#include "RuntimeBasePch.h"

namespace Js

{

const ushort CallInfo::ksizeofCount = 24;

const ushort CallInfo::ksizeofCallFlags = 8;

const uint CallInfo::kMaxCountArgs = (1 << ksizeofCount) - 1 ;

}

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

// 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

{

enum CallFlags : unsigned

{

CallFlags\_None = 0,

CallFlags\_New = 1,

CallFlags\_Value = 2,

CallFlags\_Eval = 4,

CallFlags\_ExtraArg = 8,

CallFlags\_NotUsed = 0x10,

CallFlags\_Wrapped = 0x20,

CallFlags\_NewTarget = 0x40,

CallFlags\_InternalFrame = 0x80

};

struct CallInfo

{

CallInfo() {}

/\*

\* Removed the copy constructor because it forced the 64 bit compiler

\* to pass this object by reference. Interpreter stack setup code expects

\* CallInfo to be passed by value.

\*/

CallInfo(ushort count)

: Flags(CallFlags\_None)

, Count(count)

#ifdef \_WIN64

, unused(0)

#endif

{

}

CallInfo(CallFlags flags, ushort count)

: Flags(flags)

, Count(count)

#ifdef \_WIN64

, unused(0)

#endif

{

}

CallInfo(VirtualTableInfoCtorEnum v)

{

}

// Assumes big-endian layout

// If the size of the count is changed, change should happen at following places also

// - scriptdirect.idl

// - LowererMDArch::LoadInputParamCount

//

unsigned Count : 24;

CallFlags Flags : 8;

#ifdef \_WIN64

unsigned unused : 32;

#endif

public:

static const ushort ksizeofCount;

static const ushort ksizeofCallFlags;

static const uint kMaxCountArgs;

};

struct InlineeCallInfo

{

// Assumes big-endian layout.

size\_t Count: 4;

size\_t InlineeStartOffset: sizeof(void\*) \* CHAR\_BIT - 4;

static size\_t const MaxInlineeArgoutCount = 0xF;

static bool Encode(Js::Var &callInfo, size\_t count, size\_t offset)

{

const size\_t offsetMask = (~(size\_t)0) >> 4;

const size\_t countMask = 0x0000000F;

if (count != (count & countMask))

{

return false;

}

if (offset != (offset & offsetMask))

{

return false;

}

callInfo = (Js::Var)((offset << 4) | count);

return true;

}

void Clear()

{

this->Count = 0;

this->InlineeStartOffset = 0;

}

};

}

<?xml version="1.0" encoding="utf-8"?>

<Project DefaultTargets="Build" ToolsVersion="12.0" xmlns="http://schemas.microsoft.com/developer/msbuild/2003">

<Import Condition="'$(ChakraBuildPathImported)'!='true'" Project="$(SolutionDir)Chakra.Build.Paths.props" />

<Import Project="$(BuildConfigPropsPath)Chakra.Build.ProjectConfiguration.props" />

<PropertyGroup Label="Globals">

<TargetName>Chakra.Runtime.Base</TargetName>

<ProjectGuid>{706083F7-6AA4-4558-A153-6352EF9110EE}</ProjectGuid>

<RootNamespace>JS</RootNamespace>

<Keyword>Win32Proj</Keyword>

</PropertyGroup>

<PropertyGroup Label="Configuration">

<ConfigurationType>StaticLibrary</ConfigurationType>

</PropertyGroup>

<Import Project="$(BuildConfigPropsPath)Chakra.Build.Default.props" />

<Import Project="$(VCTargetsPath)\Microsoft.Cpp.Default.props" />

<Import Project="$(VCTargetsPath)\Microsoft.Cpp.props" />

<Import Project="$(BuildConfigPropsPath)Chakra.Build.props" />

<PropertyGroup>

<\_ProjectFileVersion>10.0.30319.1</\_ProjectFileVersion>

</PropertyGroup>

<ItemDefinitionGroup>

<ClCompile>

<AdditionalIncludeDirectories>

$(MSBuildThisFileDirectory)..;

$(MSBuildThisFileDirectory)..\..\Common;

$(MSBuildThisFileDirectory)..\..\Parser;

$(MSBuildThisFileDirectory)..\..\Backend;

%(AdditionalIncludeDirectories)

</AdditionalIncludeDirectories>

<PrecompiledHeader>Use</PrecompiledHeader>

<PrecompiledHeaderFile>RuntimeBasePch.h</PrecompiledHeaderFile>

</ClCompile>

</ItemDefinitionGroup>

<ItemGroup>

<ClCompile Include="$(MSBuildThisFileDirectory)CallInfo.cpp" />

<ClCompile Include="$(MSBuildThisFileDirectory)CharStringCache.cpp" />

<ClCompile Include="$(MSBuildThisFileDirectory)Constants.cpp" />

<ClCompile Include="$(MSBuildThisFileDirectory)CrossSite.cpp" />

<ClCompile Include="$(MSBuildThisFileDirectory)Debug.cpp" />

<ClCompile Include="$(MSBuildThisFileDirectory)DelayLoadLibrary.cpp" />

<ClCompile Include="$(MSBuildThisFileDirectory)Entropy.cpp" />

<ClCompile Include="$(MSBuildThisFileDirectory)EtwTrace.cpp" />

<ClCompile Include="$(MSBuildThisFileDirectory)Exception.cpp" />

<ClCompile Include="$(MSBuildThisFileDirectory)ExpirableObject.cpp" />

<ClCompile Include="$(MSBuildThisFileDirectory)FunctionBody.cpp" />

<ClCompile Include="$(MSBuildThisFileDirectory)FunctionInfo.cpp" />

<ClCompile Include="$(MSBuildThisFileDirectory)HiResTimer.cpp" />

<ClCompile Include="$(MSBuildThisFileDirectory)LeaveScriptObject.cpp" />

<ClCompile Include="$(MSBuildThisFileDirectory)PerfHint.cpp" />

<ClCompile Include="$(MSBuildThisFileDirectory)PropertyRecord.cpp" />

<ClCompile Include="$(MSBuildThisFileDirectory)ScriptContext.cpp" />

<ClCompile Include="$(MSBuildThisFileDirectory)ScriptContextProfiler.cpp" />

<ClCompile Include="$(MSBuildThisFileDirectory)ScriptContextOptimizationOverrideInfo.cpp" />

<ClCompile Include="$(MSBuildThisFileDirectory)SourceHolder.cpp" />

<ClCompile Include="$(MSBuildThisFileDirectory)ScriptMemoryDumper.cpp" />

<ClCompile Include="$(MSBuildThisFileDirectory)StackProber.cpp" />

<ClCompile Include="$(MSBuildThisFileDirectory)TestEtwEventSink.cpp" />

<ClCompile Include="$(MSBuildThisFileDirectory)TempArenaAllocatorObject.cpp" />

<ClCompile Include="$(MSBuildThisFileDirectory)ThreadBoundThreadContextManager.cpp" />

<ClCompile Include="$(MSBuildThisFileDirectory)ThreadContext.cpp" />

<ClCompile Include="$(MSBuildThisFileDirectory)ThreadContextTLSEntry.cpp" />

<ClCompile Include="$(MSBuildThisFileDirectory)ThreadServiceWrapperBase.cpp" />

<ClCompile Include="$(MSBuildThisFileDirectory)Utf8SourceInfo.cpp" />

<ClCompile Include="$(MSBuildThisFileDirectory)WindowsFoundationAdapter.cpp" />

<ClCompile Include="$(MSBuildThisFileDirectory)WindowsGlobalizationAdapter.cpp" />

<ClCompile Include="$(MSBuildThisFileDirectory)RuntimeBasePch.cpp">

<PrecompiledHeader>Create</PrecompiledHeader>

</ClCompile>

</ItemGroup>

<ItemGroup>

<ClInclude Include="RuntimeBasePch.h" />

<ClInclude Include="CallInfo.h" />

<ClInclude Include="CharStringCache.h" />

<ClInclude Include="Constants.h" />

<ClInclude Include="CrossSite.h" />

<ClInclude Include="CrossSiteEnumerator.h" />

<ClInclude Include="CrossSiteObject.h" />

<ClInclude Include="Debug.h" />

<ClInclude Include="DelayLoadLibrary.h" />

<ClInclude Include="Entropy.h" />

<ClInclude Include="EtwTrace.h" />

<ClInclude Include="Exception.h" />

<ClInclude Include="ExpirableObject.h" />

<ClInclude Include="FunctionBody.h" />

<ClInclude Include="FunctionInfo.h" />

<ClInclude Include="HiResTimer.h" />

<ClInclude Include="JnDirectFields.h" />

<ClInclude Include="LeaveScriptObject.h" />

<ClInclude Include="PerfHint.h" />

<ClInclude Include="PerfHintDescriptions.h" />

<ClInclude Include="PropertyRecord.h" />

<ClInclude Include="RegexPatternMruMap.h" />

<ClInclude Include="ScriptContext.h" />

<ClInclude Include="ScriptContextBase.h" />

<ClInclude Include="ScriptContextOptimizationOverrideInfo.h" />

<ClInclude Include="ScriptContextProfiler.h" />

<ClInclude Include="ScriptMemoryDumper.h" />

<ClInclude Include="SourceHolder.h" />

<ClInclude Include="StackProber.h" />

<ClInclude Include="TempArenaAllocatorObject.h" />

<ClInclude Include="TestEtwEventSink.h" />

<ClInclude Include="ThreadBoundThreadContextManager.h" />

<ClInclude Include="ThreadContext.h" />

<ClInclude Include="ThreadContextTLSEntry.h" />

<ClInclude Include="ThreadServiceWrapper.h" />

<ClInclude Include="ThreadServiceWrapperBase.h" />

<ClInclude Include="Utf8SourceInfo.h" />

<ClInclude Include="WindowsFoundationAdapter.h" />

<ClInclude Include="WindowsGlobalizationAdapter.h" />

<None Include="ScriptContext.inl" />

</ItemGroup>

<Import Project="$(BuildConfigPropsPath)Chakra.Build.targets" Condition="exists('$(BuildConfigPropsPath)Chakra.Build.targets')"/>

<Import Project="$(VCTargetsPath)\Microsoft.Cpp.targets" />

</Project>

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

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

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

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

#include "RuntimeBasePch.h"

#include "Library\ProfileString.h"

#include "Library\SingleCharString.h"

namespace Js

{

CharStringCache::CharStringCache() : charStringCache(nullptr)

{

memset(charStringCacheA, 0, sizeof charStringCacheA);

}

JavascriptString\* CharStringCache::GetStringForCharA(char c)

{

AssertMsg(JavascriptString::IsASCII7BitChar(c), "GetStringForCharA must be called with ASCII 7bit chars only");

PropertyString \* str = charStringCacheA[c];

if (str == nullptr)

{

PropertyRecord const \* propertyRecord;

wchar\_t wc = c;

JavascriptLibrary \* javascriptLibrary = JavascriptLibrary::FromCharStringCache(this);

javascriptLibrary->GetScriptContext()->GetOrAddPropertyRecord(&wc, 1, &propertyRecord);

str = javascriptLibrary->CreatePropertyString(propertyRecord);

charStringCacheA[c] = str;

}

return str;

}

JavascriptString\* CharStringCache::GetStringForChar(wchar\_t c)

{

#ifdef PROFILE\_STRINGS

StringProfiler::RecordSingleCharStringRequest(JavascriptLibrary::FromCharStringCache(this)->GetScriptContext());

#endif

if (JavascriptString::IsASCII7BitChar(c))

{

return GetStringForCharA(JavascriptString::ToASCII7BitChar(c));

}

return GetStringForCharW(c);

}

JavascriptString\* CharStringCache::GetStringForCharW(wchar\_t c)

{

Assert(!JavascriptString::IsASCII7BitChar(c));

JavascriptString\* str;

ScriptContext \* scriptContext = JavascriptLibrary::FromCharStringCache(this)->GetScriptContext();

if (!scriptContext->IsClosed())

{

if (charStringCache == nullptr)

{

Recycler \* recycler = scriptContext->GetRecycler();

charStringCache = RecyclerNew(recycler, CharStringCacheMap, recycler, 17);

}

if (!charStringCache->TryGetValue(c, &str))

{

str = SingleCharString::New(c, scriptContext);

charStringCache->Add(c, str);

}

}

else

{

str = SingleCharString::New(c, scriptContext);

}

return str;

}

JavascriptString\* CharStringCache::GetStringForCharSP(codepoint\_t c)

{

Assert(c >= 0x10000);

CompileAssert(sizeof(wchar\_t) \* 2 == sizeof(codepoint\_t));

wchar\_t buffer[2];

Js::NumberUtilities::CodePointAsSurrogatePair(c, buffer, buffer + 1);

JavascriptString\* str = JavascriptString::NewCopyBuffer(buffer, 2, JavascriptLibrary::FromCharStringCache(this)->GetScriptContext());

// TODO: perhaps do some sort of cache for supplementary characters

return str;

}

};

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

// 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 CharStringCache

{

public:

CharStringCache();

JavascriptString\* GetStringForCharA(char c); // ASCII 7-bit

JavascriptString\* GetStringForCharW(wchar\_t c); // Unicode

JavascriptString\* GetStringForChar(wchar\_t c); // Either

JavascriptString\* GetStringForCharSP(codepoint\_t c); // Supplementary char

// For JIT

static const wchar\_t CharStringCacheSize = 0x80; /\*range of ASCII 7-bit chars\*/

static DWORD GetCharStringCacheAOffset() { return offsetof(CharStringCache, charStringCacheA); }

const PropertyString \* const \* GetCharStringCacheA() const { return charStringCacheA; }

static JavascriptString\* GetStringForChar(CharStringCache \*charStringCache, wchar\_t c) { return charStringCache->GetStringForChar(c); }

static JavascriptString\* GetStringForCharCodePoint(CharStringCache \*charStringCache, codepoint\_t c)

{

return (c >= 0x10000 ? charStringCache->GetStringForCharSP(c) : charStringCache->GetStringForCharW((wchar\_t)c));

}

private:

PropertyString \* charStringCacheA[CharStringCacheSize];

typedef JsUtil::BaseDictionary<wchar\_t, JavascriptString \*, Recycler, PowerOf2SizePolicy> CharStringCacheMap;

CharStringCacheMap \* charStringCache;

friend class CharStringCacheValidator;

};

class CharStringCacheValidator

{

// Lower assert that charStringCacheA is at the 0 offset

CompileAssert(offsetof(CharStringCache, charStringCacheA) == 0);

};

};

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

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

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

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

#include "RuntimeBasePch.h"

using namespace Js;

const wchar\_t Constants::AnonymousFunction[] = L"Anonymous function";

const wchar\_t Constants::Anonymous[] = L"anonymous";

const wchar\_t Constants::Empty[] = L"";

const wchar\_t Constants::FunctionCode[] = L"Function code";

const wchar\_t Constants::GlobalCode[] = L"Global code";

const wchar\_t Constants::EvalCode[] = L"eval code";

const wchar\_t Constants::GlobalFunction[] = L"glo";

const wchar\_t Constants::UnknownScriptCode[] = L"Unknown script code";

#ifdef \_M\_AMD64

const PBYTE Constants::StackLimitForScriptInterrupt = (PBYTE)0x7fffffffffffffff;

#else

const PBYTE Constants::StackLimitForScriptInterrupt = (PBYTE)0x7fffffff;

#endif

#pragma warning(push)

#pragma warning(disable:4815) // Allow no storage for zero-sized array at end of NullFrameDisplay struct.

const Js::FrameDisplay Js::NullFrameDisplay = 0;

const Js::FrameDisplay Js::StrictNullFrameDisplay = FrameDisplay(0, true);

#pragma warning(pop)

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

// 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\_X64\_OR\_ARM64

# define WIN64\_STACK\_FACTOR 3

#else

# define WIN64\_STACK\_FACTOR 1

#endif

namespace Js

{

class Constants

{

public:

// Id of the NoContextSourceContextInfo

static const uint NoSourceContext = (uint)-1;

// Variable indicating no source context cookie was passed in by the host- indicates Dynamic Script

static const DWORD\_PTR NoHostSourceContext = (DWORD\_PTR)-1;

static const DWORD\_PTR FunctionBodyUnavailable = (DWORD\_PTR)NULL; // Not a valid Var

static const LocalFunctionId NoFunctionId = (LocalFunctionId)-1;

static const uint NoStatementIndex = (uint)-1;

static const uint NoByteCodeOffset = (uint)-1;

static const PropertyId NoProperty = -1;

static const RegSlot NoRegister = (RegSlot)-1;

static const RegSlot OneByteRegister = (RegSlot\_OneByte)-1;

static const int32 InvalidOffset = -1;

static const unsigned int PropertyGroupSize = 512;

static const unsigned int PropertyGroupMask = PropertyGroupSize-1;

static const unsigned int PropertyGroupShift = 9;

static const unsigned int MaxPropertyGroups = 128;

static const PropertyIndex FieldsChunkSize = 4;

static const PropertyIndex NoSlot = (PropertyIndex)-1;

static const PropertyIndex PropertyIndexMax = 0xFFFE;

static const BigPropertyIndex NoBigSlot = (BigPropertyIndex)-1;

static const int IntMaxValue = 2147483647;

static const int Int31MinValue = -1 << 30;

static const int Int31MaxValue = ~Int31MinValue;

static const unsigned int UShortMaxValue = 0xFFFF;

static const uint InvalidSourceIndex = (uint)-1;

static const ProfileId NoProfileId = (ProfileId)-1;

static const int InvalidByteCodeOffset = -1;

static const InlineCacheIndex NoInlineCacheIndex = (InlineCacheIndex)-1;

static const uint UninitializedValue = (uint)-1;

static const ArgSlot InvalidArgSlot = (ArgSlot)-1;

static const uint32 InvalidSymID = (uint32)-1;

static const int ReservedTypeIds = 2048;

// 1 MEM\_RESERVE page at the bottom of the stack

// 1 PAGE\_GUARD | PAGE\_READWRITE page that serves as the guard page

static const unsigned NumGuardPages = 2;

// We need to keep some buffer to run our stack overflow

// handling code which throws an exception

static const unsigned StackOverflowHandlingBufferPages = 10;

// Minimum stack required to JIT-compile a function

static const unsigned MinStackJITCompile = 16 \* 0x400 \* WIN64\_STACK\_FACTOR; // 16 KB

// Maximum number of arguments allowed on an inlinee function for constant argument inlining

// It is set to 13 to ensure optimum size of callSiteInfo

static const unsigned MaximumArgumentCountForConstantArgumentInlining = 13;

//Invalid loop unrolling factor used for memory operations (memset/ memcopy)

static const byte InvalidLoopUnrollFactor = 0x7F;

// Minimum stack required to be able to execute a JITted Javascript function.

static const unsigned MinStackJIT = 0x930 \* WIN64\_STACK\_FACTOR;

#if (defined(\_M\_ARM32\_OR\_ARM64) || defined(\_M\_AMD64))

#if DBG

static const unsigned MinStackInterpreter = 0x4000; // 16 KB

#else

// Minimum stack required to be able to enter the interpreter.

static const unsigned MinStackInterpreter = 0x2000; // 8 KB

#endif

#else

#if DBG

static const unsigned MinStackInterpreter = 0x1000; // 4 KB

#else

// Minimum stack required to be able to enter the interpreter.

static const unsigned MinStackInterpreter = 0xC00; // 3 KB

#endif

#endif

// Minimum stack required by the byte code compiler.

static const unsigned MinStackCompile = 0x400 \* WIN64\_STACK\_FACTOR; // 1 KB

static const unsigned MinStackByteCodeVisitor = 0xC00 \* WIN64\_STACK\_FACTOR; // 3 KB

static const unsigned MinStackRegex = 8 \* 0x400 \* WIN64\_STACK\_FACTOR; // 8 KB

static const unsigned MinStackRuntime = 32 \* 0x0400 \* WIN64\_STACK\_FACTOR;

static const unsigned MinStackHost = 40 \* 0x0400 \* WIN64\_STACK\_FACTOR;

static const unsigned MinStackCallout = 32 \* 0x0400 \* WIN64\_STACK\_FACTOR;

static const unsigned MinStackParseOneTerm = 0x100 \* WIN64\_STACK\_FACTOR;

// MinStackDefault is the minimum amount of stack space required to

// call RaiseException to report a stack overflow.

static const unsigned MinStackDefault = 1 \* 0x0400 \* WIN64\_STACK\_FACTOR;

static const unsigned ExtraStack = 2 \* 0x0400 \* WIN64\_STACK\_FACTOR;

#if \_M\_IX86\_OR\_ARM32

static const unsigned MaxThreadJITCodeHeapSize = 28 \* 1024 \* 1024;

static const unsigned MaxProcessJITCodeHeapSize = 55 \* 1024 \* 1024;

#elif \_M\_X64\_OR\_ARM64

// larger address space means we can keep this higher on 64 bit architectures

static const unsigned MaxThreadJITCodeHeapSize = 800 \* 1024 \* 1024;

static const unsigned MaxProcessJITCodeHeapSize = 1024 \* 1024 \* 1024;

#endif

#if !\_M\_X64\_OR\_ARM64 && \_CONTROL\_FLOW\_GUARD

static const unsigned MaxThreadContextsWithPreReserveSegment = 6;

#endif

static const PBYTE StackLimitForScriptInterrupt;

// Arguments object created on the fly is 1 slot before the frame

static const int ArgumentLocationOnFrame = 1;

static const int StackNestedFuncList = 2;

static const int StackFrameDisplay = 3;

static const int StackScopeSlots = 4;

#if \_M\_IX86 || \_M\_AMD64

static const int StackNestedFuncListWithNoArg = 1;

static const int StackFrameDisplayWithNoArg = 2;

static const int StackScopeSlotsWithNoArg = 3;

#endif

static const DWORD NonWebWorkerContextId = 0;

// Inlinee constants shared between the backend and the stack walker.

