//-------------------------------------------------------------------------------------------------------

// Copyright (C) Microsoft. All rights reserved.

// Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

//-------------------------------------------------------------------------------------------------------

#pragma once

namespace Js {

class JavascriptExceptionContext

{

public:

struct StackFrame

{

private:

// Real script frames: functionBody, byteCodeOffset

// Native library builtin (or potentially virtual) frames: name

FunctionBody\* functionBody;

union

{

uint32 byteCodeOffset; // used for script functions (functionBody != nullptr)

PCWSTR name; // used for native/virtual frames (functionBody == nullptr)

};

StackTraceArguments argumentTypes;

public:

StackFrame() {}

StackFrame(JavascriptFunction\* func, const JavascriptStackWalker& walker, bool initArgumentTypes);

bool IsScriptFunction() const;

FunctionBody\* GetFunctionBody() const;

uint32 GetByteCodeOffset() const { return byteCodeOffset; }

LPCWSTR GetFunctionName() const;

HRESULT GetFunctionNameWithArguments(\_In\_ LPCWSTR \*outResult) const;

};

typedef JsUtil::List<StackFrame> StackTrace;

public:

JavascriptExceptionContext() :

m\_throwingFunction(nullptr),

m\_throwingFunctionByteCodeOffset(0),

m\_stackTrace(nullptr),

m\_originalStackTrace(nullptr)

{

}

JavascriptFunction\* ThrowingFunction() const { return m\_throwingFunction; }

uint32 ThrowingFunctionByteCodeOffset() const { return m\_throwingFunctionByteCodeOffset; }

void SetThrowingFunction(JavascriptFunction\* function, uint32 byteCodeOffset, void \* returnAddress);

bool HasStackTrace() const { return m\_stackTrace && m\_stackTrace->Count() > 0; }

StackTrace\* GetStackTrace() const { return m\_stackTrace; }

void SetStackTrace(StackTrace \*stackTrace) { m\_stackTrace = stackTrace; }

void SetOriginalStackTrace(StackTrace \*stackTrace) { Assert(m\_originalStackTrace == nullptr); m\_originalStackTrace = stackTrace; }

StackTrace\* GetOriginalStackTrace() const { return m\_originalStackTrace; }

private:

JavascriptFunction\* m\_throwingFunction;

uint32 m\_throwingFunctionByteCodeOffset;

StackTrace \*m\_stackTrace;

StackTrace \*m\_originalStackTrace;

};

}

//-------------------------------------------------------------------------------------------------------

// Copyright (C) Microsoft. All rights reserved.

// Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

//-------------------------------------------------------------------------------------------------------

#pragma once

#ifdef \_M\_AMD64

extern "C" void \*amd64\_CallWithFakeFrame(void \*target,

void \*frame,

size\_t spillSize,

size\_t argsSize,

void \*arg0 = nullptr) throw(...);

#elif defined(\_M\_ARM)

extern "C" void \*arm\_CallEhFrame(void \*target, void \*framePtr, void \*localsPtr, size\_t argsSize) throw(...);

extern "C" void \*arm\_CallCatch(void \*target, void \*framePtr, void \*localsPtr, size\_t argsSize, void \*catchObj) throw(...);

#elif defined(\_M\_ARM64)

extern "C" void \*arm64\_CallEhFrame(void \*target, void \*framePtr, void \*localsPtr, size\_t argsSize) throw(...);

extern "C" void \*arm64\_CallCatch(void \*target, void \*framePtr, void \*localsPtr, size\_t argsSize, void \*catchObj) throw(...);

#endif

namespace Js

{

class JavascriptExceptionContext;

class JavascriptExceptionOperators /\* All static \*/

{

public:

static const uint64 DefaultStackTraceLimit = 10;

static const uint64 MaxStackTraceLimit = \_UI64\_MAX;

// AutoCatchHandlerExists tracks where an exception will be caught and not propagated out.

// It should be included wherever an exception is caught and swallowed.

class AutoCatchHandlerExists

{

private:

bool m\_previousCatchHandlerExists;

bool m\_previousCatchHandlerToUserCodeStatus;

ThreadContext\* m\_threadContext;

void FetchNonUserCodeStatus(ScriptContext \*scriptContext);

public:

AutoCatchHandlerExists(ScriptContext\* scriptContext);

~AutoCatchHandlerExists();

};

static void \_\_declspec(noreturn) OP\_Throw(Var object, ScriptContext\* scriptContext);

static void \_\_declspec(noreturn) Throw(Var object, ScriptContext\* scriptContext);

static void \_\_declspec(noreturn) ThrowExceptionObject(Js::JavascriptExceptionObject\* exceptionObject, ScriptContext\* scriptContext, bool considerPassingToDebugger = false, PVOID returnAddress = NULL, bool resetStack = false);

static void \_\_declspec(noreturn) RethrowExceptionObject(Js::JavascriptExceptionObject\* exceptionObject, ScriptContext\* scriptContext, bool considerPassingToDebugger = false);

#ifdef \_M\_X64

static void \*OP\_TryCatch(void \*try\_, void \*catch\_, void \*frame, size\_t spillSize, size\_t argsSize, int hasBailedOutOffset, ScriptContext \*scriptContext);

static void \*OP\_TryFinally(void \*try\_, void \*finally\_, void \*frame, size\_t spillSize, size\_t argsSize, ScriptContext \*scriptContext);

#elif defined(\_M\_ARM32\_OR\_ARM64)

static void\* OP\_TryCatch(void\* continuationAddr, void\* handlerAddr, void\* framePtr, void \*localsPtr, size\_t argsSize, int hasBailedOutOffset, ScriptContext\* scriptContext);

static void\* OP\_TryFinally(void\* continuationAddr, void\* handlerAddr, void\* framePtr, void \*localsPtr, size\_t argsSize, ScriptContext\* scriptContext);

#else

static void\* OP\_TryCatch(void\* continuationAddr, void\* handlerAddr, void\* framePtr, int hasBailedOutOffset, ScriptContext\* scriptContext);

static void\* OP\_TryFinally(void\* continuationAddr, void\* handlerAddr, void\* framePtr, ScriptContext\* scriptContext);

#endif

#if defined(DBG) && defined(\_M\_IX86)

static void DbgCheckEHChain();

#endif

static JavascriptExceptionObject\* GetOutOfMemoryExceptionObject(ScriptContext\* scriptContext);

static Var OP\_RuntimeTypeError(MessageId messageId, ScriptContext\* scriptContext);

static Var OP\_RuntimeRangeError(MessageId messageId, ScriptContext\* scriptContext);

static Var OP\_RuntimeReferenceError(MessageId messageId, ScriptContext\* scriptContext);

static void \_\_declspec(noreturn) ThrowOutOfMemory(ScriptContext\* scriptContext);

static void \_\_declspec(noreturn) ThrowStackOverflow(ScriptContext\* scriptContext, PVOID returnAddress);

static uint64 GetStackTraceLimit(Var thrownObject, ScriptContext\* scriptContext);

static Var ThrowTypeErrorAccessor(RecyclableObject\* function, CallInfo callInfo, ...);

static Var ThrowTypeErrorCallerAccessor(RecyclableObject\* function, CallInfo callInfo, ...);

static Var ThrowTypeErrorCalleeAccessor(RecyclableObject\* function, CallInfo callInfo, ...);

static Var ThrowTypeErrorArgumentsAccessor(RecyclableObject\* function, CallInfo callInfo, ...);

static Var StackTraceAccessor(RecyclableObject\* function, CallInfo callInfo, ...);

static void WalkStackForExceptionContext(ScriptContext& scriptContext, JavascriptExceptionContext& exceptionContext, Var thrownObject, uint64 stackCrawlLimit, PVOID returnAddress, bool isThrownException = true, bool resetSatck = false);

static void AddStackTraceToObject(Var obj, JavascriptExceptionContext::StackTrace\* stackTrace, ScriptContext& scriptContext, bool isThrownException = true, bool resetSatck = false);

static uint64 StackCrawlLimitOnThrow(Var thrownObject, ScriptContext& scriptContext);

class EntryInfo

{

public:

static FunctionInfo ThrowTypeErrorAccessor;

static FunctionInfo StackTraceAccessor;

