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。

```
public synchronized <T extends JavaScriptModule> T getJavaScriptModule(
      CatalystInstance instance,
      Class<T> moduleInterface) {
    JavaScriptModule module = mModuleInstances.get(moduleInterface);
    if (module != null) {
      return (T) module;
    }

    JavaScriptModule interfaceProxy = (JavaScriptModule) Proxy.newProxyInstance(
        moduleInterface.getClassLoader(),
        new Class[]{moduleInterface},
        new JavaScriptModuleInvocationHandler(instance, moduleInterface));
     mModuleInstances.put(moduleInterface, interfaceProxy);
     return (T) interfaceProxy;
   }
```

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。

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

CatalystInstanceImpl.java

```
@Override
public void callFunction(final String module, final String method, final NativeArray arguments) {
  callFunction(new PendingJSCall(module, method, arguments));
}

public void callFunction(PendingJSCall function) {
```

```
  if (mDestroyed) {
    final String call = function.toString();
    FLog.w(ReactConstants.TAG, "Calling JS function after bridge has been destroyed: " + call);
    return;
  }
  if (!mAcceptCalls) {
    // Most of the time the instance is initialized and we don't need to acquire the lock
    synchronized (mJSCallsPendingInitLock) {
      if (!mAcceptCalls) {
        mJSCallsPendingInit.add(function);
        return;
      }
    }
  }
  function.call(this);
}
```

```

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

JNI层

CatalystInstanceImpl.cpp

```
void CatalystInstanceImpl::jniCallJSFunction(std::string module, std::string method, NativeArray* a
  // We want to share the C++ code, and on iOS, modules pass module/method
  // names as strings all the way through to JS, and there's no way to do
  // string -> id mapping on the objc side.  So on Android, we convert the
  // number to a string, here which gets passed as-is to JS.  There, they they
  // used as ids if isFinite(), which handles this case, and looked up as
  // strings otherwise.  Eventually, we'll probably want to modify the stack
  // from the JS proxy through here to use strings, too.
  instance_->callJSFunction(std::move(module),
                            std::move(method),
                            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(JSException) {
  auto global = Object::getGlobalObject(m_context);
  auto batchedBridgeValue = global.getProperty("__fbBatchedBridge");
  if (batchedBridgeValue.isUndefined()) {
    throwJSExecutionException("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的一个对列中
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。

```
/* package */ Collection<JavaModuleWrapper> getJavaModules(
      JSInstance jsInstance) {
    ArrayList<JavaModuleWrapper> javaModules = new ArrayList<>();
    for (Map.Entry<Class<? extends NativeModule>, ModuleHolder> entry :
          mModules.entrySet()) {
      Class<? extends NativeModule> type = entry.getKey();
      if (!CxxModuleWrapperBase.class.isAssignableFrom(type)) {
        javaModules.add(new JavaModuleWrapper(jsInstance, type, entry.getValue()));
      }
    }
    return javaModules;
  }
```

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。

```
void CatalystInstanceImpl::initializeBridge(
    jni::alias_ref<ReactCallback::javaobject> callback,
    // This executor is actually a factory holder.
    JavaScriptExecutorHolder* jseh,
    jni::alias_ref<JavaMessageQueueThread::javaobject> jsQueue,
    jni::alias_ref<JavaMessageQueueThread::javaobject> nativeModulesQueue,
    jni::alias_ref<JavaMessageQueueThread::javaobject> uiBackgroundQueue,
    jni::alias_ref
        <jni::JCollection<JavaModuleWrapper::javaobject>::javaobject> javaModules,
    jni::alias_ref
        <jni::JCollection<ModuleHolder::javaobject>::javaobject> cxxModules) {
  moduleMessageQueue_ = std::make_shared<JMessageQueueThread>(nativeModulesQueue);
  if (uiBackgroundQueue.get() != nullptr) {
    uiBackgroundMessageQueue_ =
      std::make_shared<JMessageQueueThread>(uiBackgroundQueue);
  }

  moduleRegistry_ = std::make_shared<ModuleRegistry>(
```

```
    buildNativeModuleList(
       std::weak_ptr<Instance>(instance_),
       javaModules,
       cxxModules,
       moduleMessageQueue_,
       uiBackgroundMessageQueue_));

  instance_->initializeBridge(
    folly::make_unique<JInstanceCallback>(
    callback,
    uiBackgroundMessageQueue_ !=
          NULL ? uiBackgroundMessageQueue_ : moduleMessageQueue_),
    jseh->getExecutorFactory(),
    folly::make_unique<JMessageQueueThread>(jsQueue),
    moduleRegistry_);
}
```