static const uint InlineeMetaArgIndex\_Argc = 0;

static const uint InlineeMetaArgIndex\_FunctionObject = 1;

static const uint InlineeMetaArgIndex\_ArgumentsObject = 2;

static const uint InlineeMetaArgIndex\_Argv = 3;

static const uint InlineeMetaArgCount = 3;

static const wchar\_t AnonymousFunction[];

static const wchar\_t AnonymousCode[];

static const wchar\_t Anonymous[]; // Used in the function created due to new Function

static const wchar\_t Empty[];

static const wchar\_t FunctionCode[];

static const wchar\_t GlobalCode[];

static const wchar\_t EvalCode[];

static const wchar\_t GlobalFunction[];

static const wchar\_t UnknownScriptCode[];

static const size\_t AnonymousFunctionLength = \_countof(L"Anonymous function") - 1;

static const size\_t AnonymousLength = \_countof(L"anonymous") - 1;

static const size\_t AnonymousClassLength = \_countof(L"Anonymous class") - 1;

static const size\_t FunctionCodeLength = \_countof(L"Function code") - 1;

static const size\_t GlobalFunctionLength = \_countof(L"glo") - 1;

static const size\_t GlobalCodeLength = \_countof(L"Global code") - 1;

static const size\_t EvalCodeLength = \_countof(L"eval code") - 1;

static const size\_t UnknownScriptCodeLength = \_countof(L"Unknown script code") - 1;

};

extern const FrameDisplay NullFrameDisplay;

extern const FrameDisplay StrictNullFrameDisplay;

}

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

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

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

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

#include "RuntimeBasePch.h"

#include "Library\JavascriptProxy.h"

#include "Library\HostObjectBase.h"

#include "Types\WithScopeObject.h"

namespace Js

{

BOOL CrossSite::NeedMarshalVar(Var instance, ScriptContext \* requestContext)

{

if (TaggedNumber::Is(instance))

{

return FALSE;

}

RecyclableObject \* object = RecyclableObject::FromVar(instance);

if (object->GetScriptContext() == requestContext)

{

return FALSE;

}

if (DynamicType::Is(object->GetTypeId()))

{

return !DynamicObject::FromVar(object)->IsCrossSiteObject() && !object->IsExternal();

}

return TRUE;

}

void CrossSite::MarshalDynamicObject(ScriptContext \* scriptContext, DynamicObject \* object)

{

Assert(!object->IsExternal() && !object->IsCrossSiteObject());

object->MarshalToScriptContext(scriptContext);

if (object->GetTypeId() == TypeIds\_Function)

{

AssertMsg(object != object->GetScriptContext()->GetLibrary()->GetDefaultAccessorFunction(), "default accessor marshalled");

JavascriptFunction \* function = JavascriptFunction::FromVar(object);

// See if this function is one that the host needs to handle

HostScriptContext \* hostScriptContext = scriptContext->GetHostScriptContext();

if (!hostScriptContext || !hostScriptContext->SetCrossSiteForFunctionType(function))

{

if (function->GetDynamicType()->GetIsShared())

{

function->GetLibrary()->SetCrossSiteForSharedFunctionType(function);

}

else

{

function->SetEntryPoint(function->GetScriptContext()->CurrentCrossSiteThunk);

}

}

}

}

void CrossSite::MarshalPrototypeChain(ScriptContext\* scriptContext, DynamicObject \* object)

{

RecyclableObject \* prototype = object->GetPrototype();

while (prototype->GetTypeId() != TypeIds\_Null && prototype->GetTypeId() != TypeIds\_HostDispatch)

{

// We should not see any static type or host dispatch here

DynamicObject \* prototypeObject = DynamicObject::FromVar(prototype);

if (prototypeObject->IsCrossSiteObject())

{

break;

}

if (scriptContext != prototypeObject->GetScriptContext() && !prototypeObject->IsExternal())

{

MarshalDynamicObject(scriptContext, prototypeObject);

}

prototype = prototypeObject->GetPrototype();

}

}

void CrossSite::MarshalDynamicObjectAndPrototype(ScriptContext\* scriptContext, DynamicObject \* object)

{

MarshalDynamicObject(scriptContext, object);

MarshalPrototypeChain(scriptContext, object);

}

Var CrossSite::MarshalFrameDisplay(ScriptContext\* scriptContext, FrameDisplay \*display)

{

uint16 length = display->GetLength();

FrameDisplay \*newDisplay =

RecyclerNewPlus(scriptContext->GetRecycler(), length \* sizeof(Var), FrameDisplay, length);

for (uint16 i = 0; i < length; i++)

{

Var value = display->GetItem(i);

if (WithScopeObject::Is(value))

{

// Here we are marshalling the wrappedObject and then ReWrapping th object in the new context.

value = JavascriptOperators::ToWithObject(CrossSite::MarshalVar(scriptContext, WithScopeObject::FromVar(value)->GetWrappedObject()), scriptContext);

}

else

{

value = CrossSite::MarshalVar(scriptContext, value);

}

newDisplay->SetItem(i, value);

}

return (Var)newDisplay;

}

/\*

Enumerators are always created in current script context, and if the underlying object is a cross

site object, we'll marshal the enumerator by changing the vtbl of the base enumerator, such that

we will marshal the return index before it's returned to caller.

Notice that enumerator marshalling is somewhat different from the object marshalling. We have only

one instance of object in cross site usage, but we can create multiple enumerators from different

script context for the same cross site object.

\*/

Var CrossSite::MarshalEnumerator(ScriptContext\* scriptContext, Var value)

{

TypeId typeId = JavascriptOperators::GetTypeId(value);

if (typeId != TypeIds\_Enumerator)

{

AssertMsg(FALSE, "invalid enumerator");

return value;

}

JavascriptEnumerator\* enumerator = JavascriptEnumerator::FromVar(value);

enumerator->MarshalToScriptContext(scriptContext);

return enumerator;

}

// static

\_\_inline Var CrossSite::MarshalVar(ScriptContext\* scriptContext, Var value, bool fRequestWrapper)

{

// value might be null from disable implicit call

if (value == nullptr || Js::TaggedNumber::Is(value))

{

return value;

}

Js::RecyclableObject\* object = RecyclableObject::FromVar(value);

if (fRequestWrapper || scriptContext != object->GetScriptContext())

{

return MarshalVarInner(scriptContext, object, fRequestWrapper);

}

return value;

}

bool CrossSite::DoRequestWrapper(Js::RecyclableObject\* object, bool fRequestWrapper)

{

return fRequestWrapper && JavascriptFunction::Is(object) && JavascriptFunction::FromVar(object)->IsExternalFunction();

}

Var CrossSite::MarshalVarInner(ScriptContext\* scriptContext, \_\_in Js::RecyclableObject\* object, bool fRequestWrapper)

{

if (scriptContext == object->GetScriptContext())

{

if (DoRequestWrapper(object, fRequestWrapper))

{

// If we get here then we need to either wrap in the caller's type system or we need to return undefined.

// VBScript will pass in the scriptContext (requestContext) from the JavascriptDispatch and this will be the

// same as the object's script context and so we have to safely pretend this value doesn't exist.

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

}

return object;

}

AssertMsg(scriptContext->GetThreadContext() == object->GetScriptContext()->GetThreadContext(), "ScriptContexts should belong to same threadcontext for marshalling.");

// In heapenum, we are traversing through the object graph to dump out the content of recyclable objects. The content

// of the objects are duplicated to the heapenum result, and we are not storing/changing the object graph during heap enum.

// We don't actually need to do cross site thunk here.

if (scriptContext->GetRecycler()->IsHeapEnumInProgress())

{

return object;

}

#if ENABLE\_COPYONACCESS\_ARRAY

JavascriptLibrary::CheckAndConvertCopyOnAccessNativeIntArray<Var>(object);

#endif

TypeId typeId = object->GetTypeId();

AssertMsg(typeId != TypeIds\_Enumerator, "enumerator shouldn't be marshalled here");

// At the moment the mental model for WithScopeObject Marshaling is this:

// Are we trying to marshal a WithScopeObject in the Frame Display? - then 1) unwrap in MarshalFrameDisplay,

// 2) marshal the wrapped object, 3) Create a new WithScopeObject in the current scriptContext and re-wrap.

// We can avoid copying the WithScopeObject because it has no properties and never should.

// Thus creating a new WithScopeObject per context in MarshalFrameDisplay should be kosher.

// If it is not a FrameDisplay then we should not marshal. We can wrap cross context objects with a

// withscopeObject in a different context. When we unwrap for property lookups and the wrapped object

// is cross context, then we marshal the wrapped object into the current scriptContext, thus avoiding

// the need to copy the WithScopeObject itself. Thus We don't have to handle marshaling the WithScopeObject

// in non-FrameDisplay cases.

AssertMsg(typeId != TypeIds\_WithScopeObject, "WithScopeObject shouldn't be marshalled here");

if (StaticType::Is(typeId))

{

return object->CloneToScriptContext(scriptContext);

}

if (typeId == TypeIds\_ModuleRoot)

{

RootObjectBase \*moduleRoot = static\_cast<RootObjectBase\*>(object);

HostObjectBase \* hostObject = moduleRoot->GetHostObject();

// When marshaling module root, all we need is the host object.

// So, if the module root which is being marshaled has host object, marshal it.

if (hostObject)

{

Var hostDispatch = hostObject->GetHostDispatchVar();

return CrossSite::MarshalVar(scriptContext, hostDispatch);

}

}

if (typeId == TypeIds\_Function)

{

if (object == object->GetScriptContext()->GetLibrary()->GetDefaultAccessorFunction() )

{

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

}

if (DoRequestWrapper(object, fRequestWrapper))

{

// Marshal as a cross-site thunk if necessary before re-wrapping in an external function thunk.

MarshalVarInner(scriptContext, object, false);

return scriptContext->GetLibrary()->CreateWrappedExternalFunction(static\_cast<JavascriptExternalFunction\*>(object));

}

}

// We have a object marshaled, we need to keep track of the related script context

// so optimization overrides can be updated as a group

scriptContext->optimizationOverrides.Merge(&object->GetScriptContext()->optimizationOverrides);

DynamicObject \* dynamicObject = DynamicObject::FromVar(object);

if (!dynamicObject->IsExternal())

{

if (!dynamicObject->IsCrossSiteObject())

{

MarshalDynamicObjectAndPrototype(scriptContext, dynamicObject);

}

}

else

{

MarshalPrototypeChain(scriptContext, dynamicObject);

if (Js::JavascriptConversion::IsCallable(dynamicObject))

{

dynamicObject->MarshalToScriptContext(scriptContext);

}

}

return dynamicObject;

}

bool CrossSite::IsThunk(JavascriptMethod thunk)

{

return (thunk == CrossSite::ProfileThunk || thunk == CrossSite::DefaultThunk);

}

Var CrossSite::ProfileThunk(RecyclableObject\* callable, CallInfo callInfo, ...)

{

JavascriptFunction\* function = JavascriptFunction::FromVar(callable);

Assert(function->GetTypeId() == TypeIds\_Function);

Assert(function->GetEntryPoint() == CrossSite::ProfileThunk);

RUNTIME\_ARGUMENTS(args, callInfo);

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

// It is not safe to access the function body if the script context is not alive.

scriptContext->VerifyAliveWithHostContext(!function->IsExternal(),

scriptContext->GetThreadContext()->GetPreviousHostScriptContext());

JavascriptMethod entryPoint;

FunctionInfo \*funcInfo = function->GetFunctionInfo();

if (funcInfo->HasBody())

{

#if ENABLE\_DEBUG\_CONFIG\_OPTIONS

wchar\_t debugStringBuffer[MAX\_FUNCTION\_BODY\_DEBUG\_STRING\_SIZE];

#endif

entryPoint = (JavascriptMethod)ScriptFunction::FromVar(function)->GetEntryPointInfo()->address;

if (funcInfo->IsDeferred() && scriptContext->IsProfiling())

{

// if the current entrypoint is deferred parse we need to update it appropriately for the profiler mode.

entryPoint = Js::ScriptContext::GetProfileModeThunk(entryPoint);

}

OUTPUT\_TRACE(Js::ScriptProfilerPhase, L"CrossSite::ProfileThunk FunctionNumber : %s, Entrypoint : 0x%08X\n", funcInfo->GetFunctionProxy()->GetDebugNumberSet(debugStringBuffer), entryPoint);

}

else

{

entryPoint = ProfileEntryThunk;

}

return CommonThunk(function, entryPoint, args);

}

Var CrossSite::DefaultThunk(RecyclableObject\* callable, CallInfo callInfo, ...)

{

JavascriptFunction\* function = JavascriptFunction::FromVar(callable);

Assert(function->GetTypeId() == TypeIds\_Function);

Assert(function->GetEntryPoint() == CrossSite::DefaultThunk);

RUNTIME\_ARGUMENTS(args, callInfo);

// It is not safe to access the function body if the script context is not alive.

function->GetScriptContext()->VerifyAliveWithHostContext(!function->IsExternal(),

ThreadContext::GetContextForCurrentThread()->GetPreviousHostScriptContext());

JavascriptMethod entryPoint;

FunctionInfo \*funcInfo = function->GetFunctionInfo();

if (funcInfo->HasBody())

{

entryPoint = (JavascriptMethod)ScriptFunction::FromVar(function)->GetEntryPointInfo()->address;

}

else

{

entryPoint = funcInfo->GetOriginalEntryPoint();

}

return CommonThunk(function, entryPoint, args);

}

Var CrossSite::CommonThunk(RecyclableObject\* recyclableObject, JavascriptMethod entryPoint, Arguments args)

{

DynamicObject\* function = DynamicObject::FromVar(recyclableObject);

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

Assert(!targetScriptContext->IsClosed());

Assert(function->IsExternal() || function->IsCrossSiteObject());

Assert(targetScriptContext->GetThreadContext()->IsScriptActive());

HostScriptContext\* calleeHostScriptContext = targetScriptContext->GetHostScriptContext();

HostScriptContext\* callerHostScriptContext = targetScriptContext->GetThreadContext()->GetPreviousHostScriptContext();

if (callerHostScriptContext == calleeHostScriptContext || (callerHostScriptContext == nullptr && !calleeHostScriptContext->HasCaller()))

{

return JavascriptFunction::CallFunction<true>(function, entryPoint, args);

}

#if DBG\_DUMP || defined(PROFILE\_EXEC) || defined(PROFILE\_MEM)

calleeHostScriptContext->EnsureParentInfo(callerHostScriptContext->GetScriptContext());

#endif

uint i = 0;

if (args.Values[0] == nullptr)

{

i = 1;

Assert(args.Info.Flags & CallFlags\_New);

Assert(JavascriptFunction::Is(function) && JavascriptFunction::FromVar(function)->GetFunctionInfo()->GetAttributes() & FunctionInfo::SkipDefaultNewObject);

}

uint count = args.Info.Count;

if ((args.Info.Flags & CallFlags\_ExtraArg) && ((args.Info.Flags & CallFlags\_NewTarget) == 0))

{

// The final eval arg is a frame display that needs to be marshaled specially.

args.Values[count-1] = CrossSite::MarshalFrameDisplay(targetScriptContext, (FrameDisplay\*)args.Values[count-1]);

count--;

}

for (; i < count; i++)

{

args.Values[i] = CrossSite::MarshalVar(targetScriptContext, args.Values[i]);

}

#if ENABLE\_NATIVE\_CODEGEN

CheckCodeGenFunction checkCodeGenFunction = GetCheckCodeGenFunction(entryPoint);

if (checkCodeGenFunction != nullptr)

{

ScriptFunction\* callFunc = ScriptFunction::FromVar(function);

entryPoint = checkCodeGenFunction(callFunc);

Assert(CrossSite::IsThunk(function->GetEntryPoint()));

}

#endif

// We need to setup the caller chain when we go across script site boundary. Property access

// is OK, and we need to let host know who the caller is when a call is from another script site.

// CrossSiteObject is the natural place but it is in the target site. We build up the site

// chain through PushDispatchExCaller/PopDispatchExCaller, and we call SetCaller in the target site

// to indicate who the caller is. We first need to get the site from the previously pushed site

// and set that as the caller for current call, and push a new DispatchExCaller for future calls

// off this site. GetDispatchExCaller and ReleaseDispatchExCaller is used to get the current caller.

// currentDispatchExCaller is cached to avoid multiple allocations.

IUnknown\* sourceCaller = nullptr, \*previousSourceCaller = nullptr;

HRESULT hr = NOERROR;

Var result = nullptr;

BOOL wasDispatchExCallerPushed = FALSE, wasCallerSet = FALSE;

\_\_try

{

hr = callerHostScriptContext->GetDispatchExCaller((void\*\*)&sourceCaller);

if (SUCCEEDED(hr))

{

hr = calleeHostScriptContext->SetCaller((IUnknown\*)sourceCaller, (IUnknown\*\*)&previousSourceCaller);

}

if (SUCCEEDED(hr))

{

wasCallerSet = TRUE;

hr = calleeHostScriptContext->PushHostScriptContext();

}

if (FAILED(hr))

{

// CONSIDER: Should this be callerScriptContext if we failed?

JavascriptError::MapAndThrowError(targetScriptContext, hr);

}

wasDispatchExCallerPushed = TRUE;

result = JavascriptFunction::CallFunction<true>(function, entryPoint, args);

ScriptContext\* callerScriptContext = callerHostScriptContext->GetScriptContext();

result = CrossSite::MarshalVar(callerScriptContext, result);

}

\_\_finally

{

if (sourceCaller != nullptr)

{

callerHostScriptContext->ReleaseDispatchExCaller(sourceCaller);

}

IUnknown\* originalCaller = nullptr;

if (wasDispatchExCallerPushed)

{

calleeHostScriptContext->PopHostScriptContext();

}

if (wasCallerSet)

{

calleeHostScriptContext->SetCaller(previousSourceCaller, &originalCaller);

if (previousSourceCaller)

{

previousSourceCaller->Release();

}

if (originalCaller)

{

originalCaller->Release();

}

}

}

Assert(result != nullptr);

return result;

}

// For prototype chain to install cross-site thunk.

// When we change prototype using \_\_proto\_\_, those prototypes might not have cross-site thunks

// installed even though the CEO is accessed from a different context. During ChangePrototype time

// we don't really know where the requestContext is.

// Force installing cross-site thunk for all prototype changes. It's a relatively less frequently used

// scenario.

void CrossSite::ForceCrossSiteThunkOnPrototypeChain(RecyclableObject\* object)

{

if (TaggedNumber::Is(object))

{

return;

}

while (DynamicType::Is(object->GetTypeId()) && !JavascriptProxy::Is(object))

{

DynamicObject\* dynamicObject = DynamicObject::FromVar(object);

if (!dynamicObject->IsCrossSiteObject() && !dynamicObject->IsExternal())

{

// force to install cross-site thunk on prototype objects.

dynamicObject->MarshalToScriptContext(nullptr);

}

object = object->GetPrototype();

}

return;

}

};

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

// 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

class DOMFastPathInfo;

namespace Js

{

class CrossSite

{

friend class ExternalType;

friend class DOMFastPathInfo;

public:

static bool IsThunk(JavascriptMethod thunk);

static BOOL NeedMarshalVar(Var instance, ScriptContext \* requestContext);

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

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

static Var MarshalVar(ScriptContext\* scriptContext, Var value, bool fRequestWrapper = false);

static Var MarshalEnumerator(ScriptContext\* scriptContext, Var value);

static void MarshalDynamicObjectAndPrototype(ScriptContext \* scriptContext, DynamicObject \* object);

static void ForceCrossSiteThunkOnPrototypeChain(RecyclableObject\* object);

// If we know the type of the object, use this to do a virtual table check instead of a virtual call

template <typename T>

static BOOL IsCrossSiteObjectTyped(T \* obj)

{

BOOL ret = VirtualTableInfo<CrossSiteObject<T>>::HasVirtualTable(obj);

Assert(ret || VirtualTableInfo<T>::HasVirtualTable(obj));

Assert(ret == obj->IsCrossSiteObject());

return ret;

}

private:

static Var MarshalVarInner(ScriptContext\* scriptContext, \_\_in Js::RecyclableObject\* object, bool fRequestWrapper);

static Var CommonThunk(RecyclableObject \* function, JavascriptMethod entryPoint, Arguments args);

static void MarshalDynamicObject(ScriptContext \* scriptContext, DynamicObject \* object);

static void MarshalPrototypeChain(ScriptContext \* scriptContext, DynamicObject \* object);

static Var MarshalFrameDisplay(ScriptContext\* scriptContext, FrameDisplay \*display);

static bool DoRequestWrapper(Js::RecyclableObject\* object, bool fRequestWrapper);

};

};

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

// 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

{

// for enumerators, the scriptContext of the enumerator is different

// from the scriptContext of the object.

