RAM Bundles and Inline Requires

If you have a large app you may want to consider the Random Access Modules (RAM) bundle format, and using inline requires. This is useful for apps that have a large number of screens which may not ever be opened during a typical usage of the app. Generally it is useful to apps that have large amounts of code that are not needed for a while after startup. For instance the app includes complicated profile screens or lesser used features, but most sessions only involve visiting the main screen of the app for updates. We can optimize the loading of the bundle by using the RAM format and requiring those features and screens inline (when they are actually used).

Loading JavaScript

Before react-native can execute JS code, that code must be loaded into memory and parsed. With a standard bundle if you load a 50mb bundle, all 50mb must be loaded and parsed before any of it can be executed. The optimization behind RAM bundles is that you can load only the portion of the 50mb that you actually need at startup, and progressively load more of the bundle as those sections are needed.

Inline Requires

Inline requires delay the requiring of a module or file until that file is actually needed. A basic example would look like this:

VeryExpensive.tsx

```
import React, {Component} from 'react';
import {Text} from 'react-native';
// ... import some very expensive modules

// You may want to log at the file level to verify when this is happening
console.log('VeryExpensive component loaded');

export default class VeryExpensive extends Component {
```

```
  // lots and lots of code
  render() {
    return <Text>Very Expensive Component</Text>;
  }
}
```

Optimized.tsx

```
import React, {Component} from 'react';
import {TouchableOpacity, View, Text} from 'react-native';

let VeryExpensive = null;

export default class Optimized extends Component {
  state = {needsExpensive: false};

  didPress = () => {
    if (VeryExpensive == null) {
      VeryExpensive = require('./VeryExpensive').default;
    }

    this.setState(() => ({
      needsExpensive: true,
    }));
  };

  render() {
    return (
      <View style={{marginTop: 20}}>
        <TouchableOpacity onPress={this.didPress}>
          <Text>Load</Text>
        </TouchableOpacity>
        {this.state.needsExpensive ? <VeryExpensive /> : null}
      </View>
    );
  }
}
```

Even without the RAM format, inline requires can lead to startup time improvements, because the code within VeryExpensive.js will only execute once it is required for the first time.

Enable the RAM format

On iOS using the RAM format will create a single indexed file that react native will load one module at a time. On Android, by default it will create a set of files for each module. You can force Android to create a single file, like iOS, but using multiple files can be more performant and requires less memory.

Enable the RAM format in Xcode by editing the build phase "Bundle React Native code and images". Before `../node_modules/react-native/scripts/react-native-xcode.sh` add `export BUNDLE_COMMAND="ram-bundle"`:

```
export BUNDLE_COMMAND="ram-bundle"
export NODE_BINARY=node
../node_modules/react-native/scripts/react-native-xcode.sh
```

On Android enable the RAM format by editing your `android/app/build.gradle` file. Before the line `apply from: "../../node_modules/react-native/react.gradle"` add or amend the `project.ext.react` block:

```
project.ext.react = [
  bundleCommand: "ram-bundle",
]
```

Use the following lines on Android if you want to use a single indexed file:

```
project.ext.react = [
  bundleCommand: "ram-bundle",
  extraPackagerArgs: ["--indexed-ram-bundle"]
]
```

> **INFO**
>
> If you are using Hermes JS Engine, you **should not** have RAM bundles feature enabled. In Hermes, when loading the bytecode, `mmap` ensures that the entire file is not loaded. Using

> Hermes with RAM bundles might lead to issues, because those mechanisms are not compatible with each other.

Configure Preloading and Inline Requires

Now that we have a RAM bundle, there is overhead for calling `require`. `require` now needs to send a message over the bridge when it encounters a module it has not loaded yet. This will impact startup the most, because that is where the largest number of require calls are likely to take place while the app loads the initial module. Luckily we can configure a portion of the modules to be preloaded. In order to do this, you will need to implement some form of inline require.

Investigating the Loaded Modules

In your root file (index.(ios|android).js) you can add the following after the initial imports:

```
const modules = require.getModules();
const moduleIds = Object.keys(modules);
const loadedModuleNames = moduleIds
  .filter(moduleId => modules[moduleId].isInitialized)
  .map(moduleId => modules[moduleId].verboseName);
const waitingModuleNames = moduleIds
  .filter(moduleId => !modules[moduleId].isInitialized)
  .map(moduleId => modules[moduleId].verboseName);

// make sure that the modules you expect to be waiting are actually waiting
console.log(
  'loaded:',
  loadedModuleNames.length,
  'waiting:',
  waitingModuleNames.length,
);

// grab this text blob, and put it in a file named packager/modulePaths.js
console.log(
  `module.exports = ${JSON.stringify(
    loadedModuleNames.sort(),
    null,
    2,
```

```
  )};`,
);
```

When you run your app, you can look in the console and see how many modules have been loaded, and how many are waiting. You may want to read the moduleNames and see if there are any surprises. Note that inline requires are invoked the first time the imports are referenced. You may need to investigate and refactor to ensure only the modules you want are loaded on startup. Note that you can change the Systrace object on require to help debug problematic requires.

```
require.Systrace.beginEvent = message => {
  if (message.includes(problematicModule)) {
    throw new Error();
  }
};
```

Every app is different, but it may make sense to only load the modules you need for the very first screen. When you are satisfied, put the output of the loadedModuleNames into a file named `packager/modulePaths.js`.

Updating the metro.config.js

We now need to update `metro.config.js` in the root of the project to use our newly generated `modulePaths.js` file:

```
const {getDefaultConfig, mergeConfig} = require('@react-native/metro-config');
const fs = require('fs');
const path = require('path');
const modulePaths = require('./packager/modulePaths');

const config = {
  transformer: {
    getTransformOptions: () => {
      const moduleMap = {};
      modulePaths.forEach(modulePath => {
        if (fs.existsSync(modulePath)) {
          moduleMap[path.resolve(modulePath)] = true;
        }
```

```
        });
        return {
          preloadedModules: moduleMap,
          transform: {inlineRequires: {blockList: moduleMap}},
        };
      },
    },
  };

module.exports = mergeConfig(getDefaultConfig(__dirname), config);
```

See also **Configuring Metro**.

The `preloadedModules` entry in the config indicates which modules should be marked as preloaded when building a RAM bundle. When the bundle is loaded, those modules are immediately loaded, before any requires have even executed. The `blockList` entry indicates that those modules should not be required inline. Because they are preloaded, there is no performance benefit from using an inline require. In fact the generated JavaScript spends extra time resolving the inline require every time the imports are referenced.

Test and Measure Improvements

You should now be ready to build your app using the RAM format and inline requires. Make sure you measure the before and after startup times.

Is this page useful? 

Edit this page

*Last updated on **Jul 3, 2023***