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。

```
void CatalystInstanceImpl::jniCallJSFunction(std::string module,
        std::string method, NativeArray* arguments) {
  instance_->callJSFunction(std::move(module),
                            std::move(method),
                            arguments->consume());
}
```

```
void CatalystInstanceImpl::jniCallJSCallback(jint callbackId,
         NativeArray* arguments) {
  instance_->callJSCallback(callbackId, arguments->consume());
}
```

ModuleRegistry.cpp

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

```
folly::Optional<ModuleConfig> ModuleRegistry::getConfig(const std::string& name) {
        ``````
}
```

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

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

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。

```
MethodCallResult ModuleRegistry::callSerializableNativeHook(unsigned int moduleId,
```
```

```
        unsigned int methodId, folly::dynamic&& params) {
  if (moduleId >= modules_.size()) {
    throw std::runtime_error(
      folly::to<std::string>("moduleId ", moduleId,
        "out of range [0..", modules_.size(), ")"));
  }
  return modules_[moduleId]->
       callSerializableNativeHook(methodId, std::move(params));
}
```

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

```
void JavaNativeModule::invoke(unsigned int reactMethodId,
        folly::dynamic&& params, int callId) {
  messageQueueThread_->runOnQueue(
        [this, reactMethodId, params=std::move(params), callId] {
    static auto invokeMethod = wrapper_->getClass()->getMethod
               <void(jint, ReadableNativeArray::javaobject)>("invoke");
    #ifdef WITH_FBSYSTRACE
    if (callId != -1) {
      fbsystrace_end_async_flow(TRACE_TAG_REACT_APPS, "native", callId);
    }
    #endif
    invokeMethod(
      wrapper_,
      static_cast<jint>(reactMethodId),
      ReadableNativeArray::newObjectCxxArgs(std::move(params)).get());
  });
}
```

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。

```
MethodCallResult JavaNativeModule::callSerializableNativeHook(
          unsigned int reactMethodId, folly::dynamic&& params) {
  // TODO: evaluate whether calling through invoke is potentially faster
  if (reactMethodId >= syncMethods_.size()) {
    throw std::invalid_argument(
      folly::to<std::string>("methodId ", reactMethodId,
           " out of range [0..", syncMethods_.size(), "]"));
  }

  auto& method = syncMethods_[reactMethodId];
  CHECK(method.hasValue() && method->isSyncHook())
        << "Trying to invoke a asynchronous method as synchronous hook";
  return method->invoke(instance_, wrapper_->getModule(), params);
}
```

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。

```
void Instance::initializeBridge(
```

```
    std::unique_ptr<InstanceCallback> callback,
    std::shared_ptr<JSExecutorFactory> jsef,
    std::shared_ptr<MessageQueueThread> jsQueue,
    std::shared_ptr<ModuleRegistry> moduleRegistry) {
  callback_ = std::move(callback);
  moduleRegistry_ = std::move(moduleRegistry);

  jsQueue->runOnQueueSync([this, &jsef, jsQueue]() mutable {
    nativeToJsBridge_ = folly::make_unique<NativeToJsBridge>(
        jsef.get(), moduleRegistry_, jsQueue, callback_);

    std::lock_guard<std::mutex> lock(m_syncMutex);
    m_syncReady = true;
    m_syncCV.notify_all();
  });

  CHECK(nativeToJsBridge_);
}
```

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JSCExecutor.cpp实例的工厂类。

```
void Instance::callJSFunction(std::string &&module, std::string &&method,
                              folly::dynamic &&params) {
  callback_->incrementPendingJSCalls();
  nativeToJsBridge_->callFunction(std::move(module), std::move(method),
                                  std::move(params));
}
```

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

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

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JSCExecutor.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

