

RN Java调用JS流程

通信机制主要分成三部分，java端，c++端以及js端。调用的方式分成两种JavaScriptModule和NativeModule。

1. JavaScriptModule：表示js端提供的模型以及方法，java调用模型的方法既可以通知js端，典型的有AppRegistry。
java端就是一个接口AppRegistry.java,而js端AppRegistry.js提供了具体操作。
2. NativeModule：表示java端提供了模型和方法，供js端来回调。

JavaScriptModuleRegistry

通过动态代理的方式，回调CatalystInstance.callFunction方法，
然后调用CatalystInstance.jniCallJSFunction方法，
从而调用CatalystInstance.cpp对应jniCallJSFunction，
最终调用到js端MessageQueue.js的__callFunction方法，
然后就可以找到js端JavaScriptModule对应的方法并调用。

```
1 public synchronized <T extends JavaScriptModule> T getJavaScriptModule(  
2     CatalystInstance instance,  
3     Class<T> moduleInterface) {  
4     JavaScriptModule module = mModuleInstances.get(moduleInterface);  
5     if (module != null) {  
6         return (T) module;  
7     }  
8  
9     JavaScriptModule interfaceProxy = (JavaScriptModule) Proxy.newProxyInstance(  
10         moduleInterface.getClassLoader(),  
11         new Class[] {moduleInterface},  
12         new JavaScriptModuleInvocationHandler(instance, moduleInterface));  
13     mModuleInstances.put(moduleInterface, interfaceProxy);  
14     return (T) interfaceProxy;  
15 }
```

通过JavaScriptModule的class来获取它的代理实例。

我们并不需要JavaScriptModule的真正的实例，因为它的实现是在js端，java端只需要提供对应的module模板名，method方法名以及args调用参数，就可以回调js端对应的方法了。

```
1 @Override  
2 public @Nullable Object invoke(Object proxy, Method method,  
3     @Nullable Object[] args) throws Throwable {  
4     NativeArray jsArgs = args != null  
5         ? Arguments.fromJavaArgs(args)  
6         : new WritableNativeArray();  
7     mCatalystInstance.callFunction(getJSModuleName(), method.getName(), jsArgs);  
8     return null;  
9 }
```

CatalystInstanceImpl.java

```
1 @Override  
2 public void callFunction(final String module, final String method, final NativeArray arguments) {  
3     callFunction(new PendingJSCall(module, method, arguments));  
4 }  
5  
6 public void callFunction(PendingJSCall function) {
```

```

7   if (mDestroyed) {
8       final String call = function.toString();
9       FLog.w(ReactConstants.TAG, "Calling JS function after bridge has been destroyed: " + call);
10      return;
11  }
12  if (!mAcceptCalls) {
13      // Most of the time the instance is initialized and we don't need to acquire the lock
14      synchronized (mJSCallsPendingInitLock) {
15          if (!mAcceptCalls) {
16              mJSCallsPendingInit.add(function);
17              return;
18          }
19      }
20  }
21  function.call(this);
22  }

```

```

1
2  public static class PendingJSCall {
3      void call(CatalystInstanceImpl catalystInstance) {
4          NativeArray arguments = mArguments != null ? mArguments : new WritableNativeArray();
5          catalystInstance.jniCallJSFunction(mModule, mMethod, arguments);
6      }

```

JNI层

CatalystInstanceImpl.cpp

```

1  void CatalystInstanceImpl::jniCallJSFunction(std::string module, std::string method, NativeArray* a
2
3  // We want to share the C++ code, and on iOS, modules pass module/method
4  // names as strings all the way through to JS, and there's no way to do
5  // string -> id mapping on the objc side. So on Android, we convert the
6  // number to a string, here which gets passed as-is to JS. There, they they
7  // used as ids if isFinite(), which handles this case, and looked up as
8  // strings otherwise. Eventually, we'll probably want to modify the stack
9  // from the JS proxy through here to use strings, too.
10 void CatalystInstanceImpl::jniCallJSFunction(std::move(module),
11                                             std::move(method),
12                                             arguments->consume());

