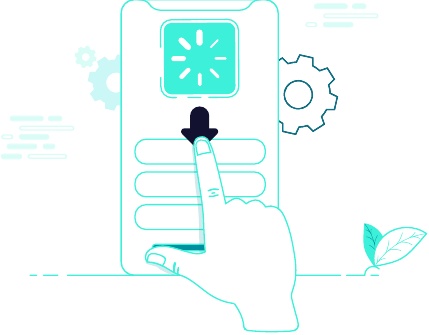
在原始组件之上，使用一组为特定目的而设计和优化的高阶组件对原生舰船做出反应。

如果没有注意到它们，或者没有在所有地方使用它们，可能会潜在地影响应用程序性能，特别是在使用实际生产数据填充状态时。应用程序的糟糕性能可能会严重损害用户体验。因此，它会使你的客户对你的产品不满意，并使他们转向你的竞争对手。

**不使用专门的组件将影响您的性能和用户体验**

**数据的增长**

如果您不使用专门的组件，那么在您的应用程序进入生产环境时，您就放弃了性能改进，并且有降低用户体验的风险。值得注意的是，在开发应用程序时，某些问题仍然没有被注意到，因为模拟数据通常很小，不能反映生产数据库的大小。



专用组件更全面，有更广泛的API来覆盖绝大多数移动场景。

**解决方案:总是使用专门的组件，例如列表的FlatList**

让我们以长列表为例。每个应用程序在某个时候都包含一个列表。

创建元素列表的最快也是最糟糕的方法是组合ScrollView和View原始组件。

**{callstack.com}**

19

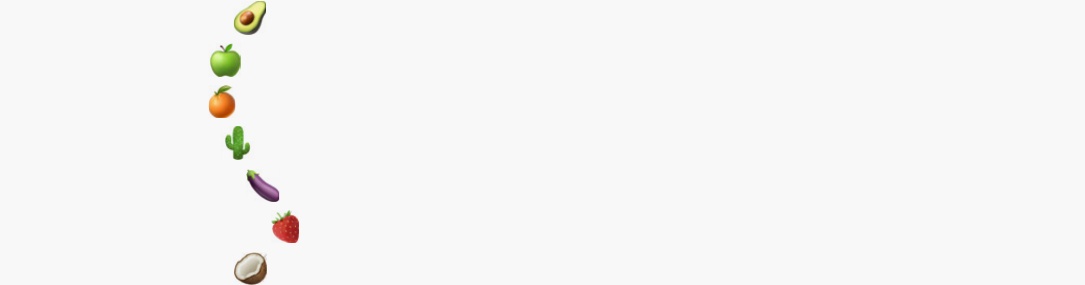
**最终引导反应原生优化|第一组**

但是，当数据增长时，这样的示例很快就会陷入麻烦。处理大型数据集、无限滚动和内存管理是FlatList背后的动机——React Native中的专用组件，用于显示和处理这样的数据结构。比较基于ScrollView添加新列表元素的性能，

从' React '导入React， {Component, useCallback, useState};

从' response -native '导入{ScrollView, View, Text, Button};

const objects = [



[“鳄梨”、“],[‘苹果’,‘],[' orage ', ' '],[“仙人掌”、“],[‘茄子’,‘],[‘草莓’,‘],[“椰子”、“],

];

getRanomItem = () => {

const item = objects[~~(Math.random() \* objects.length)];

返回{

名称:项目[0],

图标:条目[1],

id: Date.now() + Math.random()，

};

};

const \_items = Array.from(new Array(5000)).map(() => getRanomItem());

导出默认函数列表(){

const [items, setItems] = useState(\_items);

const addItem = useCallback() => {

setItems ([getRanomItem ()] .concat(项目));

**{callstack.com}**

20.

**最终引导反应原生优化|第一组**

},[项目]);

回报(

<视图风格= {{marginTop: 20}} >

<Button title= " add item " onPress={addItem} />

<滚动视图>

{物品。map(({name, icon}) => (

<视图

风格= {{

borderWidth: 1、

保证金:3,

填充:5

flexDirection:“行”,

}}>

<Text style={{fontSize: 20, width: 150}

<Text style={{fontSize: 20}}>{icon}</Text>

< /视图>

)}

< /滚动视图>

< /视图>

);

}

阅读更多:https://snack.expo.io/qjtEVHrdV

以FlastList为基础的列表。

从' React '导入React， {Component, useCallback, useState};

从' response -native '导入{View, Text, Button, FlatList, SafeAreaVie};

const objects = [

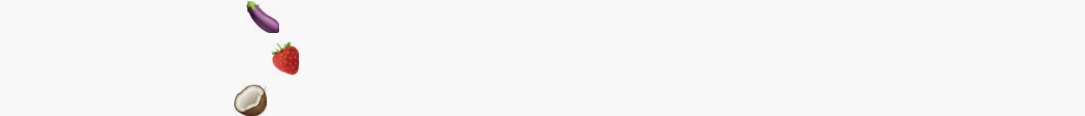


[“鳄梨”、“],[‘苹果’,‘],[' orage ', ' '],[“仙人掌”、“],

**{callstack.com}**

21

**最终引导反应原生优化|第一组**



(“茄子”、“),

[草莓，']，[椰子，']，

];

getRanomItem = () => {

const item = objects[~~(Math.random() \* objects.length)]。分割(' ');

返回{

名称:项目[0],

图标:条目[1],

id: Date.now() + Math.random()，

};

};

const \_items = Array.from(new Array(5000)).map(() => getRanomItem());

导出默认函数列表(){

const [items, setItems] = useState(\_items);

const addItem = useCallback() => {

setItems ([getRanomItem ()] .concat(项目));

},[项目]);

回报(

<View style={{marginTop: 20}}>

<Button title= " add item " onPress={addItem} />

< FlatList

data ={物品}

keyExtractor={({id}) => id}

renderItem={({item: {name, icon}}) => (

<视图

风格= {{

borderWidth: 1、

保证金:3,

填充:5

**{callstack.com}**

22

**最终引导反应原生优化|第一组**

flexDirection:“行”,

}}>

<Text style={{fontSize: 20, width: 150}}>{item[0]}</Text>

<Text style={{fontSize: 20}}>{item[1]}</Text>

< /视图>

)}

/>

< /视图>

);

}

阅读更多:https://snack.expo.io/1muB1wKya

差别很大，不是吗?在提供的5000个列表项的示例中，ScrollView版本甚至不能流畅地滚动。

最后，FlatList还使用了ScrollView和View组件——这是怎么回事呢?

嗯，关键在于在FlatList组件中抽象出来的逻辑。它包含许多启发式和高级JavaScript计算，以减少在屏幕上显示数据时发生的不必要的渲染，并使滚动体验始终以60帧每秒的速度运行。

在某些情况下，仅仅使用FlatList可能还不够。*FlatList性能优化支持不呈现当前未显示在屏幕上的元素。*这个过程中最昂贵的部分是布局测量。*FlatList必须测量你的布局，以确定在滚动区域应该为即将到来的元素保留多少空间。*

对于复杂的列表元素，它可能会显著降低与平面列表的交互速度。每次FlatList将接近渲染下一批数据，它将不得不等待所有新项目渲染来测量它们的高度。

**{callstack.com}**

23

**最终引导反应原生优化|第一组**



但是，您可以实现getItemHeight()来预先定义元素高度，而不需要度量。对于没有恒定高度的项目来说，它是不直接的。您可以根据文本行数和其他布局约束计算该值。我们建议使用response -native-text-size库一次性计算所有列表项显示的文本的高度。在我们的例子中，它显著提高了android上FlatList滚动事件的响应能力。

**好处:你的应用程序工作更快，显示复杂的数据结构，你可以选择进一步改进**

由于使用了专门的组件，您的应用程序将始终尽可能快地运行。您可以自动选择React Native到目前为止所做的所有性能优化，并订阅后续更新。

与此同时，您还节省了从头重新实现大多数常见UI模式的大量时间。粘性部分标题，拉刷新-你的名字。如果选择使用FlastList，默认情况下已经支持这些功能。

**{callstack.com}**

24

**最终引导反应原生优化|第一组**

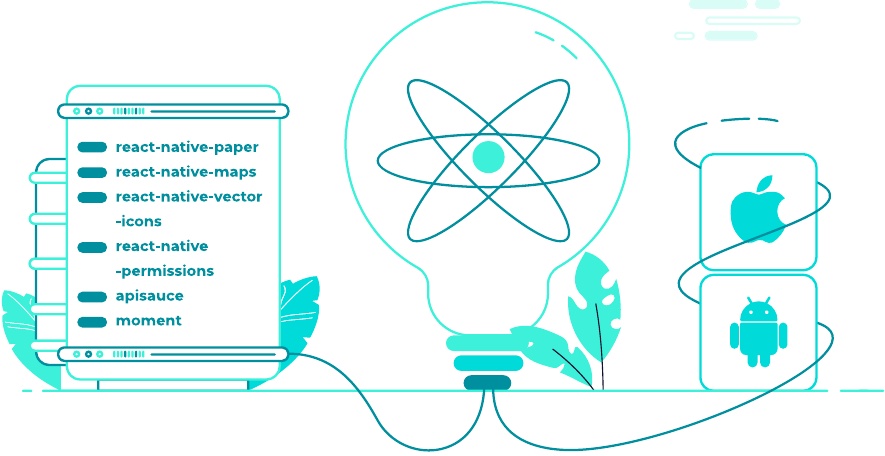
**3.在选择外部库之前请三思**



如何使用正确的JavaScript库来帮助您提高应用程序的速度和性能。

**问题:你在选择库时没有检查里面有什么**

JavaScript开发就像是用较小的块组装应用程序。在某种程度上，它非常类似于构建React本机应用程序。您不必从头开始创建React组件，而是要寻找能够帮助您实现目标的JavaScript库。JavaScript生态系统促进了这种开发方法，并鼓励围绕小而可重用的模块构造应用程序。



这种生态系统有很多优点，但也有一些严重的缺点。其中之一是开发人员很难从支持相同用例的多个库中进行选择。

**{callstack.com}**

25

**最终引导反应原生优化|第一组**

在选择下一个项目要使用的项目时，他们通常会研究一些指标，这些指标告诉他们这个库是否健康和维护良好，比如Github之星、问题数量、贡献者和PRs。

他们往往忽略了库的大小、支持的特性数量和外部依赖关系。他们认为，既然React Native都是关于JavaScript的，并且包含了现有的工具链，那么他们将使用与制作web应用程序时相同的约束和最佳实践。

事实是——它们不会，因为移动开发从根本上是不同的，并且有自己的一套规则。例如，虽然在web应用程序的情况下，资产的大小是至关重要的，但是在React Native中，资产位于文件系统中，它并不同等重要。

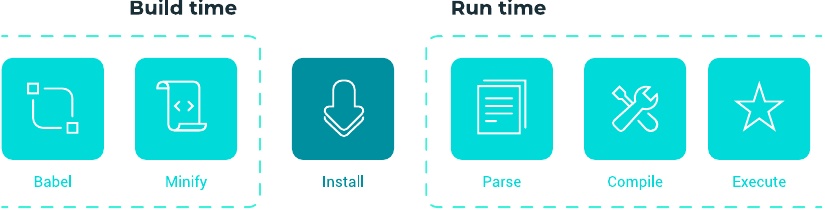
关键的区别在于移动设备和用于绑定和编译应用程序的工具的性能。

虽然您不能对设备限制做太多，但您可以控制您的JavaScript代码。通常，更少的代码意味着更快的打开时间。而影响代码总体大小的最重要因素之一就是库。

**复杂的库会影响应用程序的速度**

与完全本机应用程序不同，React本机应用程序包含一个需要加载到内存中的JavaScript包。然后由JavaScript VM解析和执行。JavaScript代码的总体大小是一个重要因素。

IMAGE



阅读更多:https://snack.expo.io/7H5S504j3

**{callstack.com}**

26

**最终引导反应原生优化|第一组**

当这种情况发生时，应用程序仍然处于加载状态。我们经常把这个过程描述为交互式的TTI - time。从图标从应用程序折叠项中被选中到它完全交互之间的时间间隔(希望是毫秒)。

不幸的是，Metro——默认的React本机绑定器——目前还不支持树抖动。如果您不熟悉这个概念，请阅读本文。

这意味着您从“npm”中提取并导入到项目中的所有代码都将出现在您的生产包中，加载到内存中并进行解析。

这可能会对应用程序的总启动时间产生负面影响。

**解决方案:更有选择性，使用更小的、专门的库**

克服这个问题的最简单的方法是使用正确的策略预先构建项目。

如果您准备提取一个复杂的库，请检查是否有较小的替代库具有您正在寻找的功能。

这里有一个例子:最常见的操作之一是操作日期。让我们假设您将要计算经过的时间。而不是拉下整个moment.js库(67.9 Kb)来解析日期本身:

从“瞬间”中引入瞬间

日期=时刻(“1995年12月25日”，“MM-DD-YYYY”);

用moment.js解析日期

我们可以使用day.js(只有2Kb)，它要小得多，并且只提供我们想要的功能。

**{callstack.com}**

27

**最终引导反应原生优化|第一组**

从' dayjs '导入dayjs

const date = dayjs(" 12-25-1995 "， " MM-DD-YYYY ");

使用day - js解析日期

如果没有其他选择，好的经验法则是检查是否可以导入库的较小部分。

例如，像lodash这样的许多库已经将自己分割成更小的实用程序集，并支持无法消除死代码的环境。

假设你想要使用lodash map。而不是导入整个库，如下所示:

从“lodash”导入{map};

const square = x => x \* x;

地图([4、8],广场);

通过导入整个库使用lodash地图

您只能导入单个包:

从“lodash/map”导入地图;

const square = x => x \* x;

地图([4、8],广场);

通过导入单个函数使用lodash映射

因此，您可以从作为lodash包一部分的实用程序中获益，而无需将它们全部放入应用程序包中。

**好处:你的应用程序加载速度更快，这可以起到很大的作用**

移动领域是一个竞争极其激烈的环境，许多应用程序都是为了类似的目的而设计的，它们都在争夺相同的客户。更快的启动时间、更流畅的交互以及整体的外观和感觉可能是你脱颖而出的唯一方法。

**{callstack.com}**

28

**最终引导反应原生优化|第一组**

根据Akamai的在线零售业绩报告，只要移动加载时间延迟一秒钟，转化率就会下降20%。

这就是为什么您不应该忽视选择正确的库集的重要性。

对第三方依赖项进行更有选择性乍一看似乎是不相关的。但是随着时间的推移，所有节省下来的毫秒都会累积成显著的收益。

**{callstack.com}**

29

**最终引导反应原生优化|第一组**

**4.总是记得**



**使用专用于移动平台的库**

使用专门用于移动设备的库，同时在许多平台上更快地构建特性，而不会影响性能和用户体验。

**问题:您使用的web库没有针对移动设备进行优化**

正如前面所讨论的，关于React Native最好的事情之一是，您可以使用JavaScript编写移动应用程序，重用一些React组件，并使用您喜欢的状态管理库执行业务逻辑。

虽然React Native提供了与web兼容的类web功能，但重要的是要理解它不是同一个环境。它有自己的一套最佳实践、快速成功和约束。

例如，在处理web应用程序时，我们不必过多担心应用程序所需的总体CPU资源。毕竟，大多数网站运行的设备要么是插入网络的，要么是大电池的。

不难想象，移动设备是不同的。有许多不同的设备具有不同的架构和资源可用。大多数时候，它们是在电池上运行的，而由应用程序造成的电量耗尽可能是许多开发人员的决定因素。

**{callstack.com}**

30.