// For strict mode

static FunctionInfo ThrowTypeErrorCallerAccessor;

static FunctionInfo ThrowTypeErrorCalleeAccessor;

static FunctionInfo ThrowTypeErrorArgumentsAccessor;

};

private:

static JavascriptFunction \* WalkStackForExceptionContextInternal(ScriptContext& scriptContext, JavascriptExceptionContext& exceptionContext, Var thrownObject, uint32& callerByteCodeOffset,

uint64 stackCrawlLimit, PVOID returnAddress, bool isThrownException, bool resetStack = false);

static void ThrowExceptionObjectInternal(Js::JavascriptExceptionObject \* exceptionObject, ScriptContext\* scriptContext, bool fillExceptionContext, bool considerPassingToDebugger, PVOID returnAddress, bool resetStack);

static BOOL GetCaller(JavascriptStackWalker& walker, JavascriptFunction\*& jsFunc);

static void DumpStackTrace(JavascriptExceptionContext& exceptionContext, bool isThrownException = true);

static JavascriptExceptionContext::StackTrace\* TrimStackTraceForThrownObject(JavascriptExceptionContext::StackTrace\* stackTraceOriginal, Var thrownObject, ScriptContext& scriptContext);

static void AppendExternalFrameToStackTrace(CompoundString\* bs, LPCWSTR functionName, LPCWSTR fileName, ULONG lineNumber, LONG characterPosition);

static void AppendLibraryFrameToStackTrace(CompoundString\* bs, LPCWSTR functionName);

static bool IsErrorInstance(Var thrownObject);

static bool CrawlStackForWER(Js::ScriptContext& scriptContext);

static void DispatchExceptionToDebugger(Js::JavascriptExceptionObject \* exceptionObject, ScriptContext\* scriptContext);

};

} // namespace Js

//-------------------------------------------------------------------------------------------------------

// Copyright (C) Microsoft. All rights reserved.

// Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

//-------------------------------------------------------------------------------------------------------

#include "RuntimeLanguagePch.h"

#include "shlwapi.h"

#include "Language\InterpreterStackFrame.h"

#ifdef \_M\_IX86

#ifdef \_CONTROL\_FLOW\_GUARD

extern "C" PVOID \_\_guard\_check\_icall\_fptr;

#endif

#endif

namespace Js

{

void JavascriptExceptionOperators::AutoCatchHandlerExists::FetchNonUserCodeStatus(ScriptContext \* scriptContext)

{

Assert(scriptContext);

bool fFound = false;

// If the outer try catch was already in the user code, no need to go any further.

if (!m\_previousCatchHandlerToUserCodeStatus)

{

Js::JavascriptFunction\* caller;

if (JavascriptStackWalker::GetCaller(&caller, scriptContext))

{

Js::FunctionBody \*funcBody = NULL;

if (caller != NULL && (funcBody = caller->GetFunctionBody()) != NULL)

{

m\_threadContext->SetIsUserCode(funcBody->IsNonUserCode() == false);

fFound = true;

}

}

}

if (!fFound)

{

// If not successfully able to find the caller, set this catch handler belongs to the user code.

m\_threadContext->SetIsUserCode(true);

}

}

JavascriptExceptionOperators::AutoCatchHandlerExists::AutoCatchHandlerExists(ScriptContext\* scriptContext)

{

Assert(scriptContext);

m\_threadContext = scriptContext->GetThreadContext();

Assert(m\_threadContext);

m\_previousCatchHandlerExists = m\_threadContext->HasCatchHandler();

m\_threadContext->SetHasCatchHandler(TRUE);

m\_previousCatchHandlerToUserCodeStatus = m\_threadContext->IsUserCode();

if (scriptContext->IsInDebugMode())

{

FetchNonUserCodeStatus(scriptContext);

}

}

JavascriptExceptionOperators::AutoCatchHandlerExists::~AutoCatchHandlerExists()

{

m\_threadContext->SetHasCatchHandler(m\_previousCatchHandlerExists);

m\_threadContext->SetIsUserCode(m\_previousCatchHandlerToUserCodeStatus);

}

bool JavascriptExceptionOperators::CrawlStackForWER(Js::ScriptContext& scriptContext)

{

return Js::Configuration::Global.flags.WERExceptionSupport && !scriptContext.GetThreadContext()->HasCatchHandler();

}

uint64 JavascriptExceptionOperators::StackCrawlLimitOnThrow(Var thrownObject, ScriptContext& scriptContext)

{

return CrawlStackForWER(scriptContext) ? MaxStackTraceLimit : GetStackTraceLimit(thrownObject, &scriptContext);

}

#ifdef \_M\_X64

void \*JavascriptExceptionOperators::OP\_TryCatch(void \*tryAddr,

void \*catchAddr,

void \*frame,

size\_t spillSize,

size\_t argsSize,

int hasBailedOutOffset,

ScriptContext \*scriptContext)

{

void \*continuation = nullptr;

JavascriptExceptionObject \*exception = nullptr;

PROBE\_STACK(scriptContext, Constants::MinStackDefault + spillSize + argsSize);

try

{

Js::JavascriptExceptionOperators::AutoCatchHandlerExists autoCatchHandlerExists(scriptContext);

continuation = amd64\_CallWithFakeFrame(tryAddr, frame, spillSize, argsSize);

}

catch (JavascriptExceptionObject \*caughtException)

{

exception = caughtException;

}

if (exception)

{

exception = exception->CloneIfStaticExceptionObject(scriptContext);

bool hasBailedOut = \*(bool\*)((char\*)frame + hasBailedOutOffset); // stack offsets are negative

if (hasBailedOut)

{

// If we have bailed out, this exception is coming from the interpreter. It should not have been caught;

// it so happens that this catch was on the stack and caught the exception.