```

这次会走到JS的MessageQueue.callFunctionReturnFlushedQueue()中了。

JS接收调用和处理

先来解释下为什么会走到callFunctionReturnFlushedQueue。

1. 在生成的bundle.js中会把MessageQueue对象放到一个全局的属性中

```
Object.defineProperty(global, "__fbBatchedBridge", {configurable:!0,value:BatchedBridge})
```

这里明明是BatchedBridge，为什么说是MessageQueue的对象呢，原来在BatchedBridge.js中有这样几句代码

```
const BatchedBridge = new MessageQueue(
() => global.__fbBatchedBridgeConfig,
serializeNativeParams
);
```

2. 在上面加载bundle文件的时候，会执行下面的方法

```
void JSCExecutor::bindBridge() throw(JSEException) {
    auto global = Object::getGlobalObject(m_context);
    auto batchedBridgeValue = global.getProperty("__fbBatchedBridge");
    if (batchedBridgeValue.isUndefined()) {
        throwJSEExecutionException("Could not get BatchedBridge, make sure your bundle is packaged correctly");
    }
}
```

```

auto batchedBridge = batchedBridgeValue.asObject();
m_callFunctionReturnFlushedQueueJS = batchedBridge.getProperty("callFunctionReturnFlushedQueue").asObject();
m_invokeCallbackAndReturnFlushedQueueJS = batchedBridge.getProperty("invokeCallbackAndReturnFlushedQueue").asObject();
m_flushedQueueJS = batchedBridge.getProperty("flushedQueue").asObject();
}

```

这里会把MessageQueue的三个方法会当作对象保存在c++中，当我们调用JS的方法时会直接用到。

```

void JSCEXecutor::callFunction(const std::string& moduleId, const std::string& methodId, const folly::dynamic& arguments) {
    try {
        auto result = m_callFunctionReturnFlushedQueueJS->callAsFunction({
            Value(m_context, String::createExpectingAscii(moduleId)),
            Value(m_context, String::createExpectingAscii(methodId)),
            Value::fromDynamic(m_context, std::move(arguments))
        });
        auto calls = Value(m_context, result).toJSONString();
        m_delegate->callNativeModules(*this, std::move(calls), true);
    } catch (...) {
        std::throw_with_nested(std::runtime_error("Error calling function: " + moduleId + ":" + methodId));
    }
}

Value Object::callAsFunction(JSObjectRef thisObj, int nArgs, const JSValueRef args[]) const {
    JSValueRef exn;
    JSValueRef result = JSObjectCallAsFunction(m_context, m_obj, thisObj, nArgs, args, &exn);
    if (!result) {
        std::string exceptionText = Value(m_context, exn).toString().str();
        throwJSExecutionException("Exception calling object as function: %s", exceptionText.c_str());
    }
    return Value(m_context, result);
}

```

最终还是通过JavaScriptCore的方法JSObjectCallAsFunction来调用JS的。下面就好办了，直接分析JS代码吧。

在callFunctionReturnFlushedQueue这个方法主要调用了__callFunction，来看一下它的实现：

```

__callFunction(module: string, method: string, args: any) {
    ...
    const moduleMethods = this._callableModules[module];
    ...
    const result = moduleMethods[method].apply(moduleMethods, args);
    Systrace.endEvent();
    return result;
}

```

方法是从_callableModules中取出来的，那他的值是从哪里来的呢，看了下这个文件原来答案是有往里添加的方法

```

registerCallableModule(name, methods) {
    this._callableModules[name] = methods;
}

```

也就是说所有的JS Module都需要把该Module中可供Native调用的方法都放到这里来，这样才能够执行。以AppRegistry.js为例，来看看它是怎么往里添加的

```

var AppRegistry = {
    registerConfig: function(config: Array) {...},

    registerComponent: function(appKey: string, getComponentFunc: ComponentProvider): string {...},

    registerRunnable: function(appKey: string, func: Function): string {...},

    getAppKeys: function(): Array {...},

    runApplication: function(appKey: string, appParameters: any): void {...},

    unmountApplicationComponentAtRootTag: function(rootTag : number) {...},
};

BatchedBridge.registerCallableModule(
    'AppRegistry',
    AppRegistry
);