**最终引导反应原生优化|第一组**



换句话说，如何优化前台和后台的电池消耗会产生很大的影响。

**没有优化的库会导致电池消耗和应用程序运行速度变慢。操作系统可能会限制应用程序的功能。**

虽然React Native使得在移动设备上运行与在浏览器中相同的JavaScript成为可能，但这并不意味着您应该每次都这样做。每个规则都有例外。

如果库严重依赖于网络，比如实时消息传递，或者提供了渲染高级图形(3D结构、图表)的能力，那么很可能最好使用专用的移动库。

原因很简单——这些库首先是在web环境中开发的，假定浏览器的功能和限制。使用流行SDK的web版本的结果很可能会导致额外的CPU和内存消耗。

某些操作系统，如iOS，会不断分析应用程序消耗的资源，以优化电池寿命。如果您的应用程序注册来执行后台活动，而这些活动占用了太多的资源，那么应用程序的间隔可能会被调整，从而降低您最初注册的后台更新的频率。

**{callstack.com}**

31

**最终引导反应原生优化|第一组**

**解决方案:使用专用的、特定于平台的库版本**

让我们以Firebase为例。Firebase是谷歌的一个移动平台，它可以让你更快地构建你的应用程序。它是一个工具和库的集合，可以在应用程序中立即启用某些特性。

Firebase包含了用于网络和移动设备的sdk——分别是iOS和Android。每个SDK都包含对实时数据库的支持。



多亏了React Native，你可以运行它的web版本而没有大的问题:

从' firebase/database '导入数据库;

数据库()

.ref(/用户/ 123)

.on(' value '， snapshot => {

控制台。log(' User data: '， snapshot.val());

});

从Firebase实时数据库中读取数据的示例

然而，这不是您应该做的。虽然上面的例子工作没有问题，但它不提供相同的性能等效的移动。SDK本身包含的特性也更少——这并不奇怪——web是不同的，没有理由Firebase.js应该提供对移动特性的支持。

**{callstack.com}**

32

**最终引导反应原生优化|第一组**

在这个特定的示例中，最好使用一个专用的Firebase库，它在专用的本地sdk之上提供一个薄层，并提供与其他本地应用程序相同的性能和稳定性。



下面是上面的例子的样子:

从' @ response -native-firebase/database '导入数据库;

数据库()

.ref(/用户/ 123)

.on(' value '， snapshot => {

控制台。log(' User data: '， snapshot.val());

});

从Firebase实时数据库中读取数据的示例

如您所见，差异很小，可以归结为不同的import语句。在这种情况下，库的作者在模仿API方面做得很好，以减少在web和移动上下文之间来回切换时可能产生的混乱。

**{callstack.com}**

33

**最终引导反应原生优化|第一组**

**优点:在不损害电池寿命的情况下提供最快和最高性能的支持**

React Native给了您选择如何构建应用程序的控制权和自由。

对于简单的事情和最大的可重用性，您可以选择使用该库的web版本。这将使您能够以相对较低的代价访问与浏览器相同的功能。

对于高级用例，您可以轻松地使用本地功能扩展React Native，并直接与移动sdk对话。这样的“逃生舱”使React Native极其通用，并为企业做好了准备。

它能让你在许多平台上同时更快地构建特性，而不会牺牲性能和用户体验——这是其他混合框架做出的标准折衷。

**{callstack.com}**

34

**最终引导反应原生优化|第一组**

**5.找到平衡点**



**在原生和JavaScript之间**

找到本机和JavaScript之间的和谐，以构建快速工作和易于维护的应用程序。

**问题:在处理本机模块时，您在本机和JavaScript抽象之间划了一条错误的线**

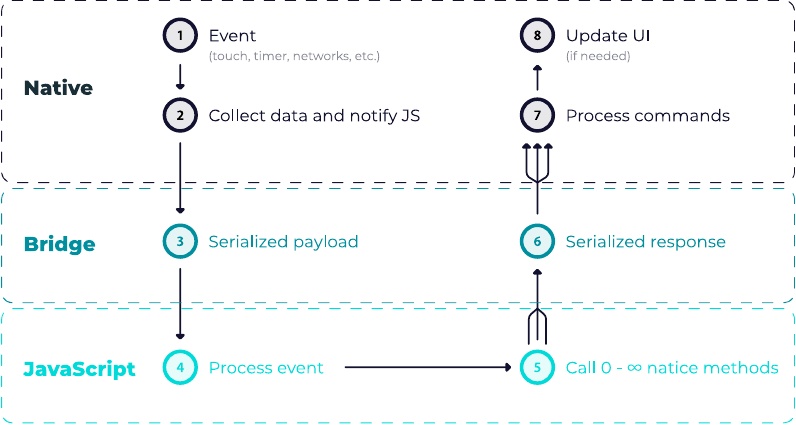
在使用React Native时，大部分时间都是在开发JavaScript。但是，在某些情况下，您需要编写一些本机代码。例如，您正在使用的第三方SDK还没有正式的React本机支持。在这种情况下，您需要创建一个本机模块，该模块包装底层本机方法并将它们导出到React本机领域。

所有本机方法都需要实际的参数才能工作。React本机构建在称为桥的抽象之上，桥提供了JavaScript和本机世界之间的双向通信。因此，JavaScript可以执行本机api并传递必要的上下文来接收所需的返回值。通信本身是异步的——这意味着当调用者在等待本地端的结果到达时，JavaScript仍然在运行，并且可能已经在执行其他任务了。

**{callstack.com}**

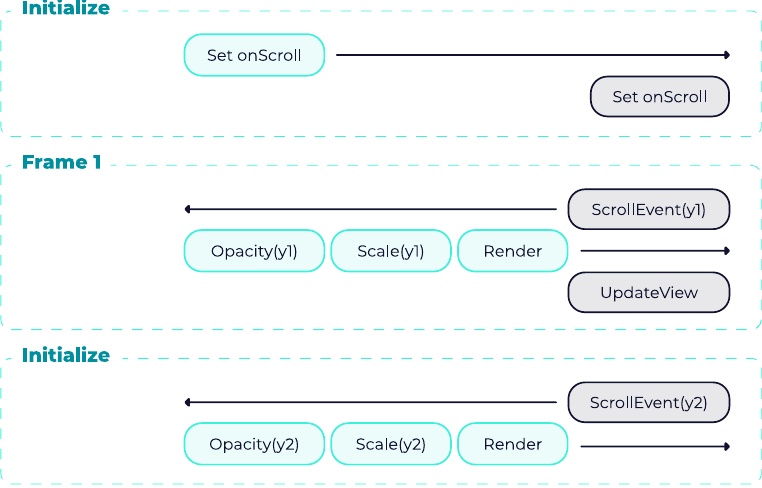
35

**最终引导反应原生优化|第一组**



到达桥接的JavaScript调用的数量是不确定的，并且可能随着时间的推移而变化，这取决于您在应用程序中进行的交互的数量。此外，每次调用都需要时间，因为JavaScript参数需要被stringified为JSON, JSON是这两个领域可以理解的已建立的格式。

例如，当网桥正忙于处理数据时，另一个调用将不得不阻塞并等待。如果交互与手势和动画有关，那么很有可能您有一个被删除的帧——某个操作没有执行，导致UI中出现抖动。



**{callstack.com}**

36

**最终引导反应原生优化|第一组**

某些库,如动画提供特殊的解决方法——在这种情况下,使用NativeDriver——序列化动画,通过一次前期本机线程,不过桥动画运行时,防止它受到意外下降而发生的另一种工作。这就是为什么保持桥梁沟通的高效和快速是很重要的。

**过桥的车辆越多，其他东西的空间就越少**

在桥上通过更多的车辆意味着有更少的空间用于其他重要的反应可能想要在那时转移。因此，在执行本机调用时，应用程序可能对手势或其他交互失去响应。

如果您在桥接器上执行某些本机调用时发现UI性能下降，或者看到大量的CPU消耗，那么应该仔细查看使用外部库所做的事情。很可能转移的资金比应该转移的要多。

**解决方案:在JS端使用适当数量的抽象——提前验证和检查类型**

在构建本机模块时，很容易将调用立即代理到本机端，并让它完成其余的工作。但是，在一些情况下，比如无效参数，会导致不必要的桥上往返，结果发现我们没有提供正确的参数集。

让我们以一个简单的JavaScript模块为例，它仅将调用直接代理给底层本机模块。

从“反应-本机”导入{NativeModules};

= NativeModules;

输出const show =(消息，持续时间)=> {

ToastExample。显示(消息、持续时间)

};

绕过参数到本机模块

**{callstack.com}**

37

**最终引导反应原生优化|第一组**

如果参数不正确或丢失，本机模块可能会抛出异常。React Native的当前版本没有提供for的抽象

确保JavaScript参数和本机代码所需的参数同步。您的调用将被序列化为JSON，传输到本机端并执行。

该操作将不会出现任何问题，即使我们没有传递它工作所需的完整参数列表。当本机端处理调用并从本机模块接收到异常时，错误将在下一个时刻到达。

在这种情况下，您会浪费一些时间等待可以预先检查的异常。

从“反应-本机”导入{NativeModules};

= NativeModules;

输出const show =(消息，持续时间)=> {

if (typeof message !== ' string ' ||消息。长度> 100){

抛出新的错误(“无效的吐司内容!”)

}

if (!Number.isInteger(duration) 2万){

抛出新的错误(“无效的吐司持续时间!”)

}

ToastExample。显示(消息、持续时间)

}

使用带参数验证的本机模块

以上内容不仅与本机模块本身绑定。值得注意的是，每个React Native原语组件都有它的原生对等物，而且每次发生渲染时组件道具都会通过桥传递——就像使用JavaScript参数执行原生方法一样。

为了更好地理解这一点，让我们仔细看看React原生应用程序中的样式。

**{callstack.com}**

38

**最终引导反应原生优化|第一组**

从‘React’导入\* as React;

从“反应-本机”导入{View};

导出默认类App extends React。组件{

呈现(){

回报(

<View style={{flex: 1, justifyContent: ' center '， alignItems: ' center '}}>

<视图风格= {{

写成backgroundColor:“珊瑚”,

宽度:200年,

身高:200

}} / >

< /视图>

);

}

}

阅读更多:https://snack.expo.io/7H5S504j3

给组件设置样式的最简单方法是给它传递一个带有样式的对象。当它起作用的时候，你不会看到它经常发生。它通常被认为是一种反模式，除非您处理的是动态值，比如根据状态更改组件的样式。

从‘React’导入\* as React;

从' response -native '导入{View, StyleSheet};

导出默认类App extends React。组件{

呈现(){

回报(

<视图风格= {styles.caontainer} >

<视图风格={风格。盒子}/ >

< /视图>

);

}

}

**{callstack.com}**

39

**最终引导反应原生优化|第一组**

const styles =样式表。create({

caontainer: {

flex: 1、

justifyContent:“中心”,

alignItems:“中心”

},

箱:{

写成backgroundColor:“珊瑚”,

宽度:200年,

身高:200,

},

});

阅读更多:https://snack.expo.io/GUFPWl8BD

React Native大多数时候使用样式表API通过桥来传递样式。这个API处理你的样式并确保它们只通过桥传递一次。在运行时，它用一个数字唯一标识符替换style prop的值，该标识符对应于本机端缓存的样式。

因此，桥接器现在必须处理数字数组，这样处理和传输就容易得多了，而不是每次都发送大量的对象来响应本机，而是重新呈现它的UI。

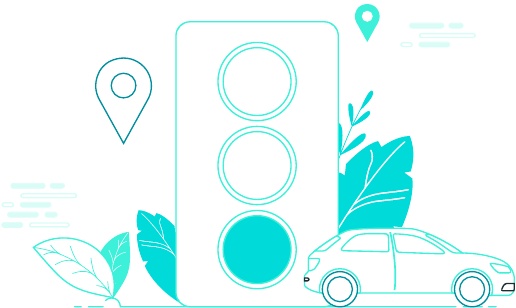
**优点:代码库更快，更容易维护**

无论您现在是否面临性能方面的挑战，围绕本机模块实现一组最佳实践都是一个很好的实践，因为其好处不仅在于速度，而且还在于用户体验。

**{callstack.com}**

40

**最终引导反应原生优化|第一组**



当然，保持过桥流量的适当数量最终将有助于提高应用程序的性能并使其平稳运行。如您所见，本节中提到的某些技术已经在React Native中得到积极使用，以提供令人满意的开箱即用性能。了解它们将帮助您创建在高负载下性能更好的应用程序。

然而，值得指出的另一个好处是维护。

在JavaScript端保持繁重和高级的抽象(如验证)将导致非常薄的本机层，它只不过是一个底层本机SDK的包装器。换句话说，模块的本机部分看起来更像是直接从文档复制粘贴过来的——非常简单且容易理解。

掌握这种开发本机模块的方法是很多人的原因

JavaScript开发人员可以很容易地通过附加功能扩展他们的应用程序，而不需要专门学习Objective-C或Java。

**{callstack.com}**

41

**最终引导反应原生优化|第一组**

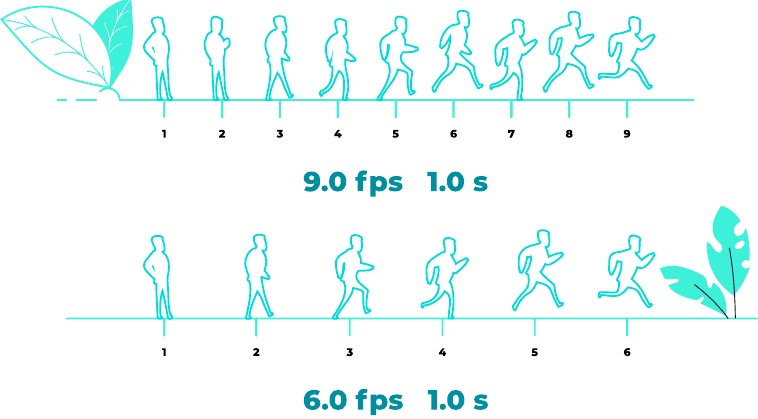
**6.以60FPS的速度动画**



使用本地解决方案实现60FPS的流畅动画和手势驱动界面。

**问题:js驱动的动画占用了桥上的交通并降低了应用程序的速度**

移动用户习惯于流畅、设计良好的界面，能够快速响应他们的交互并提供及时的视觉反馈。当涉及到web开发时，这并不是一个真正的情况，但在移动设备上却是不同的。因此，应用程序必须在许多地方注册大量的动画，当其他工作正在进行时，这些动画必须运行。



**{callstack.com}**

42

**最终引导反应原生优化|第一组**

正如我们从上一节了解到的，可以通过网桥传递的信息量是有限的。目前还没有内置的优先队列。换句话说，应该由您来构造和设计应用程序，使业务逻辑和动画都能在不中断的情况下运行。这与我们用来执行动画的方式不同。例如，在iOS上，内置的api提供了前所未有的性能，并且总是以适当的优先级进行调度。长话短说——我们不必太担心确保它们以60帧每秒的速度运行。

对于React Native，这个故事有点不同。如果您事先不考虑自顶向下的动画，并选择正确的工具来解决这个问题，那么您迟早会遇到丢失帧的情况。

**粗糙或缓慢的动画会影响应用程序的感觉，使它看起来缓慢和未完成**

在当今应用程序的海洋中，提供流畅和交互式的UI可能是赢得客户选择应用程序的唯一途径之一。如果您的应用程序不能提供与用户交互(如手势)良好工作的响应界面，不仅会影响新客户，而且还会降低ROI和用户情绪。

移动用户喜欢跟随他们的界面，看起来是一流的，确保动画总是顺利运行是建立这种体验的基本部分。

**解决方案:如果可能的话，使用本地和正确的动画**

**一次性的动画**

启用本机驱动程序是快速提高动画性能的最简单方法。然而，可以与本机驱动程序一起使用的样式道具子集是有限的。你可以将它用于非布局属性，如变换和不透明度。它与颜色、高度和其他元素都不兼容。这些已经足够执行大多数了

**{callstack.com}**

43

**最终引导反应原生优化|第一组**

因为你通常想要显示/隐藏一些东西或者改变它的位置。

const fadeAnim = useRef(新建动画。value (0)).current;

常量fadeIn = () => {

动画。时间(fadeAnim, {

toValue: 1、

持续时间:1000

useNativeDriver: true， //启用本地驱动

}) .start ();

};

/ /[…]

<动画。视图样式={{不透明度:fadeAnim}}/>

为不透明度动画启用本地驱动程序

对于更复杂的用例，您可以使用React Native Reanimated库。它的API与基本的动画库兼容，并引入了一组底层函数来控制你的动画。更重要的是，它介绍了可能性动画所有可能的风格道具与本地驱动程序。所以动画的高度和颜色将不再是一个问题。然而，转换和不透明度动画仍然会稍微快一些，因为他们是GPU加速。但是普通用户应该看不到任何区别。

**Gesture-driven动画**

最期望的效果，可以实现的动画是能够控制动画与手势。对于您的客户来说，这是界面中最令人愉快的部分。它建立了强烈的情感，使应用程序感觉非常平稳和响应。Plain React Native在结合手势和本机驱动动画时非常有限。你可以利用ScrollView滚动事件来构建平滑可折叠的标题。

**{callstack.com}**

44

**最终引导反应原生优化|第一组**

对于更复杂的用例，有一个很棒的库——React Native Gesture Handler——它允许你原生地处理不同的手势，并将它们插入到动画中。您可以通过将其与动画相结合来构建一个简单的可滑动元素。然而，它仍然需要JS回调，但有一个补救办法!

手势驱动动画最强大的一对工具是使用手势处理程序和恢复动画。它们被设计成一起工作，并提供了构建复杂的手势驱动动画的可能性，这些动画完全在本地端计算。这里唯一的限制是您的想象力(和编码技能，因为恢复低级API不是那么简单)。

从“React”导入React， {Component};

从'反应-本机'导入{样式表，视图};

从' response -native-gesture-handler '导入{PanGestureHandler, State};

从“reaction -native-reanimated”导入动画;

从' ./runSpring '中导入runSpring;

const {

集,

气孔导度,

情商,

添加、

用,

不超过,

春天,

startClock,

stopClock,

clockRunning,

子,

定义,

值,

时钟,

事件,

SpringUtils,

} =动画;

**{callstack.com}**

45

**最终引导反应原生优化|第一组**

类可获取扩展组件{

构造函数(道具){

超级(道具);

const TOSS\_SEC = 0.2;

const dragX = new Value(0);

const state = new Value(-1);

const dragVX =新值(0);

这一点。\_onGestureEvent =事件([

{nativeEvent: {translationX: dragX, velocityX: dragVX, state: state}}，

]);

const transX = new Value();

const prevDragX = new Value(0);

const clock = new clock ();

cond(

(add(transX, multiply(TOSS\_SEC, dragVX))， 0)，

-100年,

One hundred.