// Re-throw!

throw exception;

}

Var exceptionObject = exception->GetThrownObject(scriptContext);

AssertMsg(exceptionObject, "Caught object is null.");

continuation = amd64\_CallWithFakeFrame(catchAddr, frame, spillSize, argsSize, exceptionObject);

}

return continuation;

}

void \*JavascriptExceptionOperators::OP\_TryFinally(void \*tryAddr,

void \*finallyAddr,

void \*frame,

size\_t spillSize,

size\_t argsSize,

ScriptContext \*scriptContext)

{

void \*tryContinuation = nullptr;

void \*finallyContinuation = nullptr;

JavascriptExceptionObject \*exception = nullptr;

PROBE\_STACK(scriptContext, Constants::MinStackDefault + spillSize + argsSize);

try

{

tryContinuation = amd64\_CallWithFakeFrame(tryAddr, frame, spillSize, argsSize);

}

catch (JavascriptExceptionObject \*caughtException)

{

exception = caughtException;

}

if (exception)

{

// Clone static exception object early in case finally block overwrites it

exception = exception->CloneIfStaticExceptionObject(scriptContext);

}

finallyContinuation = amd64\_CallWithFakeFrame(finallyAddr, frame, spillSize, argsSize);

if (finallyContinuation)

{

return finallyContinuation;

}

if (exception)

{

throw exception;

}

return tryContinuation;

}

#elif defined(\_M\_ARM32\_OR\_ARM64)

void \*JavascriptExceptionOperators::OP\_TryCatch(

void \*tryAddr,

void \*catchAddr,

void \*framePtr,

void \*localsPtr,

size\_t argsSize,

int hasBailedOutOffset,

ScriptContext \*scriptContext)

{

void \*continuation = nullptr;

JavascriptExceptionObject \*exception = nullptr;

PROBE\_STACK(scriptContext, Constants::MinStackDefault + argsSize);

try

{

Js::JavascriptExceptionOperators::AutoCatchHandlerExists autoCatchHandlerExists(scriptContext);

#if defined(\_M\_ARM)

continuation = arm\_CallEhFrame(tryAddr, framePtr, localsPtr, argsSize);

#elif defined(\_M\_ARM64)

continuation = arm64\_CallEhFrame(tryAddr, framePtr, localsPtr, argsSize);

#endif

}

catch (JavascriptExceptionObject \*caughtException)

{

exception = caughtException;

}

if (exception)

{

exception = exception->CloneIfStaticExceptionObject(scriptContext);

bool hasBailedOut = \*(bool\*)((char\*)localsPtr + hasBailedOutOffset); // stack offsets are sp relative

if (hasBailedOut)

{

// If we have bailed out, this exception is coming from the interpreter. It should not have been caught;

// it so happens that this catch was on the stack and caught the exception.

// Re-throw!

throw exception;

}

Var exceptionObject = exception->GetThrownObject(scriptContext);

AssertMsg(exceptionObject, "Caught object is null.");

#if defined(\_M\_ARM)

continuation = arm\_CallCatch(catchAddr, framePtr, localsPtr, argsSize, exceptionObject);

#elif defined(\_M\_ARM64)

continuation = arm64\_CallCatch(catchAddr, framePtr, localsPtr, argsSize, exceptionObject);

#endif

}

return continuation;

}

void \*JavascriptExceptionOperators::OP\_TryFinally(

void \*tryAddr,

void \*finallyAddr,

void \*framePtr,

void \*localsPtr,

size\_t argsSize,

ScriptContext \*scriptContext)

{

void \*tryContinuation = nullptr;

void \*finallyContinuation = nullptr;

JavascriptExceptionObject \*exception = nullptr;

PROBE\_STACK(scriptContext, Constants::MinStackDefault + argsSize);

try

{

#if defined(\_M\_ARM)

tryContinuation = arm\_CallEhFrame(tryAddr, framePtr, localsPtr, argsSize);

#elif defined(\_M\_ARM64)

tryContinuation = arm64\_CallEhFrame(tryAddr, framePtr, localsPtr, argsSize);

#endif

}

catch (JavascriptExceptionObject \*caughtException)

{

exception = caughtException;

}

if (exception)

{

// Clone static exception object early in case finally block overwrites it

exception = exception->CloneIfStaticExceptionObject(scriptContext);

}

#if defined(\_M\_ARM)

finallyContinuation = arm\_CallEhFrame(finallyAddr, framePtr, localsPtr, argsSize);

#elif defined(\_M\_ARM64)

finallyContinuation = arm64\_CallEhFrame(finallyAddr, framePtr, localsPtr, argsSize);

#endif

if (finallyContinuation)

{

return finallyContinuation;

}

if (exception)

{

throw exception;

}

return tryContinuation;

}

#else

#pragma warning(push)

#pragma warning(disable:4731) // frame pointer register 'ebp' modified by inline assembly code

void\* JavascriptExceptionOperators::OP\_TryCatch(void\* tryAddr, void\* handlerAddr, void\* framePtr, int hasBailedOutOffset, ScriptContext \*scriptContext)

{

void\* continuationAddr = NULL;

Js::JavascriptExceptionObject\* pExceptionObject = NULL;

PROBE\_STACK(scriptContext, Constants::MinStackDefault);

try

{

Js::JavascriptExceptionOperators::AutoCatchHandlerExists autoCatchHandlerExists(scriptContext);

// Adjust the frame pointer and call into the try.

// If the try completes without raising an exception, it will pass back the continuation address.

// Bug in compiler optimizer: try-catch can be optimized away if the try block contains \_\_asm calls into function

// that may throw. The current workaround is to add the following dummy throw to prevent this optimization.

if (!tryAddr)

{

Js::Throw::InternalError();

}

#ifdef \_M\_IX86

void \*savedEsp;

\_\_asm

{

// Save and restore the callee-saved registers around the call.

// TODO: track register kills by region and generate per-region prologs and epilogs

push esi

push edi

push ebx

// 8-byte align frame to improve floating point perf of our JIT'd code.

// Save ESP

mov ecx, esp

mov savedEsp, ecx

and esp, -8

// Set up the call target, save the current frame ptr, and adjust the frame to access

// locals in native code.

mov eax, tryAddr

#if 0 && defined(\_CONTROL\_FLOW\_GUARD)

// verify that the call target is valid

mov ebx, eax ; save call target

mov ecx, eax

call [\_\_guard\_check\_icall\_fptr]

mov eax, ebx ; restore call target

#endif

push ebp

mov ebp, framePtr

call eax

pop ebp

// The native code gives us the address where execution should continue on exit

// from the region.

mov continuationAddr, eax

// Restore ESP

mov ecx, savedEsp

mov esp, ecx

pop ebx

pop edi

pop esi

}

#else

AssertMsg(FALSE, "Unsupported native try-catch handler");

#endif

}

catch(Js::JavascriptExceptionObject \* exceptionObject)

{

pExceptionObject = exceptionObject;

}

// Let's run user catch handler code only after the stack has been unwound.

if(pExceptionObject)

{

pExceptionObject = pExceptionObject->CloneIfStaticExceptionObject(scriptContext);

bool hasBailedOut = \*(bool\*)((char\*)framePtr + hasBailedOutOffset); // stack offsets are negative

if (hasBailedOut)

{

// If we have bailed out, this exception is coming from the interpreter. It should not have been caught;

// it so happens that this catch was on the stack and caught the exception.

// Re-throw!

throw pExceptionObject;

}

Var catchObject = pExceptionObject->GetThrownObject(scriptContext);

AssertMsg(catchObject, "Caught object is NULL");

#ifdef \_M\_IX86

void \*savedEsp;

\_\_asm

{

// Save and restore the callee-saved registers around the call.