```

到这里Native调用JS就已经完成了。

总结一下整个流程：

1. MessageQueue把Native调用的方法放到JavaScriptCore中

2. JS Module把可以调用的方法放到MessageQueue的一个队列中
3. Native从JavaScriptCore中拿到JS的调用入口，并把Module Name、Method Name、Parameters传过去
4. 执行JS Module的方法

NativeModuleRegistry

向c++端提供java端所有的NativeModule。主要是两个方法：getJavaModules和getCxxModules。这个两个方法都是c++端直接调用的。

```
1  /* package */ Collection<JavaModuleWrapper> getJavaModules(  
2      JSInstance jsInstance) {  
3      ArrayList<JavaModuleWrapper> javaModules = new ArrayList<>();  
4      for (Map.Entry<Class<? extends NativeModule>, ModuleHolder> entry :  
5          mModules.entrySet()) {  
6          Class<? extends NativeModule> type = entry.getKey();  
7          if (!CxxModuleWrapperBase.class.isAssignableFrom(type)) {  
8              javaModules.add(new JavaModuleWrapper(jsInstance, type, entry.getValue()));  
9          }  
10     }  
11     return javaModules;  
12 }
```

返回一个JavaModuleWrapper的集合。

接受一个参数JSInstance，它是一个接口，只有一个invokeCallback方法，作用是向js端传递数据。考虑下面情况，我们在js端调用NativeModule的方法，如果这是一个异步方法，那么这个方法的结果值怎么回传给js端呢，就是通过这个JSInstance(注意如果是同步方法就不用了，因为同步方法结果值是直接返回的)。

NativeModule:

js端调用java端方法的方式,对应js端NativeModules.js。分为同步方法和异步方法：

1. 同步方法会在c++端直接调用，得到结果值，返回给js端。
2. 异步方法由c++端调用JavaModuleWrapper的invoke方法，再调用JavaMethodWrapper的invoke方法，最终会调用到CatalystInstanceImpl的jniCallJSCallback方法，最后会调用到MessageQueue.js的__invokeCallback方法。

CatalystInstanceImpl.cpp

对应CatalystInstanceImpl.java类，它是java端native方法的具体实现。

```
1  void CatalystInstanceImpl::initializeBridge(  
2      jni::alias_ref<ReactCallback::javaobject> callback,  
3      // This executor is actually a factory holder.  
4      JavaScriptExecutorHolder* jseh,  
5      jni::alias_ref<JavaMessageQueueThread::javaobject> jsQueue,  
6      jni::alias_ref<JavaMessageQueueThread::javaobject> nativeModulesQueue,  
7      jni::alias_ref<JavaMessageQueueThread::javaobject> uiBackgroundQueue,  
8      jni::alias_ref  
9          <jni::JCollection<JavaModuleWrapper::javaobject>::javaobject> javaModules,  
10     jni::alias_ref  
11         <jni::JCollection<ModuleHolder::javaobject>::javaobject> cxxModules) {  
12     moduleMessageQueue_ = std::make_shared<JMessageQueueThread>(nativeModulesQueue);  
13     if (uiBackgroundQueue.get() != nullptr) {  
14         uiBackgroundMessageQueue_ =  
15             std::make_shared<JMessageQueueThread>(uiBackgroundQueue);  
16     }  
17  
18     moduleRegistry_ = std::make_shared<ModuleRegistry>(  
19         callback,  
20         jseh,  
21         jsQueue,  
22         nativeModulesQueue,  
23         uiBackgroundQueue,  
24         javaModules,  
25         cxxModules,  
26         moduleMessageQueue_,  
27         uiBackgroundMessageQueue_);  
28 }
```

```

19     buildNativeModuleList(
20         std::weak_ptr<Instance>(instance_),
21         javaModules,
22         cxxModules,
23         moduleMessageQueue_,
24         uiBackgroundMessageQueue_));
25
26     instance_->initializeBridge(
27         folly::make_unique<JInstanceCallback>(
28             callback,
29             uiBackgroundMessageQueue_ !=
30                 NULL ? uiBackgroundMessageQueue_ : moduleMessageQueue_),
31         jseh->getExecutorFactory(),
32         folly::make_unique<JMessageQueueThread>(jsQueue),
33         moduleRegistry_);
34 }