#if !defined(USED\_IN\_STATIC\_LIB)

#define DEFINE\_MARSHAL\_ENUMERATOR\_TO\_SCRIPT\_CONTEXT(T) \

friend class Js::CrossSiteEnumerator<T>; \

virtual void MarshalToScriptContext(Js::ScriptContext \* scriptContext) override \

{ \

Assert(GetScriptContext() == scriptContext); \

AssertMsg(VirtualTableInfo<T>::HasVirtualTable(this) || VirtualTableInfo<Js::CrossSiteEnumerator<T>>::HasVirtualTable(this), "Derived class need to define marshal to script context"); \

VirtualTableInfo<Js::CrossSiteEnumerator<T>>::SetVirtualTable(this); \

}

#else

#define DEFINE\_MARSHAL\_ENUMERATOR\_TO\_SCRIPT\_CONTEXT(T) \

virtual void MarshalToScriptContext(Js::ScriptContext \* scriptContext) override {Assert(FALSE);}

#endif

template <typename T>

class CrossSiteEnumerator : public T

{

private:

DEFINE\_VTABLE\_CTOR(CrossSiteEnumerator<T>, T);

public:

virtual Var GetCurrentIndex() override;

virtual Var GetCurrentValue() override;

virtual void Reset() override;

virtual BOOL MoveNext(PropertyAttributes\* attributes = nullptr) override;

virtual Var GetCurrentAndMoveNext(PropertyId& propertyId, PropertyAttributes\* attributes = nullptr) override;

virtual BOOL IsCrossSiteEnumerator() override

{

return true;

}

};

template<typename T>

Var CrossSiteEnumerator<T>::GetCurrentIndex()

{

Var result = \_\_super::GetCurrentIndex();

if (result)

{

result = CrossSite::MarshalVar(GetScriptContext(), result);

}

return result;

}

template <typename T>

Var CrossSiteEnumerator<T>::GetCurrentValue()

{

Var result = \_\_super::GetCurrentValue();

if (result)

{

result = CrossSite::MarshalVar(GetScriptContext(), result);

}

return result;

}

template <typename T>

BOOL CrossSiteEnumerator<T>::MoveNext(PropertyAttributes\* attributes)

{

return \_\_super::MoveNext(attributes);

}

template <typename T>

void CrossSiteEnumerator<T>::Reset()

{

\_\_super::Reset();

}

template <typename T>

Var CrossSiteEnumerator<T>::GetCurrentAndMoveNext(PropertyId& propertyId, PropertyAttributes\* attributes)

{

Var result = \_\_super::GetCurrentAndMoveNext(propertyId, attributes);

if (result)

{

result = CrossSite::MarshalVar(GetScriptContext(), result);

}

return result;

}

};

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

// 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

{

template <typename T>

class CrossSiteObject : public T

{

private:

DEFINE\_VTABLE\_CTOR(CrossSiteObject<T>, T);

public:

virtual BOOL GetProperty(Var originalInstance, PropertyId propertyId, Var\* value, PropertyValueInfo\* info, ScriptContext\* requestContext) override;

virtual BOOL GetProperty(Var originalInstance, JavascriptString\* propertyNameString, Var\* value, PropertyValueInfo\* info, ScriptContext\* requestContext) override;

virtual BOOL GetAccessors(PropertyId propertyId, Var\* getter, Var\* setter, ScriptContext \* requestContext) override;

virtual BOOL GetPropertyReference(Var originalInstance, PropertyId propertyId, Var\* value, PropertyValueInfo\* info, ScriptContext\* requestContext) override;

virtual BOOL SetProperty(PropertyId propertyId, Var value, PropertyOperationFlags flags, PropertyValueInfo\* info) override;

virtual BOOL SetProperty(JavascriptString\* propertyNameString, Var value, PropertyOperationFlags flags, PropertyValueInfo\* info) override;

virtual BOOL InitProperty(PropertyId propertyId, Var value, PropertyOperationFlags flags = PropertyOperation\_None, PropertyValueInfo\* info = NULL) override;

virtual BOOL SetPropertyWithAttributes(PropertyId propertyId, Var value, PropertyAttributes attributes, PropertyValueInfo\* info, PropertyOperationFlags flags = PropertyOperation\_None, SideEffects possibleSideEffects = SideEffects\_Any) override;

virtual BOOL InitPropertyScoped(PropertyId propertyId, Var value) override;

virtual BOOL InitFuncScoped(PropertyId propertyId, Var value) override;

virtual BOOL GetItem(Var originalInstance, uint32 index, Var\* value, ScriptContext \* requestContext) override;

virtual BOOL GetItemReference(Var originalInstance, uint32 index, Var\* value, ScriptContext \* requestContext) override;

virtual DescriptorFlags GetItemSetter(uint32 index, Var\* setterValue, ScriptContext\* requestContext) override;

virtual BOOL SetItem(uint32 index, Var value, PropertyOperationFlags flags) override;

virtual BOOL GetEnumerator(BOOL enumNonEnumerable, Var\* enumerator, ScriptContext \* requestContext, bool preferSnapshotSemantics, bool enumSymbols = false) override;

virtual Var GetHostDispatchVar() override;

virtual DescriptorFlags GetSetter(PropertyId propertyId, Var\* setterValue, PropertyValueInfo\* info, ScriptContext\* requestContext) override;

virtual DescriptorFlags GetSetter(JavascriptString\* propertyNameString, Var\* setterValue, PropertyValueInfo\* info, ScriptContext\* requestContext) override;

virtual BOOL SetAccessors(PropertyId propertyId, Var getter, Var setter, PropertyOperationFlags flags) override;

virtual void RemoveFromPrototype(ScriptContext \* requestContext) override;

virtual void AddToPrototype(ScriptContext \* requestContext) override;

virtual void SetPrototype(RecyclableObject\* newPrototype) override;

virtual BOOL IsCrossSiteObject() const override { return TRUE; }

virtual void MarshalToScriptContext(ScriptContext \* requestContext) override

{

AssertMsg(false, "CrossSite::MarshalVar should have handled this");

}

};

template <typename T>

BOOL CrossSiteObject<T>::GetProperty(Var originalInstance, PropertyId propertyId, Var\* value, PropertyValueInfo\* info, ScriptContext\* requestContext)

{

originalInstance = CrossSite::MarshalVar(GetScriptContext(), originalInstance);

BOOL result = \_\_super::GetProperty(originalInstance, propertyId, value, info, requestContext);

if (result)

{

\*value = CrossSite::MarshalVar(requestContext, \*value);

}

return result;

}

template <typename T>

BOOL CrossSiteObject<T>::GetProperty(Var originalInstance, JavascriptString\* propertyNameString, Var\* value, PropertyValueInfo\* info, ScriptContext\* requestContext)

{

BOOL result = \_\_super::GetProperty(originalInstance, propertyNameString, value, info, requestContext);

if (result)

{

\*value = CrossSite::MarshalVar(requestContext, \*value);

}

return result;

}

template <typename T>

BOOL CrossSiteObject<T>::GetAccessors(PropertyId propertyId, Var\* getter, Var\* setter, ScriptContext \* requestContext)

{

BOOL result = \_\_super::GetAccessors(propertyId, getter, setter, requestContext);

if (result)

{

if (\*getter != nullptr)

{

\*getter = CrossSite::MarshalVar(requestContext, \*getter);

}

if (\*setter != nullptr)

{

\*setter = CrossSite::MarshalVar(requestContext, \*setter);

}

}

return result;

}

template <typename T>

BOOL CrossSiteObject<T>::GetPropertyReference(Var originalInstance, PropertyId propertyId, Var\* value, PropertyValueInfo\* info, ScriptContext\* requestContext)

{

originalInstance = CrossSite::MarshalVar(GetScriptContext(), originalInstance);

BOOL result = \_\_super::GetPropertyReference(originalInstance, propertyId, value, info, requestContext);

if (result)

{

\*value = CrossSite::MarshalVar(requestContext, \*value);

}

return result;

}

template <typename T>

BOOL CrossSiteObject<T>::SetProperty(PropertyId propertyId, Var value, PropertyOperationFlags flags, PropertyValueInfo\* info)

{

value = CrossSite::MarshalVar(GetScriptContext(), value);

return \_\_super::SetProperty(propertyId, value, flags, info);

}

template <typename T>

BOOL CrossSiteObject<T>::SetProperty(JavascriptString\* propertyNameString, Var value, PropertyOperationFlags flags, PropertyValueInfo\* info)

{

value = CrossSite::MarshalVar(GetScriptContext(), value);

return \_\_super::SetProperty(propertyNameString, value, flags, info);

}

template <typename T>

BOOL CrossSiteObject<T>::InitProperty(PropertyId propertyId, Var value, PropertyOperationFlags flags, PropertyValueInfo\* info)

{

value = CrossSite::MarshalVar(GetScriptContext(), value);

return \_\_super::InitProperty(propertyId, value, flags, info);

}

template <typename T>

BOOL CrossSiteObject<T>::SetPropertyWithAttributes(PropertyId propertyId, Var value, PropertyAttributes attributes, PropertyValueInfo\* info, PropertyOperationFlags flags, SideEffects possibleSideEffects = SideEffects\_Any)

{

value = CrossSite::MarshalVar(GetScriptContext(), value);

return \_\_super::SetPropertyWithAttributes(propertyId, value, attributes, info, flags, possibleSideEffects);

}

template <typename T>

BOOL CrossSiteObject<T>::InitPropertyScoped(PropertyId propertyId, Var value)

{

value = CrossSite::MarshalVar(GetScriptContext(), value);

return \_\_super::InitPropertyScoped(propertyId, value);

}

template <typename T>

BOOL CrossSiteObject<T>::InitFuncScoped(PropertyId propertyId, Var value)

{

value = CrossSite::MarshalVar(GetScriptContext(), value);

return \_\_super::InitFuncScoped(propertyId, value);

}

template <typename T>

BOOL CrossSiteObject<T>::GetItem(Var originalInstance, uint32 index, Var\* value, ScriptContext \* requestContext)

{

originalInstance = CrossSite::MarshalVar(GetScriptContext(), originalInstance);

BOOL result = \_\_super::GetItem(originalInstance, index, value, requestContext);

if (result)

{

\*value = CrossSite::MarshalVar(requestContext, \*value);

}

return result;

}

template <typename T>

BOOL CrossSiteObject<T>::GetItemReference(Var originalInstance, uint32 index, Var\* value, ScriptContext \* requestContext)

{

originalInstance = CrossSite::MarshalVar(GetScriptContext(), originalInstance);

BOOL result = \_\_super::GetItemReference(originalInstance, index, value, requestContext);

if (result)

{

\*value = CrossSite::MarshalVar(requestContext, \*value);

}

return result;

}

template <typename T>

DescriptorFlags CrossSiteObject<T>::GetItemSetter(uint32 index, Var \*setterValue, ScriptContext\* requestContext)

{

DescriptorFlags flags = \_\_super::GetItemSetter(index, setterValue, requestContext);

if ((flags & Accessor) == Accessor && \*setterValue)

{

\*setterValue = CrossSite::MarshalVar(requestContext, \*setterValue);

}

return flags;

}

template <typename T>

BOOL CrossSiteObject<T>::SetItem(uint32 index, Var value, PropertyOperationFlags flags)

{

value = CrossSite::MarshalVar(GetScriptContext(), value);

return \_\_super::SetItem(index, value, flags);

}

template <typename T>

DescriptorFlags CrossSiteObject<T>::GetSetter(PropertyId propertyId, Var\* setterValue, PropertyValueInfo\* info, ScriptContext\* requestContext)

{

DescriptorFlags flags = \_\_super::GetSetter(propertyId, setterValue, info, requestContext);

if ((flags & Accessor) == Accessor && \*setterValue)

{

PropertyValueInfo::SetNoCache(info, this);

\*setterValue = CrossSite::MarshalVar(requestContext, \*setterValue);

}

return flags;

}

template <typename T>

DescriptorFlags CrossSiteObject<T>::GetSetter(JavascriptString\* propertyNameString, Var\* setterValue, PropertyValueInfo\* info, ScriptContext\* requestContext)

{

DescriptorFlags flags = \_\_super::GetSetter(propertyNameString, setterValue, info, requestContext);

if ((flags & Accessor) == Accessor && \*setterValue)

{

PropertyValueInfo::SetNoCache(info, this);

\*setterValue = CrossSite::MarshalVar(requestContext, \*setterValue);

}

return flags;

}

template <typename T>

BOOL CrossSiteObject<T>::SetAccessors(PropertyId propertyId, Var getter, Var setter, PropertyOperationFlags flags)

{

if (getter != nullptr)

{

getter = CrossSite::MarshalVar(GetScriptContext(), getter);

}

if (setter != nullptr)

{

setter = CrossSite::MarshalVar(GetScriptContext(), setter);

}

return \_\_super::SetAccessors(propertyId, getter, setter, flags);

}

template <typename T>

void CrossSiteObject<T>::RemoveFromPrototype(ScriptContext \* requestContext)

{

if (GetScriptContext() == requestContext)

{

\_\_super::RemoveFromPrototype(requestContext);

}

// else do nothing because we never cache cross-context

}

template <typename T>

void CrossSiteObject<T>::AddToPrototype(ScriptContext \* requestContext)

{

if (GetScriptContext() == requestContext)

{

\_\_super::AddToPrototype(requestContext);

}

// else do nothing because we never cache cross-context

}

template <typename T>

void CrossSiteObject<T>::SetPrototype(RecyclableObject\* newPrototype)

{

newPrototype = (RecyclableObject\*)CrossSite::MarshalVar(GetScriptContext(), newPrototype);

\_\_super::SetPrototype(newPrototype);

}

template <typename T>

BOOL CrossSiteObject<T>::GetEnumerator(BOOL enumNonEnumerable, Var\* enumerator, ScriptContext \* requestContext, bool preferSnapshotSemantics, bool enumSymbols)

{

BOOL result = \_\_super::GetEnumerator(enumNonEnumerable, enumerator, requestContext, preferSnapshotSemantics, enumSymbols);

if (result)

{

\*enumerator = CrossSite::MarshalEnumerator(requestContext, \*enumerator);

}

return result;

}

template <typename T>

Var CrossSiteObject<T>::GetHostDispatchVar()

{

Var hostDispatch = \_\_super::GetHostDispatchVar();

AssertMsg(hostDispatch, "hostDispatch");

hostDispatch = CrossSite::MarshalVar(GetScriptContext(), hostDispatch);

return hostDispatch;

}

}

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

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

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

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

#include "RuntimeBasePch.h"

#if DBG\_DUMP

#include <strsafe.h>

#include "Language\JavascriptStackWalker.h"

WCHAR\* DumpCallStack(uint frameCount) { return DumpCallStackFull(frameCount, /\*print\*/ true); }

WCHAR\* DumpCallStackFull(uint frameCount, bool print)

{

Js::ScriptContext\* scriptContext = ThreadContext::GetContextForCurrentThread()->GetScriptContextList();

Js::JavascriptStackWalker walker(scriptContext);

WCHAR buffer[512];

Js::StringBuilder<ArenaAllocator> sb(scriptContext->GeneralAllocator());

uint fc = 0;

while (walker.Walk())

{

void \* codeAddr = walker.GetCurrentCodeAddr();

if (walker.IsJavascriptFrame())

{

wchar\_t debugStringBuffer[MAX\_FUNCTION\_BODY\_DEBUG\_STRING\_SIZE];

StringCchPrintf(buffer, \_countof(buffer), L"0x%p ", codeAddr);

sb.AppendSz(buffer);

// Found a JavascriptFunction. Dump its name and parameters.

Js::JavascriptFunction \*jsFunc = walker.GetCurrentFunction();

Js::FunctionBody \* jsBody = jsFunc->GetFunctionBody();

Js::CallInfo const \* callInfo = walker.GetCallInfo();

const WCHAR\* sourceFileName = L"NULL";

ULONG line = 0; LONG column = 0;

walker.GetSourcePosition(&sourceFileName, &line, &column);

StringCchPrintf(buffer, \_countof(buffer), L"%s [%s] (0x%p, Args=%u", jsBody->GetDisplayName(), jsBody->GetDebugNumberSet(debugStringBuffer), jsFunc,

callInfo->Count);

sb.AppendSz(buffer);

for (uint i = 0; i < callInfo->Count; i++)

{

StringCchPrintf(buffer, \_countof(buffer), L", 0x%p", walker.GetJavascriptArgs()[i]);

sb.AppendSz(buffer);

}

StringCchPrintf(buffer, \_countof(buffer), L")[%s (%u, %d)]\n", sourceFileName, line + 1, column + 1);

sb.AppendSz(buffer);

fc++;

if(fc >= frameCount)

{

break;

}

}

}

sb.AppendCppLiteral(L"----------------------------------------------------------------------\n");

WCHAR\* stack = sb.Detach();

if(print)

{

Output::Print(stack);

}

return stack;

}

#endif

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

// 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

#if DBG\_DUMP

WCHAR\* DumpCallStackFull(uint frameCount = -1, bool print = true);

WCHAR\* DumpCallStack(uint frameCount = -1);

#endif

#if DBG\_DUMP

#define OUTPUT\_TRACE\_WITH\_STACK(Phase, ...) OUTPUT\_TRACE\_WITH\_STACK\_COUNT(Phase, /\*frameCount\*/ 3, \_\_VA\_ARGS\_\_)

#define OUTPUT\_TRACE\_WITH\_STACK\_COUNT(Phase, frameCount, ...) \

if(Js::Configuration::Global.flags.Verbose) \

{ \

Output::TraceWithCallback( Phase, [] () { \

return DumpCallStackFull(frameCount, /\*print\*/false); \

}, \_\_VA\_ARGS\_\_); \

}

#else

#define OUTPUT\_TRACE\_WITH\_STACK(Phase, ...)

#define OUTPUT\_TRACE\_WITH\_STACK\_COUNT(Phase, frameCount, ...)

#endif

#if ENABLE\_DEBUG\_CONFIG\_OPTIONS

#define OUTPUT\_PRINT(FunctionBody) \

Output::Print(L"Function %s (#%d.%u, #%u) ", (FunctionBody)->GetDisplayName(), \

(int)(FunctionBody)->GetSourceContextId(), (FunctionBody)->GetLocalFunctionId(), (FunctionBody)->GetFunctionNumber());

#define OUTPUT\_TRACE2(Phase, FunctionBody, ...) \

if(Js::Configuration::Global.flags.Trace.IsEnabled((Phase))) \

{ \

WCHAR prefixValue[512]; \

swprintf\_s(prefixValue, L"Function %s (#%d.%u, #%u)", (FunctionBody)->GetDisplayName(), \

(int)(FunctionBody)->GetSourceContextId(), (FunctionBody)->GetLocalFunctionId(), (FunctionBody)->GetFunctionNumber()); \

Output::TraceWithPrefix((Phase), prefixValue, \_\_VA\_ARGS\_\_); \

}

#define OUTPUT\_TRACE\_FUNC(Phase, Func, ...) \

if(PHASE\_TRACE((Phase), (Func))) \

{ \

WCHAR workItem[256]; \

Func->m\_workItem->GetDisplayName(workItem, \_countof(workItem)); \

WCHAR prefixValue[512]; \

swprintf\_s(prefixValue, L"%s (#%d.%u, #%u)", workItem, \

(int)(Func)->GetJnFunction()->GetSourceContextId(), (Func)->GetJnFunction()->GetLocalFunctionId(), (Func)->GetJnFunction()->GetFunctionNumber()); \

Output::TraceWithPrefix((Phase), prefixValue, \_\_VA\_ARGS\_\_); \

}

#define OUTPUT\_VERBOSE\_TRACE2(Phase, FunctionBody, ...) \

if(Js::Configuration::Global.flags.Verbose) \

{ \

OUTPUT\_TRACE2((Phase), (FunctionBody), \_\_VA\_ARGS\_\_); \

}

#define OUTPUT\_VERBOSE\_TRACE\_FUNC(Phase, Func, ...) \

if(Js::Configuration::Global.flags.Verbose) \

{ \

OUTPUT\_TRACE\_FUNC((Phase), (Func), \_\_VA\_ARGS\_\_); \

}

#else

#define OUTPUT\_PRINT(FunctionBody)

#define OUTPUT\_TRACE2(Phase, FunctionBody, ...)

#define OUTPUT\_VERBOSE\_TRACE2(Phase, FunctionBody, ...)

#define OUTPUT\_TRACE\_FUNC(Phase, Func, ...)

#define OUTPUT\_VERBOSE\_TRACE\_FUNC(Phase, Func, ...)

#endif

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

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

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

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

#include "RuntimeBasePch.h"

#ifdef \_CONTROL\_FLOW\_GUARD

#if !defined(DELAYLOAD\_SET\_CFG\_TARGET)

extern "C"

WINBASEAPI

BOOL

WINAPI

SetProcessValidCallTargets(

\_In\_ HANDLE hProcess,

\_In\_ PVOID VirtualAddress,

\_In\_ SIZE\_T RegionSize,

\_In\_ ULONG NumberOfOffets,

\_In\_reads\_(NumberOfOffets) PCFG\_CALL\_TARGET\_INFO OffsetInformation

);

#endif

#endif

namespace Js

{

HRESULT DelayLoadWinRtString::WindowsCreateString(\_In\_reads\_opt\_(length) const WCHAR \* sourceString, UINT32 length, \_Outptr\_result\_maybenull\_ \_Result\_nullonfailure\_ HSTRING \* string)

{

if (m\_hModule)

{

if (m\_pfnWindowsCreateString == nullptr)

{

m\_pfnWindowsCreateString = (PFNCWindowsCreateString)GetFunction("WindowsCreateString");

if (m\_pfnWindowsCreateString == nullptr)

{

\*string = nullptr;

return E\_UNEXPECTED;

}

}

Assert(m\_pfnWindowsCreateString != nullptr);

return m\_pfnWindowsCreateString(sourceString, length, string);

}

\*string = nullptr;

return E\_NOTIMPL;

}

HRESULT DelayLoadWinRtString::WindowsCreateStringReference(\_In\_reads\_opt\_(length + 1) const WCHAR \*sourceString, UINT32 length, \_Out\_ HSTRING\_HEADER \*hstringHeader, \_Outptr\_result\_maybenull\_ \_Result\_nullonfailure\_ HSTRING \* string)

{

if (m\_hModule)

{

if (m\_pfnWindowsCreateStringReference == nullptr)

{

m\_pfnWindowsCreateStringReference = (PFNCWindowsCreateStringReference)GetFunction("WindowsCreateStringReference");

if (m\_pfnWindowsCreateStringReference == nullptr)

{

\*string = nullptr;

return E\_UNEXPECTED;

}

}

Assert(m\_pfnWindowsCreateStringReference != nullptr);

return m\_pfnWindowsCreateStringReference(sourceString, length, hstringHeader, string);

}

\*string = nullptr;

return E\_NOTIMPL;

}

HRESULT DelayLoadWinRtString::WindowsDeleteString(\_In\_opt\_ HSTRING string)

{

if (m\_hModule)

{

if (m\_pfnWindowsDeleteString == nullptr)

{

m\_pfnWindowsDeleteString = (PFNCWindowsDeleteString)GetFunction("WindowsDeleteString");

if (m\_pfnWindowsDeleteString == nullptr)

{

return E\_UNEXPECTED;

}

}

Assert(m\_pfnWindowsDeleteString != nullptr);

HRESULT hr = m\_pfnWindowsDeleteString(string);

Assert(SUCCEEDED(hr));

return hr;

}

return E\_NOTIMPL;

}

PCWSTR DelayLoadWinRtString::WindowsGetStringRawBuffer(\_In\_opt\_ HSTRING string, \_Out\_opt\_ UINT32 \* length)

{

if (m\_hModule)

{

if (m\_pfWindowsGetStringRawBuffer == nullptr)

{

m\_pfWindowsGetStringRawBuffer = (PFNCWindowsGetStringRawBuffer)GetFunction("WindowsGetStringRawBuffer");

if (m\_pfWindowsGetStringRawBuffer == nullptr)

{

if (length)

{

\*length = 0;

}

return L"\0";

}

}

Assert(m\_pfWindowsGetStringRawBuffer != nullptr);

return m\_pfWindowsGetStringRawBuffer(string, length);

}

if (length)

{

\*length = 0;

}

return L"\0";

}

HRESULT DelayLoadWinRtString::WindowsCompareStringOrdinal(\_In\_opt\_ HSTRING string1, \_In\_opt\_ HSTRING string2, \_Out\_ INT32 \* result)