// TODO: track register kills by region and generate per-region prologs and epilogs

push esi

push edi

push ebx

// 8-byte align frame to improve floating point perf of our JIT'd code.

// Save ESP

mov ecx, esp

mov savedEsp, ecx

and esp, -8

// Set up the call target

mov ecx, handlerAddr

#if 0 && defined(\_CONTROL\_FLOW\_GUARD)

// verify that the call target is valid

mov ebx, ecx ; save call target

call [\_\_guard\_check\_icall\_fptr]

mov ecx, ebx ; restore call target

#endif

// Set up catch object, save the current frame ptr, and adjust the frame to access

// locals in native code.

mov eax, catchObject

push ebp

mov ebp, framePtr

call ecx

pop ebp

// The native code gives us the address where execution should continue on exit

// from the region.

mov continuationAddr, eax

// Restore ESP

mov ecx, savedEsp

mov esp, ecx

pop ebx

pop edi

pop esi

}

#else

AssertMsg(FALSE, "Unsupported native try-catch handler");

#endif

}

return continuationAddr;

}

void\* JavascriptExceptionOperators::OP\_TryFinally(void\* tryAddr, void\* handlerAddr, void\* framePtr, ScriptContext \*scriptContext)

{

Js::JavascriptExceptionObject\* pExceptionObject = NULL;

void\* continuationAddr = NULL;

PROBE\_STACK(scriptContext, Constants::MinStackDefault);

try

{

// Bug in compiler optimizer: try-catch can be optimized away if the try block contains \_\_asm calls into function

// that may throw. The current workaround is to add the following dummy throw to prevent this optimization.

// It seems like compiler got smart and still optimizes if the exception is not JavascriptExceptionObject (see catch handler below).

// In order to circumvent that we are throwing OutOfMemory.

if (!tryAddr)

{

Assert(false);

ThrowOutOfMemory(scriptContext);

}

#ifdef \_M\_IX86

void \*savedEsp;

\_\_asm

{

// Save and restore the callee-saved registers around the call.

// TODO: track register kills by region and generate per-region prologs and epilogs

push esi

push edi

push ebx

// 8-byte align frame to improve floating point perf of our JIT'd code.

// Save ESP

mov ecx, esp

mov savedEsp, ecx

and esp, -8

// Set up the call target, save the current frame ptr, and adjust the frame to access

// locals in native code.

mov eax, tryAddr

#if 0 && defined(\_CONTROL\_FLOW\_GUARD)

// verify that the call target is valid

mov ebx, eax ; save call target

mov ecx, eax

call [\_\_guard\_check\_icall\_fptr]

mov eax, ebx ; restore call target

#endif

push ebp

mov ebp, framePtr

call eax

pop ebp

// The native code gives us the address where execution should continue on exit

// from the region.

mov continuationAddr, eax

// Restore ESP

mov ecx, savedEsp

mov esp, ecx

pop ebx

pop edi

pop esi

}

#else

AssertMsg(FALSE, "Unsupported native try-finally handler");

#endif

}

catch(Js::JavascriptExceptionObject\* e)

{

pExceptionObject = e;

}

if (pExceptionObject)

{

// Clone static exception object early in case finally block overwrites it

pExceptionObject = pExceptionObject->CloneIfStaticExceptionObject(scriptContext);

}

void\* newContinuationAddr = NULL;

#ifdef \_M\_IX86

void \*savedEsp;

\_\_asm

{

// Save and restore the callee-saved registers around the call.

// TODO: track register kills by region and generate per-region prologs and epilogs

push esi

push edi

push ebx

// 8-byte align frame to improve floating point perf of our JIT'd code.

// Save ESP

mov ecx, esp

mov savedEsp, ecx

and esp, -8

// Set up the call target

mov eax, handlerAddr

#if 0 && defined(\_CONTROL\_FLOW\_GUARD)

// verify that the call target is valid

mov ebx, eax ; save call target

mov ecx, eax

call [\_\_guard\_check\_icall\_fptr]

mov eax, ebx ; restore call target

#endif

// save the current frame ptr, and adjust the frame to access

// locals in native code.

push ebp

mov ebp, framePtr

call eax

pop ebp

// The native code gives us the address where execution should continue on exit

// from the finally, but only if flow leaves the finally before it completes.

mov newContinuationAddr, eax

// Restore ESP

mov ecx, savedEsp

mov esp, ecx

pop ebx

pop edi

pop esi

}

#else

AssertMsg(FALSE, "Unsupported native try-finally handler");

#endif

if (newContinuationAddr != NULL)

{

// Non-null return value from the finally indicates that the finally seized the flow

// with a jump/return out of the region. Continue at that address instead of handling

// the exception.

return newContinuationAddr;

}

if (pExceptionObject)

{

throw pExceptionObject;

}

return continuationAddr;

}

#endif

void \_\_declspec(noreturn) JavascriptExceptionOperators::OP\_Throw(Var object, ScriptContext\* scriptContext)

{

Throw(object, scriptContext);

}

#if defined(DBG) && defined(\_M\_IX86)

extern "C" void \* \_except\_handler4;

void JavascriptExceptionOperators::DbgCheckEHChain()

{

#if 0

// This debug check is disabled until we figure out how to trace an fs:0 chain if we throw from inside

// a finally.

void \*currentFS0;

ThreadContext \* threadContext = ThreadContext::GetContextForCurrentThread();

if (!threadContext->IsScriptActive())

{

return;

}

// Walk the FS:0 chain of exception handlers, until the FS:0 handler in CallRootFunction.

// We should only see SEH frames on the way.

// We do allow C++ EH frames as long as there is no active objects (state = -1).

// That's because we may see frames that have calls to new(). This introduces an EH frame

// to call delete if the constructor throws. Our constructors shouldn't throw, so we should be fine.

currentFS0 = (void\*)\_\_readfsdword(0);

while (currentFS0 != threadContext->callRootFS0)

{

// EH struct:

// void \* next;

// void \* handler;

// int state;

AssertMsg(\*((void\*\*)currentFS0 + 1) == &\_except\_handler4

|| \*((int\*)currentFS0 + 2) == -1, "Found a non SEH exception frame on stack");

currentFS0 = \*(void\*\*)currentFS0;

}

#endif

}

#endif

void JavascriptExceptionOperators::Throw(Var object, ScriptContext \* scriptContext)

{

#if defined(DBG) && defined(\_M\_IX86)

DbgCheckEHChain();

#endif

Assert(scriptContext != nullptr);

// TODO: FastDOM Trampolines will throw JS Exceptions but are not isScriptActive

//AssertMsg(scriptContext->GetThreadContext()->IsScriptActive() ||

// (JavascriptError::Is(object) && (JavascriptError::FromVar(object))->IsExternalError()),

// "Javascript exception raised when script is not active");

AssertMsg(scriptContext->GetThreadContext()->IsInScript() ||

(JavascriptError::Is(object) && (JavascriptError::FromVar(object))->IsExternalError()),

"Javascript exception raised without being in CallRootFunction");

JavascriptError \*javascriptError = nullptr;

if (JavascriptError::Is(object))

{

// We keep track of the JavascriptExceptionObject that was created when this error

// was first thrown so that we can always get the correct metadata.

javascriptError = JavascriptError::FromVar(object);

JavascriptExceptionObject \*exceptionObject = javascriptError->GetJavascriptExceptionObject();

if (exceptionObject)

{

JavascriptExceptionOperators::ThrowExceptionObject(exceptionObject, scriptContext, true);

}

}

JavascriptExceptionObject \* exceptionObject =

RecyclerNew(scriptContext->GetRecycler(), JavascriptExceptionObject, object, scriptContext, NULL);

bool resetStack = false;

if (javascriptError)

{

if (!javascriptError->IsStackPropertyRedefined())

{

/\*

Throwing an error object. Original stack property will be pointing to the stack created at time of Error constructor.