```

1. 创建一个ModuleRegistry实例， buildNativeModuleList是ModuleRegistryBuilder.cpp中的方法，用来将javaModules和cxxModules这些java实例集合转换成c++的NativeModule实例集合。
2. 调用instance_的initializeBridge方法。 instance_是instance.cpp的实例。

```

1 void CatalystrInstanceImpl::jniCallJSFunction(std::string module,
2         std::string method, NativeArray* arguments) {
3     instance_->callJSFunction(std::move(module),
4                               std::move(method),
5                               arguments->consume());
6 }

```

```

1 void CatalystrInstanceImpl::jniCallJSCallback(jint callbackId,
2         NativeArray* arguments) {
3     instance_->callJSCallback(callbackId, arguments->consume());
4 }

```

ModuleRegistry.cpp

存NativeModule.cpp列表modules_，用来回调java端的NativeModule方法。

```

1 folly::Optional<ModuleConfig> ModuleRegistry::getConfig(const std::string& name) {
2     .....
3 }

```

主要是得到NativeModule的方法名列表和常量列表，传给js端NativeModules.js中。

```

1 void ModuleRegistry::callNativeMethod(unsigned int moduleId,
2         unsigned int methodId, folly::dynamic&& params, int callId) {
3     if (moduleId >= modules_.size()) {
4         throw std::runtime_error(
5             folly::to<std::string>("moduleId ", moduleId,
6                 " out of range [0..", modules_.size(), ")"));
7     }
8     modules_[moduleId]->invoke(methodId, std::move(params), callId);
9 }

```

异步NativeModule方法的调用。调用JavaModuleWrapper.cpp的invoke方法，会调用JavaModuleWrapper.java的invoke方法。

```

1 MethodCallResult ModuleRegistry::callSerializableNativeHook(unsigned int moduleId,

```

```

2     unsigned int methodId, folly::dynamic&& params) {
3     if (moduleId >= modules_.size()) {
4         throw std::runtime_error(
5             folly::to<std::string>("moduleId ", moduleId,
6                 "out of range [0..", modules_.size(), ")"));
7     }
8     return modules_[moduleId]->
9         callSerializableNativeHook(methodId, std::move(params));
10 }

```

同步NativeModule方法的调用。调用JavaModuleWrapper.cpp的callSerializableNativeHook方法，这个方法会直接调用NativeModule方法，得到结果值并返回。

JavaModuleWrapper.cpp

```

1 void JavaNativeModule::invoke(unsigned int reactMethodId,
2     folly::dynamic&& params, int callId) {
3     messageQueueThread->runOnQueue(
4         [this, reactMethodId, params=std::move(params), callId] {
5         static auto invokeMethod = wrapper->getClass()->getMethod
6             <void(jint, ReadableNativeArray::javaobject)>("invoke");
7         #ifdef WITH_FBSYSTRACE
8         if (callId != -1) {
9             fbsysstrace_end_async_flow(TRACE_TAG_REACT_APPS, "native", callId);
10        }
11        #endif
12        invokeMethod(
13            wrapper_,
14            static_cast<jint>(reactMethodId),
15            ReadableNativeArray::newObjectCxxArgs(std::move(params)).get());
16    });
17 }

```

这个方法主要作用调用JavaModuleWrapper.java的invoke方法。wrapper_就是JavaModuleWrapper.java对应的实例。

```

1 MethodCallResult JavaNativeModule::callSerializableNativeHook(
2     unsigned int reactMethodId, folly::dynamic&& params) {
3     // TODO: evaluate whether calling through invoke is potentially faster
4     if (reactMethodId >= syncMethods_.size()) {
5         throw std::invalid_argument(
6             folly::to<std::string>("methodId ", reactMethodId,
7                 " out of range [0..", syncMethods_.size(), "]"));
8     }
9
10    auto& method = syncMethods_[reactMethodId];
11    CHECK(method.hasValue() && method->isSyncHook())
12        << "Trying to invoke a asynchronous method as synchronous hook";
13    return method->invoke(instance_, wrapper->getModule(), params);
14 }