);

这一点。\_transX =电导率(

eq(状态,State.ACTIVE),

(

stopClock(时钟),

设置(transX，添加(transX，子(dragX, prevDragX))，

集(prevDragX dragX),

transX,

),

(

集(prevDragX, 0),

**{callstack.com}**

46

**最终引导反应原生优化|第一组**

集(

transX,

cond(定义(transX)， runSpring(时钟，transX, dragVX, snapPoint)， 0)

),

]

);

}

呈现(){

孩子们，…rest} = this.props;

回报(

< PanGestureHandler

其他{…}

maxPointers = {1}

minDist = {10}

onGestureEvent = {this.\_onGestureEvent}

onHandlerStateChange = {this.\_onGestureEvent} >

<动画。视图样式={{转换:[{translateX: this。\_transX}]}} >

{孩子}

< / Animated.View >

< / PanGestureHandler >

);

}

}

导出默认类示例扩展组件{

呈现(){

回报(

<视图风格= {styles.container} >

< Snappable >

<视图风格={风格。盒子}/ >

< / Snappable >

< /视图>

);

}

}

**{callstack.com}**

47

**最终引导反应原生优化|第一组**

const BOX\_SIZE = 100;

const styles =样式表。create({

容器:{

flex: 1、

justifyContent:“中心”,

alignItems:“中心”,

写成backgroundColor:“# F5FCFF ',

},

箱:{

宽度:BOX\_SIZE,

高度:BOX\_SIZE,

borderColor:“# F5FCFF ',

alignSelf:“中心”,

写成backgroundColor:“李子”,

页边距:BOX\_SIZE / 2，

},

});

阅读更多:https://snack.expo.io/EM0KZfwJd

手势的低级处理可能不是一件容易的事，但幸运的是，已经有第三方库利用上面提到的工具并公开callbacknode。*CallbackNode只不过是一个动画。价值，但它是从特定的手势行为派生出来的。*它的值范围通常从0到1，跟随手势的进程。您可以将值插入到屏幕上的动画元素中。公开CallbackNode的一个很好的库示例是reanimed -bottom-sheet和response -native-tab-view。

从‘React’导入\* as React;

从' response -native '中导入{样式表，文本，视图};

从“reaction -native-reanimated”导入动画;

从“reanimate -bottom-sheet”导入底板;

从' ./Lorem '导入Lorem;

const {block, set, greaterThan, lessThan, Value, cond, sub，插值}=

**{callstack.com}**

48

**最终引导反应原生优化|第一组**

动画;

导出默认类示例扩展React。组件{

gestureCallbackNode = new Value(0);

contentPos = this.gestureCallbackNode;

renderHeader = name => (

<视图

风格= {{

宽度:100%,

写成backgroundColor:“lightgrey”,

高度:40岁

borderWidth: 2

}}>

<Text style={{textAlign: ' center '， fontSize: 20, padding: 5}}>拖动</Text>

< /视图>

);

renderInner = () => (

<视图风格={{写成backgroundColor:‘lightblue}} >

< Animated.View

风格= {{

透明度:插入(这一点。contentPos, {inputRange: [0, 1],

outputRange (1,0):}),

变换:[{

translateY:插入(这一点。contentPos, {inputRange: [0, 1],

outputRange: [0100]}),

})

}}>

< Lorem / >

< Lorem / >

< / Animated.View >

< /视图>

);

**{callstack.com}**

49

**最终引导反应原生优化|第一组**

呈现(){

回报(

<视图风格= {styles.container} >

< BottomSheet

callbackNode = {this.gestureCallbackNode}

snapPoints = {[400]}

initialSnap = {1}

renderHeader = {this.renderHeader}

renderContent = {this.renderInner}

/>

< /视图>

);

}

}

const IMAGE\_SIZE = 200;

const styles =样式表。create({

容器:{

flex: 1、

},

});

阅读更多:https://snack.expo.io/KFpkVKYB9

**降低你的JS操作的优先级**

完全控制动画的实现方式并不总是可能的。例如，React导航使用了React原生手势处理程序和Animated，而后者仍然需要JavaScript来控制动画运行时。因此，如果您正在导航以加载繁重的UI的屏幕，您的动画可能会开始闪烁。幸运的是，您可以使用InteractionManager延迟这类操作的执行。

**{callstack.com}**

50

**最终引导反应原生优化|第一组**

从' React '导入React， {useState, useRef};

进口{

文本,

看来,

样式表,

按钮,

动画,

InteractionManager,

平台,

从“react-native”};

从“expo-constants”导入常量;

const ExpensiveTaskStates = {

notStared:“没有开始”,

计划:“安排”,

:“做”,

};

导出默认函数App() {

useRef(新建animationValue (100));

const [animationState, setAnimationState] = useState(false);

const [expensiveTaskState, setExpensiveTaskState] = useState

ExpensiveTaskStates.notStared

);

const startAnimationAndSchedlueExpensiveTask = () = bbb {

Animated.timing (animationValue。目前,{

持续时间:2000

toValue: animationState ?100: 300,

useNativeDriver:假的,

= > {}) .start (()

setAnimationState (! animationState);

});

setExpensiveTaskState (ExpensiveTaskStates.scheduled);

InteractionManager.runAfterInteractions (() = > {

**{callstack.com}**

51

**最终引导反应原生优化|第一组**

setExpensiveTaskState (ExpensiveTaskStates.done);

});

};

回报(

<视图风格= {styles.container} >

{平台。OS === ' web ' ?(

<Text style={{textAlign: ' center '}}>

!InteractionManager只能在本地平台上工作。iOS上的开放示例

或Android !

< /文本>

):(

<>

<按钮

title= "开始动画并安排昂贵的任务"

onPress = {startAnimationAndSchedlueExpensiveTask}

/>

< Animated.View

风格={[风格。框，{width: animationValue。当前}]}>

> <文本框动画文本> < /

< / Animated.View >

<文本样式= {styles.paragraph} >

昂贵的任务状态:{' '}

<Text style={{fontWeight: ' bold '}}>{expensiveTaskState}</Text>

< /文本>

</>

)}

< /视图>

);

}

const styles =样式表。create({

容器:{

flex: 1、

justifyContent:“中心”,

alignItems:“中心”,

**{callstack.com}**

52

**最终引导反应原生优化|第一组**

paddingTop: Constants.statusBarHeight,

填充:8

},

段:{

保证金:24日

字形大小:18,

textAlign:“中心”,

},

箱:{

写成backgroundColor:“珊瑚”,

marginVertical: 20,

高度:50,

},

});

阅读更多:https://snack.expo.io/Wv8u ! mKwJ

这个方便的React本机模块允许在所有运行动画完成后执行任何代码。在实践中，您可以显示一个占位符，等待动画完成，然后呈现实际的UI。它将使您的JavaScript动画顺利运行，并避免被其他操作中断。通常足够平滑，可以提供很好的体验。

**优点:60fps享受流畅的动画和手势驱动的界面**

在React Native中没有唯一正确的动画制作方法。这个生态系统充满了不同的图书馆和处理交互的方法。本节建议的思想只是建议您不要想当然地认为平滑的界面是理所当然的。

更重要的是在您的头脑中描绘出应用程序中所有交互的自顶向下的画面，并选择正确的方法来处理它们。在有些情况下，javascript驱动的动画会工作得很好。与此同时，在交互中，本地动画(或完全本地视图)将是使其平滑的唯一方法。

**{callstack.com}**

53

**最终引导反应原生优化|第一组**

使用这种方法，您创建的应用程序将更加平滑和快速。它不仅会让您的用户感到愉快，而且会让您在开发过程中调试并从中获得乐趣。

**如果您需要帮助的性能，稳定性，用户体验或其他复杂的问题-联系我们!作为反应本机的核心贡献者和**

**社区的领导们，我们会很乐意帮忙的。**

**{callstack.com}**

54



**第二组**

通过使用最新的React本机特性来提高性能。

**{callstack.com}**

**最终引导反应原生优化|第二组**

**介绍**

**React Native正在快速增长，其特性的数量也在增长**

去年，开发者为React Native core做出了3670多个承诺。这个数字看起来令人印象深刻，但实际上它甚至更大，因为它不包括在React Native Community组织下做出的较小的贡献(9678个提交)。

所有这些都证明了“自然反应”正在以一个真正健康的速度发展。社区和Facebook的贡献使得这个框架能够使用越来越多的高级用例。Hermes就是一个很好的例子——这是一个全新的JavaScript引擎，专门为React Native和Android设计。Hermes的目标是取代JavaScriptCore, JavaScriptCore之前在安卓和iOS上都使用过。它还带来了许多企业级的优化——它改善了你的Android应用程序的性能，启动时间，并减少了它的整体大小。

在本节中，我们将向您展示一些可以立即启用的特性，以便开始优化过程。我们还鼓励您跟踪React Native的所有新特性，以确保您充分发挥框架的潜力。

**{callstack.com}**

**最终引导反应原生优化|第二组**

**1.总是使用最新的**

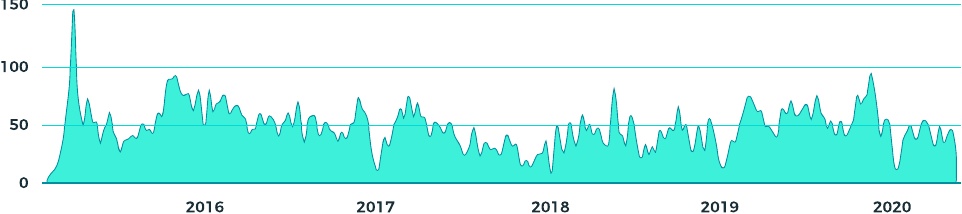


**响应本机版本以访问新特性**

升级你的应用程序到最新版本，以获得更多的功能和更好的支持。

**问题:您运行的是一个旧的、不受支持的React Native版本，使您无法获得新的改进和特性**

让您的应用程序与您使用的框架保持同步是至关重要的。这样，您就可以订阅最新的特性、性能改进和安全修复程序。JavaScript生态系统在这方面特别有趣，因为它发展得非常快。如果你不定期更新你的应用程序，你的代码很可能会远远落后于升级它将变得痛苦和危险。



React Native就是这些快速发展的库之一。

每天，来自世界各地的开发人员都会引入新的特性、关键的bug修复和安全补丁。平均而言，每个版本包含大约500次提交。社区中最流行的变化包括快速刷新或

**{callstack.com}**

57

**最终引导反应原生优化|第二组**

*自动油墨-这两者我们将在下面的部分中描述。*

在React Native生态系统中，库不向后兼容是很常见的。新特性通常使用了以前版本中没有的好东西。这意味着，如果您的应用程序运行在较旧的React本机版本上，您最终将开始错过最新的改进。

|  |  |
| --- | --- |
| @react-native-community / cli | react-native |
| ^ 5.0.0(下) | 主 |
| ^ 4.0.0(主) | ^ 0.62.0 |
| ^ 3.0.0 | ^ 0.61.0 |
| ^ 2.0.0 | ^ 0.60.0 |
| ^ 1.0.0 | ^ 0.59.0 |

这就是为什么，跟上最新的React本地升级似乎是唯一的办法。

不幸的是，在每个新版本中都要升级您的React本机代码，因此需要进行一些严肃的工作。它的数量将取决于本地功能和核心部分的底层更改的数量。大多数时候，您必须仔细分析和比较您的项目与最新版本，并自己做出调整。如果您已经适应了在本地环境中移动，那么这个任务会更容易完成。但如果你和我们大多数人一样，这可能有点挑战性。

例如，您在代码中使用的模块和组件可能不再是反应本机核心的一部分。

**{callstack.com}**

58

**最终引导反应原生优化|第二组**

这可能是因为Facebook在“精益核心”(lean core)过程中引入的变化。这项努力的目标是:

-通过提取核心的某些部分并将其移动到response -native-community存储库，使response -native包更小、更灵活、更容易维护。

-将提取模块的维护转移到社区。

这一过程加速了特定模块的成长，并使整个生态系统更好地组织起来。但它也对反应性升级产生了一些负面影响。现在，您必须安装提取的包作为附加依赖，直到您这样做，您的应用程序将不会编译或崩溃在运行时。然而，从开发人员的角度来看，迁移到社区包通常不过是引入新的依赖并重写导入。

另一个重要的问题是第三方的支持。您的代码通常依赖于外部库，并且存在与最新的React本机版本不兼容的风险。

至少有两种方法可以解决这个问题:

-等待项目维护人员执行必要的调整之前，你升级，-寻找替代方案或补丁自己的模块-使用一个方便的实用程序

称为补丁包或创建具有必要修复的临时fork。

**在旧版本上运行意味着附带一些问题，可能会让用户感到泄气**

如果您运行的是旧版本，那么您很可能落后于使用最新版本框架的竞争对手。

React Native框架中的修复、改进和改进的数量确实令人印象深刻。如果你玩的是追赶游戏，你可以选择退出大量的更新，这会让你的生活变得更容易。进行常规升级所涉及的工作负载和成本总是被直接的DX(开发人员体验)增强所抵消。

在本节中，我们将介绍一些完善的实践，以简化升级对新版本的反应。

**{callstack.com}**

59

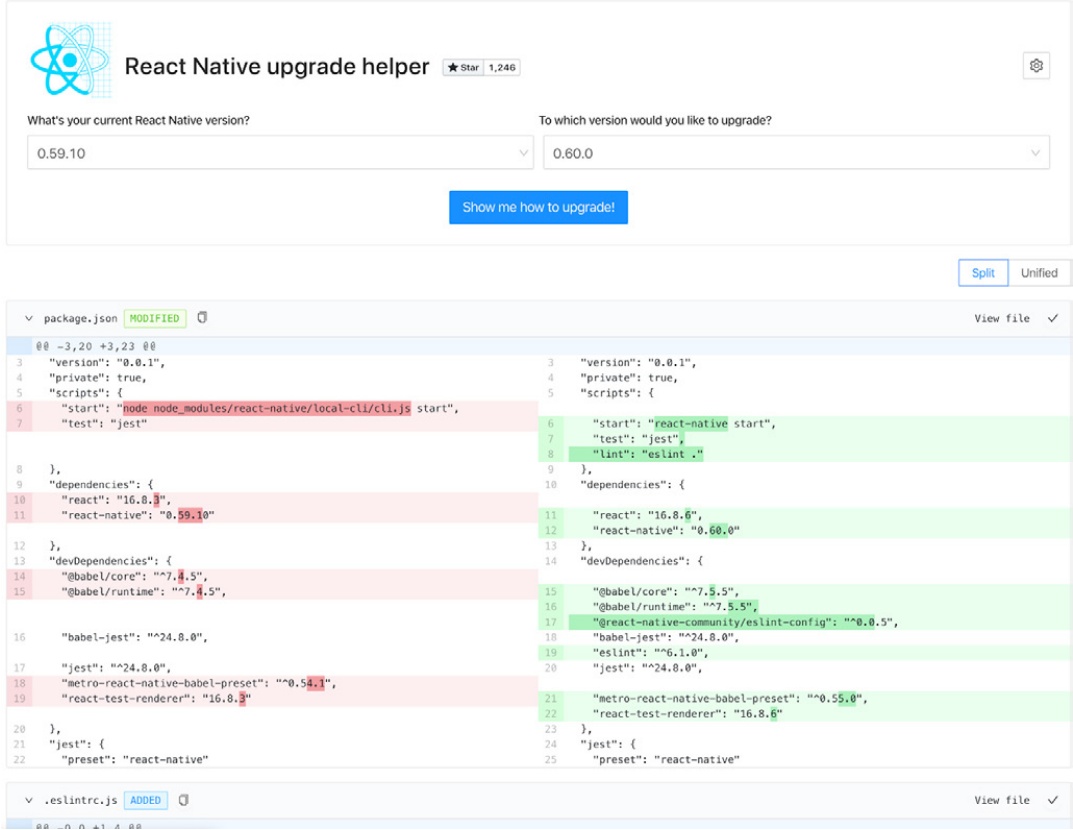
**最终引导反应原生优化|第二组**

**解决方案:升级到最新的React原生版本(我们将向您展示如何进行)**

升级反应原生可能不是世界上最容易的事情。但是有一些工具可以使这个过程变得更简单，并且消除大多数问题。实际的工作量将取决于更改的数量和您的基本版本。但是，无论应用程序的状态如何，本节提供的步骤都可以应用于每次升级。

**为升级做准备**

React Native Upgrade Helper是一个很好的起点。在较高的层次上，它提供了自上次升级本地版本以来响应本机的更改的概述。



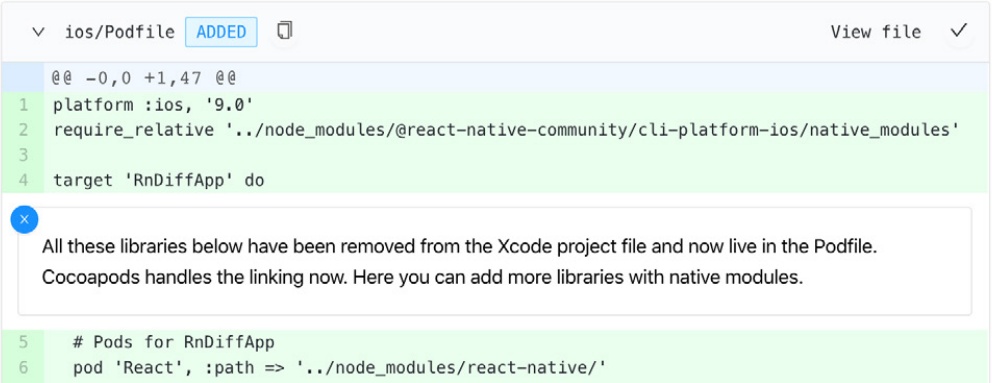
不同的包。json '之间反应本机0.59和反应本机0.60

**{callstack.com}**

60

**最终引导反应原生优化|第二组**

为此，helper将运行' npx response - Native init '创建的bare React本机项目与您的版本和您正在升级的版本进行比较。接下来，它显示了项目之间的差异，让您了解在此期间发生的每一个小修改。一些变化可能会被附加注释一个特殊的信息，这将给为什么某些事情发生提供更多的上下文。



对用户文件的更有趣更改的附加说明

对变更有一个更好的概述将帮助您更快地行动并更有信心地行动。



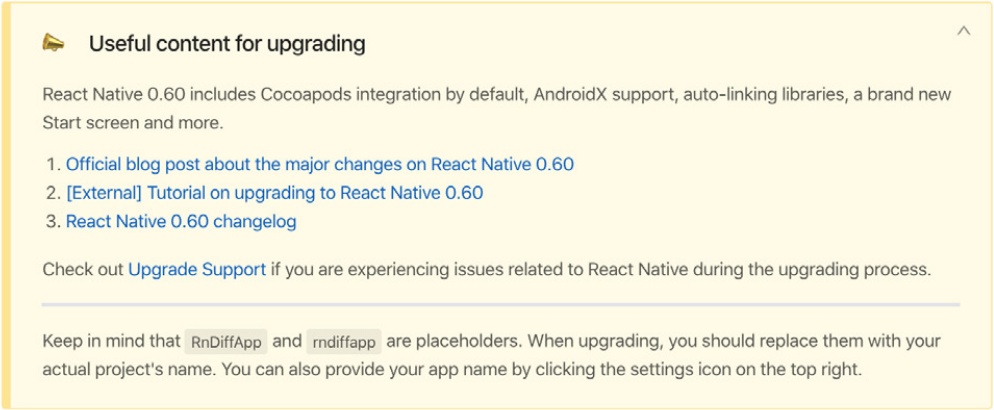
**注意:拥有更多的上下文是非常重要的，因为在升级时没有自动化——你必须自己应用变更。**