{

if (m\_hModule)

{

if (m\_pfnWindowsCompareStringOrdinal == nullptr)

{

m\_pfnWindowsCompareStringOrdinal = (PFNCWindowsCompareStringOrdinal)GetFunction("WindowsCompareStringOrdinal");

if (m\_pfnWindowsCompareStringOrdinal == nullptr)

{

return E\_UNEXPECTED;

}

}

Assert(m\_pfnWindowsCompareStringOrdinal != nullptr);

return m\_pfnWindowsCompareStringOrdinal(string1,string2,result);

}

return E\_NOTIMPL;

}

HRESULT DelayLoadWinRtString::WindowsDuplicateString(\_In\_opt\_ HSTRING original, \_Outptr\_result\_maybenull\_ \_Result\_nullonfailure\_ HSTRING \*newString)

{

if(m\_hModule)

{

if(m\_pfnWindowsDuplicateString == nullptr)

{

m\_pfnWindowsDuplicateString = (PFNCWindowsDuplicateString)GetFunction("WindowsDuplicateString");

if(m\_pfnWindowsDuplicateString == nullptr)

{

\*newString = nullptr;

return E\_UNEXPECTED;

}

}

Assert(m\_pfnWindowsDuplicateString != nullptr);

return m\_pfnWindowsDuplicateString(original, newString);

}

\*newString = nullptr;

return E\_NOTIMPL;

}

HRESULT DelayLoadWinRtTypeResolution::RoParseTypeName(\_\_in HSTRING typeName, \_\_out DWORD \*partsCount, \_\_RPC\_\_deref\_out\_ecount\_full\_opt(\*partsCount) HSTRING \*\*typeNameParts)

{

if (m\_hModule)

{

if (m\_pfnRoParseTypeName == nullptr)

{

m\_pfnRoParseTypeName = (PFNCWRoParseTypeName)GetFunction("RoParseTypeName");

if (m\_pfnRoParseTypeName == nullptr)

{

return E\_UNEXPECTED;

}

}

Assert(m\_pfnRoParseTypeName != nullptr);

return m\_pfnRoParseTypeName(typeName, partsCount, typeNameParts);

}

return E\_NOTIMPL;

}

HRESULT DelayLoadWindowsGlobalization::DllGetActivationFactory(

\_\_in HSTRING activatibleClassId,

\_\_out IActivationFactory\*\* factory)

{

if (m\_hModule)

{

if (m\_pfnFNCWDllGetActivationFactory == nullptr)

{

m\_pfnFNCWDllGetActivationFactory = (PFNCWDllGetActivationFactory)GetFunction("DllGetActivationFactory");

if (m\_pfnFNCWDllGetActivationFactory == nullptr)

{

return E\_UNEXPECTED;

}

}

Assert(m\_pfnFNCWDllGetActivationFactory != nullptr);

return m\_pfnFNCWDllGetActivationFactory(activatibleClassId, factory);

}

return E\_NOTIMPL;

}

HRESULT DelayLoadWinRtFoundation::RoGetActivationFactory(

\_\_in HSTRING activatibleClassId,

\_\_in REFIID iid,

\_\_out IActivationFactory\*\* factory)

{

if (m\_hModule)

{

if (m\_pfnFNCWRoGetActivationFactory == nullptr)

{

m\_pfnFNCWRoGetActivationFactory = (PFNCWRoGetActivationFactory)GetFunction("RoGetActivationFactory");

if (m\_pfnFNCWRoGetActivationFactory == nullptr)

{

return E\_UNEXPECTED;

}

}

Assert(m\_pfnFNCWRoGetActivationFactory != nullptr);

return m\_pfnFNCWRoGetActivationFactory(activatibleClassId, iid, factory);

}

return E\_NOTIMPL;

}

HRESULT DelayLoadWinRtTypeResolution::RoResolveNamespace(

\_\_in\_opt const HSTRING namespaceName,

\_\_in\_opt const HSTRING windowsMetaDataPath,

\_\_in const DWORD packageGraphPathsCount,

\_\_in\_opt const HSTRING \*packageGraphPaths,

\_\_out DWORD \*metaDataFilePathsCount,

HSTRING \*\*metaDataFilePaths,

\_\_out DWORD \*subNamespacesCount,

HSTRING \*\*subNamespaces)

{

if (m\_hModule)

{

if (m\_pfnRoResolveNamespace == nullptr)

{

m\_pfnRoResolveNamespace = (PFNCRoResolveNamespace)GetFunction("RoResolveNamespace");

if (m\_pfnRoResolveNamespace == nullptr)

{

return E\_UNEXPECTED;

}

}

Assert(m\_pfnRoResolveNamespace != nullptr);

return m\_pfnRoResolveNamespace(namespaceName, windowsMetaDataPath, packageGraphPathsCount, packageGraphPaths,

metaDataFilePathsCount, metaDataFilePaths, subNamespacesCount, subNamespaces);

}

return E\_NOTIMPL;

}

void DelayLoadWindowsGlobalization::Ensure(Js::DelayLoadWinRtString \*winRTStringLibrary)

{

if (!this->m\_isInit)

{

DelayLoadLibrary::EnsureFromSystemDirOnly();

#if DBG

// This unused variable is to allow one to see the value of lastError in case both LoadLibrary (DelayLoadLibrary::Ensure has one) fail.

// As the issue might be with the first one, as opposed to the second

DWORD errorWhenLoadingBluePlus = GetLastError();

Unused(errorWhenLoadingBluePlus);

#endif

//Perform a check to see if Windows.Globalization.dll was loaded; if not try loading jsIntl.dll as we are on Win7.

if (m\_hModule == nullptr)

{

m\_hModule = LoadLibraryEx(GetWin7LibraryName(), nullptr, LOAD\_LIBRARY\_SEARCH\_SYSTEM32);

}

this->winRTStringLibrary = winRTStringLibrary;

this->winRTStringsPresent = GetFunction("WindowsDuplicateString") != nullptr;

}

}

HRESULT DelayLoadWindowsGlobalization::WindowsCreateString(\_In\_reads\_opt\_(length) const WCHAR \* sourceString, UINT32 length, \_Outptr\_result\_maybenull\_ \_Result\_nullonfailure\_ HSTRING \* string)

{

//If winRtStringLibrary isn't nullptr, that means it is available and we are on Win8+

if(!winRTStringsPresent && winRTStringLibrary->IsAvailable())

{

return winRTStringLibrary->WindowsCreateString(sourceString, length, string);

}

return DelayLoadWinRtString::WindowsCreateString(sourceString, length, string);

}

HRESULT DelayLoadWindowsGlobalization::WindowsCreateStringReference(\_In\_reads\_opt\_(length + 1) const WCHAR \* sourceString, UINT32 length, \_Out\_ HSTRING\_HEADER \* header, \_Outptr\_result\_maybenull\_ \_Result\_nullonfailure\_ HSTRING \* string)

{

//First, we attempt to use the WinStringRT api encapsulated in the globalization dll; if it is available then it is a downlevel dll.

//Otherwise; we might run into an error where we are using the Win8 (because testing is being done for instance) with the downlevel dll, and that would cause errors.

if(!winRTStringsPresent && winRTStringLibrary->IsAvailable())

{

return winRTStringLibrary->WindowsCreateStringReference(sourceString, length, header, string);

}

return DelayLoadWinRtString::WindowsCreateStringReference(sourceString, length, header, string);

}

HRESULT DelayLoadWindowsGlobalization::WindowsDeleteString(\_In\_opt\_ HSTRING string)

{

//First, we attempt to use the WinStringRT api encapsulated in the globalization dll; if it is available then it is a downlevel dll.

//Otherwise; we might run into an error where we are using the Win8 (because testing is being done for instance) with the downlevel dll, and that would cause errors.

if(!winRTStringsPresent && winRTStringLibrary->IsAvailable())

{

return winRTStringLibrary->WindowsDeleteString(string);

}

return DelayLoadWinRtString::WindowsDeleteString(string);

}

PCWSTR DelayLoadWindowsGlobalization::WindowsGetStringRawBuffer(\_In\_opt\_ HSTRING string, \_Out\_opt\_ UINT32 \* length)

{

//First, we attempt to use the WinStringRT api encapsulated in the globalization dll; if it is available then it is a downlevel dll.

//Otherwise; we might run into an error where we are using the Win8 (because testing is being done for instance) with the downlevel dll, and that would cause errors.

if(!winRTStringsPresent && winRTStringLibrary->IsAvailable())

{

return winRTStringLibrary->WindowsGetStringRawBuffer(string, length);

}

return DelayLoadWinRtString::WindowsGetStringRawBuffer(string, length);

}

HRESULT DelayLoadWindowsGlobalization::WindowsCompareStringOrdinal(\_In\_opt\_ HSTRING string1, \_In\_opt\_ HSTRING string2, \_Out\_ INT32 \* result)

{

//First, we attempt to use the WinStringRT api encapsulated in the globalization dll; if it is available then it is a downlevel dll.

//Otherwise; we might run into an error where we are using the Win8 (because testing is being done for instance) with the downlevel dll, and that would cause errors.

if(!winRTStringsPresent && winRTStringLibrary->IsAvailable())

{

return winRTStringLibrary->WindowsCompareStringOrdinal(string1, string2, result);

}

return DelayLoadWinRtString::WindowsCompareStringOrdinal(string1, string2, result);

}

HRESULT DelayLoadWindowsGlobalization::WindowsDuplicateString(\_In\_opt\_ HSTRING original, \_Outptr\_result\_maybenull\_ \_Result\_nullonfailure\_ HSTRING \*newString)

{

//First, we attempt to use the WinStringRT api encapsulated in the globalization dll; if it is available then it is a downlevel dll.

//Otherwise; we might run into an error where we are using the Win8 (because testing is being done for instance) with the downlevel dll, and that would cause errors.

if(!winRTStringsPresent && winRTStringLibrary->IsAvailable())

{

return winRTStringLibrary->WindowsDuplicateString(original, newString);

}

return DelayLoadWinRtString::WindowsDuplicateString(original, newString);

}

#ifdef ENABLE\_PROJECTION

HRESULT DelayLoadWinRtError::RoClearError()

{

if (m\_hModule)

{

if (m\_pfnRoClearError == nullptr)

{

m\_pfnRoClearError = (PFNCRoClearError)GetFunction("RoClearError");

if (m\_pfnRoClearError == nullptr)

{

return E\_UNEXPECTED;

}

}

Assert(m\_pfnRoClearError != nullptr);

m\_pfnRoClearError();

return S\_OK;

}

return E\_NOTIMPL;

}

BOOL DelayLoadWinRtError::RoOriginateLanguageException(\_\_in HRESULT error, \_\_in\_opt HSTRING message, \_\_in IUnknown \* languageException)

{

if (m\_hModule)

{

if (m\_pfnRoOriginateLanguageException == nullptr)

{

m\_pfnRoOriginateLanguageException = (PFNCRoOriginateLanguageException)GetFunction("RoOriginateLanguageException");

if (m\_pfnRoOriginateLanguageException == nullptr)

{

return FALSE;

}

}

Assert(m\_pfnRoOriginateLanguageException != nullptr);

return m\_pfnRoOriginateLanguageException(error, message, languageException);

}

return FALSE;

}

#endif

#ifdef \_CONTROL\_FLOW\_GUARD

// Note. \_\_declspec(guard(nocf)) causes the CFG check to be removed

// inside this function, and is needed only for test binaries (chk and FRETEST)

#if defined(DELAYLOAD\_SET\_CFG\_TARGET)

DECLSPEC\_GUARDNOCF

#endif

BOOL DelayLoadWinCoreMemory::SetProcessCallTargets(\_In\_ HANDLE hProcess,

\_In\_ PVOID VirtualAddress,

\_In\_ SIZE\_T RegionSize,

\_In\_ ULONG NumberOfOffets,

\_In\_reads\_(NumberOfOffets) PCFG\_CALL\_TARGET\_INFO OffsetInformation)

{

#if defined(DELAYLOAD\_SET\_CFG\_TARGET)

if (m\_hModule)

{

if (m\_pfnSetProcessValidCallTargets == nullptr)

{

m\_pfnSetProcessValidCallTargets = (PFNCSetProcessValidCallTargets) GetFunction("SetProcessValidCallTargets");

if (m\_pfnSetProcessValidCallTargets == nullptr)

{

return FALSE;

}

}

Assert(m\_pfnSetProcessValidCallTargets != nullptr);

return m\_pfnSetProcessValidCallTargets(hProcess, VirtualAddress, RegionSize, NumberOfOffets, OffsetInformation);

}

return FALSE;

#else

return SetProcessValidCallTargets(hProcess, VirtualAddress, RegionSize, NumberOfOffets, OffsetInformation);

#endif

}

#endif

BOOL DelayLoadWinCoreProcessThreads::GetMitigationPolicyForProcess(

\_\_in HANDLE hProcess,

\_\_in PROCESS\_MITIGATION\_POLICY MitigationPolicy,

\_\_out\_bcount(nLength) PVOID lpBuffer,

\_\_in SIZE\_T nLength

)

{

#if defined(DELAYLOAD\_SET\_CFG\_TARGET)

if (m\_hModule)

{

if (m\_pfnGetProcessMitigationPolicy == nullptr)

{

m\_pfnGetProcessMitigationPolicy = (PFNCGetMitigationPolicyForProcess) GetFunction("GetProcessMitigationPolicy");

if (m\_pfnGetProcessMitigationPolicy == nullptr)

{

return FALSE;

}

}

Assert(m\_pfnGetProcessMitigationPolicy != nullptr);

return m\_pfnGetProcessMitigationPolicy(hProcess, MitigationPolicy, lpBuffer, nLength);

}

return FALSE;

#else

return BinaryFeatureControl::GetMitigationPolicyForProcess(hProcess, MitigationPolicy, lpBuffer, nLength);

#endif // ENABLE\_DEBUG\_CONFIG\_OPTIONS

}

BOOL DelayLoadWinCoreProcessThreads::GetProcessInformation(

\_\_in HANDLE hProcess,

\_\_in PROCESS\_INFORMATION\_CLASS ProcessInformationClass,

\_\_out\_bcount(nLength) PVOID lpBuffer,

\_\_in SIZE\_T nLength

)

{

#if defined(DELAYLOAD\_SET\_CFG\_TARGET) || defined(\_M\_ARM)

if (m\_hModule)

{

if (m\_pfnGetProcessInformation == nullptr)

{

m\_pfnGetProcessInformation = (PFNCGetProcessInformation) GetFunction("GetProcessInformation");

if (m\_pfnGetProcessInformation == nullptr)

{

return FALSE;

}

}

Assert(m\_pfnGetProcessInformation != nullptr);

return m\_pfnGetProcessInformation(hProcess, ProcessInformationClass, lpBuffer, nLength);

}

#endif

return FALSE;

}

// Implement this function inlined so that WinRT.lib can be used without the runtime.

HRESULT DelayLoadWinType::RoGetMetaDataFile(

\_In\_ const HSTRING name,

\_In\_opt\_ IMetaDataDispenserEx \*metaDataDispenser,

\_Out\_opt\_ HSTRING \*metaDataFilePath,

\_Outptr\_opt\_ IMetaDataImport2 \*\*metaDataImport,

\_Out\_opt\_ mdTypeDef \*typeDefToken)

{

if (m\_hModule)

{

if (m\_pfnRoGetMetadataFile == nullptr)

{

m\_pfnRoGetMetadataFile = (PFNCWRoGetMettadataFile)GetFunction("RoGetMetaDataFile");

if (m\_pfnRoGetMetadataFile == nullptr)

{

return E\_UNEXPECTED;

}

}

Assert(m\_pfnRoGetMetadataFile != nullptr);

return m\_pfnRoGetMetadataFile(name, metaDataDispenser, metaDataFilePath, metaDataImport, typeDefToken);

}

return E\_NOTIMPL;

}

}

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

// 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

//#include <roapi.h>

#include "activation.h"

#include <winstring.h>

// cor.h includes corhdr.h which is not clean with warning 4091

#pragma warning(push)

#pragma warning(disable: 4091) /\* warning C4091: 'typedef ': ignored on left of '' when no variable is declared \*/

#include <cor.h>

#pragma warning(pop)

#include "RoParameterizedIID.h"

namespace Js

{

class DelayLoadWinRtString : public DelayLoadLibrary

{

private:

// WinRTString specific functions

typedef HRESULT FNCWindowsCreateString(const WCHAR \*, UINT32, \_Outptr\_result\_maybenull\_ \_Result\_nullonfailure\_ HSTRING \*);

typedef FNCWindowsCreateString\* PFNCWindowsCreateString;

PFNCWindowsCreateString m\_pfnWindowsCreateString;

typedef HRESULT FNCWindowsCreateStringReference(const WCHAR \*, UINT32, HSTRING\_HEADER \*, \_Outptr\_result\_maybenull\_ \_Result\_nullonfailure\_ HSTRING \*);

typedef FNCWindowsCreateStringReference\* PFNCWindowsCreateStringReference;

PFNCWindowsCreateStringReference m\_pfnWindowsCreateStringReference;

typedef PCWSTR FNCWindowsGetStringRawBuffer(HSTRING, UINT32\*);

typedef FNCWindowsGetStringRawBuffer\* PFNCWindowsGetStringRawBuffer;

PFNCWindowsGetStringRawBuffer m\_pfWindowsGetStringRawBuffer;

typedef HRESULT FNCWindowsDeleteString(HSTRING);

typedef FNCWindowsDeleteString\* PFNCWindowsDeleteString;

PFNCWindowsDeleteString m\_pfnWindowsDeleteString;

typedef HRESULT FNCWindowsCompareStringOrdinal(HSTRING,HSTRING,INT32\*);

typedef FNCWindowsCompareStringOrdinal\* PFNCWindowsCompareStringOrdinal;

PFNCWindowsCompareStringOrdinal m\_pfnWindowsCompareStringOrdinal;

typedef HRESULT FNCWindowsDuplicateString(HSTRING, \_Outptr\_result\_maybenull\_ \_Result\_nullonfailure\_ HSTRING\*);

typedef FNCWindowsDuplicateString\* PFNCWindowsDuplicateString;

PFNCWindowsDuplicateString m\_pfnWindowsDuplicateString;

public:

DelayLoadWinRtString() : DelayLoadLibrary(),

m\_pfnWindowsCreateString(NULL),

m\_pfWindowsGetStringRawBuffer(NULL),

m\_pfnWindowsDeleteString(NULL),

m\_pfnWindowsCreateStringReference(NULL),

m\_pfnWindowsDuplicateString(NULL),

m\_pfnWindowsCompareStringOrdinal(NULL) { }

virtual ~DelayLoadWinRtString() { }

LPCTSTR GetLibraryName() const { return L"api-ms-win-core-winrt-string-l1-1-0.dll"; }

virtual HRESULT WindowsCreateString(\_In\_reads\_opt\_(length) const WCHAR \* sourceString, UINT32 length, \_Outptr\_result\_maybenull\_ \_Result\_nullonfailure\_ HSTRING \* string);

virtual HRESULT WindowsCreateStringReference(\_In\_reads\_opt\_(length + 1) const WCHAR \* sourceString, UINT32 length, \_Out\_ HSTRING\_HEADER \* header, \_Outptr\_result\_maybenull\_ \_Result\_nullonfailure\_ HSTRING \* string);

virtual HRESULT WindowsDeleteString(\_In\_opt\_ HSTRING string);

virtual PCWSTR WindowsGetStringRawBuffer(\_In\_opt\_ HSTRING string, \_Out\_opt\_ UINT32 \* length);

virtual HRESULT WindowsCompareStringOrdinal(\_In\_opt\_ HSTRING string1, \_In\_opt\_ HSTRING string2, \_Out\_ INT32 \* result);

virtual HRESULT WindowsDuplicateString(\_In\_opt\_ HSTRING original, \_Outptr\_result\_maybenull\_ \_Result\_nullonfailure\_ HSTRING \* newString);

};

class DelayLoadWinRtTypeResolution sealed : public DelayLoadLibrary

{

private:

// WinRtTypeResolution specific functions

typedef HRESULT FNCWRoParseTypeName(HSTRING, DWORD \*, HSTRING \*\*);

typedef FNCWRoParseTypeName\* PFNCWRoParseTypeName;

PFNCWRoParseTypeName m\_pfnRoParseTypeName;

typedef HRESULT FNCRoResolveNamespace(const HSTRING, const HSTRING, const DWORD, const HSTRING\*, DWORD\*, HSTRING\*\*, DWORD\*, HSTRING\*\*);

typedef FNCRoResolveNamespace\* PFNCRoResolveNamespace;

PFNCRoResolveNamespace m\_pfnRoResolveNamespace;

public:

DelayLoadWinRtTypeResolution() : DelayLoadLibrary(),

m\_pfnRoParseTypeName(nullptr) { }

virtual ~DelayLoadWinRtTypeResolution() { }

LPCTSTR GetLibraryName() const { return L"api-ms-win-ro-typeresolution-l1-1-0.dll"; }

HRESULT RoParseTypeName(\_\_in HSTRING typeName, \_\_out DWORD \*partsCount, \_\_RPC\_\_deref\_out\_ecount\_full\_opt(\*partsCount) HSTRING \*\*typeNameParts);

HRESULT RoResolveNamespace(

\_\_in\_opt const HSTRING namespaceName,

\_\_in\_opt const HSTRING windowsMetaDataPath,

\_\_in const DWORD packageGraphPathsCount,

\_\_in\_opt const HSTRING \*packageGraphPaths,

\_\_out DWORD \*metaDataFilePathsCount,

HSTRING \*\*metaDataFilePaths,

\_\_out DWORD \*subNamespacesCount,

HSTRING \*\*subNamespaces);

};

class DelayLoadWinType sealed : public DelayLoadLibrary

{

private:

// WinRtTypeResolution specific functions

typedef HRESULT FNCWRoGetMetaDataFile(

\_In\_ const HSTRING name,

\_In\_opt\_ IMetaDataDispenserEx \*metaDataDispenser,

\_Out\_opt\_ HSTRING \*metaDataFilePath,

\_Outptr\_opt\_ IMetaDataImport2 \*\*metaDataImport,

\_Out\_opt\_ mdTypeDef \*typeDefToken);

typedef FNCWRoGetMetaDataFile\* PFNCWRoGetMettadataFile;