```

直接调用NativeModule的方法，并将结果值返回。

Instance.cpp

主要作用是处理CatalystInstanceImpl对应的方法，还会创建一个NativeToJsBridge实例。

```

1 void Instance::initializeBridge(

```

```

2     std::unique_ptr<InstanceCallback> callback,
3     std::shared_ptr<JSExecutorFactory> jsef,
4     std::shared_ptr<MessageQueueThread> jsQueue,
5     std::shared_ptr<ModuleRegistry> moduleRegistry) {
6     callback_ = std::move(callback);
7     moduleRegistry_ = std::move(moduleRegistry);
8
9     jsQueue->runOnQueueSync([this, &jsef, jsQueue]() mutable {
10         nativeToJsBridge_ = folly::make_unique<NativeToJsBridge>(
11             jsef.get(), moduleRegistry_, jsQueue, callback_);
12
13         std::lock_guard<std::mutex> lock(m_syncMutex);
14         m_syncReady = true;
15         m_syncCV.notify_all();
16     });
17
18     CHECK(nativeToJsBridge_);
19 }

```

进行属性的赋值，并创建NativeToJsBridge实例。callback_是java端ReactCallback的实例，jsef用来得到生成JSExecutor.cpp实例的工厂类。

```

1 void Instance::callJSFunction(std::string &&module, std::string &&method,
2                               folly::dynamic &&params) {
3     callback_->incrementPendingJSCalls();
4     nativeToJsBridge_->callFunction(std::move(module), std::move(method),
5                                     std::move(params));
6 }

```

先回调java端ReactCallback的incrementPendingJSCalls方法，然后调用nativeToJsBridge_的对应方法。

```

1 void Instance::callJSCallback(uint64_t callbackId, folly::dynamic &&params) {
2     SystraceSection s("Instance::callJSCallback");
3     callback_->incrementPendingJSCalls();
4     nativeToJsBridge_->invokeCallback((double)callbackId, std::move(params));
5 }

```

先回调java端ReactCallback的incrementPendingJSCalls方法，然后调用nativeToJsBridge_的对应方法。

NativeToJsBridge.cpp

创建JsToNativeBridge实例和JSExecutor实例(即JSExecutor.cpp实例),然后调用JSExecutor对应方法与js交互。这个类还包含一个JsToNativeBridge类，用它来与java端交互的，即调用NativeModule，所有它拥有ModuleRegistry。

JsToNativeBridge

```

1 JsToNativeBridge(std::shared_ptr<ModuleRegistry> registry,
2                  std::shared_ptr<InstanceCallback> callback)
3 : m_registry(registry)
4 , m_callback(callback) {}

```

得到ModuleRegistry属性。callback是java端的BridgeCallback实例。

```

1 void callNativeModules(
2     JSExecutor& executor, folly::dynamic&& calls, bool isEndOfBatch) override {

```

```

3
4     CHECK(m_registry || calls.empty()) <<
5         "native module calls cannot be completed with no native modules";
6     m_batchHadNativeModuleCalls = m_batchHadNativeModuleCalls || !calls.empty();
7
8     for (auto& call : parseMethodCalls(std::move(calls))) {
9         m_registry->callNativeMethod(call.moduleId, call.methodId, std::move(call.arguments), call.callback);
10    }
11    if (isEndOfBatch) {
12        if (m_batchHadNativeModuleCalls) {
13            m_callback->onBatchComplete();
14            m_batchHadNativeModuleCalls = false;
15        }
16        m_callback->decrementPendingJSCalls();
17    }
18 }

```

我们可以看到通过m_registry的callNativeMethod方法，来回调java端对应的异步方法。parseMethodCalls是MethodCall.cpp中的方法，主要作用是將js端传递来的json格式数据转换成我们需要的数据格式。

```

1 MethodCallResult callSerializableNativeHook(
2     JSExecutor& executor, unsigned int moduleId, unsigned int methodId,
3     folly::dynamic&& args) override {
4     return m_registry->callSerializableNativeHook(moduleId, methodId, std::move(args));
5 }