React Native Upgrade Helper还建议在升级时阅读有用的内容。在大多数情况下，包括在React Native博客上发表的一篇专门的博客文章以及原始的更新日志。

**{callstack.com}**

61

**最终引导反应原生优化|第二组**



在升级到React Native 0.60时可以阅读的有用内容

我们建议您阅读推荐的资源，以更好地掌握即将发布的版本，并了解其中的亮点。

由于这一点，您不仅会意识到这些变化，而且还会理解它们背后的原因。你将准备好打开你的项目并开始工作。

**应用JavaScript更改**

升级React Native的JavaScript部分的过程类似于升级其他JavaScript框架。我们的建议是一步一步地进行升级——一次升级一个库。在我们看来，这种方法比一次升级所有东西要好，因为它给您更多的控制，并且使捕获回归更容易。

第一步是撞击React并将本机依赖关系反应到所需的版本，并执行必要的更改(包括中断更改)。为此，您可以查找React Native Upgrade Helper提供的建议并手动应用它们。一旦完成，确保重新安装你的' node\_modules '。

**{callstack.com}**

62

**最终引导反应原生优化|第二组**

**注意:在执行升级时，您可能会看到许多来自iOS项目文件的更改(.xcodeproj中的所有内容，包括.pbxproj)。**这些文件是在你使用React Native应用程序的iOS部分时Xcode生成的。

与其修改源文件，不如通过Xcode UI执行更改。这是升级到React Native 0.60的情况，本问题中描述了适当的操作。

最后，您应该尝试运行应用程序。如果一切都很好——完美。升级很顺利，你可以收工了!但更重要的是——现在你应该检查你使用的其他依赖项是否有更新版本!他们可能提供了重要的性能改进。

不幸的是，还有另一种更悲观的情况。你的应用程序可能根本无法构建，或者可能立即崩溃，出现红色屏幕。在这种情况下，很有可能您的一些第三方依赖项不能正常工作，您需要使它们与您的React本机版本兼容。

**注意:如果您的升级有问题，您可以检查升级支持项目。**它是开发人员分享经验并帮助彼此解决与升级相关的一些最具挑战性的操作的存储库。

**升级第三方库**

在大多数情况下，首先应该考虑的是React本机依赖关系。与常规的JavaScript / React包不同，它们通常依赖于本地构建系统和更高级的React本地api。当框架成熟为更稳定的API时，这将使它们暴露于潜在的错误。

如果错误是在构建期间发生的，那么只需将依赖关系提高到其最新版本就可以使其正常工作。

一旦应用程序构建完成，您就可以检查更改日志并熟悉公共API中发生的JavaScript更改了。这一步很简单

**{callstack.com}**

63

**最终引导反应原生优化|第二组**

忽略和通常是运行时异常的结果。使用Flow或TypeScript应该可以确保正确地应用更改。

如您所见，没有什么魔术可以修复所有错误并自动升级依赖项。这主要是一项需要耐心和注意的手工工作。它还需要进行大量的测试，以确保在开发过程中不会破坏任何特性。

**好处:你运行的是最新版本，这意味着更多的功能**

**更好的支持**

升级到最新的React本机版本与保持其他框架和库的更新没有什么不同。除了重要的性能和安全改进，新的React原生版本还解决了iOS和Android最新的底层变化。这包括应用于移动电话的破坏性更改，比如某些api被弃用时。

举个例子:去年谷歌宣布2019年8月1日以后所有提交给谷歌Play的Android应用程序都必须是64位的。为了继续开发您的应用程序并发布新特性，您必须升级到响应本机0.59并执行必要的调整。

这样的升级对于保持用户满意非常关键。毕竟，如果应用程序在新版操作系统下开始崩溃，或者从app Store中消失，他们会很失望。每个版本可能会带来额外的工作负载，但是保持更新会带来更快乐的用户、更稳定的应用程序和更好的开发体验。

**{callstack.com}**

64

**最终引导反应原生优化|第二组**

**2.如何用Flipper调试更快更好**



通过实现Flipper来建立一个更好的反馈循环，在你的应用程序上工作的时候会有更多的乐趣。

**问题:你正在使用Chrome调试器或其他一些奇怪的方式来调试和配置你的React本地应用程序**

调试是每个开发人员日常工作中最具挑战性的部分之一。当一切似乎都能正常工作时，引入新特性相对容易，但发现问题所在可能非常令人沮丧。我们通常会尽可能快地修复bug，尤其是当这些bug非常重要，并且导致应用程序无法正常运行时。在这个过程中，时间是一个重要的因素，我们通常必须敏捷地快速解决问题。



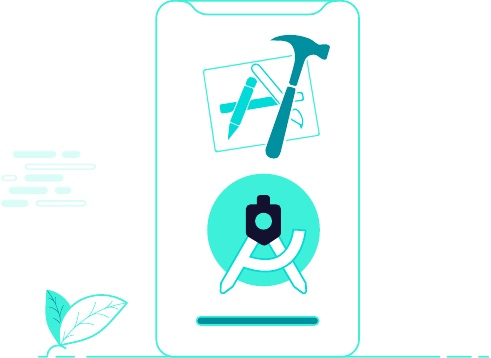
但是，本机调试并不是很直接，因为您试图解决的问题可能在不同级别上发生。即可能由以下原因引起:

**{callstack.com}**

65

**最终引导反应原生优化|第二组**

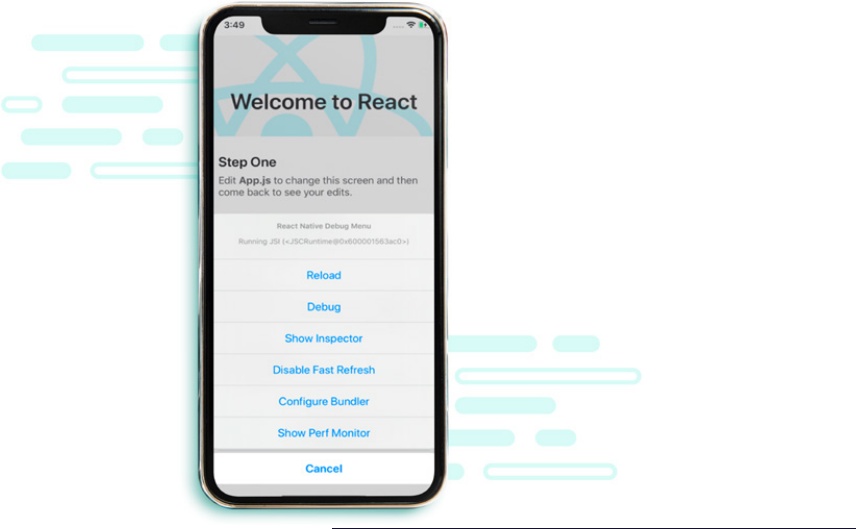
- JavaScript:你的应用程序的代码或反应本机，或-本机代码:第三方库或反应本机本身。



当需要调试本地代码时，你必须使用Android Studio和Xcode中内置的工具。

在调试JavaScript代码时，您可能会遇到一些困难。第一种也是最简单的调试方法是编写控制台。登录您的代码并检查终端中的日志。这种方法只适用于解决小bug，或者使用分治法技术。在所有其他情况下，可能需要使用外部调试器。

默认情况下，React本机附带内置调试实用程序。



**{callstack.com}**

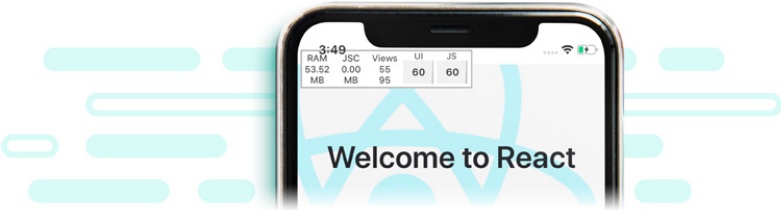
66

**最终引导反应原生优化|第二组**

最常见的是谷歌Chrome调试器。它允许您以一种比在终端中更方便的方式在代码中设置断点或预览日志。不幸的是，使用Chrome调试器可能会导致难以发现的问题。这是因为你的代码是在Chrome的V8引擎中执行的，而不是在JSC或Hermes等特定平台的引擎中执行的。

在Chrome中生成的指令通过Websocket发送到模拟器或设备。这意味着您不能真正使用调试器来分析应用程序，以便它检测性能问题。它可以让您大致了解什么可能导致问题，但是由于WebSocket消息传递的开销，您将无法调试实际情况。

另一个不方便的事实是，你不能用Chrome调试器轻松地调试网络请求(它需要额外的设置，仍然有它的限制)。为了调试所有可能的请求，您必须使用模拟器的developer菜单打开一个专用的网络调试器。然而，由于模拟器的屏幕尺寸，它的界面是非常小和不方便的。



从developer菜单中，您可以访问其他调试工具，如layout inspector或performance monitor。后者使用起来相对方便，因为它只显示一小部分信息。然而，使用前者是一种挣扎，因为它提供的工作空间有限。

**在调试和发现性能问题上花费更多的时间意味着更糟糕的开发人员体验和更少的满意度**

与本地开发人员不同，使用React native的开发人员可以使用广泛的调试工具和技术。每个系统都源自于不同的生态系统，比如iOS、Android或JS。虽然一开始听起来很棒，但你需要记住

**{callstack.com}**

67

**最终引导反应原生优化|第二组**

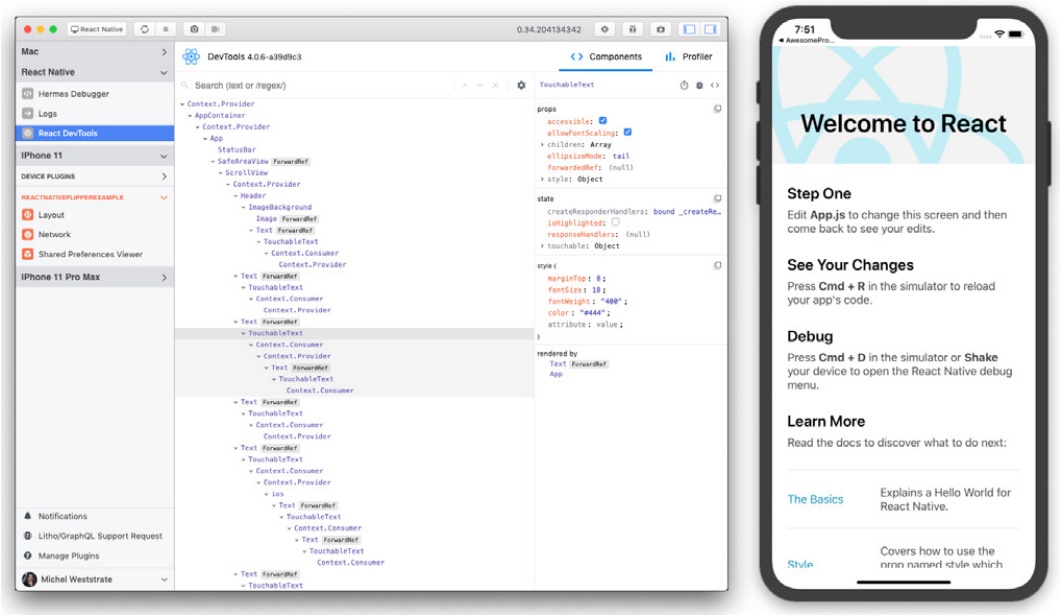
每个工具在本地开发中都需要不同水平的专业知识。这对大多数JavaScript开发人员来说是一个挑战。一个不方便的工具通常会降低团队的速度，并使其成员受挫。结果，它们没有发挥应有的作用。这影响了应用程序的质量，降低了发布频率。

**解决方案:打开Flipper，开始调试**

如果有一个综合的工具来处理上面所有的用例，不是很好吗?当然!这就是Flipper起作用的地方!



Flipper是一个移动应用程序调试平台。它还支持作为一等公民的React Native。自0.62版本以来，它默认随React Native一起发布，并于2019年9月发布。



来源:https://fbflipper.com/docs/features/react-native

**{callstack.com}**

68

**最终引导反应原生优化|第二组**

IIt是一个桌面应用程序与一个方便的界面，它直接集成您的应用程序的JS和本地代码。这意味着你不再需要担心JS运行时的差异和使用Chrome调试器的性能警告。它附带了一个网络检查器、React DevTools，甚至还有本地视图层次工具。

更重要的是，Flipper允许从本地代码预览日志和跟踪本地崩溃，所以你不需要运行Android Studio或Xcode来检查在本地发生了什么!

Flipper很容易扩展，所以社区开发的大量有用插件很有可能丰富了它。现在，您可以使用Flipper执行一些任务，比如检测内存泄漏、预览共享偏好的内容或检查已加载的图像。此外，Flipper for React Native还附带了React DevTools、Hermes调试器和Metro bundler集成。

最令人兴奋的是，所有需要的工具都放在一个桌面应用程序中。这将上下文切换最小化。如果没有Flipper，开发人员在调试一个与显示从后端获取的数据相关的问题时，必须使用Chrome调试器(来预览日志)，仿真器内的网络请求调试器，可能还有仿真器内的布局检查器或独立的React Devtools应用。有了Flipper，所有这些工具都可以作为内置插件使用。他们很容易访问从侧面面板和类似的UI和

用户体验。

**好处:你和React Native一起工作更有趣，建立了更好的反馈循环**

更好的调试过程可以使您的应用程序开发周期更快、更可预测。因此，您的团队能够生成更可靠的代码，并且更容易发现任何类型的问题。

将所有调试工具放在一个界面中绝对是符合人体工程学的，并且不会中断与模拟器或设备的任何交互。对于您的团队来说，这个过程将不那么繁重，并且将对产品开发和bug修复的速度产生积极的影响。

**{callstack.com}**

69

**最终引导反应原生优化|第二组**

**3.自动化**

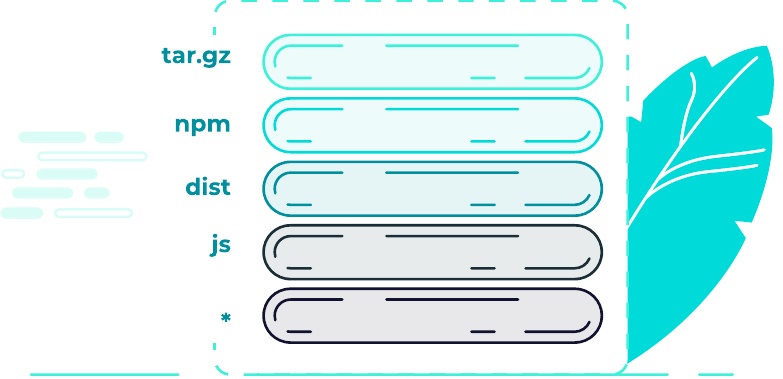


**使用“自动连接”进行依赖管理**

切换到“autolinking”来快速添加新包，而不用担心本地代码。

**问题:您是手动添加库，还是使用不受欢迎的“反应-本机链接”**

与“npm”上的大多数包不同，React本地库通常不仅仅包含一个JavaScript代码。根据它们提供的功能类型，它们可能包含用于Android和iOS平台的额外本机代码。



**{callstack.com}**

70

**最终引导反应原生优化|第二组**

例如，response - Native -fbads是一个用于与底层Facebook SDK交互的React本机模块，正如其名所示，它用于在应用程序中显示广告。为此，模块附带了允许从React原生领域调用SDK的JavaScript代码。除此之外，它还提供了Objective-C(适用于iOS)和Java(适用于Android)原生模块，这些模块将JavaScript调用代理给适当的Facebook SDK部分。它还要求Facebook SDK出现在你的应用程序中。换句话说，它让你自己来安装它。

从历史上看，React Native并没有为这种情况提供开箱即用的解决方案。鼓励开发人员遵循给定平台的最佳实践。

在Android上，推荐使用Gradle，它已经是Android社区的首选平台。React Native used Gradle用于构建它的源代码并拉出它自己的依赖项，这自然地强制所有社区包遵循相同的策略。

另一方面，在iOS上，情况就有点复杂了。默认情况下，React Native项目不使用任何复杂的工具来管理依赖关系——要靠您自己来拉下它们。一些社区模块已经开始使用CocoaPods，类似于Gradle，因为它构建项目的方式和提供了适当的依赖管理。不幸的是，CocoaPods与默认情况下用于管理依赖项的所有本地项目的反应并不兼容。

React Native试图通过引入“response - Native link”来部分解决这个问题——这是一个CLI命令，一旦运行，就会尝试为您执行所有必要的步骤。它在您的配置文件中执行了一个简单的查找和替换，并尝试添加所需的包。



