Больше интеропов, хороших и разных

Nikolay Igotti

RRI

Интеропы - суровая правда мира!

- Большие системы неизбежно требует стековой структуры (высокоуровневого и низкоуровневого)
- Разные системы исполнения и парадигмы программирования
- Интерфейсы к ОС и готовым библиотекам
- Необходимо уметь в интероперабельность!

Виды интеропов

- Межъязыковой в рамках одного рантайма: (C++/C, Kotlin JVM/Java, TypeScript/JavaScript, Swift/Objective-C)
- Межрантаймовый: JS/C (NAPI), JVM/C (JNI), JS/WASM (пограничный вариант), Kotlin JVM/Kotlin Native
- Межпроцессный: user/kernel (syscalls), guest/hypervisor, RPC
- Межмашинный: (REST, RMI, DCOM, gRPC)

Решаемые задачи (aka Lost in Translation)

- Описание интерфейса
- Вызов по интерфейсу
- Передача данных
- Правила соответствия типов
- Управление памятью
- Уровень прозрачности для пользователя

Подходы к интеропу

- Общая lingua franca на основе самого простого участника или системы исполнения (C ABI в C++/C, JVM в Kotlin JVM/Java)
- Низкоуровневый рефлективный интерфейс к системе исполнения (JNI, NAPI)
- Языко-нейтральный типизированный подход с отображением типов и сериализацией (различные IDL, gRPC, WSDL)

Несколько примеров из жизни

- Sun RPC
- JNI и NAPI
- JS и WASM
- Kotlin/Native
- Метаинтероп для кроссплатформы (тема этого доклада)

Sun RPC (1988)

- В основном исторический интерес
- Применялась для вызова сервисов NFS, port mapper и т.д.
- XDR - стандартизированный формат сериализации (также используется в NFS, ZFS, libvirt и т.п.)

JNI (1998)

```java
class Foo {
    native int getValue(int arg);
}
```

```c
struct JNINativeInterface_ {
    void *reserved0;
    void *reserved1;
    void *reserved2;

    void *reserved3;
    jint (JNICALL *GetVersion)(JNIEnv *env);

    jclass (JNICALL *DefineClass)
      (JNIEnv *env, const char *name, jobject loader, const jbyte
       jsize len);
    jclass (JNICALL *FindClass)
      (JNIEnv *env, const char *name);

    jmethodID (JNICALL *FromReflectedMethod)
      (JNIEnv *env, jobject method);
    jfieldID (JNICALL *FromReflectedField)
      (JNIEnv *env, jobject field);
```

```c
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:     Foo
 * Method:    getValue
 * Signature: (I)I
 */
JNIEXPORT jint JNICALL Java_Foo_getValue
  (JNIEnv *, jobject, jint);

#ifdef __cplusplus
}
```

JNI

- Вызов через C ABI
- Примитивы в примитивы
- Объекты и строки - в хендлы
- Рефлективный интерфейс к JVM через JNIEnv
- Достаточно медленный
- Неудобный в разработке

NAPI (2009)

```
let nativeModule = require("NativeModule")
console.log(nativeModule.bar(42))
```

```
Napi::Value fooImpl(
    const Napi::CallbackInfo& info) {
    if (1 != info.Length()) {
        Napi::Error::New(info.Env(), "Error")
                .ThrowAsJavaScriptException();
        return info.Env().Undefined();
    }
    if (!info[0].IsNumber()) {
        Napi::Error::New(info.Env(), "Expected Number
            .ThrowAsJavaScriptException();
        return 0;
    }

    int32_t result =
        info[0].As<Napi::Number>().Int32Value() + 1;
    return Napi::Number::New(info.Env(), result);
}

static Napi::Object InitModule(Napi::Env env,
    Napi::Object exports) {
    exports.Set(Napi::String::New(env, "foo"),
        Napi::Function::New(env, fooImpl, "foo"));
    return exports;
}

NODE_API_MODULE("NativeModule", InitModule)
```

NAPI

- Всё в хендлы
- Рефлективный интерфейс к JS VM через Napi::CallbackInfo
- Медленный
- Неудобный в разработке