PFNCWRoGetMettadataFile m\_pfnRoGetMetadataFile;

public:

DelayLoadWinType() : DelayLoadLibrary(),

m\_pfnRoGetMetadataFile(nullptr) { }

virtual ~DelayLoadWinType() { }

LPCTSTR GetLibraryName() const { return L"wintypes.dll"; }

HRESULT WINAPI RoGetMetaDataFile(

\_In\_ const HSTRING name,

\_In\_opt\_ IMetaDataDispenserEx \*metaDataDispenser,

\_Out\_opt\_ HSTRING \*metaDataFilePath,

\_Outptr\_opt\_ IMetaDataImport2 \*\*metaDataImport,

\_Out\_opt\_ mdTypeDef \*typeDefToken);

};

class DelayLoadWinRtRoParameterizedIID sealed : public DelayLoadLibrary

{

private:

// WinRtRoParameterizedIID specific functions

typedef HRESULT FNCWRoGetParameterizedTypeInstanceIID(UINT32, PCWSTR\*, const IRoMetaDataLocator&, GUID\*, ROPARAMIIDHANDLE\*);

typedef FNCWRoGetParameterizedTypeInstanceIID\* PFNCWRoGetParameterizedTypeInstanceIID;

PFNCWRoGetParameterizedTypeInstanceIID m\_pfnRoGetParameterizedTypeInstanceIID;

public:

DelayLoadWinRtRoParameterizedIID() : DelayLoadLibrary(),

m\_pfnRoGetParameterizedTypeInstanceIID(nullptr) { }

virtual ~DelayLoadWinRtRoParameterizedIID() { }

LPCTSTR GetLibraryName() const { return L"api-ms-win-core-winrt-roparameterizediid-l1-1-0.dll"; }

HRESULT RoGetParameterizedTypeInstanceIID(

\_\_in UINT32 nameElementCount,

\_\_in\_ecount(nameElementCount) PCWSTR\* nameElements,

\_\_in const IRoMetaDataLocator& metaDataLocator,

\_\_out GUID\* iid,

\_\_deref\_opt\_out ROPARAMIIDHANDLE\* pExtra = nullptr);

};

class DelayLoadWindowsGlobalization sealed : public DelayLoadWinRtString

{

private:

// DelayLoadWindowsGlobalization specific functions

typedef HRESULT FNCWDllGetActivationFactory(HSTRING clsid, IActivationFactory\*\* factory);

typedef FNCWDllGetActivationFactory\* PFNCWDllGetActivationFactory;

PFNCWDllGetActivationFactory m\_pfnFNCWDllGetActivationFactory;

Js::DelayLoadWinRtString \*winRTStringLibrary;

BOOL winRTStringsPresent;

public:

DelayLoadWindowsGlobalization() : DelayLoadWinRtString(),

m\_pfnFNCWDllGetActivationFactory(nullptr),

winRTStringLibrary(nullptr),

winRTStringsPresent(false) { }

virtual ~DelayLoadWindowsGlobalization() { }

LPCTSTR GetLibraryName() const

{

return L"windows.globalization.dll";

}

LPCTSTR GetWin7LibraryName() const

{

return L"jsIntl.dll";

}

void Ensure(Js::DelayLoadWinRtString \*winRTStringLibrary);

HRESULT DllGetActivationFactory(\_\_in HSTRING activatibleClassId, \_\_out IActivationFactory\*\* factory);

HRESULT WindowsCreateString(\_In\_reads\_opt\_(length) const WCHAR \* sourceString, UINT32 length, \_Outptr\_result\_maybenull\_ \_Result\_nullonfailure\_ HSTRING \* string) override;

HRESULT WindowsCreateStringReference(\_In\_reads\_opt\_(length+1) const WCHAR \* sourceString, UINT32 length, \_Out\_ HSTRING\_HEADER \* header, \_Outptr\_result\_maybenull\_ \_Result\_nullonfailure\_ HSTRING \* string) override;

HRESULT WindowsDeleteString(\_In\_opt\_ HSTRING string) override;

PCWSTR WindowsGetStringRawBuffer(\_In\_opt\_ HSTRING string, \_Out\_opt\_ UINT32 \* length) override;

HRESULT WindowsCompareStringOrdinal(\_In\_opt\_ HSTRING string1, \_In\_opt\_ HSTRING string2, \_Out\_ INT32 \* result) override;

HRESULT WindowsDuplicateString(\_In\_opt\_ HSTRING original, \_Outptr\_result\_maybenull\_ \_Result\_nullonfailure\_ HSTRING \*newString) override;

};

class DelayLoadWinRtFoundation sealed : public DelayLoadLibrary

{

private:

// DelayLoadWindowsFoundation specific functions

typedef HRESULT FNCWRoGetActivationFactory(HSTRING clsid, REFIID iid, IActivationFactory\*\* factory);

typedef FNCWRoGetActivationFactory\* PFNCWRoGetActivationFactory;

PFNCWRoGetActivationFactory m\_pfnFNCWRoGetActivationFactory;

public:

DelayLoadWinRtFoundation() : DelayLoadLibrary(),

m\_pfnFNCWRoGetActivationFactory(nullptr) { }

virtual ~DelayLoadWinRtFoundation() { }

LPCTSTR GetLibraryName() const { return L"api-ms-win-core-winrt-l1-1-0.dll"; }

HRESULT RoGetActivationFactory(

\_\_in HSTRING activatibleClassId,

\_\_in REFIID iid,

\_\_out IActivationFactory\*\* factory);

};

#ifdef ENABLE\_PROJECTION

class DelayLoadWinRtError sealed : public DelayLoadLibrary

{

private:

// DelayLoadWinRtError specific functions

typedef void FNCRoClearError();

typedef FNCRoClearError\* PFNCRoClearError;

PFNCRoClearError m\_pfnRoClearError;

typedef BOOL FNCRoOriginateLanguageException(HRESULT, HSTRING, IUnknown \*);

typedef FNCRoOriginateLanguageException\* PFNCRoOriginateLanguageException;

PFNCRoOriginateLanguageException m\_pfnRoOriginateLanguageException;

public:

DelayLoadWinRtError() : DelayLoadLibrary(),

m\_pfnRoClearError(nullptr),

m\_pfnRoOriginateLanguageException(nullptr) { }

virtual ~DelayLoadWinRtError() { }

LPCTSTR GetLibraryName() const { return L"api-ms-win-core-winrt-error-l1-1-1.dll"; }

HRESULT RoClearError();

BOOL RoOriginateLanguageException(\_\_in HRESULT error, \_\_in\_opt HSTRING message, \_\_in IUnknown \* languageException);

};

#endif

#if defined(\_CONTROL\_FLOW\_GUARD)

class DelayLoadWinCoreMemory sealed : public DelayLoadLibrary

{

private:

// LoadWinCoreMemory specific functions

typedef BOOL FNCSetProcessValidCallTargets(HANDLE, PVOID, SIZE\_T, ULONG, PCFG\_CALL\_TARGET\_INFO);

typedef FNCSetProcessValidCallTargets\* PFNCSetProcessValidCallTargets;

PFNCSetProcessValidCallTargets m\_pfnSetProcessValidCallTargets;

public:

DelayLoadWinCoreMemory() : DelayLoadLibrary(),

m\_pfnSetProcessValidCallTargets(nullptr) { }

LPCTSTR GetLibraryName() const { return L"api-ms-win-core-memory-l1-1-3.dll"; }

BOOL SetProcessCallTargets(

\_In\_ HANDLE hProcess,

\_In\_ PVOID VirtualAddress,

\_In\_ SIZE\_T RegionSize,

\_In\_ ULONG NumberOfOffets,

\_In\_reads\_(NumberOfOffets) PCFG\_CALL\_TARGET\_INFO OffsetInformation

);

};

#endif

class DelayLoadWinCoreProcessThreads sealed : public DelayLoadLibrary

{

private:

// LoadWinCoreMemory specific functions

typedef BOOL FNCGetMitigationPolicyForProcess(HANDLE, PROCESS\_MITIGATION\_POLICY, PVOID, SIZE\_T);

typedef FNCGetMitigationPolicyForProcess\* PFNCGetMitigationPolicyForProcess;

PFNCGetMitigationPolicyForProcess m\_pfnGetProcessMitigationPolicy;

typedef BOOL FNCGetProcessInformation(HANDLE, PROCESS\_INFORMATION\_CLASS, PVOID, SIZE\_T);

typedef FNCGetProcessInformation\* PFNCGetProcessInformation;

PFNCGetProcessInformation m\_pfnGetProcessInformation;

public:

DelayLoadWinCoreProcessThreads() :

DelayLoadLibrary(),

m\_pfnGetProcessMitigationPolicy(nullptr),

m\_pfnGetProcessInformation(nullptr)

{

}

LPCTSTR GetLibraryName() const { return L"api-ms-win-core-processthreads-l1-1-3.dll"; }

BOOL GetMitigationPolicyForProcess(

\_\_in HANDLE hProcess,

\_\_in PROCESS\_MITIGATION\_POLICY MitigationPolicy,

\_\_out\_bcount(nLength) PVOID lpBuffer,

\_\_in SIZE\_T nLength

);

BOOL GetProcessInformation(

\_\_in HANDLE hProcess,

\_\_in PROCESS\_INFORMATION\_CLASS ProcessInformationClass,

\_\_out\_bcount(nLength) PVOID lpBuffer,

\_\_in SIZE\_T nLength

);

};

// Implement this function inlined so that WinRT.lib can be used without the runtime.

inline HRESULT DelayLoadWinRtRoParameterizedIID::RoGetParameterizedTypeInstanceIID(

\_\_in UINT32 nameElementCount,

\_\_in\_ecount(nameElementCount) PCWSTR\* nameElements,

\_\_in const IRoMetaDataLocator& metaDataLocator,

\_\_out GUID\* iid,

\_\_deref\_opt\_out ROPARAMIIDHANDLE\* pExtra)

{

if (m\_hModule)

{

if (m\_pfnRoGetParameterizedTypeInstanceIID == NULL)

{

m\_pfnRoGetParameterizedTypeInstanceIID = (PFNCWRoGetParameterizedTypeInstanceIID)GetFunction("RoGetParameterizedTypeInstanceIID");

if (m\_pfnRoGetParameterizedTypeInstanceIID == NULL)

{

return E\_UNEXPECTED;

}

}

Assert(m\_pfnRoGetParameterizedTypeInstanceIID != NULL);

return m\_pfnRoGetParameterizedTypeInstanceIID(nameElementCount, nameElements, metaDataLocator, iid, pExtra);

}

return E\_NOTIMPL;

}

}

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

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

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

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

#include "RuntimeBasePch.h"

const uint32 Entropy::kInitIterationCount = 3;

void Entropy::BeginAdd()

{

previousValue = u.value;

}

void Entropy::AddCurrentTime()

{

LARGE\_INTEGER time = {0};

QueryPerformanceCounter(&time);

if (time.LowPart)

u.value ^= time.LowPart;

}

void Entropy::Add(const char byteValue)

{

if (byteValue)

{

u.array[currentIndex++] ^= byteValue;

currentIndex %= sizeof(unsigned \_\_int32);

}

}

/\* public API \*/

void Entropy::Initialize()

{

for (uint32 times = 0; times < Entropy::kInitIterationCount; times++)

{

AddIoCounters();

AddThreadCycleTime();

}

AddCurrentTime();

}

void Entropy::Add(const char \*buffer, size\_t size)

{

BeginAdd();

for (size\_t index = 0; index < size; index++)

{

Add(buffer[index]);

}

}

void Entropy::AddIoCounters()

{

IO\_COUNTERS ioc = {0};

if (GetProcessIoCounters(GetCurrentProcess(), &ioc))

{

Add((char \*)&ioc.ReadOperationCount, sizeof(ioc.ReadOperationCount));

Add((char \*)&ioc.WriteOperationCount, sizeof(ioc.WriteOperationCount));

Add((char \*)&ioc.OtherOperationCount, sizeof(ioc.OtherOperationCount));

Add((char \*)&ioc.ReadTransferCount, sizeof(ioc.ReadTransferCount));

Add((char \*)&ioc.WriteTransferCount, sizeof(ioc.WriteTransferCount));

Add((char \*)&ioc.OtherTransferCount, sizeof(ioc.OtherTransferCount));

}

AddCurrentTime();

}

void Entropy::AddThreadCycleTime()

{

LARGE\_INTEGER threadCycleTime = {0};

QueryThreadCycleTime(GetCurrentThread(), (PULONG64)&threadCycleTime);

Add((char \*)&threadCycleTime.LowPart, sizeof(threadCycleTime.LowPart));

AddCurrentTime();

}

unsigned \_\_int64 Entropy::GetRand() const

{

return (((unsigned \_\_int64)previousValue) << 32) | u.value;

}

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

// 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

/\*

\* This class accumulates entropy from sources such as the process' IO counts

\* and the thread's cycle counter (i.e. number of cycles spent in user + kernel

\* mode) and is used to improve the entropy of the seed used by Math.random.

\*/

class Entropy {

public:

Entropy()

{

previousValue = 0;

u.value = 0;

currentIndex = 0;

}

void Initialize();

void Add(const char \*buffer, size\_t size);

void AddIoCounters();

void AddThreadCycleTime();

unsigned \_\_int64 GetRand() const;

private:

unsigned \_\_int32 previousValue;

union

{

unsigned \_\_int32 value;

char array[sizeof(unsigned \_\_int32)];

} u;

size\_t currentIndex;

static const uint32 kInitIterationCount;

void BeginAdd();

void Add(const char byteValue);

void AddCurrentTime();

};

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

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

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

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

#include "RuntimeBasePch.h"

#ifdef ENABLE\_JS\_ETW

#include "core\EtwTraceCore.h"

#include "Base\EtwTrace.h"

#ifdef VTUNE\_PROFILING

#ifdef CDECL

#define ORIGINAL\_CDECL CDECL

#undef CDECL

#endif

// Not enabled in ChakraCore

#include "jitProfiling.h"

#ifdef ORIGINAL\_CDECL

#undef CDECL

#endif

#define CDECL ORIGINAL\_CDECL

#endif

using namespace Js;

//

// This C style callback is invoked by ETW when a trace session is started/stopped

// by an ETW controller for the Jscript and MSHTML providers.

//

static const char LoopStr[] = "Loop";

static const wchar\_t LoopWStr[] = L"Loop";

void EtwCallbackApi::OnSessionChange(ULONG controlCode, PVOID callbackContext)

{

PMCGEN\_TRACE\_CONTEXT context = (PMCGEN\_TRACE\_CONTEXT)callbackContext;

// A manifest based provider can be enabled to multiple event tracing sessions

// As long as there is at least 1 enabled session, isEnabled will be TRUE

// We only care about Jscript events.

if(context->RegistrationHandle == Microsoft\_JScriptHandle)

{

switch(controlCode)

{

case EVENT\_CONTROL\_CODE\_ENABLE\_PROVIDER:

case EVENT\_CONTROL\_CODE\_CAPTURE\_STATE:

if(McGenLevelKeywordEnabled(context,

TRACE\_LEVEL\_INFORMATION,

JSCRIPT\_RUNDOWNSTART\_KEYWORD))

{

EtwTrace::PerformRundown(/\*start\*/ true);

}

if(McGenLevelKeywordEnabled(context,

TRACE\_LEVEL\_INFORMATION,

JSCRIPT\_RUNDOWNEND\_KEYWORD))

{

EtwTrace::PerformRundown(/\*start\*/ false);

}

break;

case EVENT\_CONTROL\_CODE\_DISABLE\_PROVIDER:

break; // Do Nothing

}

}

}

#ifdef VTUNE\_PROFILING

const utf8char\_t EtwTrace::DynamicCode[] = "Dynamic code";

bool EtwTrace::isJitProfilingActive = false;

#endif

//

// Registers the ETW provider - this is usually done on Jscript DLL load

// After registration, we will receive callbacks when ETW tracing is enabled/disabled.

//

void EtwTrace::Register()

{

EtwTraceCore::Register();

#ifdef TEST\_ETW\_EVENTS

TestEtwEventSink::Load();

#endif

#ifdef VTUNE\_PROFILING

isJitProfilingActive = (iJIT\_IsProfilingActive() == iJIT\_SAMPLING\_ON);

#endif

}

//

// Unregister to ensure we do not get callbacks.

//

void EtwTrace::UnRegister()

{

EtwTraceCore::UnRegister();

#ifdef TEST\_ETW\_EVENTS

TestEtwEventSink::Unload();

#endif

#ifdef VTUNE\_PROFILING

if(isJitProfilingActive)

{

iJIT\_NotifyEvent(iJVM\_EVENT\_TYPE\_SHUTDOWN, NULL);

}

#endif

}

//

// Enumerate through all the script contexts in the process and log events

// for each function loaded. Depending on the argument, start or end events are logged.

// In particular, a rundown is needed for the 'Attach' scenario of profiling.

//

void EtwTrace::PerformRundown(bool start)

{

// Lock threadContext list during etw rundown

AutoCriticalSection autoThreadContextCs(ThreadContext::GetCriticalSection());

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

if(start)

{

JS\_ETW(EventWriteDCStartInit());

}

else

{

JS\_ETW(EventWriteDCEndInit());

}

while(threadContext != nullptr)

{

// Take etw rundown lock on this thread context

AutoCriticalSection autoEtwRundownCs(threadContext->GetEtwRundownCriticalSection());

ScriptContext\* scriptContext = threadContext->GetScriptContextList();

while(scriptContext != NULL)

{

if(scriptContext->IsClosed())

{

scriptContext = scriptContext->next;

continue;

}

if(start)

{

JS\_ETW(EventWriteScriptContextDCStart(scriptContext));

if(scriptContext->GetSourceContextInfoMap() != nullptr)

{

scriptContext->GetSourceContextInfoMap()->Map( [=] (DWORD\_PTR sourceContext, SourceContextInfo \* sourceContextInfo)

{

if (sourceContext != Constants::NoHostSourceContext)

{

JS\_ETW(LogSourceEvent(EventWriteSourceDCStart,

sourceContext,

scriptContext,

/\* sourceFlags\*/ 0,

sourceContextInfo->url));

}

});

}

}

else

{

JS\_ETW(EventWriteScriptContextDCEnd(scriptContext));

if(scriptContext->GetSourceContextInfoMap() != nullptr)

{

scriptContext->GetSourceContextInfoMap()->Map( [=] (DWORD\_PTR sourceContext, SourceContextInfo \* sourceContextInfo)

{

if (sourceContext != Constants::NoHostSourceContext)

{

JS\_ETW(LogSourceEvent(EventWriteSourceDCEnd,

sourceContext,

scriptContext,

/\* sourceFlags\*/ 0,

sourceContextInfo->url));

}

});

}

}

scriptContext->MapFunction([&start] (FunctionBody\* body)

{

if(body->HasInterpreterThunkGenerated())

{

if(start)

{

LogMethodInterpretedThunkEvent(EventWriteMethodDCStart, body);

}

else

{

LogMethodInterpretedThunkEvent(EventWriteMethodDCEnd, body);

}

}

body->MapEntryPoints([&](int index, FunctionEntryPointInfo \* entryPoint)

{

if(entryPoint->IsCodeGenDone())

{

if (start)

{

LogMethodNativeEvent(EventWriteMethodDCStart, body, entryPoint);

}

else

{

LogMethodNativeEvent(EventWriteMethodDCEnd, body, entryPoint);

}

}

});

body->MapLoopHeaders([&](uint loopNumber, LoopHeader\* header)

{

header->MapEntryPoints([&](int index, LoopEntryPointInfo \* entryPoint)

{

if(entryPoint->IsCodeGenDone())

{

if(start)

{

LogLoopBodyEvent(EventWriteMethodDCStart, body, header, entryPoint);

}

else

{

LogLoopBodyEvent(EventWriteMethodDCEnd, body, header, entryPoint);

}

}

});

});

});

scriptContext = scriptContext->next;

}

if (EventEnabledJSCRIPT\_HOSTING\_CEO\_START())

{

threadContext->EtwLogPropertyIdList();

}

threadContext = threadContext->Next();

}

if(start)

{

JS\_ETW(EventWriteDCStartComplete());

}

else

{

JS\_ETW(EventWriteDCEndComplete());

}

}

//

// Returns an ID for the source file of the function.

//

DWORD\_PTR EtwTrace::GetSourceId(FunctionBody\* body)

{

DWORD\_PTR sourceId = body->GetHostSourceContext();

// For dynamic scripts - use fixed source ID of -1.

// TODO: Find a way to generate unique ID for dynamic scripts.

if(sourceId == Js::Constants::NoHostSourceContext)

{

sourceId = (DWORD\_PTR)-1;

}

return sourceId;

}

//

// Returns an ID to identify the function.