**{callstack.com}**

71

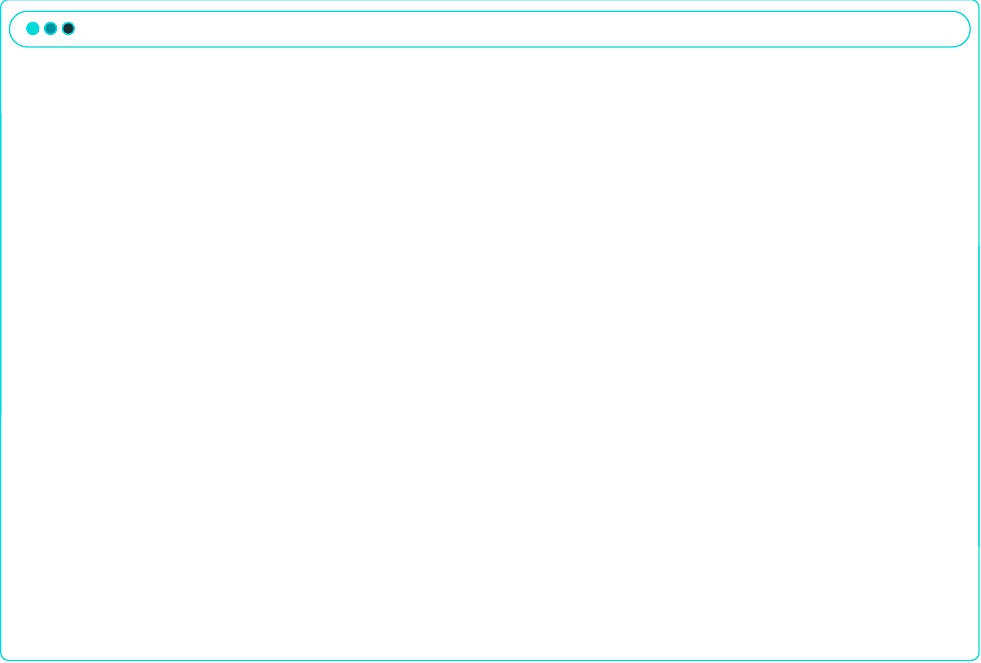
**最终引导反应原生优化|第二组**

不幸的是，总会有碰到与您管理依赖项的方式不兼容的依赖项的风险。在这种情况下，唯一的解决方案是迁移到一个工作的系统。这项任务本身并不容易——它需要大量与本地相关的知识和对构建系统的理解。如果您曾经升级到引入了某些本地更改的React本机版本，那么您将非常清楚我们在讨论什么。

随着时间的推移，CocoaPods开始在社区中变得越来越受欢迎。最终，React Native决定切换到CocoaPods，并将其作为管理iOS上外部依赖的默认方式。

因此，iOS和Android现在都为依赖管理提供了一个功能齐全的解决方案。由于这一点，开发人员可以使用类似于npm的工具来拉下依赖项，而不是手动下载文件并将它们放在磁盘上的某个位置。

虽然这有助于解决添加外部本机依赖时的困惑，但这种情况仍然需要在简单的“纱线添加”之后运行额外的步骤。



**3.安装Javascript包**

使用您喜爱的包管理器将包添加到您的项目中

$ yarn添加反应-本机-fbads链接

**对于RN < 0。60**

如果在Xcode项目中有多个目标，可以通过拖动手动链接其中一些

库/ ReactNativeAdsFacebook.xcodeproj /产品/ libReactNativeAdsFacebook。a to ' Build Phases ' -> '链接二进制与库'

**对于RN >= 0。60**

如果您使用的是RN > 0.60，请在您的Podfile pod ' ReactNativeAdsFacebook '中添加以下行:path => ' ../node\_modules/ response -native-fbads '

**{callstack.com}**

72

**最终引导反应原生优化|第二组**

由于Gradle和CocoaPods都有一个可用于操作项目的公共API, React Native团队很快发布了一个名为autolinking的特性。它自动化了上面提到的所有步骤，并消除了React本机和JavaScript包之间的差异。

$纱线添加反应本土的fbads

安装React本机包应该与常规的JavaScript库没有区别

长话短说——如果您在安装React本机模块后执行其他步骤，您应该继续阅读!

**代码基更难升级和维护，您要花更多的时间来添加额外的包**

如果您仍然按照上面描述的“遗留方式”管理您的依赖项，那么您就会错过构建改进和自动化。因此，试验新的依赖关系变得更加具有挑战性，并且需要花费更长的时间来设置它们。一些库甚至可能在开发人员将它们迁移到新的构建系统时停止工作。

此外，您需要花费更多的时间来升级到较新的React本机版本，因为有一大堆本机依赖项和本机代码需要修改和升级。

新系统基于专用的本地构建工具，如CocoaPods和Gradle。正因为如此，它能够为你处理很多细致的步骤。

**解决方案:切换到“自动上墨”(以椰子为基础)**

*Autolinking是一种管理本地依赖关系的新方法，从设计上来说，它是完全透明的，不需要您做任何额外的工作。*它很容易集成，并且它会挂在你必须自己处理的地方。

它在iOS和Android上的工作方式是一样的。在本节中，让我们关注Android。

**{callstack.com}**

73

**最终引导反应原生优化|第二组**

**之前**

**后**

app / settings.gradle

app / settings.gradle

' ' '

' ' '

rootProject.name =“HelloWorld”

rootProject.name =“HelloWorld”

包括:react-native-fs的

项目(“:react-native-fs”)。

projectDir =新文件(rootProject)

管理扫描,“. . / node\_modules /反应-

native-fs / android”)

应用:文件(“. . / node\_modules / @

react-native-community / cli-platform -

android / native\_modules.gradle”);

applyNativeModulesSettingsGradle

(设置);

包括:应用程序的

包括:应用程序的

' ' '

' ' '

app / build.gradle

app / build.gradle

' ' '

' ' '

依赖关系{

依赖关系{

(': react-native -编译项目

编译fileTree (dir:“填词”,

包括:[" \* . jar "])

fs”)

“com.android编译。

编译fileTree (dir:“填词”,

支持:appcompat-v7:23.0.1”

包括:[" \* . jar "])

“com.facebook编译。

“com.android编译。

react:react-native:+”//来自node\_

支持:appcompat-v7:23.0.1”

模块

“com.facebook编译。

}

react:react-native:+”//来自node\_

应用:文件(“. . / . . / node\_modules /

模块

}

@react-native-community / cli-platform -

android / native\_modules.gradle”);

' ' '

applyNativeModulesAppBuildGradle

(项目)

' ' '

**{callstack.com}**

74

**最终引导反应原生优化|第二组**

MainApplication.java

' ' '

进口com.rnfs.RNFSPackage;

MainApplication.java

' ' '

进口com.facebook.react。

PackageList;

…

…

@Override

保护列表< ReactPackage >

@Override

保护列表< ReactPackage >

getPackages () {

getPackages () {

返回数组。< ReactPackage > asList (

返回新PackageList(这个)。

新MainReactPackage (),

getPackages ();

新的RNFSPackage ()

}

);

}

' ' '

' ' '

您不用让Gradle知道您正在使用的每个包的详细信息，而是用调用React本机CLI的一行代码替换包列表。这个小助手检查你的“包”。json '可能对本地包做出反应，并自动执行必要的操作。

这里有一个例子:在build中。*gradle，对React本机CLI的调用会产生一组包，然后在管道中注册这些包。*值得注意的是，路径是根据源文件的位置动态计算的。因此，所有不同的非标准架构，包括流行的“monorepo”，现在默认支持。

autolinking的另一个伟大特性是，它为您在Android上生成包的列表。正因为如此，所有由外部依赖定义的包都可以自动注册，而不需要打开Android Studio并学习如何导入Java包。

这里的原则很简单——你不需要知道你下载的库是由什么组成的。探索这些细节的可能性应该留给最好奇的开发人员来选择。

**{callstack.com}**

75

**最终引导反应原生优化|第二组**

**好处:您可以快速添加新包，而不用担心本机代码**

由于使用了autolinking，您可以忘记常规JavaScript之间的所有差异，并对本地包做出反应，而专注于构建应用程序。



您不再需要担心外部依赖关系或其他构建步骤，包括提取SDK或链接资产。

从长远来看，您将欣赏这种易于维护和升级速度的方法。CocoaPods、Gradle和React原生CLI助手确保设置自动连接和在应用程序中使用自动连接所需的知识尽可能基本，并且易于JavaScript开发人员掌握。

**{callstack.com}**

76

**最终引导反应原生优化|第二组**

**4.优化你的安卓**

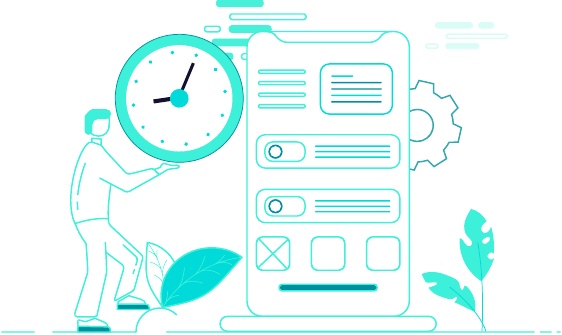


**使用Hermes的应用程序启动时间**

使用Hermes实现更好的应用性能。

**问题:你在启动时加载了很多Android软件包，这是不必要的。此外，你使用的引擎不是为Android优化的。**

用户希望应用程序具有响应性和快速加载。未能满足这些要求的用户最终会在App Store或Play Store中收到糟糕的评价。在最极端的情况下，它甚至会被竞争对手抛弃。



用单一的度量标准来描述启动时间并不容易。这是因为加载阶段有很多不同的阶段，它们会影响应用程序的“快”或“慢”程度。例如，在Lighthouse报告中，有六个性能指标用于分析web应用程序。其中之一是交互时间(简称TTI)，它测量应用程序准备好进行第一次交互之前的时间。

**{callstack.com}**

77

**最终引导反应原生优化|第二组**

当你第一次从抽屉里按下应用程序图标时，会发生很多事情。

1本地呈现

2 JS渲染

3 JS Init +要求

4本地初始化



加载过程从本地初始化(1)开始，它加载JavaScript VM并初始化所有本地模块。然后它继续从磁盘(2)读取JavaScript，将其加载到内存中，解析并开始执行。此操作的详细信息在有关为应用程序选择正确库的部分前面已经讨论过。

在下一步(3)中，React Native开始加载React组件，并将最后一组指令发送给UIManager。*最后，UIManager处理从JavaScript接收到的信息，并开始执行本机指令(4)，这将产生最终的本机接口。*

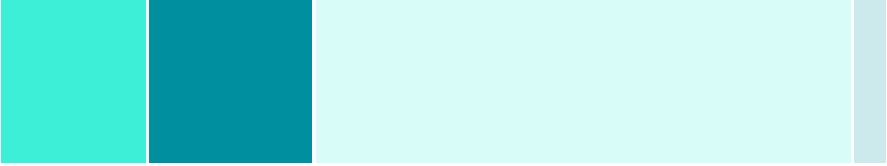
正如您在上图中看到的，有两组操作会影响应用程序的总体启动时间。

1本地呈现

2 JS渲染

3 JS Init +要求

4本地初始化



**{callstack.com}**

78

**最终引导反应原生优化|第二组**

第一个涉及图中的操作1和操作2，并描述了React Native启动(启动VM和VM执行JavaScript代码)所需的时间。另一个包含剩余的操作3和4，并与您为应用程序创建的业务逻辑相关联。这个组的长度高度依赖于组件的数量和应用程序的总体复杂性。

本节关注第一组改进—与您的配置相关的改进，而不是业务逻辑本身。

如果您还没有测量过应用程序的总体启动时间，或者还没有使用Hermes等工具，请继续阅读。

**在Android上，长启动时间和缓慢的用户体验可能是你的应用获得糟糕评价并最终被抛弃的原因之一**

创建有趣的应用程序是非常重要的，尤其是

考虑到移动市场已经是多么饱和。现在，所有的移动应用程序不仅要容易理解和直观。他们还应该是愉快的互动。

有一种常见的误解，认为与本机应用程序相比，本机应用程序具有性能权衡。事实是，通过足够的关注和配置调整，它们可以同样快速地加载，而且没有任何显著的差异。

**解决方案:开启Hermes以获得更好的性能**

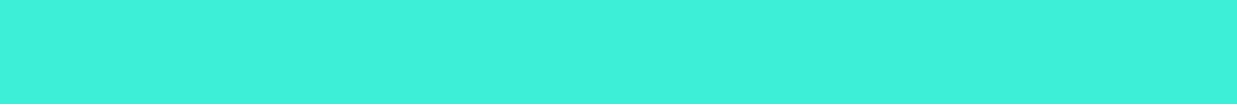
虽然React Native应用程序负责本地接口，但它仍然需要在运行时运行JavaScript逻辑。为此，它创建了自己的JavaScript虚拟机。默认情况下，它使用JavaScriptCore。这个引擎是WebKit的一部分，默认只能在iOS上使用。现在它也是Android上兼容性的首选。这是因为使用V8引擎(Chrome附带的引擎)可能会增加Android和iOS之间的差异，使得在平台之间共享代码变得更加困难。

**{callstack.com}**

79

**最终引导反应原生优化|第二组**

JavaScript引擎的生活并不轻松。它们不断地提供新的启发式方法来提高总体性能，包括加载代码和执行代码所需的时间。为此，他们对常见的JavaScript操作进行基准测试，并对完成此过程所需的CPU和内存提出质疑。



V8团队最近发表了一篇关于改进正则表达式性能的博客文章。一定要去看看。

开发人员处理JavaScript引擎的大部分工作正在主要和最流行的网站上进行测试，如Facebook或Twitter。React Native以不同的方式使用JavaScript，这并不奇怪。例如，为web设计的JavaScript引擎不必太担心启动时间。浏览器很可能在加载页面时已经运行了。由于这个原因，引擎可以将注意力转移到总体CPU和内存消耗上，因为web应用程序可以执行许多复杂的操作和计算，包括3D图形。

正如您在上一节展示的性能图表中所看到的，JavaScript虚拟机消耗了应用程序总加载时间的很大一部分。不幸的是，除非你建立自己的引擎，否则你对此无能为力。这就是Facebook团队最终所做的。



**{callstack.com}**

80

**最终引导反应原生优化|第二组**

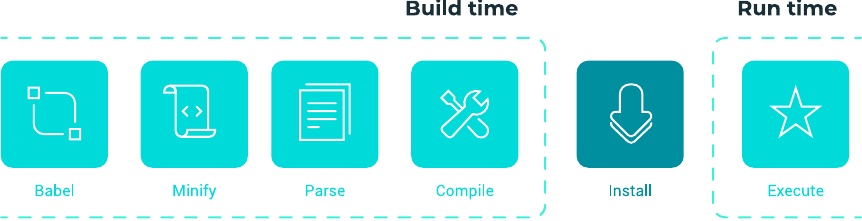
这是一个专门为React Native设计的JavaScript引擎。它针对移动设备进行了优化，专注于对cpu不敏感的指标，比如应用程序的大小或交互时间。目前，它只能在Android上使用，未来可能会支持iOS。

在深入了解在现有的React本机应用程序中启用Hermes的细节之前，让我们先看看它的一些关键架构决策。

**字节码precompilation**

通常，传统的JavaScript VM在运行时解析JavaScript源代码，然后生成字节码。结果，代码的执行被延迟到解析完成。这和爱马仕不一样。为了减少引擎执行业务逻辑所需的时间，它在构建期间生成字节码。

IMAGE



它可以花更多的时间使用各种技术来优化包，使其更小、更高效。例如，生成的字节码被设计成可以在不加载整个文件的情况下映射到内存中。优化这个过程会带来显著的TTI改进，因为移动设备上的I/O操作往往会增加总体延迟。

**没有JIT**

大多数现代浏览器引擎使用即时(JIT)编译器。这意味着将逐行翻译和执行代码。但是，JIT编译器跟踪

**{callstack.com}**

81

**最终引导反应原生优化|第二组**

*暖代码段(出现几次的代码段)和热代码段(运行多次的代码段)。*然后，这些经常出现的代码段被发送到编译器，根据它们在程序中出现的次数，编译器将它们编译成机器码，并可选地执行一些优化。

赫尔墨斯，不同于其他引擎，是AOT(提前时间)引擎。这意味着整个包提前被编译成字节码。因此，JIT编译器在热代码段上执行的某些优化没有出现。

一方面，这使得Hermes捆绑在面向CPU的基准测试中表现不佳。然而，这些基准并不能真正与现实生活中的移动应用程序体验相媲美，在现实生活中，TTI和应用程序大小是优先考虑的。

另一方面，JIT引擎减少了TTI，因为它们需要时间来解析包并及时执行它。他们还需要时间来“热身”。也就是说，他们必须运行代码几次，以检测常见的模式并开始优化它们。

如果你想开始使用Hermes，请确保你运行的至少是React Native 0.60.4，并在你的“android/app/build.gradle”中打开以下代码:

project.ext。反应= [

entryFile:“index.js”,

enableHermes:真

]

在编写此内容时，“enableHermes”设置为“false”。一定要换成“true”。

由于这一点，您的项目应该成功地清理和重新构建。如果出现这种情况，恭喜您——您的应用程序现在正在使用Hermes。

**好处:更好的启动时间会带来更好的性能。这是一个永无止境的故事。**

快速加载应用程序并不是一项简单的任务。这是一项持续的努力，其最终结果将取决于许多因素。您可以通过调整应用程序的配置和它用于编译源代码的工具来控制其中的一些。

**{callstack.com}**

82

**最终引导反应原生优化|第二组**

启用Hermes是当前可以显著提高应用程序性能的方法之一。

除此之外，您还可以查看Facebook团队提供的其他重要改进。要做到这一点，要熟悉他们对React Native performance的评论。这通常是一个小而简单的改进游戏，一旦应用就会产生很大的不同。

正如我们在关于运行最新的React本机的一节中提到的，只要您保持React本机版本的最新，Hermes就是您可以利用的资产之一。

这样做将帮助您的应用程序保持在性能游戏的顶端，并让它以最大速度运行。

**{callstack.com}**

83

**最终引导反应原生优化|第二组**

**5.优化你的安卓**



**应用程序的大小与这些等级设置**

改善TTI，减少内存使用和你的应用程序的大小调整混淆规则到你的项目。

**问题:你没有启用Proguard用于发布构建和创建APK与所有CPU架构的代码。你要运送更大的APK**

在每个React本机项目的开始阶段，您通常不关心应用程序的大小。毕竟，在这个过程的早期很难做出这样的预测。但是，根据代码库的不同，应用程序从标准的5 MB增长到10、20甚至50 MB只需要几个额外的依赖项。