```
JsToNativeBridge(std::shared_ptr<ModuleRegistry> registry,
                 std::shared_ptr<InstanceCallback> callback)
    : m_registry(registry)
    , m_callback(callback) {}
```

得到ModuleRegistry属性。callback是java端的BridgeCallback实例。

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

```
    CHECK(m_registry || calls.empty()) <<
      "native module calls cannot be completed with no native modules";
    m_batchHadNativeModuleCalls = m_batchHadNativeModuleCalls || !calls.empty();

    for (auto& call : parseMethodCalls(std::move(calls))) {
      m_registry->callNativeMethod(call.moduleId, call.methodId, std::move(call.arguments), call.ca
     }
     if (isEndOfBatch) {
       if (m_batchHadNativeModuleCalls) {
         m_callback->onBatchComplete();
         m_batchHadNativeModuleCalls = false;
       }
       m_callback->decrementPendingJSCalls();
     }
   }
```

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，主要作用是将js端传递来的json格式数据转换成我们需要的数据格式。

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

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

下面是NativeToJsBridge中的方法

```
NativeToJsBridge::NativeToJsBridge(
    JSExecutorFactory* jsExecutorFactory,
    std::shared_ptr<ModuleRegistry> registry,
    std::shared_ptr<MessageQueueThread> jsQueue,
    std::shared_ptr<InstanceCallback> callback)
    : m_destroyed(std::make_shared<bool>(false))
    , m_delegate(std::make_shared<JsToNativeBridge>(registry, callback))
    , m_executor(jsExecutorFactory->createJSExecutor(m_delegate, jsQueue))
    , 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JSCExecutor.cpp的实例)。

```

void NativeToJsBridge::callFunction(
    std::string&& module,
    std::string&& method,
    folly::dynamic&& arguments) {
  int systraceCookie = -1;
  #ifdef WITH_FBSYSTRACE
  systraceCookie = m_systraceCookie++;
  FbSystraceAsyncFlow::begin(
      TRACE_TAG_REACT_CXX_BRIDGE,
      "JSCall",
      systraceCookie);
   #endif

   runOnExecutorQueue([module = std::move(module), method = std::move(method),
            arguments = std::move(arguments), systraceCookie]
     (JSExecutor* executor) {
```

```
      #ifdef WITH_FBSYSTRACE
      FbSystraceAsyncFlow::end(
          TRACE_TAG_REACT_CXX_BRIDGE,
          "JSCall",
          systraceCookie);
      SystraceSection s("NativeToJsBridge::callFunction",
           "module", module, "method", method);
      #endif

      executor->callFunction(module, method, arguments);
    });
}
```

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

```
void NativeToJsBridge::invokeCallback(double callbackId,
         folly::dynamic&& arguments) {
  int systraceCookie = -1;
  #ifdef WITH_FBSYSTRACE
  systraceCookie = m_systraceCookie++;
  FbSystraceAsyncFlow::begin(
      TRACE_TAG_REACT_CXX_BRIDGE,
      "<callback>",
      systraceCookie);
   #endif

   runOnExecutorQueue(
        [callbackId, arguments = std::move(arguments), systraceCookie]
    (JSExecutor* executor) {
      #ifdef WITH_FBSYSTRACE
      FbSystraceAsyncFlow::end(
          TRACE_TAG_REACT_CXX_BRIDGE,
          "<callback>",
          systraceCookie);
      SystraceSection s("NativeToJsBridge::invokeCallback");
      executor->invokeCallback(callbackId, arguments);
    });
}
```

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

JSCExecutor.cpp

真正js端进行交互了，直接调用js端的方法。

```

std::unique_ptr<JSExecutor> JSCExecutorFactory::createJSExecutor(
      std::shared_ptr<ExecutorDelegate> delegate,
      std::shared_ptr<MessageQueueThread> jsQueue) {
      return folly::make_unique<JSCExecutor>(delegate, jsQueue, m_jscConfig);
    }