JS + WASM

```
WebAssembly.instantiateStreaming(fetch("module.wasm"),{})
    .then((obj) =>
        console.log(obj.instance.exports["fooImpl"](42))
    )
```

```
EMSCRIPTEN_KEEPALIVE extern "C" fooImpl(int value) {
    return value + 1;
}
```

JS + WASM

- Одна VM
- Вызов JS -> WASM через таблицу экспорта
- Передаются только примитивные типы
- Память WASM - типизированный массив для JS
- Emscripten создает более удобные мостики

Kotlin/JVM

- Автоматический и удобный интероп с Java и другими JVM языками (за счет общей VM и общих базовых типов (строки, коллекции))
- Есть непереводимые концепции, специально обрабатываемые при интеропе (inline методы и классы)
- Посредственный интероп с не JVM языками через JNI
- Экспериментальный автоматический C интероп

Kotlin/Native

- Автоматический (и неудобный) интероп с C
- Автоматический и удобный интероп с Objective-C
- C ABI, примитивные типы как есть
- Нет интеропа с другими вариантами Котлина (увы)

Метаинтероп

- Все интеропы разные
- А если нужно организовать интероп в нескольких средах исполнения?
- И не думать про интероп, а думать про бизнес-логику
- Встречайте: метаинтероп!

Требования

- Вызов нескольких библиотек на C++ скомпилированных в native и WASM из различных сред исполнения (V8, JSCore, JVM, browser JS)
- Высокая производительность
- Удобная для высокоуровневого программиста модель программирования
- С автоматическим управлением памятью
- Возможность отладки и профилировки
- Мы начинали с библиотеки Skia как целевой в интеропе

Основные принципы

- Ограниченный набор типов идущих через границу: примитивы, указатели, массивы примитивов и строки
- Ограничения на обратные вызовы native -> managed (только синхронно в некоторых вызовах), асинхронные - через события
- Пары объектов: managed peer и native
- Автоматические конверторы для допустимых типов
- Управление памятью со стороны managed runtime используя очередь финализации

Код для пользователя

```
protected drawShape(canvas: Canvas, rect: RRect) {
    if (!this.shapePaint) {
        this.shapePaint = Paint.make()
    } else {
        this.shapePaint.reset()
    }
    this.shapePaint.color = this.style.color.value
    this.style.gradient?.applyToPaint(this.shapePaint, rect)
    canvas.drawRRect(rect, this.shapePaint)
}
```

Paint и Canvas - нативные объекты

Механизм вызова

- Универсальный мост к C++
- интероп-специфичный переходник

```
KNativePointer impl_skoala_ImageFilter_MakeBlur(
    KFloat sigmaX, KFloat sigmaY, KInt tileModeInt,
    KNativePointer inputPtr, KInt* cropRectInts) {
    auto input = ptr<SkImageFilter>(inputPtr);
    auto crop = skoala::IRect::toSkIRect(cropRectInts);
    auto result = SkImageFilters::Blur(sigmaX, sigmaY,
        static_cast<SkTileMode>(tileModeInt), sk_ref_sp(input), crop.get())
        .release();
    return nativePtr(result);
}
KOALA_INTEROP_5(skoala_ImageFilter_MakeBlur, KNativePointer,
    KFloat, KFloat, KInt, KNativePointer, KInt*)
```

Переходник (NAPI)

```
#define KOALA_INTEROP_6(name, Ret, P0, P1, P2, P3, P4, P5) \
  Napi::Value Node_##name(const Napi::CallbackInfo& info) { \
      KOALA_MAYBE_LOG(name)                             \
      P0 p0 = getArgument<P0>(info, 0); \
      P1 p1 = getArgument<P1>(info, 1); \
      P2 p2 = getArgument<P2>(info, 2); \
      P3 p3 = getArgument<P3>(info, 3); \
      P4 p4 = getArgument<P4>(info, 4); \
      P5 p5 = getArgument<P5>(info, 5); \
      return makeResult<Ret>(info, impl_##name(p0, p1, p2, p3, p4, p5)); \
  } \
  MAKE_NODE_EXPORT(name)
```