```

通过m_registry的callSerializableNativeHook，进行NativeModule同步方法的调用。

下面是NativeToJsBridge中的方法

```

1 NativeToJsBridge::NativeToJsBridge(
2     JSExecutorFactory* jsExecutorFactory,
3     std::shared_ptr<ModuleRegistry> registry,
4     std::shared_ptr<MessageQueueThread> jsQueue,
5     std::shared_ptr<InstanceCallback> callback)
6     : m_destroyed(std::make_shared<bool>(false))
7     , m_delegate(std::make_shared<JsToNativeBridge>(registry, callback))
8     , m_executor(jsExecutorFactory->createJSExecutor(m_delegate, jsQueue))
9     , m_executorMessageQueueThread(std::move(jsQueue)) {}

```

我们可以看到，创建了一个JsToNativeBridge实例m_delegate，以及JSExecutor实例m_executor(即JSExecutor.cpp的实例)。

```

1
2 void NativeToJsBridge::callFunction(
3     std::string&& module,
4     std::string&& method,
5     folly::dynamic&& arguments) {
6     int systraceCookie = -1;
7     #ifdef WITH_FBSYSTRACE
8     systraceCookie = m_systraceCookie++;
9     FbSystraceAsyncFlow::begin(
10         TRACE_TAG_REACT_CXX_BRIDGE,
11         "JSCall",
12         systraceCookie);
13     #endif
14
15     runOnExecutorQueue([module = std::move(module), method = std::move(method),
16         arguments = std::move(arguments), systraceCookie]
17         (JSExecutor* executor) {

```



```

18     #ifdef WITH_FBSYSTRACE
19     FbSysTraceAsyncFlow::end(
20         TRACE_TAG_REACT_CXX_BRIDGE,
21         "JSCall",
22         sysTraceCookie);
23     SysTraceSection s("NativeToJsBridge::callFunction",
24         "module", module, "method", method);
25     #endif
26
27     executor->callFunction(module, method, arguments);
28 });
29 }

```

用JSExecutor的callFunction方法(JSExecutor.cpp中有具体实现)。

```

1 void NativeToJsBridge::invokeCallback(double callbackId,
2     folly::dynamic&& arguments) {
3     int sysTraceCookie = -1;
4     #ifdef WITH_FBSYSTRACE
5     sysTraceCookie = m_sysTraceCookie++;
6     FbSysTraceAsyncFlow::begin(
7         TRACE_TAG_REACT_CXX_BRIDGE,
8         "<callback>",
9         sysTraceCookie);
10    #endif
11
12    runOnExecutorQueue(
13        [callbackId, arguments = std::move(arguments), sysTraceCookie]
14        (JSExecutor* executor) {
15        #ifdef WITH_FBSYSTRACE
16        FbSysTraceAsyncFlow::end(
17            TRACE_TAG_REACT_CXX_BRIDGE,
18            "<callback>",
19            sysTraceCookie);
20        SysTraceSection s("NativeToJsBridge::invokeCallback");
21        executor->invokeCallback(callbackId, arguments);
22        });
23 }

```

用JSExecutor的invokeCallback方法(JSExecutor.cpp中有具体实现)。

JSExecutor.cpp

真正js端进行交互了，直接调用js端的方法。

```

1
2 std::unique_ptr<JSExecutor> JSExecutorFactory::createJSExecutor(
3     std::shared_ptr<ExecutorDelegate> delegate,
4     std::shared_ptr<MessageQueueThread> jsQueue) {
5     return folly::make_unique<JSExecutor>(delegate, jsQueue, m_jscConfig);
6 }
7

```

创建一个JSExecutor实例，持有一个JsToNativeBridge实例对象delegate，用来回调java端方法的。

```

1 void JSExecutor::bindBridge() throw(JSEException) {
2     SysTraceSection s("JSExecutor::bindBridge");
3     std::call_once(m_bindFlag, [this] {
4         auto global = Object::getGlobalObject(m_context);