//

uint EtwTrace::GetFunctionId(FunctionProxy\* body)

{

return body->GetFunctionNumber();

}

void EtwTrace::LogSourceUnloadEvents(ScriptContext\* scriptContext)

{

if(scriptContext->GetSourceContextInfoMap() != nullptr)

{

scriptContext->GetSourceContextInfoMap()->Map( [&] (DWORD\_PTR sourceContext, SourceContextInfo \* sourceContextInfo)

{

if(sourceContext != Constants::NoHostSourceContext)

{

JS\_ETW(LogSourceEvent(EventWriteSourceUnload,

sourceContext,

scriptContext,

/\* sourceFlags\*/ 0,

sourceContextInfo->url));

}

});

}

JS\_ETW(EventWriteScriptContextUnload(scriptContext));

}

void EtwTrace::LogMethodInterpreterThunkLoadEvent(FunctionBody\* body)

{

LogMethodInterpretedThunkEvent(EventWriteMethodLoad, body);

}

void EtwTrace::LogMethodNativeLoadEvent(FunctionBody\* body, FunctionEntryPointInfo\* entryPoint)

{

LogMethodNativeEvent(EventWriteMethodLoad, body, entryPoint);

#ifdef VTUNE\_PROFILING

if(isJitProfilingActive)

{

iJIT\_Method\_Load methodInfo;

memset(&methodInfo, 0, sizeof(iJIT\_Method\_Load));

const wchar\_t\* methodName = body->GetExternalDisplayName();

// Append function line number info to method name so that VTune can distinguish between polymorphic methods

wchar\_t methodNameBuffer[\_MAX\_PATH];

ULONG lineNumber = body->GetLineNumber();

wchar\_t numberBuffer[20];

\_ltow\_s(lineNumber, numberBuffer, 10);

wcscpy\_s(methodNameBuffer, methodName);

if(entryPoint->GetJitMode() == ExecutionMode::SimpleJit)

{

wcscat\_s(methodNameBuffer, L" Simple");

}

wcscat\_s(methodNameBuffer, L" {line:");

wcscat\_s(methodNameBuffer, numberBuffer);

wcscat\_s(methodNameBuffer, L"}");

size\_t methodLength = wcslen(methodNameBuffer);

Assert(methodLength < \_MAX\_PATH);

size\_t length = methodLength \* 3 + 1;

utf8char\_t\* utf8MethodName = HeapNewNoThrowArray(utf8char\_t, length);

if(utf8MethodName)

{

methodInfo.method\_id = iJIT\_GetNewMethodID();

utf8::EncodeIntoAndNullTerminate(utf8MethodName, methodNameBuffer, (charcount\_t)methodLength);

methodInfo.method\_name = (char\*)utf8MethodName;

methodInfo.method\_load\_address = (void\*)entryPoint->GetNativeAddress();

methodInfo.method\_size = (uint)entryPoint->GetCodeSize(); // Size in memory - Must be exact

LineNumberInfo numberInfo[1];

uint lineCount = (entryPoint->GetNativeOffsetMapCount()) \* 2 + 1; // may need to record both .begin and .end for all elements

LineNumberInfo\* pLineInfo = HeapNewNoThrowArray(LineNumberInfo, lineCount);

if (pLineInfo == NULL || Js::Configuration::Global.flags.DisableVTuneSourceLineInfo)

{

// resort to original implementation, attribute all samples to first line

numberInfo[0].LineNumber = lineNumber;

numberInfo[0].Offset = 0;

methodInfo.line\_number\_size = 1;

methodInfo.line\_number\_table = numberInfo;

}

else

{

int size = entryPoint->PopulateLineInfo(pLineInfo, body);

methodInfo.line\_number\_size = size;

methodInfo.line\_number\_table = pLineInfo;

}

size\_t urlLength = 0;

utf8char\_t\* utf8Url = GetUrl(body, &urlLength);

methodInfo.source\_file\_name = (char\*)utf8Url;

methodInfo.env = iJDE\_JittingAPI;

OUTPUT\_TRACE(Js::ProfilerPhase, L"Method load event: %s\n", methodNameBuffer);

iJIT\_NotifyEvent(iJVM\_EVENT\_TYPE\_METHOD\_LOAD\_FINISHED, &methodInfo);

HeapDeleteArray(lineCount, pLineInfo);

if(urlLength > 0)

{

HeapDeleteArray(urlLength, utf8Url);

}

HeapDeleteArray(length, utf8MethodName);

}

}

#endif

}

void EtwTrace::LogLoopBodyLoadEvent(FunctionBody\* body, LoopHeader\* loopHeader, LoopEntryPointInfo\* entryPoint)

{

LogLoopBodyEvent(EventWriteMethodLoad, body, loopHeader, entryPoint);

#ifdef VTUNE\_PROFILING

if(isJitProfilingActive)

{

iJIT\_Method\_Load methodInfo;

memset(&methodInfo, 0, sizeof(iJIT\_Method\_Load));

const wchar\_t\* methodName = body->GetExternalDisplayName();

size\_t methodLength = wcslen(methodName);

methodLength = min(methodLength, (size\_t)UINT\_MAX); // Just truncate if it is too big

size\_t length = methodLength \* 3 + /\* spaces \*/ 2 + \_countof(LoopStr) + /\*size of loop number\*/ 10 + /\*NULL\*/ 1;

utf8char\_t\* utf8MethodName = HeapNewNoThrowArray(utf8char\_t, length);

if(utf8MethodName)

{

methodInfo.method\_id = iJIT\_GetNewMethodID();

size\_t len = utf8::EncodeInto(utf8MethodName, methodName, (charcount\_t)methodLength);

uint loopNumber = body->GetLoopNumber(loopHeader) + 1;

sprintf\_s((char\*)(utf8MethodName + len), length - len," %s %d", LoopStr, loopNumber);

methodInfo.method\_name = (char\*)utf8MethodName;

methodInfo.method\_load\_address = (void\*)entryPoint->GetNativeAddress();

methodInfo.method\_size = (uint)entryPoint->GetCodeSize(); // Size in memory - Must be exact

size\_t urlLength = 0;

utf8char\_t\* utf8Url = GetUrl(body, &urlLength);

methodInfo.source\_file\_name = (char\*)utf8Url;

methodInfo.env = iJDE\_JittingAPI;

iJIT\_NotifyEvent(iJVM\_EVENT\_TYPE\_METHOD\_LOAD\_FINISHED, &methodInfo);

OUTPUT\_TRACE(Js::ProfilerPhase, L"Loop body load event: %s Loop %d\n", methodName, loopNumber);

if(urlLength > 0)

{

HeapDeleteArray(urlLength, utf8Url);

}

HeapDeleteArray(length, utf8MethodName);

}

}

#endif

}

void EtwTrace::LogMethodInterpreterThunkUnloadEvent(FunctionBody\* body)

{

LogMethodInterpretedThunkEvent(EventWriteMethodUnload, body);

}

void EtwTrace::LogMethodNativeUnloadEvent(FunctionBody\* body, FunctionEntryPointInfo\* entryPoint)

{

LogMethodNativeEvent(EventWriteMethodUnload, body, entryPoint);

}

void EtwTrace::LogLoopBodyUnloadEvent(FunctionBody\* body, LoopHeader\* loopHeader, LoopEntryPointInfo\* entryPoint)

{

LogLoopBodyEvent(EventWriteMethodUnload, body, loopHeader, entryPoint);

}

//

// Logs the runtime script context load event

//

void EtwTrace::LogScriptContextLoadEvent(ScriptContext\* scriptContext)

{

JS\_ETW(EventWriteScriptContextLoad(

scriptContext));

}

//

// Logs the runtime source module load event.

//

void EtwTrace::LogSourceModuleLoadEvent(ScriptContext\* scriptContext, DWORD\_PTR sourceContext, \_In\_z\_ const wchar\_t\* url)

{

AssertMsg(sourceContext != Constants::NoHostSourceContext, "We should not be logged this if there is no source code available");

JS\_ETW(LogSourceEvent(EventWriteSourceLoad,

sourceContext,

scriptContext,

/\* sourceFlags\*/ 0,

url));

}

//

// This emulates the logic used by the F12 profiler to give names to functions

//

const wchar\_t\* EtwTrace::GetFunctionName(FunctionBody\* body)

{

return body->GetExternalDisplayName();

}

size\_t EtwTrace::GetLoopBodyName(\_In\_ FunctionBody\* body, \_In\_ LoopHeader\* loopHeader, \_Out\_writes\_opt\_z\_(size) wchar\_t\* nameBuffer, \_In\_ size\_t size)

{

return body->GetLoopBodyName(body->GetLoopNumber(loopHeader), nameBuffer, size);

}

\_Success\_(return == 0)

size\_t EtwTrace::GetSimpleJitFunctionName(

Js::FunctionBody \*const body,

\_Out\_writes\_opt\_z\_(nameCharCapacity) wchar\_t \*const name,

const size\_t nameCharCapacity)

{

Assert(body);

Assert(name);

Assert(nameCharCapacity != 0);

const wchar\_t \*const suffix = L"Simple";

const size\_t suffixCharLength = \_countof(L"Simple") - 1;

const wchar\_t \*const functionName = GetFunctionName(body);

const size\_t functionNameCharLength = wcslen(functionName);

const size\_t requiredCharCapacity = functionNameCharLength + suffixCharLength + 1;

if(requiredCharCapacity > nameCharCapacity)

{

return requiredCharCapacity;

}

wcscpy\_s(name, nameCharCapacity, functionName);

wcscpy\_s(&name[functionNameCharLength], nameCharCapacity - functionNameCharLength, suffix);

return 0;

}

#ifdef VTUNE\_PROFILING

utf8char\_t\* EtwTrace::GetUrl( FunctionBody\* body, size\_t\* urlBufferLength )

{

utf8char\_t\* utf8Url = NULL;

if(!body->GetSourceContextInfo()->IsDynamic())

{

const wchar\* url = body->GetSourceContextInfo()->url;

if(url)

{

size\_t urlCharLength = wcslen(url);

urlCharLength = min(urlCharLength, (size\_t)UINT\_MAX); // Just truncate if it is too big

\*urlBufferLength = urlCharLength \* 3 + 1;

utf8Url = HeapNewNoThrowArray(utf8char\_t, \*urlBufferLength);

if (utf8Url)

{

utf8::EncodeIntoAndNullTerminate(utf8Url, url, (charcount\_t)urlCharLength);

}

}

}

else

{

utf8Url = (utf8char\_t\*)EtwTrace::DynamicCode;

}

return utf8Url;

}

#endif

#endif

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

// 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 ENABLE\_JS\_ETW

//

// ETW (Event Tracing for Windows) is a high-performance, low overhead and highly scalable

// tracing facility provided by the Windows Operating System. There are

// four main types of components in ETW: event providers, controllers, consumers, and event trace sessions.

// An event provider is a logical entity that writes events to ETW sessions. The event provider must register

// a provider ID with ETW through the registration API. A provider first registers with ETW and writes events

// from various points in the code by invoking the ETW logging API. When a provider is enabled dynamically by

// the ETW controller application, calls to the logging API sends events to a specific trace session

// designated by the controller. Javascript runtime is an event provider.

//

// When ETW's sample profile is enabled it logs the entry point of each function as part of

// stackwalking on the platform. For javascript functions these entry points are not meaningful.

// Therefore, we log events which map the entry point to the name of the function. There are two types of events

// that we log for symbol decoding:

// Rundown: Represents the state of the process and logged when tracing is enabled. Enables the 'attach' scenario.

// Runtime: Represents the change of state of the process and logged as the state changes e.g. when a jitted function gets loaded.

//

// C-style callback

extern "C" {

void EtwCallback(

ULONG controlCode,

PVOID callbackContext);

}

#include "TestEtwEventSink.h"

//

// Represents type of method entry point.

//

enum MethodType : uint16

{

MethodType\_Interpreted = 0x1,

MethodType\_Jit = 0x2,

MethodType\_LoopBody = 0x3

};

#ifdef TEST\_ETW\_EVENTS

#define WriteMethodEvent(EventName, ScriptContextID, MethodStartAddress, MethodSize, MethodID, MethodFlags, MethodAddressRangeID, SourceID, Line, Column, MethodName) \

if(TestEtwEventSink::IsLoaded()) \

{ \

TestEtwEventSink::Instance->WriteMethodEvent(EventName, ScriptContextID, MethodStartAddress, MethodSize, MethodID, MethodFlags, MethodAddressRangeID, SourceID, Line, Column, MethodName); \

}

#define WriteSourceEvent(EventName, SourceContext, ScriptContextID, SourceFlags, Url) \

if(TestEtwEventSink::IsLoaded()) \

{ \

TestEtwEventSink::Instance->WriteSourceEvent(EventName, SourceContext, ScriptContextID, SourceFlags, Url); \

}

#else

#define WriteMethodEvent(Event, ...)

#define WriteSourceEvent(Event, ...)

#endif

// Helper macro to log all the method level events

#define LogSourceEvent(Function, SourceContext, ScriptContext, SourceFlags, Url) \

JS\_ETW(Function(SourceContext, \

ScriptContext, \

SourceFlags, \

Url)); \

\

WriteSourceEvent(STRINGIZEW(Function), \

SourceContext, \

ScriptContext, \

SourceFlags, \

Url);

#define LogMethodNativeEvent(Function, Body, entryPoint) \

Assert(entryPoint->GetNativeAddress() != NULL); \

Assert(entryPoint->GetCodeSize() > 0); \

Assert(entryPoint->IsNativeCode()); \

wchar\_t functionNameArray[NameBufferLength]; \

const wchar\_t \*functionName; \

bool deleteFunctionName = false; \

const ExecutionMode jitMode = entryPoint->GetJitMode(); \

if(jitMode == ExecutionMode::SimpleJit) \

{ \

const size\_t requiredCharCapacity = \

GetSimpleJitFunctionName(Body, functionNameArray, \_countof(functionNameArray)); \

if(requiredCharCapacity == 0) \

{ \

functionName = functionNameArray; \

} \

else \

{ \

Assert(requiredCharCapacity > NameBufferLength); \

wchar\_t \*const allocatedFunctionName = new wchar\_t[requiredCharCapacity]; \

if(allocatedFunctionName) \

{ \

const size\_t newRequiredCharCapacity = \

GetSimpleJitFunctionName(Body, allocatedFunctionName, requiredCharCapacity); \

Assert(newRequiredCharCapacity == 0); \

functionName = allocatedFunctionName; \

deleteFunctionName = true; \

} \

else \

{ \

functionNameArray[0] = L'\0'; \

functionName = functionNameArray; \

} \

} \

} \

else \

{ \

functionName = GetFunctionName(Body); \

} \

JS\_ETW(Function( \

Body->GetScriptContext(), \

(void \*)entryPoint->GetNativeAddress(), \

entryPoint->GetCodeSize(), \

GetFunctionId(Body), \

0 /\* methodFlags - for future use\*/, \

MethodType\_Jit, \

GetSourceId(Body), \

Body->GetLineNumber(), \

Body->GetColumnNumber(), \

functionName)); \

WriteMethodEvent(STRINGIZEW(Function), \

Body->GetScriptContext(), \

(void \*)entryPoint->GetNativeAddress(), \

entryPoint->GetCodeSize(), \

GetFunctionId(Body), \

0 /\* methodFlags - for future use\*/, \

MethodType\_Jit, \

GetSourceId(Body), \

Body->GetLineNumber(), \

Body->GetColumnNumber(), \

functionName); \

if(deleteFunctionName) \

{ \

delete[] functionName; \

}

#define LogMethodInterpretedThunkEvent(Function, Body) \

Assert(Body->GetDynamicInterpreterEntryPoint() != NULL); \

JS\_ETW(Function(Body->GetScriptContext(), \

Body->GetDynamicInterpreterEntryPoint(), \

Body->GetDynamicInterpreterThunkSize(), \

GetFunctionId(Body), \

0 /\* methodFlags - for future use\*/, \

MethodType\_Interpreted, \

GetSourceId(Body), \

Body->GetLineNumber(), \

Body->GetColumnNumber(), \

GetFunctionName(Body))); \

\

WriteMethodEvent(STRINGIZEW(Function), \

Body->GetScriptContext(), \

Body->GetDynamicInterpreterEntryPoint(), \

Body->GetDynamicInterpreterThunkSize(), \

GetFunctionId(Body), \

0 /\* methodFlags - for future use\*/, \

MethodType\_Interpreted, \

GetSourceId(Body), \

Body->GetLineNumber(), \

Body->GetColumnNumber(), \

GetFunctionName(Body))

#define LogLoopBodyEvent(Function, Body, loopHeader, entryPoint) \

Assert(entryPoint->GetNativeAddress() != NULL); \

Assert(entryPoint->GetCodeSize() > 0); \

WCHAR loopBodyNameArray[NameBufferLength]; \

WCHAR\* loopBodyName = loopBodyNameArray; \

size\_t bufferSize = GetLoopBodyName(Body, loopHeader, loopBodyName, NameBufferLength); \

if(bufferSize > NameBufferLength) /\* insufficient buffer space\*/ \

{ \

loopBodyName = new WCHAR[bufferSize]; \

if(loopBodyName) \

{ \

GetLoopBodyName(Body, loopHeader, loopBodyName, bufferSize); \

} \

else \

{ \

loopBodyNameArray[0] = L'\0'; \

loopBodyName = loopBodyNameArray; \

} \

} \

JS\_ETW(Function(Body->GetScriptContext(), \

(void \*)entryPoint->GetNativeAddress(), \

entryPoint->GetCodeSize(), \

GetFunctionId(Body), \

0 /\* methodFlags - for future use\*/, \

MethodType\_LoopBody + (uint16)Body->GetLoopNumber(loopHeader), \

GetSourceId(Body), \

/\*line\*/ 0, \

/\*column\*/ 0, \

loopBodyName)); \

WriteMethodEvent(STRINGIZEW(Function), \

Body->GetScriptContext(), \

(void \*)entryPoint->GetNativeAddress(), \

entryPoint->GetCodeSize(), \

GetFunctionId(Body), \

0 /\* methodFlags - for future use\*/, \

MethodType\_LoopBody + (uint16)Body->GetLoopNumber(loopHeader), \

GetSourceId(Body), \

/\*line\*/ 0, \

/\*column\*/ 0, \

loopBodyName); \

if(loopBodyNameArray != loopBodyName) \

{ \

delete[] loopBodyName; \

}

//

// Encapsulates all ETW event logging and registration related to symbol decoding.

//

class EtwTrace

{

private:

static const size\_t NameBufferLength = 256;

public:

static void Register();

static void UnRegister();

static void PerformRundown(bool start);

/\* Unload events \*/

static void LogSourceUnloadEvents(Js::ScriptContext\* scriptContext);

static void LogMethodNativeUnloadEvent(Js::FunctionBody\* body, Js::FunctionEntryPointInfo\* entryPoint);

static void LogMethodInterpreterThunkUnloadEvent(Js::FunctionBody\* body);

static void LogLoopBodyUnloadEvent(Js::FunctionBody\* body, Js::LoopHeader\* loopHeader, Js::LoopEntryPointInfo\* entryPoint);

/\* Load events \*/

static void LogMethodInterpreterThunkLoadEvent(Js::FunctionBody\* body);

static void LogMethodNativeLoadEvent(Js::FunctionBody\* body, Js::FunctionEntryPointInfo\* entryPoint);

static void LogLoopBodyLoadEvent(Js::FunctionBody\* body, Js::LoopHeader\* loopHeader, Js::LoopEntryPointInfo\* entryPoint);

static void LogScriptContextLoadEvent(Js::ScriptContext\* scriptContext);

static void LogSourceModuleLoadEvent(Js::ScriptContext\* scriptContext, DWORD\_PTR sourceContext, \_In\_z\_ const wchar\_t\* url);

static const wchar\_t\* GetFunctionName(Js::FunctionBody\* body);

static size\_t GetLoopBodyName(\_In\_ Js::FunctionBody\* body, \_In\_ Js::LoopHeader\* loopHeader, \_Out\_writes\_opt\_z\_(size) wchar\_t\* nameBuffer, \_In\_ size\_t size );

\_Success\_(return == 0)

static size\_t GetSimpleJitFunctionName(Js::FunctionBody \*const body, \_Out\_writes\_opt\_z\_(nameCharCapacity) wchar\_t \*const name, const size\_t nameCharCapacity);

static DWORD\_PTR GetSourceId(Js::FunctionBody\* body);

static uint GetFunctionId(Js::FunctionProxy\* body);

#ifdef VTUNE\_PROFILING

static const utf8char\_t DynamicCode[];

static bool isJitProfilingActive;

#endif

private:

#ifdef VTUNE\_PROFILING

static utf8char\_t\* GetUrl(Js::FunctionBody\* body, size\_t\* urlLength);

#endif

};

#endif

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

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

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

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

#include "RuntimeBasePch.h"

#include "Exception.h"

namespace Js

{

bool Exception::RaiseIfScriptActive(ScriptContext \*scriptContext, unsigned kind, PVOID returnAddress)

{

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

if (threadContext != nullptr && threadContext->IsScriptActive())

{

switch (kind) {

case ExceptionKind\_OutOfMemory:

AssertMsg(returnAddress == NULL, "should not have returnAddress passed in");

JavascriptError::ThrowOutOfMemoryError(scriptContext);

case ExceptionKind\_StackOverflow:

JavascriptError::ThrowStackOverflowError(scriptContext, returnAddress);

default:

AssertMsg(false, "Invalid ExceptionKind");

}

}

return false;

}

// Recover/Release unused memory and give it back to OS.

// The function doesn't throw if the attempt to recover memory fails, in which case it simply does nothing.

// Useful when running out of memory e.g. for Arena but there is some recycler memory which has been committed but is unused.

void Exception::RecoverUnusedMemory()

{

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

if (threadContext)

{

Recycler\* threadRecycler = threadContext->GetRecycler();

if (threadRecycler)

{

try

{

threadRecycler->CollectNow<CollectOnRecoverFromOutOfMemory>();

}

catch (...)

{

// Technically, exception is a valid scenario: we asked to recover mem, and it couldn't.

// Do not let the exception leak out.