Регистрация переходника (NAPI)

```
#define MAKE_NODE_EXPORT(name) \
    __attribute__((constructor)) \
    static void __init_##name() { \
        Exports::getInstance()->addImpl("_"#name, Node_##name); \
    }
```

```
static Napi::Object InitModule(Napi::Env env, Napi::Object exports) {
    for (const auto& impl: Exports::getInstance()->getImpls()) {
        exports.Set(Napi::String::New(env, impl.first.c_str()),
                    Napi::Function::New(env, impl.second, impl.first.c_str()));
    }
    return exports;
}
NODE_API_MODULE(INTEROP_LIBRARY_NAME, InitModule)
```

Конвертор аргументов (NAPI)

```
P0 p0 = getArgument<P0>(info, 0); \
P1 p1 = getArgument<P1>(info, 1); \
P2 p2 = getArgument<P2>(info, 2); \
P3 p3 = getArgument<P3>(info, 3); \
```

```
template <typename Type>
inline Type getArgument(const Napi::CallbackInfo& info, int index) = delete;
```

```
template <>
inline KInt getArgument<int32_t>(const Napi::CallbackInfo& info, int index) {
    NAPI_ASSERT_INDEX(info, index, 0);
    Napi::Value value = info[index]
    if (!value.IsNumber()) {
        Napi::Error::New(env, "Expected Number")
            .ThrowAsJavaScriptException();
        return 0;
    }
    return value.As<Napi::Number>().Int32Value();
}
```

Переходник (JNI)

```
#define KOALA_INTEROP_6(name, Ret, P0, P1, P2, P3, P4, P5) \
  KOALA_JNI_CALL(Ret) Java_org_##name(JNIEnv* env, jclass instance, \
  InteropTypeConverter<P0>::InteropType _p0, \
  InteropTypeConverter<P1>::InteropType _p1, \
  InteropTypeConverter<P2>::InteropType _p2, \
  InteropTypeConverter<P3>::InteropType _p3, \
  InteropTypeConverter<P4>::InteropType _p4, \
  InteropTypeConverter<P5>::InteropType _p5) { \
      KOALA_MAYBE_LOG(name) \
      P0 p0 = getArgument<P0>(env, _p0); \
      P1 p1 = getArgument<P1>(env, _p1); \
      P2 p2 = getArgument<P2>(env, _p2); \
      P3 p3 = getArgument<P3>(env, _p3); \
      P4 p4 = getArgument<P4>(env, _p4); \
      P5 p5 = getArgument<P5>(env, _p5); \
      Ret rv = makeResult<Ret>(env, impl_##name(p0, p1, p2, p3, p4, p5)); \
      releaseArgument(env, _p0, p0); \
      releaseArgument(env, _p1, p1); \
      releaseArgument(env, _p2, p2); \
      releaseArgument(env, _p3, p3); \
      releaseArgument(env, _p4, p4); \
      releaseArgument(env, _p5, p5); \
      return rv; \
}
```

```
#define KOALA_JNI_CALL(type) extern "C" JNIEXPORT type JNICALL
```

Конвертор аргументов (JNI)

```
template <typename Type>
inline Type getArgument(JNIEnv* env,
typename InteropTypeConverter<Type>::InteropType arg) {
  return InteropTypeConverter<Type>::convertFrom(env, arg);
}


template <typename Type>
inline void releaseArgument(JNIEnv* env,
typename InteropTypeConverter<Type>::InteropType arg, Type data) {
  InteropTypeConverter<Type>::release(env, arg, data);
}
```

```
template <typename Type>
inline typename InteropTypeConverter<Type>::InteropType
makeResult(JNIEnv* env, Type value) {
  return InteropTypeConverter<Type>::convertTo(env, value);
}
```

```
template<>
struct InteropTypeConverter<KInt*> {
    using InteropType = jintArray;
    static KInt* convertFrom(JNIEnv* env, InteropType value) {
        return value ? reinterpret_cast<KInt*>(
            env->GetIntArrayElements(value, nullptr)) : nullptr;
    }
    static InteropType convertTo(JNIEnv* env, KInt* value) = delete;
    static void release(JNIEnv* env, InteropType value, KInt* converted) {
        env->ReleaseIntArrayElements(value,
        reinterpret_cast<jint*>(converted), 0);
    }
};
```