在移动互联网和WiFi无处不在的时代，你真的应该关注应用程序的大小吗?为什么bundle的规模增长如此之快?我们将在接下来的几段中回答这些问题。但首先，让我们看看一个典型的反应原生包是由什么组成的。

默认情况下，Android上的React本地应用程序包括:- 4组针对不同CPU架构编译的二进制文件，-包含图像、字体等资源的目录，-包含业务逻辑和你的React组件的JavaScript包，-其他文件。

**{callstack.com}**

84

**最终引导反应原生优化|第二组**

React Native提供了一些优化，允许您改进包的结构及其总体大小。但默认情况下是禁用的。

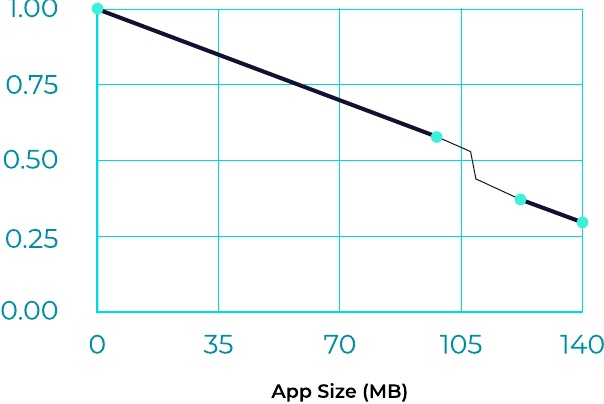
如果没有有效地使用它们，特别是在应用程序增长时，就不必要地增加了应用程序的总体大小(以字节为单位)。这会对最终用户的体验产生负面影响。我们将在下一节中讨论它。

**更大的APK大小意味着需要更多的时间从app store下载，需要更多的字节码加载到内存中**

您和您的团队使用最新的设备，并且能够快速稳定地访问Internet，这真是太好了。但你要记住，不是每个人都有同样的奢侈品。世界上仍有部分地区的网络可访问性和可靠性远非完美。Loon等项目承诺会改善这种状况，但这需要时间。

现在，仍然有市场上每兆流量是有价格的。在这些地区，应用程序的大小直接影响转换，安装/取消比例随着应用程序大小的增加而增加。

**安装速度，不同的应用程序大小**



来源:https://segment.com/blog/mobile-app-size-effect-on-downloads/

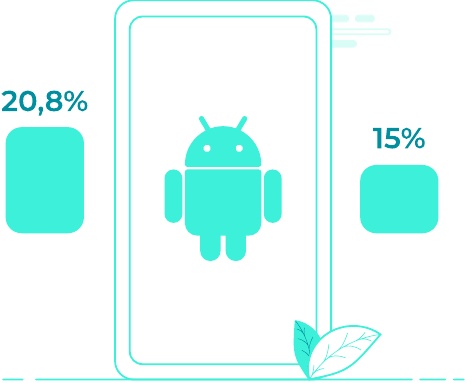
**{callstack.com}**

85

**最终引导反应原生优化|第二组**

人们普遍认为，每一个精心设计的应用程序不仅提供了漂亮的界面，而且针对终端设备进行了优化。不过，情况并不总是这样。由于Android市场竞争如此激烈，有一个很大的机会，一个小的替代那些漂亮但大的应用已经在社区中获得了更多的吸引力。

另一个重要因素是设备碎片。Android市场在这方面是非常多样化的。在处理更大的APKs时，中低端设备中有相当大的份额可能会面临问题。



**2019年3月，只有20.8%的安卓智能手机在高端市场，高于2018年3月的15%**

来源:https://newzoo.com/insights/infographics/10-key-facts-about-the-android-smartphone-market/

正如我们已经强调的那样，应用程序的启动时间是至关重要的。在打开代码时，设备需要执行的代码越多，启动应用程序并为第一次交互做好准备所需的时间就越长。

现在，让我们转移一下在此上下文中值得一提的最后一个因素——设备存储。

安装后，应用程序通常会占用更多空间。有时它们甚至可能不适合进入设备内存。在这种情况下，用户可能决定跳过安装您的产品，如果这意味着删除其他资源，如应用程序或图像。

**{callstack.com}**

86

**最终引导反应原生优化|第二组**

**解决方案:将布尔标志“enab麻风瓜releasebuild”切换为true，根据你的需要调整混淆器规则，测试发布版本是否崩溃。另外，将“enableSeparateBuildPerCPUArchitecture”翻转为true**

Android是一种运行在许多不同架构的设备上的操作系统，因此您的构建必须支持其中的大多数。React Native支持四种:armeabi-v7a、arm64-v8a、x86和x86\_64。

在开发你的应用程序时，Gradle生成可以安装在任何提到的CPU架构设备上的“apk”文件。换句话说，您的“apk”(构建过程输出的文件)实际上是四个被打包成单个文件的独立应用程序。这使得测试更加容易，因为应用程序可以同时分布到许多不同的测试设备上。

不幸的是，这种方法有其缺点。应用程序的总体大小现在比它应该的要大得多，因为它包含了所有架构所需的文件。因此，用户最终将下载无关的代码，这些代码甚至与他们的手机不兼容。

值得庆幸的是，在发布应用程序的生产版本时，可以利用应用程序包来优化发布过程。

*App Bundle是一种发布格式，允许你包含所有已编译的代码和资源。*这都是因为谷歌Play Store动态交付稍后将根据终端用户的设备构建定制的APKs。

要构建App Bundle，你只需要调用一个不同寻常的脚本。使用。/gradlew bundleRelease，而不是使用。/gradlew bundleRelease，如下所示:

$ cd安卓

美元。/ gradlew bundleRelease

构建一个React原生应用作为应用包

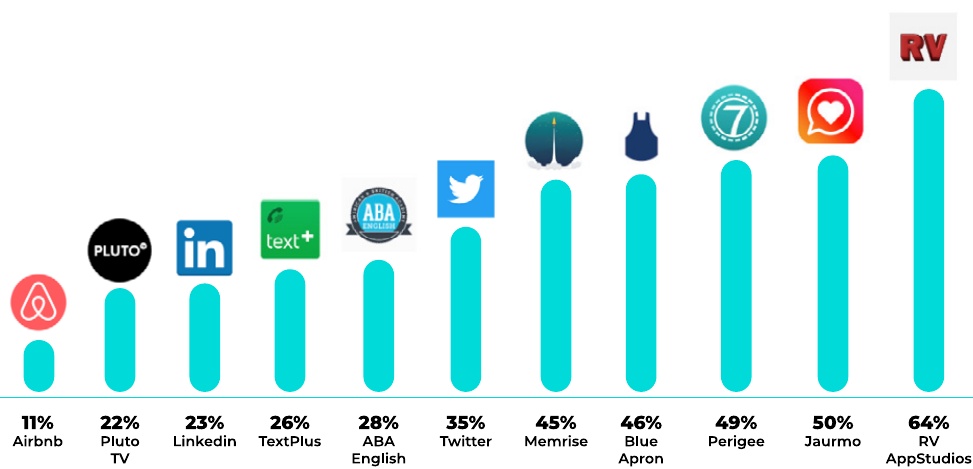
**{callstack.com}**

87

**最终引导反应原生优化|第二组**

Android App Bundle的主要优势是易于交付。毕竟，您必须只交付一个工件，而动态交付将为您实现整个奇迹。它还在受支持的平台上为您提供了更多的灵活性。你不必担心你的终端用户的设备有哪个CPU架构。据Android团队称，一个应用程序的平均尺寸减少约35%，但在某些情况下，甚至可以减少一半

IMAGE



来源:https://medium.com/google-developer-experts/exploring-the-android-app-bundle-ca16846fa3d7

另一种减少构建大小的方法是启用混淆器。Proguard的工作方式类似于从JavaScript中清除死代码——它从第三方sdk中清除无用代码，并缩小代码库。

然而，混淆器可能无法在一些项目中开箱即用，通常需要额外的设置才能达到最佳效果。在本例中，我们能够将提到的28mb版本的大小减少700Kb。这不算多，但仍是一个进步。

def enabmalguardinreleasebuild = true

在“android/app/build.gradle”中启用“proguard”

**{callstack.com}**

88

**最终引导反应原生优化|第二组**

另一个好的实践是关注资源优化。每个应用程序都包含一些可以使用免费web工具优化的svg或png图形。

减少svg中的冗余文本和压缩png图像可以在项目已经有很多字节时节省一些字节。

**优点:更小的APK，更快的TTI，更少的内存使用**

所有提到的步骤都比较容易引入，当您在与不断增长的应用程序大小作斗争时，值得采取。通过为不同的架构构建应用程序，您将获得最大的尺寸缩减。但是可能的优化还不止这些。

通过争取更小的APK大小，您将尽最大努力减少下载取消率。此外，你的客户将受益于更短的互动时间，并更倾向于更频繁地使用该应用程序。

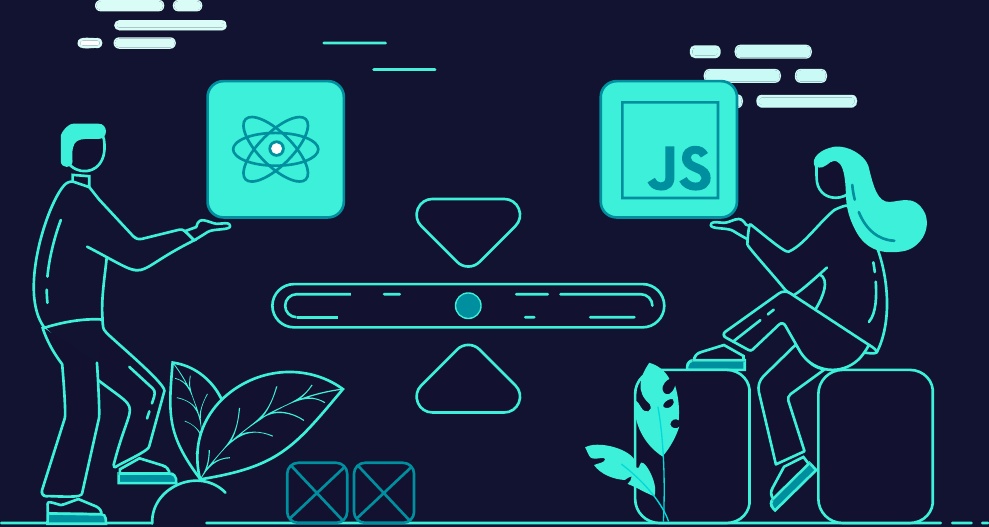
最后，您将证明您关心每一个用户，而不仅仅是那些拥有一流设备和快速互联网连接的用户。你的平台越大，支持这些小群体就越重要，因为每一个百分比的用户都会转化为成千上万的实际用户。

**如果您需要帮助的性能，稳定性，用户体验或其他复杂的问题-联系我们!作为反应本机的核心贡献者和**

**社区的领导们，我们会很乐意帮忙的。**

**{callstack.com}**

89



**第三组**

如何在稳定的开发环境下更快的发货。

**{callstack.com}**

**最终引导反应原生优化|第三组**

**介绍**

**反应自然是伟大的发货快速和自信，但你准备好了吗?**

现在，必须有一个稳定、舒适的开发设置，鼓励发布新特性，而且不会减慢开发速度。你必须快速供货，领先于竞争对手。

React Native在这样的环境下踢得很好。例如，它最大的卖点之一是，它允许你在不通过App Store提交的情况下对应用程序进行更新。它被称为空中更新，简称OTA。

问题是:您的应用程序准备好了吗?您的开发管道是否加速了React Native的开发和交付特性?

大多数时候，你都希望答案是肯定的。但在现实中，它变得很复杂。

在本节中，我们将介绍一些最佳实践和建议，这些实践和建议可以让您更快、更有信心地发布应用程序。正如大多数文章所建议的，这不仅仅是关于打开空中更新。要营造一个反应本土光芒、加快创新的稳定健康的发展环境。

这就是我们这部分指南的内容。

**{callstack.com}**

**最终引导反应原生优化|第三组**

**1.为应用程序的关键部分运行测试**



重点测试应用程序的关键部分，以便更好地了解新功能和调整。

**问题:您根本不编写测试，或者编写没有真正覆盖的低质量测试，并且只依赖于手工测试**

有信心地构建和部署应用程序是一项具有挑战性的任务。然而，验证每件事是否真的有效需要花费大量的时间和精力——不管它是否是自动化的。让人手动验证软件是否按预期工作对您的产品至关重要。

不幸的是，随着应用程序功能的增长，这个过程不能很好地扩展。它也没有向编写代码的开发人员提供直接的反馈。因此，它增加了发现和修复错误所需的时间。

那么，开发人员如何确保他们的软件始终是可生产的，并且不依赖于人工测试人员呢?他们编写自动化测试。React Native也不例外。您可以为您的JS代码(包含业务逻辑和UI)和底层使用的本地代码编写各种测试。可以通过使用端到端测试框架、旋转模拟器、模拟器甚至是真实设备来实现。React Native的一个重要特性是它捆绑到一个本地应用程序包中，因此它允许您在本地项目中使用所有喜欢的端到端测试框架。

但是要注意，编写测试本身可能是一项具有挑战性的任务，尤其是在您缺乏经验的情况下。很容易得到一个没有很好覆盖您的测试的结果

**{callstack.com}**

92

**最终引导反应原生优化|第三组**

特性。或者只测试积极的行为，而不处理异常。遇到没有提供太多价值的低质量测试是很常见的，因此不会增强您交付代码的信心。

无论您要编写哪种测试，不管是单元测试、集成测试还是E2E(端到端的缩写)，都有一条黄金法则可以避免您编写糟糕的测试。规则是“避免测试实现细节”。坚持这样做，随着时间的推移，您的测试将开始提供价值。

**你不能像你的竞争对手那样快速行动，出现倒退的几率很高，当收到不好的评论时，应用可能会被从应用商店中删除。**

测试代码的主要目标是通过最小化代码库中引入的错误数量，自信地部署代码。对于通常发布到应用程序商店的移动应用程序来说，不向用户发送错误尤其重要。因此，它们是一个漫长的审查过程的主题，可能需要几个小时到几天的时间。你最不想要的事情就是让你的用户失望的更新，使你的应用程序出错。这可能会导致你的评分下降，在极端的情况下，甚至会把应用从商店下架。

这种情况可能看起来很少见，但它们确实发生了。然后，您的团队可能会非常害怕出现另一次回归和崩溃，从而失去整个速度和信心。

**解决方案:不要以100%的覆盖率为目标，把重点放在应用程序的关键部分。使用单元测试(快照)，集成测试(排毒)**

运行测试不是“如果”的问题，而是“如何”的问题。你需要制定一个计划，让所花费的时间得到最大的价值。要让你的代码行数和依赖项都达到100%是非常困难的。而且，这通常是非常不切实际的。

大多数移动应用程序都不需要对它们所编写的代码进行完整的测试。例外情况是客户由于必须遵守政府规定而要求全面承保的情况。但是在这种情况下，您可能已经意识到了这个问题。

**{callstack.com}**

93

**最终引导反应原生优化|第三组**

对你来说，把时间集中在测试正确的事情上是至关重要的。学习识别对业务至关重要的特性和能力通常比编写测试本身更重要。毕竟，您希望增强对代码的信心，而不是为了它而进行编写测试。一旦您这样做了，您需要做的就是决定如何运行它。你有相当多的选择。

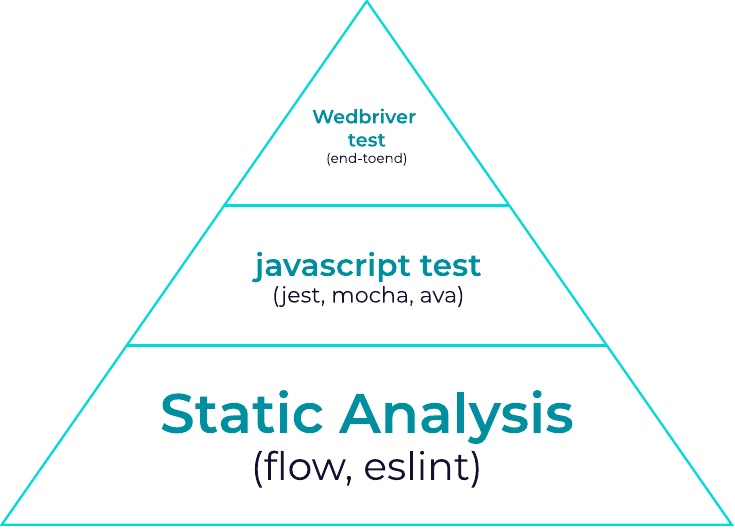
在React Native中，你的应用由多层代码组成，一些是用JS编写的，一些是用Java/Kotlin编写的，一些是用Objective-C/Swift编写的，甚至还有一些是用c++编写的，后者正在React Native core中得到广泛采用。

因此，出于实际原因，我们可以区分:

- JavaScript测试-借助Jest框架在React Native上下文中，如果您考虑“单元”或“集成”测试，这是它们最终属于的类别。从实际的角度来看，没有理由对这两类人加以区别。

-端到端应用测试-借助Detox, Appium或其他我们熟悉的移动测试框架。

因为您的大部分业务代码都使用JS，所以将精力集中在那里是有意义的。



测试金字塔。来源:https://twitter.com/aaronabramov\_/status/805913874704674816

**{callstack.com}**

94

**最终引导反应原生优化|第三组**

**JavaScript测试**

为实用函数编写测试应该非常简单。为此，您可以使用您最喜欢的测试运行器。在React Native社区中最受欢迎和推荐的是Jest。我们将在下面的部分中提到它。

但是，为了测试React组件，您需要更高级的工具。让我们以以下组件为例:

函数QuestionsBoard({questions, onSubmit}) {

= response . usestate ({});

回报(

<滚动视图>

{问题。map((q, index) => {

回报(

<查看关键= {q} >

<文本> {q} < /文本>

< TextInput

accessibilityLabel = "答案输入"

onChangeText = = >{{文本

setData(状态= > ({

状态,

[索引+ 1]:{q, a: text}，

}));