}

}

}

}

}

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

// 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 Exception

{

friend JsUtil::ExternalApi;

private:

enum {

ExceptionKind\_OutOfMemory,

ExceptionKind\_StackOverflow

};

static bool RaiseIfScriptActive(ScriptContext \*scriptContext, unsigned kind, PVOID returnAddress = NULL);

static void RecoverUnusedMemory();

};

}

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

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

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

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

#include "RuntimeBasePch.h"

ExpirableObject::ExpirableObject(ThreadContext\* threadContext):

isUsed(false),

registrationHandle(nullptr)

{

if (threadContext)

{

threadContext->RegisterExpirableObject(this);

}

}

void ExpirableObject::Finalize(bool isShutdown)

{

if (!isShutdown && this->registrationHandle != nullptr)

{

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

threadContext->UnregisterExpirableObject(this);

}

}

void ExpirableObject::Dispose(bool isShutdown)

{

if (!isShutdown && this->registrationHandle == nullptr)

{

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

threadContext->DisposeExpirableObject(this);

}

}

void ExpirableObject::EnterExpirableCollectMode()

{

this->isUsed = false;

}

bool ExpirableObject::IsObjectUsed()

{

return this->isUsed;

}

void ExpirableObject::SetIsObjectUsed()

{

this->isUsed = true;

}

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

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

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

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

class ThreadContext;

class ExpirableObject: public FinalizableObject

{

public:

friend class ThreadContext;

ExpirableObject(ThreadContext\* threadContext);

virtual void Finalize(bool isShutdown);

virtual void Dispose(bool isShutdown) override;

virtual void Mark(Recycler \*recycler) override { AssertMsg(false, "Mark called on object that isn't TrackableObject"); }

// Called when an expirable object gets expired

virtual void Expire() = 0;

virtual void EnterExpirableCollectMode();

bool IsObjectUsed();

void SetIsObjectUsed();

bool SupportsExpiration()

{

return (registrationHandle != nullptr);

}

private:

void\* registrationHandle;

bool isUsed;

};

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

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

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

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

#include "RuntimeBasePch.h"

#include "ByteCode\ByteCodeAPI.h"

#include "ByteCode\ByteCodeDumper.h"

#include "Language\AsmJsTypes.h"

#include "Language\AsmJsModule.h"

#include "ByteCode\ByteCodeSerializer.h"

#include "Language\FunctionCodeGenRuntimeData.h"

#include "ByteCode\ScopeInfo.h"

#include "Base\EtwTrace.h"

#ifdef DYNAMIC\_PROFILE\_MUTATOR

#include "Language\DynamicProfileMutator.h"

#endif

#include "Language\SourceDynamicProfileManager.h"

#include "Debug\ProbeContainer.h"

#include "Debug\DebugContext.h"

#include "Parser.h"

#include "RegexCommon.h"

#include "RegexPattern.h"

#include "Library\RegexHelper.h"

#include "Language\InterpreterStackFrame.h"

#include "Library\ModuleRoot.h"

#include "Types\PathTypeHandler.h"

namespace Js

{

#ifdef FIELD\_ACCESS\_STATS

void FieldAccessStats::Add(FieldAccessStats\* other)

{

Assert(other != nullptr);

this->totalInlineCacheCount += other->totalInlineCacheCount;

this->noInfoInlineCacheCount += other->noInfoInlineCacheCount;

this->monoInlineCacheCount += other->monoInlineCacheCount;

this->emptyMonoInlineCacheCount += other->emptyMonoInlineCacheCount;

this->polyInlineCacheCount += other->polyInlineCacheCount;

this->nullPolyInlineCacheCount += other->nullPolyInlineCacheCount;

this->emptyPolyInlineCacheCount += other->emptyPolyInlineCacheCount;

this->ignoredPolyInlineCacheCount += other->ignoredPolyInlineCacheCount;

this->highUtilPolyInlineCacheCount += other->highUtilPolyInlineCacheCount;

this->lowUtilPolyInlineCacheCount += other->lowUtilPolyInlineCacheCount;

this->equivPolyInlineCacheCount += other->equivPolyInlineCacheCount;

this->nonEquivPolyInlineCacheCount += other->nonEquivPolyInlineCacheCount;

this->disabledPolyInlineCacheCount += other->disabledPolyInlineCacheCount;

this->clonedMonoInlineCacheCount += other->clonedMonoInlineCacheCount;

this->clonedPolyInlineCacheCount += other->clonedPolyInlineCacheCount;

}

#endif

// FunctionProxy methods

FunctionProxy::FunctionProxy(JavascriptMethod entryPoint, Attributes attributes, int nestedCount, int derivedSize, LocalFunctionId functionId, ScriptContext\* scriptContext, Utf8SourceInfo\* utf8SourceInfo, uint functionNumber):

FunctionInfo(entryPoint, attributes, functionId, (FunctionBody\*) this),

m\_nestedCount(nestedCount),

m\_isTopLevel(false),

m\_isPublicLibraryCode(false),

m\_derivedSize(derivedSize),

m\_scriptContext(scriptContext),

m\_utf8SourceInfo(utf8SourceInfo),

m\_referenceInParentFunction(nullptr),

m\_functionNumber(functionNumber),

m\_defaultEntryPointInfo(nullptr),

m\_functionObjectTypeList(nullptr)

{

PERF\_COUNTER\_INC(Code, TotalFunction);

}

uint FunctionProxy::GetSourceContextId() const

{

return m\_utf8SourceInfo->GetSrcInfo()->sourceContextInfo->sourceContextId;

}

wchar\_t\* FunctionProxy::GetDebugNumberSet(wchar(&bufferToWriteTo)[MAX\_FUNCTION\_BODY\_DEBUG\_STRING\_SIZE]) const

{

// (#%u.%u), #%u --> (source file Id . function Id) , function Number

int len = swprintf\_s(bufferToWriteTo, MAX\_FUNCTION\_BODY\_DEBUG\_STRING\_SIZE, L" (#%d.%u), #%u",

(int)this->GetSourceContextId(), this->GetLocalFunctionId(), this->GetFunctionNumber());

Assert(len > 8);

return bufferToWriteTo;

}

bool

FunctionProxy::IsFunctionBody() const

{

return !IsDeferredDeserializeFunction() && GetParseableFunctionInfo()->IsFunctionParsed();

}

uint

ParseableFunctionInfo::GetSourceIndex() const

{

return this->m\_sourceIndex;

}

LPCUTF8

ParseableFunctionInfo::GetSource(const wchar\_t\* reason) const

{

return this->m\_utf8SourceInfo->GetSource(reason == nullptr ? L"ParseableFunctionInfo::GetSource" : reason) + this->StartOffset();

}

LPCUTF8

ParseableFunctionInfo::GetStartOfDocument(const wchar\_t\* reason) const

{

return this->m\_utf8SourceInfo->GetSource(reason == nullptr ? L"ParseableFunctionInfo::GetStartOfDocument" : reason);

}

bool

ParseableFunctionInfo::IsDynamicFunction() const

{

return this->m\_isDynamicFunction;

}

bool

ParseableFunctionInfo::IsDynamicScript() const

{

return this->GetSourceContextInfo()->IsDynamic();

}

charcount\_t

ParseableFunctionInfo::StartInDocument() const

{

return this->m\_cchStartOffset;

}

uint

ParseableFunctionInfo::StartOffset() const

{

return this->m\_cbStartOffset;

}

void ParseableFunctionInfo::RegisterFuncToDiag(ScriptContext \* scriptContext, wchar\_t const \* pszTitle)

{

// Register the function to the PDM as eval code (the debugger app will show file as 'eval code')

scriptContext->GetDebugContext()->RegisterFunction(this, pszTitle);

}

// Given an offset into the source buffer, determine if the end of this SourceInfo

// lies after the given offset.

bool

FunctionBody::EndsAfter(size\_t offset) const

{

return offset < this->StartOffset() + this->LengthInBytes();

}

void

FunctionBody::RecordStatementMap(StatementMap\* pStatementMap)

{

Assert(!this->m\_sourceInfo.pSpanSequence);

Recycler\* recycler = this->m\_scriptContext->GetRecycler();

StatementMapList \* statementMaps = this->GetStatementMaps();

if (!statementMaps)

{

statementMaps = RecyclerNew(recycler, StatementMapList, recycler);

this->pStatementMaps = statementMaps;

}

statementMaps->Add(pStatementMap);

}

void

FunctionBody::RecordStatementMap(SmallSpanSequenceIter &iter, StatementData \* data)

{

Assert(!this->GetStatementMaps());

if (!this->m\_sourceInfo.pSpanSequence)

{

this->m\_sourceInfo.pSpanSequence = HeapNew(SmallSpanSequence);

}

this->m\_sourceInfo.pSpanSequence->RecordARange(iter, data);

}

void

FunctionBody::RecordStatementAdjustment(uint offset, StatementAdjustmentType adjType)

{

this->EnsureAuxStatementData();

Recycler\* recycler = this->m\_scriptContext->GetRecycler();

if (this->GetStatementAdjustmentRecords() == nullptr)

{

m\_sourceInfo.m\_auxStatementData->m\_statementAdjustmentRecords = RecyclerNew(recycler, StatementAdjustmentRecordList, recycler);

}

StatementAdjustmentRecord record(adjType, offset);

this->GetStatementAdjustmentRecords()->Add(record); // Will copy stack value and put the copy into the container.

}

BOOL

FunctionBody::GetBranchOffsetWithin(uint start, uint end, StatementAdjustmentRecord\* record)

{

Assert(start < end);

if (!this->GetStatementAdjustmentRecords())

{

// No Offset

return FALSE;

}

int count = this->GetStatementAdjustmentRecords()->Count();

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

{

StatementAdjustmentRecord item = this->GetStatementAdjustmentRecords()->Item(i);

if (item.GetByteCodeOffset() > start && item.GetByteCodeOffset() < end)

{

\*record = item;

return TRUE;

}

}

// No offset found in the range.

return FALSE;

}

ScriptContext\* EntryPointInfo::GetScriptContext()

{

Assert(!IsCleanedUp());

return this->library->GetScriptContext();

}

#if DBG\_DUMP | defined(VTUNE\_PROFILING)

void

EntryPointInfo::RecordNativeMap(uint32 nativeOffset, uint32 statementIndex)

{

int count = nativeOffsetMaps.Count();

if (count)

{

NativeOffsetMap\* previous = &nativeOffsetMaps.Item(count-1);

// Check if the range is still not finished.

if (previous->nativeOffsetSpan.begin == previous->nativeOffsetSpan.end)

{

if (previous->statementIndex == statementIndex)

{

// If the statement index is the same, we can continue with the previous range

return;

}

// If the range is empty, replace the previous range.

if ((uint32)previous->nativeOffsetSpan.begin == nativeOffset)

{

if (statementIndex == Js::Constants::NoStatementIndex)

{

nativeOffsetMaps.RemoveAtEnd();

}

else

{

previous->statementIndex = statementIndex;

}

return;

}

// Close the previous range

previous->nativeOffsetSpan.end = nativeOffset;

}

}

if (statementIndex == Js::Constants::NoStatementIndex)

{

// We do not explicitly record the offsets that do not map to user code.

return;

}

NativeOffsetMap map;

map.statementIndex = statementIndex;

map.nativeOffsetSpan.begin = nativeOffset;

map.nativeOffsetSpan.end = nativeOffset;

nativeOffsetMaps.Add(map);

}

#endif

void

FunctionBody::CopySourceInfo(ParseableFunctionInfo\* originalFunctionInfo)

{

this->m\_sourceIndex = originalFunctionInfo->GetSourceIndex();

this->m\_cchStartOffset = originalFunctionInfo->StartInDocument();

this->m\_cchLength = originalFunctionInfo->LengthInChars();

this->m\_lineNumber = originalFunctionInfo->GetRelativeLineNumber();

this->m\_columnNumber = originalFunctionInfo->GetRelativeColumnNumber();

this->m\_isEval = originalFunctionInfo->IsEval();

this->m\_isDynamicFunction = originalFunctionInfo->IsDynamicFunction();

this->m\_cbStartOffset = originalFunctionInfo->StartOffset();

this->m\_cbLength = originalFunctionInfo->LengthInBytes();

this->FinishSourceInfo();

}

// When sourceInfo is complete, register this functionBody to utf8SourceInfo. This ensures we never

// put incomplete functionBody into utf8SourceInfo map. (Previously we do it in FunctionBody constructor.

// If an error occurs thereafter before SetSourceInfo, e.g. OOM, we'll have an incomplete functionBody

// in utf8SourceInfo map whose source range is unknown and can't be reparsed.)

void FunctionBody::FinishSourceInfo()

{

m\_utf8SourceInfo->SetFunctionBody(this);

}

RegSlot FunctionBody::GetFrameDisplayRegister() const

{

return this->m\_sourceInfo.frameDisplayRegister;

}

void FunctionBody::SetFrameDisplayRegister(RegSlot frameDisplayRegister)

{

this->m\_sourceInfo.frameDisplayRegister = frameDisplayRegister;

}

RegSlot FunctionBody::GetObjectRegister() const

{

return this->m\_sourceInfo.objectRegister;

}

void FunctionBody::SetObjectRegister(RegSlot objectRegister)

{

this->m\_sourceInfo.objectRegister = objectRegister;

}

ScopeObjectChain \*FunctionBody::GetScopeObjectChain() const

{

return this->m\_sourceInfo.pScopeObjectChain;

}

void FunctionBody::SetScopeObjectChain(ScopeObjectChain \*pScopeObjectChain)

{

this->m\_sourceInfo.pScopeObjectChain = pScopeObjectChain;

}

ByteBlock \*FunctionBody::GetProbeBackingBlock()

{

return this->m\_sourceInfo.m\_probeBackingBlock;

}

void FunctionBody::SetProbeBackingBlock(ByteBlock\* probeBackingBlock)

{

this->m\_sourceInfo.m\_probeBackingBlock = probeBackingBlock;

}

FunctionBody \* FunctionBody::NewFromRecycler(ScriptContext \* scriptContext, const wchar\_t \* displayName, uint displayNameLength, uint displayShortNameOffset, uint nestedCount,

Utf8SourceInfo\* sourceInfo, uint uScriptId, Js::LocalFunctionId functionId, Js::PropertyRecordList\* boundPropertyRecords, Attributes attributes

#ifdef PERF\_COUNTERS

, bool isDeserializedFunction

#endif

)

{

return FunctionBody::NewFromRecycler(scriptContext, displayName, displayNameLength, displayShortNameOffset, nestedCount, sourceInfo,

scriptContext->GetThreadContext()->NewFunctionNumber(), uScriptId, functionId, boundPropertyRecords, attributes

#ifdef PERF\_COUNTERS

, isDeserializedFunction

#endif

);

}

FunctionBody \* FunctionBody::NewFromRecycler(ScriptContext \* scriptContext, const wchar\_t \* displayName, uint displayNameLength, uint displayShortNameOffset, uint nestedCount,

Utf8SourceInfo\* sourceInfo, uint uFunctionNumber, uint uScriptId, Js::LocalFunctionId functionId, Js::PropertyRecordList\* boundPropertyRecords, Attributes attributes

#ifdef PERF\_COUNTERS

, bool isDeserializedFunction

#endif

)

{

#ifdef PERF\_COUNTERS

return RecyclerNewWithBarrierFinalizedPlus(scriptContext->GetRecycler(), nestedCount \* sizeof(FunctionBody\*), FunctionBody, scriptContext, displayName, displayNameLength, displayShortNameOffset, nestedCount, sourceInfo, uFunctionNumber, uScriptId, functionId, boundPropertyRecords, attributes, isDeserializedFunction);

#else

return RecyclerNewWithBarrierFinalizedPlus(scriptContext->GetRecycler(), nestedCount \* sizeof(FunctionBody\*), FunctionBody, scriptContext, displayName, displayNameLength, displayShortNameOffset, nestedCount, sourceInfo, uFunctionNumber, uScriptId, functionId, boundPropertyRecords, attributes);

#endif

}

FunctionBody::FunctionBody(ScriptContext\* scriptContext, const wchar\_t\* displayName, uint displayNameLength, uint displayShortNameOffset, uint nestedCount,

Utf8SourceInfo\* utf8SourceInfo, uint uFunctionNumber, uint uScriptId,

Js::LocalFunctionId functionId, Js::PropertyRecordList\* boundPropertyRecords, Attributes attributes

#ifdef PERF\_COUNTERS

, bool isDeserializedFunction

#endif

) :

ParseableFunctionInfo(scriptContext->CurrentThunk, nestedCount, sizeof(FunctionBody), functionId, utf8SourceInfo, scriptContext, uFunctionNumber, displayName, displayNameLength, displayShortNameOffset, attributes, boundPropertyRecords),

m\_uScriptId(uScriptId),

m\_varCount(0),

m\_outParamMaxDepth(0),

m\_firstTmpReg(Constants::NoRegister),

loopCount(0),

cleanedUp(false),

sourceInfoCleanedUp(false),

profiledLdElemCount(0),

profiledStElemCount(0),

profiledCallSiteCount(0),

profiledArrayCallSiteCount(0),

profiledDivOrRemCount(0),

profiledSwitchCount(0),

profiledReturnTypeCount(0),

profiledSlotCount(0),

m\_isFuncRegistered(false),

m\_isFuncRegisteredToDiag(false),

m\_hasBailoutInstrInJittedCode(false),

m\_depth(0),

inlineDepth(0),

m\_pendingLoopHeaderRelease(false),

inlineCacheCount(0),

rootObjectLoadInlineCacheStart(0),

rootObjectStoreInlineCacheStart(0),

isInstInlineCacheCount(0),

objLiteralCount(0),

literalRegexCount(0),

innerScopeCount(0),

hasCachedScopePropIds(false),

m\_byteCodeCount(0),

m\_byteCodeWithoutLDACount(0),

m\_argUsedForBranch(0),

m\_byteCodeInLoopCount(0),

m\_envDepth((uint16)-1),

flags(Flags\_HasNoExplicitReturnValue),

m\_hasFinally(false),

#if ENABLE\_PROFILE\_INFO

dynamicProfileInfo(nullptr),

polymorphicCallSiteInfoHead(nullptr),

#endif

savedInlinerVersion(0),

#if ENABLE\_NATIVE\_CODEGEN

savedImplicitCallsFlags(ImplicitCall\_HasNoInfo),

#endif

savedPolymorphicCacheState(0),

functionBailOutRecord(nullptr),

hasExecutionDynamicProfileInfo(false),

m\_hasAllNonLocalReferenced(false),

m\_hasSetIsObject(false),

m\_hasFunExprNameReference(false),

m\_CallsEval(false),

m\_ChildCallsEval(false),

m\_hasReferenceableBuiltInArguments(false),

m\_firstFunctionObject(true),

m\_inlineCachesOnFunctionObject(false),

m\_hasDoneAllNonLocalReferenced(false),

m\_hasFunctionCompiledSent(false),

byteCodeCache(nullptr),

stackNestedFuncParent(nullptr),

localClosureRegister(Constants::NoRegister),

localFrameDisplayRegister(Constants::NoRegister),

envRegister(Constants::NoRegister),

thisRegisterForEventHandler(Constants::NoRegister),

firstInnerScopeRegister(Constants::NoRegister),

funcExprScopeRegister(Constants::NoRegister),

m\_tag(TRUE),

m\_nativeEntryPointUsed(FALSE),

debuggerScopeIndex(0),

bailOnMisingProfileCount(0),

bailOnMisingProfileRejitCount(0),

auxBlock(nullptr),

auxContextBlock(nullptr),

byteCodeBlock(nullptr),

entryPoints(nullptr),

loopHeaderArray(nullptr),

m\_constTable(nullptr),

literalRegexes(nullptr),

asmJsFunctionInfo(nullptr),

asmJsModuleInfo(nullptr),

m\_codeGenRuntimeData(nullptr),

m\_codeGenGetSetRuntimeData(nullptr),

pStatementMaps(nullptr),

inlineCaches(nullptr),

polymorphicInlineCachesHead(nullptr),

cacheIdToPropertyIdMap(nullptr),

referencedPropertyIdMap(nullptr),

propertyIdsForScopeSlotArray(nullptr),

propertyIdOnRegSlotsContainer(nullptr),

executionMode(ExecutionMode::Interpreter),

interpreterLimit(0),

autoProfilingInterpreter0Limit(0),

profilingInterpreter0Limit(0),

autoProfilingInterpreter1Limit(0),

simpleJitLimit(0),

profilingInterpreter1Limit(0),

fullJitThreshold(0),

fullJitRequeueThreshold(0),

committedProfiledIterations(0),

simpleJitEntryPointInfo(nullptr),

wasCalledFromLoop(false),

hasScopeObject(false),

hasNestedLoop(false),

recentlyBailedOutOfJittedLoopBody(false),

serializationIndex(-1),

m\_isAsmJsScheduledForFullJIT(false),

m\_asmJsTotalLoopCount(0),

//

// Even if the function does not require any locals, we must always have "R0" to propagate

// a return value. By enabling this here, we avoid unnecessary conditionals during execution.

//

m\_constCount(1)

#ifdef IR\_VIEWER

,m\_isIRDumpEnabled(false)

,m\_irDumpBaseObject(nullptr)

#endif /\* IR\_VIEWER \*/

, m\_isFromNativeCodeModule(false)

, interpretedCount(0)

, loopInterpreterLimit(CONFIG\_FLAG(LoopInterpretCount))

, hasHotLoop(false)

, m\_isPartialDeserializedFunction(false)

#ifdef PERF\_COUNTERS

, m\_isDeserializedFunction(isDeserializedFunction)

#endif

#if DBG

, m\_DEBUG\_executionCount(0)

, m\_nativeEntryPointIsInterpreterThunk(false)

, m\_canDoStackNestedFunc(false)

, m\_inlineCacheTypes(nullptr)

, m\_iProfileSession(-1)

, initializedExecutionModeAndLimits(false)

#endif

#if ENABLE\_DEBUG\_CONFIG\_OPTIONS

, regAllocLoadCount(0)

, regAllocStoreCount(0)

, callCountStats(0)

#endif

{

this->SetDefaultFunctionEntryPointInfo((FunctionEntryPointInfo\*) this->GetDefaultEntryPointInfo(), DefaultEntryThunk);

this->m\_hasBeenParsed = true;

#ifdef PERF\_COUNTERS

if (isDeserializedFunction)

{

PERF\_COUNTER\_INC(Code, DeserializedFunctionBody);

}

#endif

Assert(!utf8SourceInfo || m\_uScriptId == utf8SourceInfo->GetSrcInfo()->sourceContextInfo->sourceContextId);

// Sync entryPoints changes to etw rundown lock

CriticalSection\* syncObj = scriptContext->GetThreadContext()->GetEtwRundownCriticalSection();

this->entryPoints = RecyclerNew(this->m\_scriptContext->GetRecycler(), FunctionEntryPointList, this->m\_scriptContext->GetRecycler(), syncObj);

this->AddEntryPointToEntryPointList(this->GetDefaultFunctionEntryPointInfo());

Assert(this->GetDefaultEntryPointInfo()->address != nullptr);

InitDisableInlineApply();

InitDisableInlineSpread();

}

void FunctionBody::SetDefaultFunctionEntryPointInfo(FunctionEntryPointInfo\* entryPointInfo, const JavascriptMethod originalEntryPoint)

{

Assert(entryPointInfo);

// Need to set twice since ProxyEntryPointInfo cast points to an interior pointer

this->m\_defaultEntryPointInfo = (ProxyEntryPointInfo\*) entryPointInfo;

this->defaultFunctionEntryPointInfo = entryPointInfo;

SetOriginalEntryPoint(originalEntryPoint);

}

ByteBlock\*

FunctionBody::GetAuxiliaryData()

{

return this->auxBlock;

}

ByteBlock\*

FunctionBody::GetAuxiliaryContextData()

{

return this->auxContextBlock;

}

ByteBlock\*

FunctionBody::GetByteCode()

{

return this->byteCodeBlock;

}

// Returns original bytecode without probes (such as BPs).