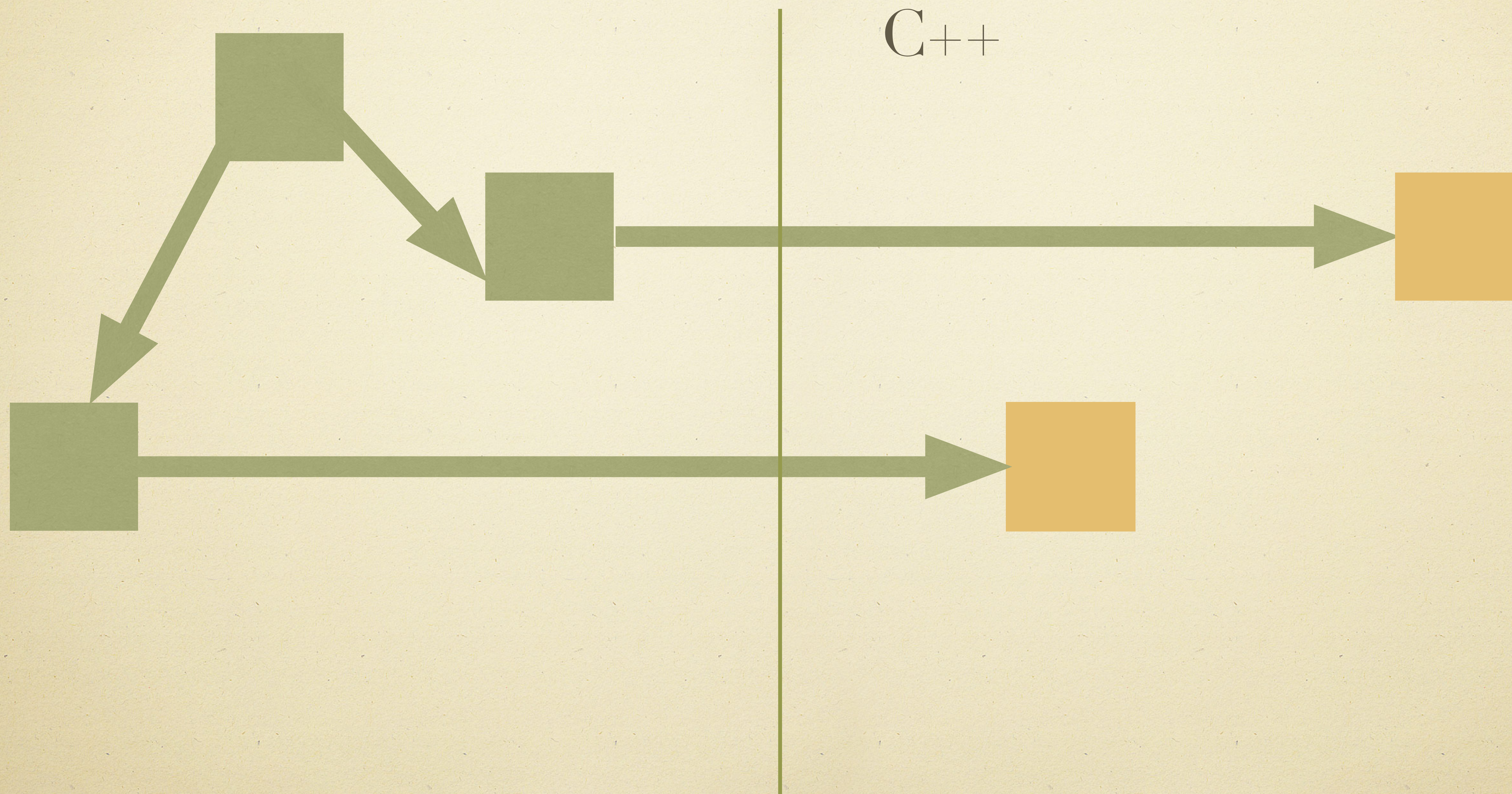
Конвертор (WASM)

```
export function withString<R>(data: string | undefined, exec: Exec<number, R>): R {
    if (data === undefined) return exec(nullptr)

    let array = encoder.encode(data, true)
    return withUint8Array(array, Access.READ, exec)
}
```

```
function withArray<C extends TypedArray, R>(
    data: C | undefined,
    access: Access,
    exec: ExecWithLength<number, R>,
    bytesPerElement: int32,
    ctor: (ptr: number, length: number) => C
): R {
    if (data === undefined || data.length === 0) {
        return exec(nullptr, 0)
    }
    let ptr = _malloc(data.length * bytesPerElement)
    let wasmArray = ctor(ptr, data.length)
    if (isRead(access)) {
        wasmArray.set(data)
    }
    let result = exec(ptr, data.length)
    if (isWrite(access)) {
        data.set(wasmArray)
    }
    _free(ptr)
    return result
}
```

```
template<class T>
struct InteropTypeConverter {
    using InteropType = T;
    static T convertFrom(InteropType value) { return value; }
    static InteropType convertTo(T value) { return value; }
};

template<>
struct InteropTypeConverter<KStringPtr> {
    using InteropType = const uint8_t*;
    static KStringPtr convertFrom(InteropType value) {
        KStringPtr result;
        if (value == nullptr) {
            return KStringPtr();
        } else {
            int len = (value[0] | (value[1] << 8) | (value[2] << 16)
            | (value[3] << 24));
            return KStringPtr((const char*)(value + 4), len);
        }
    }
    static InteropType convertTo(KStringPtr value) = delete;
};
```