}}

/>

< /视图>

);

})}

<TouchableOpacity onPress={() => onSubmit(data)}>

< >文本提交文本> < /

< / TouchableOpacity >

< /滚动视图>

);

}

**{callstack.com}**

95

**最终引导反应原生优化|第三组**

它是一个React组件，显示问题列表并允许回答这些问题。您需要通过检查回调函数是否使用用户提供的一组答案调用来确保其逻辑工作。

为此，您可以使用React核心团队提供的官方反应-测试-呈现程序库。它是一个测试渲染器——换句话说——它允许你渲染你的组件并与它的生命周期交互，而不需要实际处理原生api。有些人可能会发现它非常可怕和难以使用，因为它是低级的API。

这就是为什么React Native周围的社区推出了一些助手库，比如React Native Testing Library，为我们提供了一组很好的助手，帮助我们高效地编写高质量的测试。

这个库的一个伟大之处在于，它的API迫使您避免测试组件的实现细节，从而使它对内部重构更具弹性。

问题板组件的测试如下所示:

从' response -native-test -library '导入{render, fireEvent};

从' ../QuestionsBoard '导入QuestionsBoard;

test('表单提交两个答案'，()=> {

const allQuestions = [' q1 '， ' q2 '];

const mockFn = jest.fn();

const {getAllByA11yLabel, getByText} = render(

<QuestionsBoard questions={allQuestions} onSubmit={mockFn} />

);

const answerinput = getAllByA11yLabel(' answer input ');

fireEvent。changeText (answerInputs[0],“a1”);

fireEvent。changeText (answerInputs[1],“a2”);

fireEvent.press (getByText('提交'));

**{callstack.com}**

96

**最终引导反应原生优化|第三组**

期望(mockFn) .toBeCalledWith ({

' 1 ': {q: ' q1 '， a: ' a1 '}，

' 2 ': {q: ' q2 '， a: ' a2 '}，

});

});

测试套件取自官方RNTL文档

首先使用问题集“呈现”QuestionsBoard组件。接下来，您将根据可访问性角色查询树，以访问由组件显示的问题数组。最后，您将设置正确的答案，并按下“提交”按钮。

如果一切顺利，您的断言将通过，以确保使用正确的参数集调用了' verifyQuestions '函数。

**注意:您可能也听说过针对JS的“快照测试”技术。**当从测试中断言重复数据时，它可以在某些测试场景中帮助您。该技术在React生态系统中被广泛采用，因为Jest提供了内置的支持。但是它是一种低级的API，应该避免使用，除非您有可靠的测试经验。

如果您想了解更多关于快照的知识，请参阅Rogelio的一位(Jest贡献者之一)，他谈到快照测试和项目存储库，后者可以帮助您测试数据的不同状态之间的差异，包括React组件

**E2E测试**

在我们的测试金字塔之上的樱桃是一套端到端测试。最好从所谓的“冒烟测试”开始——一个确保你的应用在第一次运行时不会崩溃的测试。有一个这样的测试是至关重要的，因为它将帮助您避免发送错误的应用程序给您的用户。一旦完成了这些基础工作，就应该使用所选的E2E测试框架来涵盖应用程序中最重要的功能。例如，可以登录(成功或失败)、注销、接受付款、显示从您的或第三方服务器获取的数据列表。

**{callstack.com}**

97

**最终引导反应原生优化|第三组**

**注意:要注意，这些测试通常比JS的测试更难设置。**而且，由于网络、文件系统操作、存储或内存不足等问题，它们更有可能失败。更重要的是，他们提供给你很少的信息，为什么他们这样做。这个测试的质量(不仅仅是E2E测试)被称为“片状”，应该不惜一切代价避免，因为它会降低您对测试套件的信心。这就是为什么将测试断言划分为更小的组如此重要的原因，这样可以更容易地调试出错的地方。

在本节中，我们将介绍Detox—React原生社区中最流行的E2E测试运行器，它是React原生测试管道的一部分。使用它，您将能够确保您所选择的框架得到最新的React本机版本的支持。在未来可能发生在框架级别上的变化的背景下，这一点尤其重要。

在继续之前，你必须安装Detox。在准备运行您的第一个套件之前，这个过程需要您采取一些额外的“本机步骤”。请遵循官方文档，因为这些步骤将来可能会更改。

一旦成功安装和配置了排毒，就可以开始第一个测试了。

它('应该显示问题'，async () => {

等待device.reloadReactNative ();

等待元素(by.text (allQuestions [0])) .toBeVisible ();

});

上面显示的这个简短片段将确保显示第一个问题。在执行断言之前，您应该重新加载React本机实例，以确保以前的状态不会干扰结果。

**注意:当你处理多个元素时(例如，在我们的例子中，一个组件呈现多个问题)，一个好的做法是为元素的索引指定一个后缀来查询特定的一个。**这和其他一些有趣的技巧是官方的排毒建议。

**{callstack.com}**

98

**最终引导反应原生优化|第三组**

有各种匹配器和期望可以帮助您以您想要的方式构建您的测试套件。

**好处:您对新特性和调整有更好的了解，可以满怀信心地交付，并且当测试是绿色的时—您节省了其他人(QA团队)的时间**

为您的核心特性提供足够覆盖范围的高质量测试套件是对团队速度的投资。毕竟，你只能在你的信心允许的范围内行动。这些测试都是为了确保你在正确的方向上前进。

React本地社区正在努力使测试变得尽可能简单和愉快——对您的团队和QA团队都是如此。正因为如此，您可以将更多的时间花在创新和用华丽的新功能取悦用户上，而不是一次又一次地消除bug和倒退。

**{callstack.com}**

99

**最终引导反应原生优化|第三组**

**2.有一个工作**



**持续集成(CI)到位**

使用CI提供者来改进应用程序的构建、测试和分发。

**问题:缺少CI或不稳定的CI意味着更长的反馈循环——你不知道你的代码是否能正常工作，而且你与其他开发人员合作的速度很慢**

正如您已经从上一节了解到的，通过测试覆盖您的代码对提高应用程序的整体可靠性非常有帮助。然而，尽管测试您的产品是至关重要的，但它不是您更快地发货和更自信地发货的唯一先决条件。

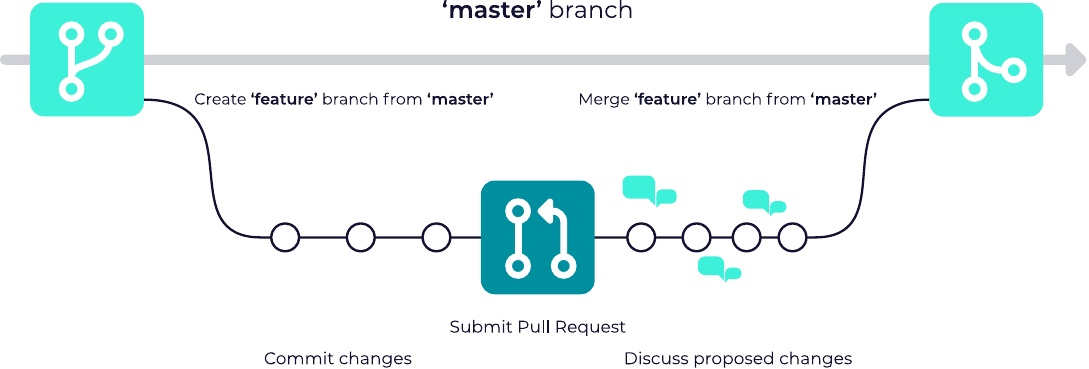
同样重要的是，您可以多快地发现潜在的回归，以及发现它们是否属于日常开发生命周期的一部分。换句话说，这一切都归结为反馈循环。

为了获得更好的上下文，让我们看看开发过程的早期。当您开始时，您的重点是尽可能快地发布第一个迭代(MVP)。因此，您可能会忽视体系结构本身的重要性。当您完成更改时，您将它们提交到存储库，让您的团队的其他成员知道该特性已准备好接受检查。

**{callstack.com}**

One hundred.

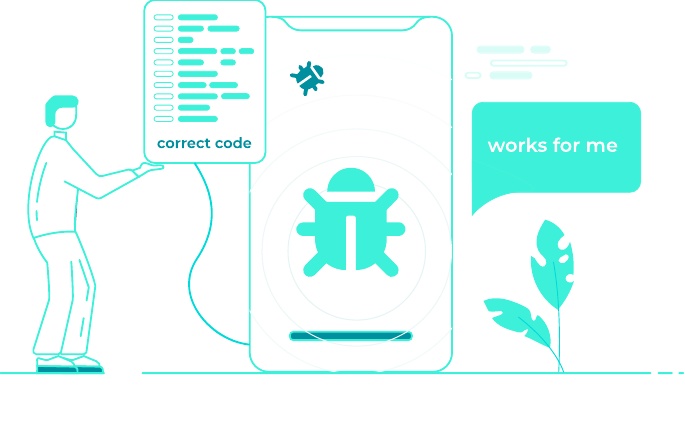
**最终引导反应原生优化|第三组**



这是Github上工作流的一个例子，其中的变更以PR的形式提出

虽然这项技术可能非常有用，但它本身也有潜在的危险，特别是当您的团队规模扩大时。在您准备接受PR之前，您不仅应该检查代码，还应该将其克隆到您的环境中并彻底地测试它。在这个过程的最后，提出的变更可能会引入原始作者没有发现的回归。

原因很简单——我们都有不同的配置、环境和工作方式。



虽然依赖于相同的配置，类似的开发原则和对细节的关注有助于在早期阶段更快地进行，但它可能会导致发布一些破坏测试的东西。

**{callstack.com}**

101

**最终引导反应原生优化|第三组**

**在你的组织中招募新成员是比较困难的。您不能在PRs和不同的贡献发生时就发布和测试它们。**

如果您手动测试更改，那么您不仅增加了回归生产的机会。您还降低了开发的总体速度。幸运的是，有了正确的方法集和一点自动化，您就可以一劳永逸地克服这个挑战。

这就是持续集成(CI)发挥作用的时候。CI是一种开发实践，在这种实践中，开发团队每天将建议的更改签入到上游存储库中数次。接下来，它们被一个自动化的构建验证，允许团队及早检测变更。

自动化构建由专用的基于云的CI提供者执行，该提供者通常与存储代码的位置集成。目前可用的大多数云提供商都支持Github，这是微软拥有的一个平台，用于在使用git作为版本控制系统的项目上进行合作。

CI系统实时提取更改并执行一组选定的测试，从而为您的结果提供早期反馈。这种方法为测试引入了单一的真实来源，并允许使用不同环境的开发人员获得方便和可靠的信息。

使用CI服务，您不仅可以测试代码，还可以为项目构建新版本的文档，构建应用程序并将其分发给测试人员或发行版。这种技术称为持续部署，关注于发布的自动化。本节对它进行了更深入的讨论。

**解决方案:使用CI提供程序(例如CircleCI)构建应用程序。运行所有必需的测试，并尽可能发布预览版本。**

有很多CI提供商可供选择，最流行的是CircleCI、Travis和最近发布的Github Actions。出于本节的目的，我们选择了CircleCI。

**{callstack.com}**

102

**最终引导反应原生优化|第三组**

它是React Native及其社区创建的所有项目的缺省CI提供者。实际上，有一个示例项目演示了如何使用React Native进行CI。你可以在这里了解更多。我们将在本节的后面使用它来介绍不同的CI概念。

**注意:好的经验法则是利用React Native / React Native社区项目已经使用的资源。**按照这种方式，您可以确保您所选择的提供者能够使用React Native，并且核心团队已经解决了最常见的挑战。

对于大多数CI提供程序，在做其他事情之前研究它们的配置文件是极其重要的。

让我们来看看CircleCI的一个配置文件示例，它来自前面提到的React本地示例:

版本:2

工作:

android:

working\_directory: ~ /回购

码头工人:

——图片:reactnativecommunity / react-native-android

步骤:

- - - - - -付款

-运行:npm i -g envinfo && envinfo

-运行:纱线安装

-运行:cd android && chmod +x gradlew && ./gradlew assembly elease

工作流程:

版本:2

build\_and\_test:

工作:

——安卓

.circleci / config.yml的例子

**{callstack.com}**

103

**最终引导反应原生优化|第三组**

该结构是用于基于文本的配置文件的标准Yaml语法。在进一步进行之前，您可能想了解它的基础知识。

**注意:许多CI服务，如CircleCI或Github操作，都基于Docker容器和将不同的作业组合到工作流中的思想。**Github和它的Github操作就是这样的提供者。您可能会发现这些服务之间有许多相似之处。

CircleCI配置有三个最重要的构建块:命令、作业和工作流。

命令只不过是一个shell脚本。它在指定的环境中执行。而且，它是执行云中的实际工作的工具。它可以是任何东西，从一个简单的命令来安装你的依赖，比如“纱线安装”(如果你正在使用纱线)到稍微复杂一点的一个。*编译Android文件的。*

*Job是一系列命令——被描述为步骤——专注于实现一个单一的、明确的目标。通过选择适当的Docker容器，作业可以在不同的环境中运行。*

例如，如果您只需要运行React单元测试，则可能需要使用节点容器。因此，容器将更小，依赖更少，安装速度更快。如果你想在云中构建一个React本地应用，你可以选择一个不同的容器，例如使用Android NDK/SDK或者使用OS X构建苹果平台的容器。

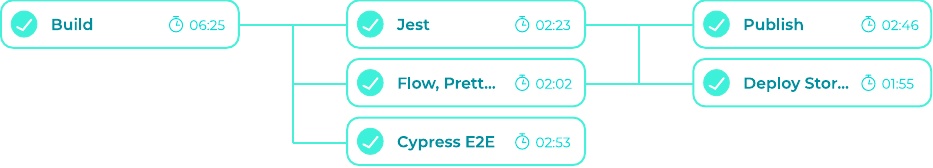
**注意:为了帮助您选择在运行React本机测试时使用的容器，该团队准备了response - Native - Android Docker容器，其中包括执行Android构建和测试所需的节点和Android依赖项。**

为了执行一个作业，它必须被分配到一个工作流中。默认情况下，作业将在工作流中并行执行，但可以通过指定作业的需求来更改这一点。

**{callstack.com}**

104

**最终引导反应原生优化|第三组**



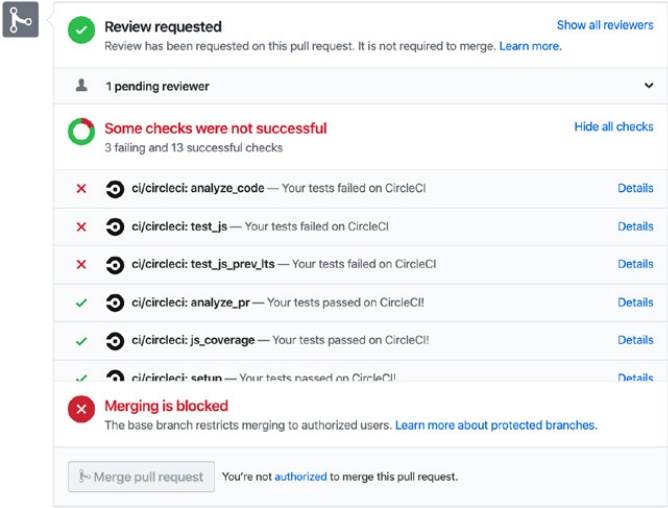
工作流包含可以分组以按顺序或并行运行的作业

您还可以通过添加筛选器来修改作业执行计划，因此，例如，只有在代码中的更改引用主分支时，部署作业才会运行。

你可以为不同的目的定义许多工作流，例如，一个用于打开PR后运行的测试，另一个用于部署应用程序的新版本。这就是React Native每隔一段时间自动发布新版本所做的。

**好处:您可以获得对添加的特性的早期反馈，快速发现倒退。此外，您也不会浪费其他开发人员的时间来测试那些不起作用的更改。**

在发布应用程序的新版本时，正确配置和工作的CI提供者可以节省大量时间。



Github UI报告CircleCI作业的状态，这个例子来自React本地存储库

通过提前发现错误，您可以减少评审PRs所需的工作，并保护您的产品免受可能直接减少您收入的倒退和bug的影响。

**{callstack.com}**

105

**最终引导反应原生优化|第三组**

**3.不要害怕装运**



**快速连续部署**

建立一个持续部署设置，以更快地发布新特性并验证关键的错误。

**问题:手工构建和发布应用程序是一个复杂和耗时的过程**

正如您在前一部分中了解到的，开发生命周期的关键部分的自动化可以帮助您提高总体开发速度和安全性。反馈周期越短，团队对产品本身进行迭代的速度就越快。

然而，测试和开发只是您在开发产品时必须执行的活动的一部分。另一个重要步骤是部署——构建应用程序并将其分发到生产环境中。大多数时候，这个过程是手动的。

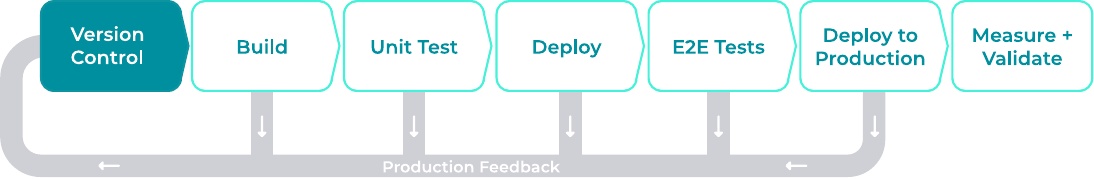
原因很简单——部署需要时间来设置，而且比在云中运行测试要复杂得多。例如，在iOS上，Xcode会自动配置许多设置和证书。这确保了在本地应用程序上工作的人有更好的开发人员体验。习惯了这种方法的开发人员通常会发现，将部署转移到云上并手动设置证书等内容是一项挑战。