Reset the stack property to match IE11 behavior

\*/

resetStack = true;

}

javascriptError->SetJavascriptExceptionObject(exceptionObject);

}

JavascriptExceptionOperators::ThrowExceptionObject(exceptionObject, scriptContext, /\*considerPassingToDebugger=\*/ true, /\*returnAddress=\*/ nullptr, resetStack);

}

void

JavascriptExceptionOperators::WalkStackForExceptionContext(ScriptContext& scriptContext, JavascriptExceptionContext& exceptionContext, Var thrownObject, uint64 stackCrawlLimit, PVOID returnAddress, bool isThrownException, bool resetSatck)

{

uint32 callerBytecodeOffset;

JavascriptFunction \* jsFunc = WalkStackForExceptionContextInternal(scriptContext, exceptionContext, thrownObject, callerBytecodeOffset, stackCrawlLimit, returnAddress, isThrownException, resetSatck);

if (jsFunc)

{

// If found, the caller is a function, and we can retrieve the debugger info from there

// otherwise it's probably just accessing property. While it is still possible to throw

// from that context, we just won't be able to get the line number etc., which make sense.

exceptionContext.SetThrowingFunction(jsFunc, callerBytecodeOffset, returnAddress);

}

}

JavascriptFunction \*

JavascriptExceptionOperators::WalkStackForExceptionContextInternal(ScriptContext& scriptContext, JavascriptExceptionContext& exceptionContext, Var thrownObject,

uint32& callerByteCodeOffset, uint64 stackCrawlLimit, PVOID returnAddress, bool isThrownException, bool resetStack)

{

JavascriptStackWalker walker(&scriptContext, true, returnAddress);

JavascriptFunction\* jsFunc = nullptr;

if (!GetCaller(walker, jsFunc))

{

return nullptr;

}

// Skip to first non-Library code

// Similar behavior to GetCaller returning false

if(jsFunc->IsLibraryCode() && !walker.GetNonLibraryCodeCaller(&jsFunc))

{

return nullptr;

}

JavascriptFunction \* caller = jsFunc;

callerByteCodeOffset = walker.GetByteCodeOffset();

Assert(!caller->IsLibraryCode());

// NOTE Don't set the throwing exception here, because we might need to box it and will cause a nested stack walker

// instead, return it to be set in WalkStackForExceptionContext

if (stackCrawlLimit == 0)

{

return caller;

}

const bool crawlStackForWER = CrawlStackForWER(scriptContext);

// If we take an OOM (JavascriptException for OOM if script is active), just bail early and return what we've got

HRESULT hr;

JavascriptExceptionContext::StackTrace \*stackTrace = NULL;

BEGIN\_TRANSLATE\_EXCEPTION\_AND\_ERROROBJECT\_TO\_HRESULT\_NESTED

{

// In WER scenario, we should combine the original stack with latest throw stack as the final throw might be coming form

// a different stack.

uint64 i = 1;

if (crawlStackForWER && thrownObject && Js::JavascriptError::Is(thrownObject))

{

Js::JavascriptError\* errorObject = Js::JavascriptError::FromVar(thrownObject);

Js::JavascriptExceptionContext::StackTrace \*originalStackTrace = NULL;

const Js::JavascriptExceptionObject\* originalExceptionObject = errorObject->GetJavascriptExceptionObject();

if (!resetStack && errorObject->GetInternalProperty(errorObject, InternalPropertyIds::StackTrace, (Js::Var\*) &originalStackTrace, NULL, &scriptContext) &&

(originalStackTrace != nullptr))

{

exceptionContext.SetOriginalStackTrace(originalStackTrace);

}

else

{

if (originalExceptionObject != nullptr)

{

exceptionContext.SetOriginalStackTrace(originalExceptionObject->GetExceptionContext()->GetStackTrace());

}

}

}

stackTrace = RecyclerNew(scriptContext.GetRecycler(), JavascriptExceptionContext::StackTrace, scriptContext.GetRecycler());

do

{

JavascriptExceptionContext::StackFrame stackFrame(jsFunc, walker, crawlStackForWER);

stackTrace->Add(stackFrame);

} while (walker.GetDisplayCaller(&jsFunc) && i++ < stackCrawlLimit);

}

END\_TRANSLATE\_EXCEPTION\_AND\_ERROROBJECT\_TO\_HRESULT\_INSCRIPT(hr);

exceptionContext.SetStackTrace(stackTrace);

DumpStackTrace(exceptionContext, isThrownException);

return caller;

}

// We might be trying to raise a stack overflow exception from the interpreter before

// we've executed code in the current script stack frame. In that case the current byte

// code offset is 0. In such cases walk to the caller's caller.

BOOL JavascriptExceptionOperators::GetCaller(JavascriptStackWalker& walker, JavascriptFunction\*& jsFunc)

{

if (! walker.GetCaller(&jsFunc))

{

return FALSE;

}

if (! walker.GetCurrentInterpreterFrame() ||

walker.GetCurrentInterpreterFrame()->GetReader()->GetCurrentOffset() > 0)

{

return TRUE;

}

return walker.GetCaller(&jsFunc);

}

void JavascriptExceptionOperators::DumpStackTrace(JavascriptExceptionContext& exceptionContext, bool isThrownException)

{

#ifdef ENABLE\_DEBUG\_CONFIG\_OPTIONS

if (! exceptionContext.GetStackTrace()

|| ! Configuration::Global.flags.Dump.IsEnabled(ExceptionStackTracePhase)

|| ! isThrownException)

{

return;

}

Output::Print(L"\nStack trace for thrown exception\n");

JavascriptExceptionContext::StackTrace \*stackTrace = exceptionContext.GetStackTrace();

for (int i=0; i < stackTrace->Count(); i++)

{

Js::JavascriptExceptionContext::StackFrame currFrame = stackTrace->Item(i);

ULONG lineNumber = 0;

LONG characterPosition = 0;

if (currFrame.IsScriptFunction() && !currFrame.GetFunctionBody()->GetUtf8SourceInfo()->GetIsLibraryCode())

{

currFrame.GetFunctionBody()->GetLineCharOffset(currFrame.GetByteCodeOffset(), &lineNumber, &characterPosition);

}

Output::Print(L" %3d: %s (%d, %d)\n", i, currFrame.GetFunctionName(), lineNumber, characterPosition);

}

Output::Flush();

#endif

}

/// ---------------------------------------------------------------------------------------------------

/// When allocators throw out of memory exception - scriptContext is NULL

/// ---------------------------------------------------------------------------------------------------

JavascriptExceptionObject \* JavascriptExceptionOperators::GetOutOfMemoryExceptionObject(ScriptContext \*scriptContext)

{

ThreadContext \*threadContext = scriptContext ?