```

```

5      auto batchedBridgeValue = global.getProperty("__fbBatchedBridge");
6      if (batchedBridgeValue.isUndefined()) {
7          auto requireBatchedBridge =
8              global.getProperty("__fbRequireBatchedBridge");
9          if (!requireBatchedBridge.isUndefined()) {
10              batchedBridgeValue =
11                  requireBatchedBridge.asObject().callAsFunction({});
12          }
13          if (batchedBridgeValue.isUndefined()) {
14              throw JSEException("Could not get BatchedBridge,
15                  make sure your bundle is packaged correctly");
16          }
17      }
18
19      auto batchedBridge = batchedBridgeValue.asObject();
20      m_callFunctionReturnFlushedQueueJS = batchedBridge.getProperty("callFunctionReturnFlushedQueueJS");
21      m_invokeCallbackAndReturnFlushedQueueJS = batchedBridge.getProperty("invokeCallbackAndReturnFlushedQueueJS");
22      m_flushedQueueJS = batchedBridge.getProperty("flushedQueue").asObject();
23      m_callFunctionReturnResultAndFlushedQueueJS = batchedBridge.getProperty("callFunctionReturnResultAndFlushedQueueJS");
24  });
25  std::call_once(m_bindFlag, [this] {
26  }

```

这个方法搭建通信桥，拿到js端的实例对象和方法，用它们直接调用js端代码，'__fbBatchedBridge'是在BatchedBridge.js中定义的，其实是一个MessageQueue.js对象实例，得到四个方法的引用，通过它们直接调用js方法。

m_callFunctionReturnFlushedQueueJS：调用js端JavaScriptModule对应方法，不接收调用的结果值。

m_invokeCallbackAndReturnFlushedQueueJS：NativeModule异步方法得到结果值的回调。

m_flushedQueueJS：得到js端发起的NativeModule异步方法请求列表。

m_callFunctionReturnResultAndFlushedQueueJS：与m_callFunctionReturnFlushedQueueJS相同，但是能够接受js端调用的结果值。可惜这个方法没有被java端调用过。

```

1  void JSCExecutor::callFunction(const std::string& moduleId,
2      const std::string& methodId, const folly::dynamic& arguments) {
3      SysTraceSection s("JSCExecutor::callFunction");
4      auto result = [&] {
5          JSContextLock lock(m_context);
6          try {
7              if (!m_callFunctionReturnResultAndFlushedQueueJS) {
8                  bindBridge();
9              }
10             return m_callFunctionReturnFlushedQueueJS->callAsFunction({
11                 Value(m_context, String::createExpectingAscii(m_context, moduleId)),
12                 Value(m_context, String::createExpectingAscii(m_context, methodId)),
13                 Value::fromDynamic(m_context, std::move(arguments))
14             });
15         } catch (...) {
16             std::throw_with_nested(
17                 std::runtime_error("Error calling " + moduleId + "." + methodId));
18         }
19     }();
20     callNativeModules(std::move(result));
21 }

```

这个方法会调用js端JavaScriptModule对应方法，并得到当前js端发起的NativeModule异步方法请求列表，然后调用callNativeModules方法,这个方法会调用JsToNativeBridge的callNativeModules方法，最终调用java端代码。

```

1 void JSCExecutor::invokeCallback(const double callbackId,
2     const folly::dynamic& arguments) {
3     SystraceSection s("JSCExecutor::invokeCallback");
4     auto result = [&] {
5         JSContextLock lock(m_context);
6         try {
7             if (!m_invokeCallbackAndReturnFlushedQueueJS) {
8                 bindBridge();
9             }
10            return m_invokeCallbackAndReturnFlushedQueueJS->callAsFunction({
11                Value::makeNumber(m_context, callbackId),
12                Value::fromDynamic(m_context, std::move(arguments))
13            });
14        } catch (...) {
15            std::throw_with_nested(
16                std::runtime_error(folly::to<std::string>("Error invoking callback ",
17                    callbackId)));
18        }
19    }();
20    callNativeModules(std::move(result));
21 }