Peer объекты



Peer объекты

```
export abstract class Finalizable {
    finalizer: pointer
    cleaner?: NativeThunk = undefined
    managed: boolean
    constructor(public ptr: pointer, finalizer: pointer, managed: boolean) {
        this.finalizer = finalizer
        this.managed = managed
        const handle = this.createHandle()
        if (this.managed) {
            const thunk = this.makeNativeThunk(ptr, finalizer, handle)
            finalizerRegister(this, thunk)
            this.cleaner = thunk
        }
```

```
export class Paint extends Finalizable {
    constructor(ptr: pointer) { super(ptr, Paint.getFinalizer()) }

    public static getFinalizer(): KNativePointer {
        return Module._skoala_Paint_GetFinalizer()
    }

    public static make(): Paint {
        const ptr = Module._skoala_Paint_Make()
        return new Paint(ptr)
    }
```

```
KNativePointer impl_skoala_Paint_Make(){
    return new SkPaint();
}
KOALA_INTEROP_0(skoala_Paint_Make, KNativePointer)
```

```
static void deletePaint(SkPaint* paint) {
    delete paint;
}

KNativePointer impl_skoala_Paint_GetFinalizer(){
    return fnPtr<SkPaint>(&deletePaint);
}
KOALA_INTEROP_0(skoala_Paint_GetFinalizer, KNativePointer)
```

Управление памятью

```
declare const FinalizationRegistry: FinalizationRegistryConstructor

const registry = new FinalizationRegistry((thunk: Thunk) => {
    thunk.clean()
})

export function finalizerRegister(target: object, thunk: object) {
    registry.register(target, thunk)
}

export function finalizerUnregister(target: object) {
    registry.unregister(target)
}
```

```
clean() {
    if (!isNullPtr(this.obj)) {
        this.destroyNative(this.obj, this.finalizer)
    }
    this.obj = nullptr
}
```

```
class NativeThunkImpl extends NativeThunk {
    constructor(obj: pointer, finalizer: pointer, name: string|undefined) {
        super(obj, finalizer, name)
    }
    destroyNative(ptr: pointer, finalizer: pointer): void {
        Module._InvokeFinalizer(ptr, finalizer)
    }
}
```

```
void impl_InvokeFinalizer(KNativePointer obj, KNativePointer final
    auto finalizer_f = reinterpret_cast<void (*)(KNativePointer)>(finalizer);
    finalizer_f(obj);
}
KOALA_INTEROP_V2(InvokeFinalizer, KNativePointer, KNativePointer)
```