scriptContext->GetThreadContext() :

ThreadContext::GetContextForCurrentThread();

JavascriptExceptionObject \*oomExceptionObject = threadContext->GetPendingOOMErrorObject();

Assert(oomExceptionObject);

return oomExceptionObject;

}

void JavascriptExceptionOperators::ThrowOutOfMemory(ScriptContext \*scriptContext)

{

ThreadContext \*threadContext = scriptContext ?

scriptContext->GetThreadContext() :

ThreadContext::GetContextForCurrentThread();

threadContext->ClearDisableImplicitFlags();

JavascriptExceptionObject \*oom = JavascriptExceptionOperators::GetOutOfMemoryExceptionObject(scriptContext);

JavascriptExceptionOperators::ThrowExceptionObject(oom, scriptContext);

}

void JavascriptExceptionOperators::ThrowStackOverflow(ScriptContext \*scriptContext, PVOID returnAddress)

{

Assert(scriptContext);

ThreadContext \*threadContext = scriptContext->GetThreadContext();

JavascriptExceptionObject \*so = threadContext->GetPendingSOErrorObject();

Assert(so);

// Disable implicit call before calling into recycler (to prevent QueryContinue/dispose from leave script and stack overflow again)

threadContext->DisableImplicitCall();

Var thrownObject = scriptContext->GetLibrary()->CreateStackOverflowError();

so->SetThrownObject(thrownObject);

// NOTE: Do not ClearDisableImplicitFlags() here. We still need to allocate StackTrace, etc. Keep implicit call disabled till actual

// throw (ThrowExceptionObjectInternal will ClearDisableImplicitFlags before throw). If anything wrong happens in between which throws

// a new exception, the new throw will ClearDisableImplicitFlags.

JavascriptExceptionOperators::ThrowExceptionObject(so, scriptContext, false, returnAddress);

}

void JavascriptExceptionOperators::ThrowExceptionObjectInternal(Js::JavascriptExceptionObject \* exceptionObject, ScriptContext\* scriptContext, bool fillExceptionContext, bool considerPassingToDebugger, PVOID returnAddress, bool resetStack)

{

if (scriptContext)

{

if (fillExceptionContext)

{

Assert(exceptionObject);

JavascriptExceptionContext exceptionContext;

Var thrownObject = exceptionObject->GetThrownObject(nullptr);

WalkStackForExceptionContext(\*scriptContext, exceptionContext, thrownObject, StackCrawlLimitOnThrow(thrownObject, \*scriptContext), returnAddress, /\*isThrownException=\*/ true, resetStack);

exceptionObject->FillError(exceptionContext, scriptContext);

AddStackTraceToObject(thrownObject, exceptionContext.GetStackTrace(), \*scriptContext, /\*isThrownException=\*/ true, resetStack);

if (considerPassingToDebugger)

{

DispatchExceptionToDebugger(exceptionObject, scriptContext);

}

}

Assert(!scriptContext ||

// If we disabled implicit calls and we did record an implicit call, do not throw.

// Check your helper to see if a call recorded an implicit call that might cause an invalid value

!(

scriptContext->GetThreadContext()->IsDisableImplicitCall() &&

scriptContext->GetThreadContext()->GetImplicitCallFlags() & (~ImplicitCall\_None)

) ||

// Make sure we didn't disable exceptions

!scriptContext->GetThreadContext()->IsDisableImplicitException()

);

scriptContext->GetThreadContext()->ClearDisableImplicitFlags();

}

if (exceptionObject->IsPendingExceptionObject())

{

ThreadContext \* threadContext = scriptContext? scriptContext->GetThreadContext() : ThreadContext::GetContextForCurrentThread();

threadContext->SetHasThrownPendingException();

}

throw exceptionObject;

}

void JavascriptExceptionOperators::DispatchExceptionToDebugger(Js::JavascriptExceptionObject \* exceptionObject, ScriptContext\* scriptContext)

{

Assert(exceptionObject != NULL);

Assert(scriptContext != NULL);

if (scriptContext->IsInDebugMode()

&& scriptContext->GetDebugContext()->GetProbeContainer()->HasAllowedForException(exceptionObject))

{

InterpreterHaltState haltState(STOP\_EXCEPTIONTHROW, /\*executingFunction\*/nullptr);

haltState.exceptionObject = exceptionObject;

scriptContext->GetDebugContext()->GetProbeContainer()->DispatchExceptionBreakpoint(&haltState);

}

}

void JavascriptExceptionOperators::ThrowExceptionObject(Js::JavascriptExceptionObject \* exceptionObject, ScriptContext\* scriptContext, bool considerPassingToDebugger, PVOID returnAddress, bool resetStack)

{

ThrowExceptionObjectInternal(exceptionObject, scriptContext, true, considerPassingToDebugger, returnAddress, resetStack);

}

// The purpose of RethrowExceptionObject is to determine if ThrowExceptionObjectInternal should fill in the exception context.

//

// We pretty much always want to fill in the exception context when we throw an exception. The only case where we don't want to do it

// is if we are rethrowing and have the JavascriptExceptionObject from the previous throw with its exception context intact. If

// RethrowExceptionObject is passed a JavascriptExceptionObject with the function already there, that implies we have existing

// exception context and shouldn't step on it on the throw.

//

// RethrowExceptionObject is called only for cross-host calls. When throwing across host calls, we stash our internal JavascriptExceptionObject

// in the TLS. When we are throwing on the same thread (e.g. a throw from one frame to another), we can retrieve that stashed JavascriptExceptionObject

// from the TLS and rethrow it with its exception context intact, so we don't want to step on it. In other cases, e.g. when we throw across threads,

// we cannot retrieve the internal JavascriptExceptionObject from the TLS and have to create a new one. In this case, we need to fill the exception context.

//

void JavascriptExceptionOperators::RethrowExceptionObject(Js::JavascriptExceptionObject \* exceptionObject, ScriptContext\* scriptContext, bool considerPassingToDebugger)

{

ThrowExceptionObjectInternal(exceptionObject, scriptContext, ! exceptionObject->GetFunction(), considerPassingToDebugger, /\*returnAddress=\*/ nullptr, /\*resetStack=\*/ false);

}

// Trim the stack trace down to the amount specified for Error.stackTraceLimit. This happens when we do a full crawl for WER, but we only want to store the specified amount in the error object for consistency.

JavascriptExceptionContext::StackTrace\* JavascriptExceptionOperators::TrimStackTraceForThrownObject(JavascriptExceptionContext::StackTrace\* stackTraceIn, Var thrownObject, ScriptContext& scriptContext)

{

Assert(CrawlStackForWER(scriptContext)); // Don't trim if crawl for Error.stack

Assert(stackTraceIn);

int stackTraceLimit = static\_cast<int>(GetStackTraceLimit(thrownObject, &scriptContext));

Assert(stackTraceLimit == 0 || IsErrorInstance(thrownObject));

if (stackTraceIn->Count() <= stackTraceLimit)

{

return stackTraceIn;

}

JavascriptExceptionContext::StackTrace\* stackTraceTrimmed = NULL;

if (stackTraceLimit > 0)

{

HRESULT hr;

BEGIN\_TRANSLATE\_EXCEPTION\_AND\_ERROROBJECT\_TO\_HRESULT\_NESTED

{

stackTraceTrimmed = RecyclerNew(scriptContext.GetRecycler(), JavascriptExceptionContext::StackTrace, scriptContext.GetRecycler());

for (int i = 0; i < stackTraceLimit; i++)

{

stackTraceTrimmed->Add(stackTraceIn->Item(i));

}

}

END\_TRANSLATE\_EXCEPTION\_AND\_ERROROBJECT\_TO\_HRESULT\_INSCRIPT(hr);

}

// ignore OOM and just return what we can get

return stackTraceTrimmed;

}

//

// Check if thrownObject is instanceof Error (but not an Error prototype).