```

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。

```
void JSCExecutor::bindBridge() throw(JSException) {
      SystraceSection s("JSCExecutor::bindBridge");
      std::call_once(m_bindFlag, [this] {
        auto global = Object::getGlobalObject(m_context);
```

```
        auto batchedBridgeValue = global.getProperty("__fbBatchedBridge");
        if (batchedBridgeValue.isUndefined()) {
          auto requireBatchedBridge =
               global.getProperty("__fbRequireBatchedBridge");
          if (!requireBatchedBridge.isUndefined()) {
             batchedBridgeValue =
                requireBatchedBridge.asObject().callAsFunction({});
           }
           if (batchedBridgeValue.isUndefined()) {
             throw JSException("Could not get BatchedBridge,
                make sure your bundle is packaged correctly");
           }
         }

         auto batchedBridge = batchedBridgeValue.asObject();
         m_callFunctionReturnFlushedQueueJS = batchedBridge.getProperty("callFunctionReturnFlushedQ
         m_invokeCallbackAndReturnFlushedQueueJS = batchedBridge.getProperty("invokeCallbackAndRetur
         m_flushedQueueJS = batchedBridge.getProperty("flushedQueue").asObject();
         m_callFunctionReturnResultAndFlushedQueueJS = batchedBridge.getProperty("callFunctionReturn
       });
       std::call_once(m_bindFlag, [this] {
     }
```

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。

```
void JSCExecutor::callFunction(const std::string& moduleId,
       const std::string& methodId, const folly::dynamic& arguments) {
      SystraceSection s("JSCExecutor::callFunction");
      auto result = [&] {
        JSContextLock lock(m_context);
        try {
          if (!m_callFunctionReturnResultAndFlushedQueueJS) {
            bindBridge();
          }
          return m_callFunctionReturnFlushedQueueJS->callAsFunction({
            Value(m_context, String::createExpectingAscii(m_context, moduleId)),
            Value(m_context, String::createExpectingAscii(m_context, methodId)),
            Value::fromDynamic(m_context, std::move(arguments))
          });
        } catch (...) {
          std::throw_with_nested(
            std::runtime_error("Error calling " + moduleId + "." + methodId));
        }
      }();
      callNativeModules(std::move(result));
    }
```

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。

```
void JSCExecutor::invokeCallback(const double callbackId,
       const folly::dynamic& arguments) {
      SystraceSection s("JSCExecutor::invokeCallback");
      auto result = [&] {
        JSContextLock lock(m_context);
        try {
          if (!m_invokeCallbackAndReturnFlushedQueueJS) {
            bindBridge();
          }
          return m_invokeCallbackAndReturnFlushedQueueJS->callAsFunction({
            Value::makeNumber(m_context, callbackId),
            Value::fromDynamic(m_context, std::move(arguments))
          });
        } catch (...) {
          std::throw_with_nested(
          std::runtime_error(folly::to<std::string>("Error invoking callback ",
                   callbackId)));
        }
      }();
      callNativeModules(std::move(result));
    }
```

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。

```
void JSCExecutor::callNativeModules(Value&& value) {
      SystraceSection s("JSCExecutor::callNativeModules");
      CHECK(m_delegate) << "Attempting to use native modules without a delegate";
      try {
        auto calls = value.toJSONString();
        m_delegate->callNativeModules(*this, folly::parseJson(calls), true);
      } catch (...) {
        std::string message = "Error in callNativeModules()";
        try {
          message += ":" + value.toString().str();
        } catch (...) {
          // ignored
        }
        std::throw_with_nested(std::runtime_error(message));
      }
    }
```

会调用JsToNativeBridge的callNativeModules方法。

```

JSValueRef JSCExecutor::nativeCallSyncHook(
          size_t argumentCount,
         const JSValueRef arguments[]) {
      if (argumentCount != 3) {
        throw std::invalid_argument("Got wrong number of args");
      }

      unsigned int moduleId = Value(m_context, arguments[0]).asUnsignedInteger();
       unsigned int methodId = Value(m_context, arguments[1]).asUnsignedInteger();
       folly::dynamic args =
            folly::parseJson(Value(m_context, arguments[2]).toJSONString());

       if (!args.isArray()) {
```

```
        throw std::invalid_argument(
              folly::to<std::string>(
               "method parameters should be array, but are ", args.typeName()));
      }

      MethodCallResult result = m_delegate->callSerializableNativeHook(
                                                              *this,
                                                             moduleId,
                                                             methodId,
                                                             std::move(args));
      if (!result.hasValue()) {
        return Value::makeUndefined(m_context);
      }
      return Value::fromDynamic(m_context, result.value());
    }
```

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。

`