```

与上个方法流程相同，调用js端对应方法，得到当前js端发起的NativeModule异步方法请求列表，再调用callNativeModules方法。

```

1 void JSCExecutor::callNativeModules(Value&& value) {
2     SystraceSection s("JSCExecutor::callNativeModules");
3     CHECK(m_delegate) << "Attempting to use native modules without a delegate";
4     try {
5         auto calls = value.toJSONString();
6         m_delegate->callNativeModules(*this, folly::parseJson(calls), true);
7     } catch (...) {
8         std::string message = "Error in callNativeModules()";
9         try {
10            message += ":" + value.toString().str();
11        } catch (...) {
12            // ignored
13        }
14        std::throw_with_nested(std::runtime_error(message));
15    }
16 }

```

会调用JsToNativeBridge的callNativeModules方法。

```

1
2 JSValueRef JSCExecutor::nativeCallSyncHook(
3     size_t argumentCount,
4     const JSValueRef arguments[]) {
5     if (argumentCount != 3) {
6         throw std::invalid_argument("Got wrong number of args");
7     }
8
9     unsigned int moduleId = Value(m_context, arguments[0]).asUnsignedInteger();
10    unsigned int methodId = Value(m_context, arguments[1]).asUnsignedInteger();
11    folly::dynamic args =
12        folly::parseJson(Value(m_context, arguments[2]).toJSONString());
13
14    if (!args.isArray()) {

```

```

15         throw std::invalid_argument(
16             folly::to<std::string>(
17                 "method parameters should be array, but are ", args.typeName());
18     }
19
20     MethodCallResult result = m_delegate->callSerializableNativeHook(
21                                     *this,
22                                     moduleId,
23                                     methodId,
24                                     std::move(args));
25     if (!result.hasValue()) {
26         return Value::makeUndefined(m_context);
27     }
28     return Value::fromDynamic(m_context, result.value());
29 }

```

这个方法是由js端NativeModules.js中调用，当js端调用NativeModule的一个同步方法时，它就会调用到这个方法。

由它调用JsToNativeBridge->callSerializableNativeHook方法，然后调用ModuleRegistry->callSerializableNativeHook方法，继续调用JavaModuleWrapper->callSerializableNativeHook方法，这个方法直接调用java端NativeModule对应方法，并返回结果值(注意这里没有调用JavaModuleWrapper.java和JavaMethodWrapper.java的invoke方法，因为通过回调的方法向js端传递结果值，不能直接返回结果值)。

这里说一下java和c++代码的相互调用，js和c++代码的相互调用。我们都知道java可以通过jni来实现与c++端代码的相互调用的。那么js怎么实现的呢，通过全局变量global，c++通过global获取js端的实例和方法，也可以向global全局变量中注册c++本地的方法，然后js端就可以通过global来调用c++方法了。例如installNativeHook<&JSCExecutor::nativeCallSyncHook>("nativeCallSyncHook");就是向全局变量中注册nativeCallSyncHook这个本地方法。

总结

java端->js端：

从CatalystInstanceImpl.cpp开始，调用Instance.cpp对应方法，再调用NativeToJsBridge.cpp对应方法，再调用JSCExecutor.cpp对应方法，最终由JSCExecutor调用js端方法(因为JSCExecutor持有js对象方法的实例)

js端->java端：

异步方法的调用流程：通过JSCExecutor.cpp的callNativeModules方法，回调JsToNativeBridge(在NativeToJsBridge.cpp中)的callNativeModules方法,回调ModuleRegistry.cpp的callNativeMethod方法，再调用JavaModuleWrapper.cpp的invoke方法，然后就调用到JavaModuleWrapper.java中的invoke方法。

同步方法的调用流程：通过JSCExecutor.cpp的nativeCallSyncHook方法，回调JsToNativeBridge(在NativeToJsBridge.cpp中)的callSerializableNativeHook方法,回调ModuleRegistry.cpp的callSerializableNativeHook方法，回调JavaModuleWrapper.cpp的callSerializableNativeHook方法，然后直接调用java端NativeModule对应的方法，返回结果值。

、