//

bool JavascriptExceptionOperators::IsErrorInstance(Var thrownObject)

{

if (thrownObject && JavascriptError::Is(thrownObject))

{

return !JavascriptError::FromVar(thrownObject)->IsPrototype();

}

if (thrownObject && RecyclableObject::Is(thrownObject))

{

RecyclableObject\* obj = RecyclableObject::FromVar(thrownObject);

while (true)

{

obj = JavascriptOperators::GetPrototype(obj);

if (JavascriptOperators::GetTypeId(obj) == TypeIds\_Null)

{

break;

}

if (JavascriptError::Is(obj))

{

return true;

}

}

}

return false;

}

void JavascriptExceptionOperators::AddStackTraceToObject(Var targetObject, JavascriptExceptionContext::StackTrace\* stackTrace, ScriptContext& scriptContext, bool isThrownException, bool resetStack)

{

if (!stackTrace || stackTrace->Count() == 0 || !scriptContext.GetConfig()->IsErrorStackTraceEnabled())

{

return;

}

if (isThrownException && CrawlStackForWER(scriptContext)) // Trim stack trace for WER

{

stackTrace = TrimStackTraceForThrownObject(stackTrace, targetObject, scriptContext);

if (!stackTrace)

{

return;

}

}

// If we still have stack trace to store and obj is a thrown exception object, obj must be an Error instance.

Assert(!isThrownException || IsErrorInstance(targetObject));

RecyclableObject\* obj = RecyclableObject::FromVar(targetObject);

if (!resetStack && obj->HasProperty(PropertyIds::stack))

{

return; // we don't want to overwrite an existing "stack" property

}

JavascriptFunction\* accessor = scriptContext.GetLibrary()->GetStackTraceAccessorFunction();

PropertyDescriptor stackPropertyDescriptor;

stackPropertyDescriptor.SetSetter(accessor);

stackPropertyDescriptor.SetGetter(accessor);

stackPropertyDescriptor.SetConfigurable(true);

stackPropertyDescriptor.SetEnumerable(false);

HRESULT hr;

BEGIN\_TRANSLATE\_EXCEPTION\_AND\_ERROROBJECT\_TO\_HRESULT\_NESTED

{

if (JavascriptOperators::DefineOwnPropertyDescriptor(obj, PropertyIds::stack, stackPropertyDescriptor, false, &scriptContext))

{

obj->SetInternalProperty(InternalPropertyIds::StackTrace, stackTrace, PropertyOperationFlags::PropertyOperation\_None, NULL);

obj->SetInternalProperty(InternalPropertyIds::StackTraceCache, NULL, PropertyOperationFlags::PropertyOperation\_None, NULL);

}

}

END\_TRANSLATE\_EXCEPTION\_AND\_ERROROBJECT\_TO\_HRESULT\_INSCRIPT(hr)

}

Var JavascriptExceptionOperators::OP\_RuntimeTypeError(MessageId messageId, ScriptContext \*scriptContext)

{

JavascriptError::ThrowTypeError(scriptContext, MAKE\_HR(messageId));

}

Var JavascriptExceptionOperators::OP\_RuntimeRangeError(MessageId messageId, ScriptContext \*scriptContext)

{

JavascriptError::ThrowRangeError(scriptContext, MAKE\_HR(messageId));

}

Var JavascriptExceptionOperators::OP\_RuntimeReferenceError(MessageId messageId, ScriptContext \*scriptContext)

{

JavascriptError::ThrowReferenceError(scriptContext, MAKE\_HR(messageId));

}

Var JavascriptExceptionOperators::ThrowTypeErrorAccessor(RecyclableObject\* function, CallInfo callInfo, ...)

{

JavascriptError::ThrowTypeError(function->GetScriptContext(), VBSERR\_ActionNotSupported);

}

// Throw type error on access caller when in a restricted context

Var JavascriptExceptionOperators::ThrowTypeErrorCallerAccessor(RecyclableObject\* function, CallInfo callInfo, ...)

{

JavascriptError::ThrowTypeError(function->GetScriptContext(), JSERR\_AccessCallerRestricted);

}

// Throw type error on access on callee when strict mode

Var JavascriptExceptionOperators::ThrowTypeErrorCalleeAccessor(RecyclableObject\* function, CallInfo callInfo, ...)

{

JavascriptError::ThrowTypeError(function->GetScriptContext(), JSERR\_AccessCallee);

}

// Throw type error on access arguments when in a restricted context

Var JavascriptExceptionOperators::ThrowTypeErrorArgumentsAccessor(RecyclableObject\* function, CallInfo callInfo, ...)

{

JavascriptError::ThrowTypeError(function->GetScriptContext(), JSERR\_AccessArgumentsRestricted);

}

Var JavascriptExceptionOperators::StackTraceAccessor(RecyclableObject\* function, CallInfo callInfo, ...)

{

ARGUMENTS(args, callInfo);

AssertMsg(args.Info.Count > 0, "Should always have implicit 'this'");

ScriptContext \*scriptContext = function->GetScriptContext();

AnalysisAssert(scriptContext);

// If the first argument to the accessor is not a recyclable object, return undefined

// for compat with other browsers

if (!RecyclableObject::Is(args[0]))

{

return scriptContext->GetLibrary()->GetUndefined();

}

RecyclableObject \*obj = RecyclableObject::FromVar(args[0]);

// If an argument was passed to the accessor, it is being called as a setter.

// Set the internal StackTraceCache property accordingly.

if (args.Info.Count > 1)

{

obj->SetInternalProperty(InternalPropertyIds::StackTraceCache, args[1], PropertyOperationFlags::PropertyOperation\_None, NULL);

if (JavascriptError::Is(obj))

{

((JavascriptError \*)obj)->SetStackPropertyRedefined(true);

}

return scriptContext->GetLibrary()->GetEmptyString();

}

// Otherwise, the accessor is being called as a getter.

// Return existing cached value, or obtain the string representation of the StackTrace to return.