**{callstack.com}**

106

**最终引导反应原生优化|第三组**

手工方法最大的缺点是它需要时间，而且不能伸缩。因此，不投资于此过程改进的团队最终会以较慢的速度发布他们的软件。



持续部署是一种策略，在这种策略中，软件通过一组自动化脚本频繁发布。它的目标是更快、更频繁地构建、测试和发布软件。通过允许对生产中的应用程序进行更多增量更新，该方法有助于降低交付更改的成本、时间和风险。

**您没有尽快交付新特性和修复程序**

不管团队有多大，手动构建和分发应用程序都会降低开发过程的速度。即使是在5人左右的小型产品团队中，自动化构建管道也能让每个人的工作更轻松、更轻松

不必要的沟通。这对于远程公司尤其重要。

持续部署还允许您引入关注于提高应用程序整体性能的标准和最佳实践。其中一些已经在本指南中讨论过。由于部署所需的所有步骤都在一个地方，因此很容易确保所有发布都以相同的方式完成，并采用公司范围的标准。

**解决方案:建立一个持续部署设置(基于‘fastlane’)，用于构建并生成变更日志。立即发送给用户**

当涉及到自动化移动应用程序的部署时，有两种方法。

**{callstack.com}**

107

**最终引导反应原生优化|第三组**

第一种方法是通过直接与' xcode '和' gradle '交互，从头开始编写一组脚本。不幸的是，Android和iOS工具之间存在着显著的差异，没有多少开发人员有足够的经验来处理这种自动化。最重要的是，iOS比Android复杂得多，因为它有先进的代码签名和分发策略。就像我们之前说的，如果你手动操作，即使Xcode也无法通过它的魔力来帮助你。



第二种方法是使用开发人员已经处理了大部分用例的预先存在的工具。我们最喜欢的一个是fastlane——一套用Ruby编写的模块化实用程序，通过在配置文件中编写一组指令，您可以构建iOS和Android应用程序。

成功构建二进制文件之后，就可以将其部署到目的地了。同样，你既可以手动上传文件到你想要的服务(例如App Store)，也可以使用工具为你处理这些文件。出于同样的原因，我们更喜欢使用现有的解决方案——在本例中是微软的AppCenter。

IMAGE

*AppCenter是一个云服务，带有自动化和部署应用程序的工具。*它最大的优点是许多设置都可以通过图形界面进行配置。以这种方式设置应用程序商店和播放商店部署比使用命令行上传要容易得多。

在本节中，我们将在CircleCI管道中使用fastlane和AppCenter来完全自动化将应用程序传递给最终用户的过程。

**注意:描述设置的细节会使本文太长。**这就是为什么我们选择只参考特定的文档。我们的目标是提供一个概述，而不是一步一步的指导，因为每个项目的最终配置都是不同的。

**{callstack.com}**

108

**最终引导反应原生优化|第三组**

**设置Fastlane**

在详细了解Android和iOS之前，你必须确保Fastlane已经安装并配置在我们的设备上。

接下来，您必须在React本机项目中运行init命令。我们将从每个本机文件夹运行两次' fastlane '命令。这是因为React Native在底层实际上是两个独立的应用程序。

> cd ./ios && fastlane init

> cd ./android && fastlane init

因此，该命令将在“ios”和“android”文件夹中生成安装文件。每个文件夹中的主文件将被称为Fastfile，所有车道都将在此配置。在fastlane术语中，lane就像工作流—一个分组部署应用程序的低级操作的部分。

底层操作可以通过调用actions来执行——预定义的fastlane操作可以简化您的工作流。我们将在下一节中向您展示它们是如何工作的。

**在Android上设置Fastlane**

既然已经在我们的项目中成功地设置了fastlane，就可以自动部署我们的Android应用程序了。要做到这一点，你可以选择一个Android特定的动作——在这个例子中是gradle。顾名思义，Gradle是一个动作，它可以让你获得与独立使用Android Gradle类似的结果。

我们的lane使用gradle操作首先清理构建文件夹，然后根据传递的参数装配带有签名的APK。

default\_platform (android):

project\_dir = File.expand\_path(“. .”,Dir.pwd)

平台:安卓做

lane:构建执行|选项|

if (ENV[' ANDROID\_KEYSTORE\_PASSWORD '] && ENV[' ANDROID\_KEY\_PASSWORD '])

109

**{callstack.com}**

**最终引导反应原生优化|第三组**

属性= {

“RELEASE\_STORE\_PASSWORD”= > ENV(“ANDROID\_KEYSTORE\_PASSWORD”),

“RELEASE\_KEY\_PASSWORD”= > ENV(“ANDROID\_KEY\_PASSWORD”)

}

结束

gradle (

任务:“干净”,

project\_dir: project\_dir,

属性:属性,

print\_command:假

)

gradle (

任务:“组装”,

build\_type:“释放”,

project\_dir: project\_dir,

属性:属性,

print\_command:假

)

结束

结束

android/fastlane/Fastfile的一部分，它定义了android lane，名为build

你应该能够运行lane build实现:$ cd ./android && fastlane build

这应该会产生一个签名的Android apk。

**注意:不要忘记设置访问密钥存储库的环境变量。**这些都是

' RELEASE\_STORE\_PASSWORD '和' RELEASE\_KEY\_PASSWORD '和在上面的示例中已经设置。

**{callstack.com}**

110

**最终引导反应原生优化|第三组**

**在iOS上设置Fastlane**

随着Android构建被自动化，你已经准备好转向iOS了。正如我们前面讨论的，由于认证和配置配置文件，iOS有点复杂。它们是苹果公司为了增加安全性而设计的。幸运的是，fastlane提供了一些专门的行动，帮助我们克服这些复杂性。

您可以从匹配操作开始。它有助于管理和分发iOS证书，并在团队成员中提供配置文件。你可以在合作设计中了解match背后的想法。指导的概念。

简单地说，match负责设置您的设备，使其能够成功地构建将被Apple服务器验证和接受的应用程序。

**注意:在进行进一步操作之前，请确保您的init与项目匹配。**它将生成所需的证书，并将它们存储在一个中央存储库中，您的团队和其他自动化工具可以在其中获取它们。

除了match，你可以使用的另一个动作是gym。gym在某种程度上类似于Gradle action，它实际上执行你的应用程序的构建。为此，它使用先前从match获取的证书和签名设置。

default\_platform (ios):

ios\_directory = File.expand\_path(“. .”,Dir.pwd)

base\_path = File.expand\_path(“. .”,ios\_directory)

ios\_workspace\_path = " # {ios\_directory} / YOUR\_WORKSPACE.xcworkspace "

ios\_output\_dir = File.expand\_path(”。/输出”,base\_path)

com . example”ios\_app\_id = '

ios\_app\_scheme = ' MyScheme '

before\_all做

如果is\_ci ?& & FastlaneCore: Helper.mac吗?

setup\_circle\_ci

结束

结束

**{callstack.com}**

111

**最终引导反应原生优化|第三组**

ios平台:做

lane:构建执行|选项|

匹配(

类型:选择(类型):

只读的,真的,

app\_identifier: ios\_app\_id,

)

cocoapods (podfile ios\_directory):

健身房(

配置:“释放”,

方案:ios\_app\_scheme,

export\_method:“特别的”,

工作区:ios\_workspace\_path,

output\_directory: ios\_output\_dir,

清洁:没错,

xcargs:“-UseModernBuildSystem =不”

)

结束

结束

定义ios lane的部分ios/fastlane/Fastfile

你应该能够通过运行Android相同的命令来运行lane build:

$ cd ./ios && fastlane版本

这应该也产生了一个iOS应用程序。

**部署二进制文件**

既然您已经自动化了构建，那么您就能够自动化流程的最后一部分——部署本身。为此，您可以使用App Center，正如本指南前面所讨论的。

**{callstack.com}**

112

**最终引导反应原生优化|第三组**

**注意:你必须在App中心创建一个账户，在仪表盘上创建Android和iOS应用，并为每个应用生成访问令牌。**

您还需要一个特殊的Fastlane插件，为工具带提供适当的操作。为此，运行“fastlane add\_plugin appcenter”。

一旦完成了项目的配置，就可以继续编写lane了，lane将打包生成的二进制文件并将它们上传到App Center。

莱恩:部署

构建

appcenter\_upload (

api\_token: ENV(“APPCENTER\_TOKEN”),

owner\_name:“ORGANIZATION\_OR\_USER\_NAME”,

owner\_type: " organization "， # ' user ' | ' organization '

app\_name:“YOUR\_APP\_NAME”,

文件:“# {ios\_output\_dir} / YOUR\_WORKSPACE.ipa”,

notify\_testers:真

)

结束

部分ios/fastlane/快速文件上传车道

就是这样!现在是时候通过在本地机器上执行deploy lane来部署应用程序了。

**结合CircleCI**

使用所有这些命令，您就能够在本地构建和分发应用程序。现在，您可以配置CI服务器，使它在每次提交master时都执行相同的操作。为此，您将使用CircleCI——我们在整个指南中一直使用的提供商。

**注意:在CI服务器上运行Fastlane通常需要一些额外的设置。**请参阅官方文档，以更好地理解本地环境和CI环境中的设置之间的区别。

**{callstack.com}**

113

**最终引导反应原生优化|第三组**

要从CircleCI部署应用程序，您可以配置专门的工作流，该工作流将专注于构建和部署应用程序。它将包含一个名为deploy\_ios的作业，该作业将执行fastlane命令。

deploy\_ios:

macos:

xcode:“11.3.1”

working\_directory: ~ / CI-CD

步骤:

- - - - - -付款

- attach\_workspace:

: ~ / CI-CD

运行:HOMEBREW\_NO\_AUTO\_UPDATE=1 brew安装watchman

-运行:捆绑安装

-运行:cd ios && bundle exec fastlane部署

工作流程:

版本:2

部署:

工作:

——deploy\_ios

实行快速道的环形结构的一部分建道

Android的管道看起来非常类似。主要的区别在于遗嘱执行人。应该使用docker的“reactnativecommunity/react-native-android”图像，而不是macOS。

**注意:这只是CircleCI中的一个示例用法。**在您的情况下，为确保deploy\_ios在正确的时间点运行，在其他作业上定义过滤器和依赖关系可能更有意义。

您可以修改或参数化呈现的通道，以便在其他类型的部署中使用它们，例如针对特定于平台的应用程序商店。要了解这些高级用例的详细信息，请熟悉官方的Fastlane文档。

**{callstack.com}**

114

**最终引导反应原生优化|第三组**

**好处:短的反馈循环以及每晚或每周的构建让您更快地验证特性，更频繁地发布关键的bug。**

通过自动化部署，您不再浪费时间进行手工构建，也不再将工件发送到测试设备或应用程序商店。涉众能够更快地验证特性，并进一步缩短反馈循环。使用常规构建，您将能够轻松地捕获或发布任何关键错误的修复。

**{callstack.com}**

115

**最终引导反应原生优化|第三组**

**4.在紧急情况下使用OTA(无线)**



通过OTA及时提交关键的更新和修复。

**问题:传统的应用程序更新方式会变慢，你会失去宝贵的时间**

在移动设备上发送更新的传统模式与我们在其他平台上编写JavaScript应用程序的模式有本质上的不同。与web不同，移动部署要复杂得多，而且具有更好的开箱即用安全性。在上一节中我们已经详细讨论了CI/CD。

这对你的生意意味着什么?

每次更新，不管开发者发布的速度有多快，通常都要等一段时间，App Store和Play Store团队会根据他们的策略和最佳实践来审查你的产品。

这一过程在所有苹果平台上都特别具有挑战性，因为在这些平台上，应用程序常常会因为不遵守某些政策或用户界面不符合要求的标准而被撤下或拒绝。值得庆幸的是，您的应用程序被React Native拒绝的风险降低到了最低限度，因为您正在处理应用程序的JavaScript部分。React Native核心团队确保对框架所做的所有更改不会影响应用程序提交的成功。

因此，提交过程需要一段时间。如果你要发布一个重要的更新，每一分钟都很重要。

**{callstack.com}**

116

**最终引导反应原生优化|第三组**

幸运的是，有了React Native，就可以直接动态地将JavaScript更改发送给用户，而不用经历App Store的审核过程。这种技术通常被称为空中更新。它允许您根据所选择的技术为所有用户立即更改应用程序的外观。

**当严重的错误发生时——分钟和小时可能是关键的。不要等着苹果和谷歌来审核你的应用。**

如果你的应用程序没有准备好ota，你可能会在很多设备上留下一个严重的错误，只要苹果/谷歌检查你的产品并允许它发布。

尽管回顾时间在整个过程中变得更好了

多年来，它仍然是一个很好的逃生口，能够立即从错误中恢复过来，通过测试管道进入生产。

**解决方案:使用App Center / CodePush实现OTA更新**

正如前面提到的，React Native已经为OTA准备好了。这意味着它的架构和设计选择使这种更新成为可能。但是，它没有附带执行此类操作的基础设施。为此，您需要集成一个带有自己的基础设施的第三方服务。

最受欢迎和广泛使用的OTA更新工具是CodePush，它现在是微软应用程序中心套件的一部分。

**注意:你必须在App中心创建一个账号才能继续。**如果您正在阅读前面的部分，那么您应该已经有一个了。OTA选项将在您创建的应用程序下可见。通常使用OTA和App Center的发布功能是一个很好的做法，这样配置更容易。

**{callstack.com}**

117

**最终引导反应原生优化|第三组**

**配置本机端**

要将CodePush集成到您的应用程序中，请分别按照iOS和Android所需的步骤操作。我们决定链接到官方指南，而不是在这里包含步骤，因为其中包含要应用的附加本机代码，而这在未来几个月很可能会改变。

**配置JavaScript端**

在本机端设置了服务之后，可以使用JavaScript API启用更新并定义更新发生的时间。在应用启动时获取更新最简单的方法是使用“codePush”包装器包装你的主要组件。

从“反应-本地代码推送”导入代码推送;

类MyApp扩展组件{}

MyApp = codePush (MyApp);

基本CodePush集成

就是这样!如果您已经在本机端执行了所有更改，那么您的应用程序现在就可以使用OTA了。

对于更高级的用例，您还可以更改何时检查更新以及何时下载和应用更新的默认设置。例如，你可以强制CodePush在每次应用被带回到前台时检查更新，并在下一个简历中安装更新。

下面的图表代码片段演示了这种解决方案:

类MyApp扩展组件{}

MyApp = codePush ({

updateDialog:没错,

checkFrequency: codePush.CheckFrequency.ON\_APP\_RESUME,

**{callstack.com}**

118

**最终引导反应原生优化|第三组**

installMode: codePush.InstallMode.ON\_NEXT\_RESUME

}) (MyApp);

自定义CodePush设置发送更新到应用程序

在JavaScript和React native的本地端配置完代码后，就可以启动更新，让您的新客户享受更新了。为此，我们可以通过使用App Center CLI从命令行执行此操作。

npm install -g appcenter-cli

appcenter登录

然后，发布命令捆绑本地资产和文件，并将它们发送到云:

appcenter代码发布-react -a <ownerName>/<appName>

完成这些步骤后，运行您的应用程序的所有用户将使用您在上一节中配置的体验接收更新。

**注意:在发布新的CodePush版本之前，你必须在App Center的仪表板上创建一个应用程序。**这会给你你要找的" ownerName "和" appName "如前所述，您可以通过访问App Center的UI来完成此操作，也可以使用App Center CLI。

**优点:即时向用户提供关键修复和一些内容**

与OTA更新集成到您的应用程序，您可以发送您的JavaScript更新到所有用户在几分钟的事情。这种可能性对于修复重大bug或发送即时补丁至关重要。

例如，可能会发生后端停止工作并导致启动时崩溃的情况。这可能是一个处理不当的错误——在开发过程中，您从未遇到过后端故障，而忘记了处理这种边缘情况。

**{callstack.com}**

119

**最终引导反应原生优化|第三组**

解决这个问题的潜在方法很简单——只要显示一条回退消息并通知用户这个问题就足够了。虽然开发将花费您大约一个小时，但实际的更新和审查过程可能需要数小时，甚至数天。

通过OTA的更新，你可以在几分钟内做出反应，而不用担心糟糕的用户体验会影响到大多数用户。

**如果您需要帮助的性能，稳定性，用户体验或其他复杂的问题-联系我们!作为反应本机的核心贡献者和**

**社区的领导们，我们会很乐意帮忙的。**

**{callstack.com}**

120

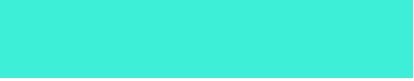
**反应原生优化的最终指南**

**谢谢你！**



我们希望您会发现上述React Native optimization的最佳实践是有用的，它们将使您的工作更容易。我们尽了最大努力使这个指南更加全面，并且描述了优化过程的技术和业务方面。如果您喜欢它，请毫不犹豫地与您的朋友分享，他们也在他们的项目中使用React Native。

**如果您在跨平台或React本地开发方面有更多问题或需要帮助，我们将很乐意提供免费咨询。**



**就联系我们**

**{callstack.com}**

121

**作者**





**麦克格拉博夫斯基**

Callstack联合创始人兼首席技术官。Mike是React Native的核心贡献者，也是RNPM和Haul等库的作者。迈克确保我们的基础设施在其极限下运行良好。

**Jakub Mazurek**

体验过React和Node的TypeScript爱好者。他最擅长的是从零开始创建解决方案、为概念项目做证明以及试验新技术。在他的空闲时间，他喜欢各种户外活动，如徒步旅行和骑自行车。

**反应原生优化的最终指南，提高用户体验，性能，和稳定性的应用程序。**

电子书



我们是React Native的Facebook官方合作伙伴。我们从事React Native项目已经超过5年了，为客户提供了高质量的解决方案，并为React Native生态系统做出了巨大贡献。我们的开源项目帮助成千上万的开发人员应对他们的挑战，使他们每天的工作更加轻松。

**{callstack.com}**

2020 Callstack。保留所有权利。