Обратные вызовы в ВМ

- Обратные вызовы == проблемы с производительностью для ВМ
- Нужны не так часто, большинство вызовов - прямые
- Требуют передачи объекта-функции
- Ухудшают читаемость кода
- Иногда неизбежны!
- Асинхронные обратные вызовы - еще хуже

Реализация обратных вызовов (ВМ специфичная)

```
export function wrapCallback(callback: (args: Int32Array) => number,
    autoDisposable: boolean = true): int32 {
    return CallbackRegistry.instance.wrap(callback, autoDisposable)
}

export function disposeCallback(id: int32) {
    CallbackRegistry.instance.dispose(id)
}
```

```
static Napi::Reference<Napi::Function> g_koalaNapiCallbackDispatcher;

void impl_SetCallbackDispatcher(Napi::Object dispatcher) {
    g_koalaNapiCallbackDispatcher = Napi::Reference<Napi::Function>::New(
        dispatcher.As<Napi::Function>(), 1);
}
KOALA_INTEROP_V1(SetCallbackDispatcher, Napi::Object)
```

```
let callbackId = wrapCallback((args: Int32Array) => {
    return this.peer.customizationCallback(args)
}, false)
Module._SetCustomCallbackId(ptr, callbackId)
this.customCallbackId = callbackId
```

```
Module._SetCallbackDispatcher(callKoalaCallback)
```

```
export function callKoalaCallback(id: int32, args: Int32Array): number {
    return CallbackRegistry.instance.call(id, args)
}
```

Асинхронные обратные вызовы и цикл обработки сообщений

```
while (true) {
    processEvents()
    drawUI()
    await waitForVSync()
}
```

```
function waitForVSync(deviceId: KInt): Promise<void> {
    return new Promise((resolve, reject) => {
        Module._SetVsyncCallback(deviceId,
            wrapCallback((args: Int32Array) => {
            if (args[0] != 0)
                resolve()
            else
                reject(new Error("vsync failed"))
            return 0
        }))
    })
}
```

Частные случаи: строки и указатели

- Важный практический случай
- В JS VM нет 64-битного примитивного типа
- Специфично для рантайма и нативной стороны
- Важно избегать копирования при возможности

Строки (реализация)

```
void impl_skoala_Font_TextToGlyphs(KNativePointer fontPtr,
    const KStringPtr& text, KInt maxGlyphsCount, KUShort* resultGlyphs) {
    ptr<SkFont>(fontPtr)->textToGlyphs(text.c_str(), text.length(),
        SkTextEncoding::kUTF8, resultGlyphs, maxGlyphsCount);
}
```

```
template<>
struct InteropTypeConverter<KStringPtr> {
    using InteropType = jstring;
    static KStringPtr convertFrom(JNIEnv* env, InteropType value) {
        if (value == nullptr) return KStringPtr();
        KStringPtr result;
        int len = env->GetStringLength(value);
        result.resize(len);
        env->GetStringUTFRegion(value, 0, len, result.data());
        return result;
    }
    static InteropType convertTo(JNIEnv* env, KStringPtr value) = delete;
    static void release(JNIEnv* env, InteropType value,
        const KStringPtr& converted) {}
};
```

Указатели (JS VM)

- Пробовали Uint32Array[2] и biging
- bigint быстрее, но на JSCore - нельзя создать bigint с нативно стороны
- В WASM - достаточно int32
- Также можно ассоциировать указатель с JS объектом, но неудобно манипулировать на JS стороне

Итоги

- Реализован практический механизм обеспечения мультиплатформенного интеропа, отделяющий интероп от бизнес-логики
- Достаточный для обеспечения нужд большого реального проекта
- Поддерживается большой (и расширяемый) набор пар managed/unmanaged: V8, JSCore, JVM, Browser JS/C++,WASM
- Подход позволяет гибко добавлять новые интеропные типы с кастомизированными конверторами

Перспективы

- Комбинация с IDL для автоматической генерации универсальных мостов
- Комбинация с @FastNative and @CriticalNative (и похожими механизмами других ВМ)
- Автоматическое отражение типовой системы на нативной стороне
- Автоматическая сериализация сложных типов расширениями компилятора

Спасибо!