Var cache = NULL;

if (obj->GetInternalProperty(obj,InternalPropertyIds::StackTraceCache, (Var\*)&cache, NULL, scriptContext) && cache)

{

return cache;

}

JavascriptString\* stringMessage = scriptContext->GetLibrary()->GetEmptyString();

HRESULT hr;

BEGIN\_TRANSLATE\_EXCEPTION\_AND\_ERROROBJECT\_TO\_HRESULT\_NESTED

{

Js::JavascriptExceptionContext::StackTrace \*stackTrace = NULL;

if (!obj->GetInternalProperty(obj,InternalPropertyIds::StackTrace, (Js::Var\*) &stackTrace, NULL, scriptContext) ||

stackTrace == nullptr)

{

obj->SetInternalProperty(InternalPropertyIds::StackTraceCache, stringMessage, PropertyOperationFlags::PropertyOperation\_None, NULL);

return stringMessage;

}

if (IsErrorInstance(obj))

{

stringMessage = JavascriptConversion::ToString(obj, scriptContext);

}

CompoundString \*const stringBuilder = CompoundString::NewWithCharCapacity(40, scriptContext->GetLibrary());

stringBuilder->AppendChars(stringMessage);

for (int i = 0; i < stackTrace->Count(); i++)

{

Js::JavascriptExceptionContext::StackFrame currentFrame = stackTrace->Item(i);

// Defend in depth. Discard cross domain frames if somehow they creped in.

if (currentFrame.IsScriptFunction())

{

ScriptContext\* funcScriptContext = currentFrame.GetFunctionBody()->GetScriptContext();

AnalysisAssert(funcScriptContext);

if (scriptContext != funcScriptContext && FAILED(scriptContext->GetHostScriptContext()->CheckCrossDomainScriptContext(funcScriptContext)))

{

continue; // Ignore this frame

}

}

FunctionBody\* functionBody = currentFrame.GetFunctionBody();

const bool isLibraryCode = !functionBody || functionBody->GetUtf8SourceInfo()->GetIsLibraryCode();

if (isLibraryCode)

{

AppendLibraryFrameToStackTrace(stringBuilder, currentFrame.GetFunctionName());

}

else

{

LPCWSTR pUrl = NULL;

ULONG lineNumber = 0;

LONG characterPosition = 0;

functionBody->GetLineCharOffset(currentFrame.GetByteCodeOffset(), &lineNumber, &characterPosition);

pUrl = functionBody->GetSourceName();

LPCWSTR functionName = nullptr;

if (CONFIG\_FLAG(ExtendedErrorStackForTestHost))

{

BEGIN\_LEAVE\_SCRIPT\_INTERNAL(scriptContext)

{

if (currentFrame.GetFunctionNameWithArguments(&functionName) != S\_OK)

{

functionName = functionBody->GetExternalDisplayName();

}

}

END\_LEAVE\_SCRIPT\_INTERNAL(scriptContext)

}

else

{

functionName = functionBody->GetExternalDisplayName();

}

AppendExternalFrameToStackTrace(stringBuilder, functionName, pUrl ? pUrl : L"", lineNumber + 1, characterPosition + 1);

}

}

// Try to create the string object even if we did OOM, but if can't, just return what we've got. We catch and ignore OOM so it doesn�t propagate up.

// With all the stack trace functionality, we do best effort to produce the stack trace in the case of OOM, but don�t want it to trigger an OOM. Idea is if do take

// an OOM, have some chance of producing a stack trace to see where it happened.

stringMessage = stringBuilder;

}

END\_TRANSLATE\_EXCEPTION\_AND\_ERROROBJECT\_TO\_HRESULT\_INSCRIPT(hr);

obj->SetInternalProperty(InternalPropertyIds::StackTraceCache, stringMessage, PropertyOperationFlags::PropertyOperation\_None, NULL);

return stringMessage;

}

uint64 JavascriptExceptionOperators::GetStackTraceLimit(Var thrownObject, ScriptContext\* scriptContext)

{

uint64 limit = 0;

if (scriptContext->GetConfig()->IsErrorStackTraceEnabled()

&& IsErrorInstance(thrownObject))

{

HRESULT hr = JavascriptError::GetRuntimeError(RecyclableObject::FromVar(thrownObject), NULL);

JavascriptFunction\* error = scriptContext->GetLibrary()->GetErrorConstructor();

// If we are throwing StackOverflow and Error.stackTraceLimit is a custom getter, we can't make the getter

// call as we don't have stack space. Just bail out without stack trace in such case. Only proceed to get

// Error.stackTraceLimit property if we are not throwing StackOverflow, or there is no implicitCall (in getter case).

DisableImplicitFlags disableImplicitFlags = scriptContext->GetThreadContext()->GetDisableImplicitFlags();

if (hr == VBSERR\_OutOfStack)

{

scriptContext->GetThreadContext()->SetDisableImplicitFlags(DisableImplicitCallAndExceptionFlag);

}

Var var;

if (JavascriptOperators::GetProperty(error, PropertyIds::stackTraceLimit, &var, scriptContext))

{

// Only accept the value if it is a "Number". Avoid potential valueOf() call.

switch (JavascriptOperators::GetTypeId(var))

{

case TypeIds\_Integer:

case TypeIds\_Number:

case TypeIds\_Int64Number:

case TypeIds\_UInt64Number:

double value = JavascriptConversion::ToNumber(var, scriptContext);

limit = JavascriptNumber::IsNan(value) ? 0 :

(NumberUtilities::IsFinite(value) ? JavascriptConversion::ToUInt32(var, scriptContext) : MaxStackTraceLimit);

break;

}

}

if (hr == VBSERR\_OutOfStack)

{

scriptContext->GetThreadContext()->SetDisableImplicitFlags(disableImplicitFlags);

}

}

return limit;

}

void JavascriptExceptionOperators::AppendExternalFrameToStackTrace(CompoundString\* bs, LPCWSTR functionName, LPCWSTR fileName, ULONG lineNumber, LONG characterPosition)

{

// format is equivalent to printf("\n at %s (%s:%d:%d)", functionName, filename, lineNumber, characterPosition);

const CharCount maxULongStringLength = 10; // excluding null terminator

const auto ConvertULongToString = [](const ULONG value, wchar\_t \*const buffer, const CharCount charCapacity)

{

const errno\_t err = \_ultow\_s(value, buffer, charCapacity, 10);

Assert(err == 0);

};

if (CONFIG\_FLAG(ExtendedErrorStackForTestHost))

{

bs->AppendChars(L"\n\tat ");

}

else

{

bs->AppendChars(L"\n at ");

}

bs->AppendCharsSz(functionName);

bs->AppendChars(L" (");

if (CONFIG\_FLAG(ExtendedErrorStackForTestHost) && \*fileName != L'\0')

{

wchar\_t shortfilename[\_MAX\_FNAME];

wchar\_t ext[\_MAX\_EXT];

errno\_t err = \_wsplitpath\_s(fileName, NULL, 0, NULL, 0, shortfilename, \_MAX\_FNAME, ext, \_MAX\_EXT);

if (err != 0)

{

bs->AppendCharsSz(fileName);

}

else

{

bs->AppendCharsSz(shortfilename);

bs->AppendCharsSz(ext);

}

}

else

{

bs->AppendCharsSz(fileName);

}

bs->AppendChars(L':');

bs->AppendChars(lineNumber, maxULongStringLength, ConvertULongToString);

bs->AppendChars(L':');

bs->AppendChars(characterPosition, maxULongStringLength, ConvertULongToString);

bs->AppendChars(L')');

}

void JavascriptExceptionOperators::AppendLibraryFrameToStackTrace(CompoundString\* bs, LPCWSTR functionName)

{

// format is equivalent to printf("\n at %s (native code)", functionName);

bs->AppendChars(L"\n at ");

bs->AppendCharsSz(functionName);

bs->AppendChars(L" (native code)");

}

} // namespace Js