ByteBlock\*

FunctionBody::GetOriginalByteCode()

{

if (m\_sourceInfo.m\_probeBackingBlock)

{

return m\_sourceInfo.m\_probeBackingBlock;

}

else

{

return this->GetByteCode();

}

}

const ByteCodeCache \*

FunctionBody::GetByteCodeCache() const

{

return byteCodeCache;

}

const int

FunctionBody::GetSerializationIndex() const

{

return serializationIndex;

}

const wchar\_t\* ParseableFunctionInfo::GetExternalDisplayName() const

{

return GetExternalDisplayName(this);

}

RegSlot

FunctionBody::GetLocalsCount()

{

return m\_constCount + m\_varCount;

}

RegSlot

FunctionBody::GetVarCount()

{

return m\_varCount;

}

// Returns the number of non-temp local vars.

uint32

FunctionBody::GetNonTempLocalVarCount()

{

Assert(this->GetEndNonTempLocalIndex() >= this->GetFirstNonTempLocalIndex());

return this->GetEndNonTempLocalIndex() - this->GetFirstNonTempLocalIndex();

}

uint32

FunctionBody::GetFirstNonTempLocalIndex()

{

// First local var starts when the const vars end.

return m\_constCount;

}

uint32

FunctionBody::GetEndNonTempLocalIndex()

{

// It will give the index on which current non temp locals ends, which is a first temp reg.

return m\_firstTmpReg != Constants::NoRegister ? m\_firstTmpReg : GetLocalsCount();

}

bool

FunctionBody::IsNonTempLocalVar(uint32 varIndex)

{

return GetFirstNonTempLocalIndex() <= varIndex && varIndex < GetEndNonTempLocalIndex();

}

bool

FunctionBody::GetSlotOffset(RegSlot slotId, int32 \* slotOffset, bool allowTemp)

{

if (IsNonTempLocalVar(slotId) || allowTemp)

{

\*slotOffset = (slotId - GetFirstNonTempLocalIndex()) \* DIAGLOCALSLOTSIZE;

return true;

}

return false;

}

void

FunctionBody::SetConstantCount(

RegSlot cNewConstants) // New register count

{

CheckNotExecuting();

AssertMsg(m\_constCount <= cNewConstants, "Cannot shrink register usage");

m\_constCount = cNewConstants;

}

void

FunctionBody::SetVarCount(

RegSlot cNewVars) // New register count

{

CheckNotExecuting();

AssertMsg(m\_varCount <= cNewVars, "Cannot shrink register usage");

m\_varCount = cNewVars;

}

RegSlot

FunctionBody::GetYieldRegister()

{

return GetEndNonTempLocalIndex() - 1;

}

RegSlot

FunctionBody::GetFirstTmpReg()

{

AssertMsg(m\_firstTmpReg != Constants::NoRegister, "First temp hasn't been set yet");

return m\_firstTmpReg;

}

void

FunctionBody::SetFirstTmpReg(

RegSlot firstTmpReg)

{

CheckNotExecuting();

AssertMsg(m\_firstTmpReg == Constants::NoRegister, "Should not be resetting the first temp");

m\_firstTmpReg = firstTmpReg;

}

RegSlot

FunctionBody::GetTempCount()

{

return GetLocalsCount() - m\_firstTmpReg;

}

void

FunctionBody::SetOutParamDepth(

RegSlot cOutParamsDepth)

{

CheckNotExecuting();

m\_outParamMaxDepth = cOutParamsDepth;

}

RegSlot

FunctionBody::GetOutParamsDepth()

{

return m\_outParamMaxDepth;

}

ModuleID

FunctionBody::GetModuleID() const

{

return this->GetHostSrcInfo()->moduleID;

}

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

///

/// FunctionBody::BeginExecution

///

/// BeginExecution() is called by InterpreterStackFrame when a function begins execution.

/// - Once started execution, the function may not be modified, as it would

/// change the stack-frame layout:

/// - This is a debug-only check because of the runtime cost. At release time,

/// a stack-walk will be performed by GC to determine which functions are

/// executing.

///

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

void

FunctionBody::BeginExecution()

{

#if DBG

m\_DEBUG\_executionCount++;

#endif

// Don't allow loop headers to be released while the function is executing

::InterlockedIncrement(this->m\_depth.AddressOf());

}

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

///

/// FunctionBody::CheckEmpty

///

/// CheckEmpty() validates that the given instance has not been given an

/// implementation yet.

///

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

void

FunctionBody::CheckEmpty()

{

AssertMsg((this->byteCodeBlock == nullptr) && (this->auxBlock == nullptr) && (this->auxContextBlock == nullptr), "Function body may only be set once");

}

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

///

/// FunctionBody::CheckNotExecuting

///

/// CheckNotExecuting() checks that function is not currently executing when it

/// is being modified. See BeginExecution() for details.

///

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

void

FunctionBody::CheckNotExecuting()

{

AssertMsg(m\_DEBUG\_executionCount == 0, "Function cannot be executing when modified");

}

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

///

/// FunctionBody::EndExecution

///

/// EndExecution() is called by InterpreterStackFrame when a function ends execution.

/// See BeginExecution() for details.

///

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

void

FunctionBody::EndExecution()

{

#if DBG

AssertMsg(m\_DEBUG\_executionCount > 0, "Must have a previous execution to end");

m\_DEBUG\_executionCount--;

#endif

uint depth = ::InterlockedDecrement(this->m\_depth.AddressOf());

// If loop headers were determined to be no longer needed

// during the execution of the function, we release them now

if (depth == 0 && this->m\_pendingLoopHeaderRelease)

{

this->m\_pendingLoopHeaderRelease = false;

ReleaseLoopHeaders();

}

}

void FunctionBody::AddEntryPointToEntryPointList(FunctionEntryPointInfo\* entryPointInfo)

{

ThreadContext::AutoDisableExpiration disableExpiration(this->m\_scriptContext->GetThreadContext());

Recycler\* recycler = this->m\_scriptContext->GetRecycler();

entryPointInfo->entryPointIndex = this->entryPoints->Add(recycler->CreateWeakReferenceHandle(entryPointInfo));

}

BOOL FunctionBody::IsInterpreterThunk() const

{

bool isInterpreterThunk = this->originalEntryPoint == DefaultEntryThunk;

#if DYNAMIC\_INTERPRETER\_THUNK

isInterpreterThunk = isInterpreterThunk || IsDynamicInterpreterThunk();

#endif

return isInterpreterThunk;

}

BOOL FunctionBody::IsDynamicInterpreterThunk() const

{

#if DYNAMIC\_INTERPRETER\_THUNK

return this->GetScriptContext()->IsDynamicInterpreterThunk(this->originalEntryPoint);

#else

return FALSE;

#endif

}

FunctionEntryPointInfo \* FunctionBody::TryGetEntryPointInfo(int index) const

{

// If we've already freed the recyclable data, we're shutting down the script context so skip clean up

if (this->entryPoints == nullptr) return 0;

Assert(index < this->entryPoints->Count());

FunctionEntryPointInfo\* entryPoint = this->entryPoints->Item(index)->Get();

return entryPoint;

}

FunctionEntryPointInfo \* FunctionBody::GetEntryPointInfo(int index) const

{

FunctionEntryPointInfo\* entryPoint = TryGetEntryPointInfo(index);

Assert(entryPoint);

return entryPoint;

}

uint32 FunctionBody::GetFrameHeight(EntryPointInfo\* entryPointInfo) const

{

return entryPointInfo->frameHeight;

}

void FunctionBody::SetFrameHeight(EntryPointInfo\* entryPointInfo, uint32 frameHeight)

{

entryPointInfo->frameHeight = frameHeight;

}

#if ENABLE\_NATIVE\_CODEGEN

void

FunctionBody::SetNativeThrowSpanSequence(SmallSpanSequence \*seq, uint loopNum, LoopEntryPointInfo\* entryPoint)

{

Assert(loopNum != LoopHeader::NoLoop);

LoopHeader \*loopHeader = this->GetLoopHeader(loopNum);

Assert(loopHeader);

Assert(entryPoint->loopHeader == loopHeader);

entryPoint->SetNativeThrowSpanSequence(seq);

}

void

FunctionBody::RecordNativeThrowMap(SmallSpanSequenceIter& iter, uint32 nativeOffset, uint32 statementIndex, EntryPointInfo\* entryPoint, uint loopNum)

{

SmallSpanSequence \*pSpanSequence;

pSpanSequence = entryPoint->GetNativeThrowSpanSequence();

if (!pSpanSequence)

{

if (statementIndex == -1)

{

return; // No need to initialize native throw map for non-user code

}

pSpanSequence = HeapNew(SmallSpanSequence);

if (loopNum == LoopHeader::NoLoop)

{

((FunctionEntryPointInfo\*) entryPoint)->SetNativeThrowSpanSequence(pSpanSequence);

}

else

{

this->SetNativeThrowSpanSequence(pSpanSequence, loopNum, (LoopEntryPointInfo\*) entryPoint);

}

}

else if (iter.accumulatedSourceBegin == static\_cast<int>(statementIndex))

{

return; // Compress adjacent spans which share the same statementIndex

}

StatementData data;

data.sourceBegin = static\_cast<int>(statementIndex); // sourceBegin represents statementIndex here

data.bytecodeBegin = static\_cast<int>(nativeOffset); // bytecodeBegin represents nativeOffset here

pSpanSequence->RecordARange(iter, &data);

}

#endif

bool

ParseableFunctionInfo::IsTrackedPropertyId(PropertyId pid)

{

Assert(this->m\_boundPropertyRecords != nullptr);

PropertyRecordList\* trackedProperties = this->m\_boundPropertyRecords;

const PropertyRecord\* prop = nullptr;

if (trackedProperties->TryGetValue(pid, &prop))

{

Assert(prop != nullptr);

return true;

}

return this->m\_scriptContext->IsTrackedPropertyId(pid);

}

PropertyId

ParseableFunctionInfo::GetOrAddPropertyIdTracked(JsUtil::CharacterBuffer<WCHAR> const& propName)

{

Assert(this->m\_boundPropertyRecords != nullptr);

const Js::PropertyRecord\* propRecord = nullptr;

this->m\_scriptContext->GetOrAddPropertyRecord(propName, &propRecord);

PropertyId pid = propRecord->GetPropertyId();

this->m\_boundPropertyRecords->Item(pid, propRecord);

return pid;

}

#if ENABLE\_NATIVE\_CODEGEN

void

FunctionBody::RecordNativeBaseAddress(BYTE\* baseAddress, ptrdiff\_t size, NativeCodeData \* data, NativeCodeData \* transferData,

CodeGenNumberChunk \* numberChunks, EntryPointInfo\* entryPoint, uint loopNum)

{

entryPoint->SetCodeGenRecorded(baseAddress, size, data, transferData, numberChunks);

}

#endif

int

FunctionBody::GetNextDebuggerScopeIndex()

{

return this->debuggerScopeIndex++;

}

SmallSpanSequence::SmallSpanSequence()

: pStatementBuffer(nullptr),

pActualOffsetList(nullptr),

baseValue(0)

{

}

BOOL SmallSpanSequence::RecordARange(SmallSpanSequenceIter &iter, StatementData \* data)

{

Assert(data);

if (!this->pStatementBuffer)

{

this->pStatementBuffer = JsUtil::GrowingUint32HeapArray::Create(4);

baseValue = data->sourceBegin;

Reset(iter);

}

SmallSpan span(0);

span.sourceBegin = GetDiff(data->sourceBegin, iter.accumulatedSourceBegin);

span.bytecodeBegin = GetDiff(data->bytecodeBegin, iter.accumulatedBytecodeBegin);

this->pStatementBuffer->Add((uint32)span);

// Update iterator for the next set

iter.accumulatedSourceBegin = data->sourceBegin;

iter.accumulatedBytecodeBegin = data->bytecodeBegin;

return TRUE;

}

// FunctionProxy methods

ScriptContext\*

FunctionProxy::GetScriptContext() const

{

return m\_scriptContext;

}

void FunctionProxy::Copy(FunctionProxy\* other)

{

Assert(other);

other->SetIsTopLevel(this->m\_isTopLevel);

if (this->IsPublicLibraryCode())

{

other->SetIsPublicLibraryCode();

}

}

void ParseableFunctionInfo::Copy(FunctionBody\* other)

{

#define CopyDeferParseField(field) other->field = this->field;

CopyDeferParseField(m\_isDeclaration);

CopyDeferParseField(m\_isAccessor);

CopyDeferParseField(m\_isStrictMode);

CopyDeferParseField(m\_isGlobalFunc);

CopyDeferParseField(m\_doBackendArgumentsOptimization);

CopyDeferParseField(m\_isEval);

CopyDeferParseField(m\_isDynamicFunction);

CopyDeferParseField(m\_hasImplicitArgIns);

CopyDeferParseField(m\_dontInline);

CopyDeferParseField(m\_inParamCount);

CopyDeferParseField(m\_grfscr);

CopyDeferParseField(m\_scopeInfo);

CopyDeferParseField(m\_utf8SourceHasBeenSet);

#if DBG

CopyDeferParseField(deferredParseNextFunctionId);

CopyDeferParseField(scopeObjectSize);

#endif

CopyDeferParseField(scopeSlotArraySize);

CopyDeferParseField(cachedSourceString);

CopyDeferParseField(deferredStubs);

CopyDeferParseField(m\_isAsmjsMode);

CopyDeferParseField(m\_isAsmJsFunction);

#undef CopyDeferParseField

other->CopySourceInfo(this);

}

void FunctionProxy::SetReferenceInParentFunction(FunctionProxyPtrPtr reference)

{

if (reference)

{

// Tag the reference so that the child function doesn't

// keep the parent alive. If the parent function is going away,

// it'll clear its children's references.

this->m\_referenceInParentFunction = reference;

}

else

{

this->m\_referenceInParentFunction = nullptr;

}

}

void FunctionProxy::UpdateReferenceInParentFunction(FunctionProxy\* newFunctionInfo)

{

if (this->m\_referenceInParentFunction)

{

#ifdef RECYCLER\_WRITE\_BARRIER

if (newFunctionInfo == nullptr)

{

(\*m\_referenceInParentFunction).NoWriteBarrierSet(nullptr);

return;

}

#endif

(\*m\_referenceInParentFunction) = newFunctionInfo;

}

}

// DeferDeserializeFunctionInfo methods

DeferDeserializeFunctionInfo::DeferDeserializeFunctionInfo(int nestedCount, LocalFunctionId functionId, ByteCodeCache\* byteCodeCache, const byte\* serializedFunction, Utf8SourceInfo\* sourceInfo, ScriptContext\* scriptContext, uint functionNumber, const wchar\_t\* displayName, uint displayNameLength, uint displayShortNameOffset, NativeModule \*nativeModule, Attributes attributes) :

FunctionProxy(DefaultDeferredDeserializeThunk, (Attributes)(attributes | DeferredDeserialize), nestedCount, sizeof(DeferDeserializeFunctionInfo), functionId, scriptContext, sourceInfo, functionNumber),

m\_cache(byteCodeCache),

m\_functionBytes(serializedFunction),

m\_displayName(nullptr),

m\_displayNameLength(0),

m\_nativeModule(nativeModule)

{

this->m\_defaultEntryPointInfo = RecyclerNew(scriptContext->GetRecycler(), ProxyEntryPointInfo, DefaultDeferredDeserializeThunk);

PERF\_COUNTER\_INC(Code, DeferDeserializeFunctionProxy);

SetDisplayName(displayName, displayNameLength, displayShortNameOffset, FunctionProxy::SetDisplayNameFlagsDontCopy);

}

DeferDeserializeFunctionInfo\* DeferDeserializeFunctionInfo::New(ScriptContext\* scriptContext, int nestedCount, LocalFunctionId functionId, ByteCodeCache\* byteCodeCache, const byte\* serializedFunction, Utf8SourceInfo\* sourceInfo, const wchar\_t\* displayName, uint displayNameLength, uint displayShortNameOffset, NativeModule \*nativeModule, Attributes attributes)

{

return RecyclerNewFinalized(scriptContext->GetRecycler(),

DeferDeserializeFunctionInfo,

nestedCount,

functionId,

byteCodeCache,

serializedFunction,

sourceInfo,

scriptContext,

scriptContext->GetThreadContext()->NewFunctionNumber(),

displayName,

displayNameLength,

displayShortNameOffset,

nativeModule,

attributes);

}

const wchar\_t\*

DeferDeserializeFunctionInfo::GetDisplayName() const

{

return this->m\_displayName;

}

// ParseableFunctionInfo methods

ParseableFunctionInfo::ParseableFunctionInfo(JavascriptMethod entryPoint, int nestedCount, int derivedSize,

LocalFunctionId functionId, Utf8SourceInfo\* sourceInfo, ScriptContext\* scriptContext, uint functionNumber,

const wchar\_t\* displayName, uint displayNameLength, uint displayShortNameOffset, Attributes attributes, Js::PropertyRecordList\* propertyRecords) :

FunctionProxy(entryPoint, attributes, nestedCount, derivedSize, functionId, scriptContext, sourceInfo, functionNumber),

#if DYNAMIC\_INTERPRETER\_THUNK

m\_dynamicInterpreterThunk(nullptr),

#endif

m\_hasBeenParsed(false),

m\_isGlobalFunc(false),

m\_isDeclaration(false),

m\_isNamedFunctionExpression(false),

m\_isNameIdentifierRef (true),

m\_isStaticNameFunction(false),

m\_doBackendArgumentsOptimization(true),

m\_isStrictMode(false),

m\_isAsmjsMode(false),

m\_dontInline(false),

m\_hasImplicitArgIns(true),

m\_grfscr(0),

m\_inParamCount(0),

m\_reportedInParamCount(0),

m\_sourceIndex(Js::Constants::InvalidSourceIndex),

m\_utf8SourceHasBeenSet(false),

m\_cchLength(0),

m\_cbLength(0),

m\_cchStartOffset(0),

m\_cbStartOffset(0),

m\_lineNumber(0),

m\_columnNumber(0),

m\_isEval(false),

m\_isDynamicFunction(false),

m\_scopeInfo(nullptr),

m\_displayName(nullptr),

m\_displayNameLength(0),

m\_displayShortNameOffset(0),

deferredStubs(nullptr),

scopeSlotArraySize(0),

cachedSourceString(nullptr),

m\_boundPropertyRecords(propertyRecords),

m\_reparsed(false),

m\_isAsmJsFunction(false),

#if DBG

m\_wasEverAsmjsMode(false),

scopeObjectSize(0),

#endif

isByteCodeDebugMode(false)

{

if ((attributes & Js::FunctionInfo::DeferredParse) == 0)

{

void\* validationCookie = nullptr;

#if ENABLE\_NATIVE\_CODEGEN

validationCookie = (void\*)scriptContext->GetNativeCodeGenerator();

#endif

this->m\_defaultEntryPointInfo = RecyclerNewFinalized(scriptContext->GetRecycler(),

FunctionEntryPointInfo, this, entryPoint, scriptContext->GetThreadContext(), validationCookie);

}

else

{

this->m\_defaultEntryPointInfo = RecyclerNew(scriptContext->GetRecycler(), ProxyEntryPointInfo, entryPoint);

}

SetDisplayName(displayName, displayNameLength, displayShortNameOffset);

this->originalEntryPoint = DefaultEntryThunk;

}

ParseableFunctionInfo\* ParseableFunctionInfo::New(ScriptContext\* scriptContext, int nestedCount,

LocalFunctionId functionId, Utf8SourceInfo\* sourceInfo, const wchar\_t\* displayName, uint displayNameLength, uint displayShortNameOffset, Js::PropertyRecordList\* propertyRecords, Attributes attributes)

{

Assert(scriptContext->DeferredParsingThunk == ProfileDeferredParsingThunk

|| scriptContext->DeferredParsingThunk == DefaultDeferredParsingThunk);

#ifdef PERF\_COUNTERS

PERF\_COUNTER\_INC(Code, DeferedFunction);

#endif

uint newFunctionNumber = scriptContext->GetThreadContext()->NewFunctionNumber();

if (!sourceInfo->GetSourceContextInfo()->IsDynamic())

{

PHASE\_PRINT\_TESTTRACE1(Js::DeferParsePhase, L"Function was deferred from parsing - ID: %d; Display Name: %s; Utf8SourceInfo ID: %d; Source Length: %d; Source Url:%s\n", newFunctionNumber, displayName, sourceInfo->GetSourceInfoId(), sourceInfo->GetCchLength(), sourceInfo->GetSourceContextInfo()->url);

}

else

{

PHASE\_PRINT\_TESTTRACE1(Js::DeferParsePhase, L"Function was deferred from parsing - ID: %d; Display Name: %s; Utf8SourceInfo ID: %d; Source Length: %d;\n", newFunctionNumber, displayName, sourceInfo->GetSourceInfoId(), sourceInfo->GetCchLength());

}

// When generating a new defer parse function, we always use a new function number

return RecyclerNewWithBarrierFinalizedPlus(scriptContext->GetRecycler(),

nestedCount \* sizeof(FunctionBody\*),

ParseableFunctionInfo,

scriptContext->DeferredParsingThunk,

nestedCount,

sizeof(ParseableFunctionInfo),

functionId,

sourceInfo,

scriptContext,

newFunctionNumber,

displayName,

displayNameLength,

displayShortNameOffset,

(Attributes)(attributes | DeferredParse),

propertyRecords);

}

DWORD\_PTR FunctionProxy::GetSecondaryHostSourceContext() const

{

return this->m\_utf8SourceInfo->GetSecondaryHostSourceContext();

}

DWORD\_PTR FunctionProxy::GetHostSourceContext() const

{

return this->GetSourceContextInfo()->dwHostSourceContext;

}

SourceContextInfo \* FunctionProxy::GetSourceContextInfo() const

{

return this->GetHostSrcInfo()->sourceContextInfo;

}

SRCINFO const \* FunctionProxy::GetHostSrcInfo() const

{

return m\_utf8SourceInfo->GetSrcInfo();

}

//

// Returns the start line for the script buffer (code buffer for the entire script tag) of this current function.

// We subtract the lnMinHost because it is the number of lines we have added to augment scriptlets passed through

// ParseProcedureText to have a function name.

//

ULONG FunctionProxy::GetHostStartLine() const

{

return this->GetHostSrcInfo()->dlnHost - this->GetHostSrcInfo()->lnMinHost;

}

//

// Returns the start column of the first line for the script buffer of this current function.

//

ULONG FunctionProxy::GetHostStartColumn() const

{

return this->GetHostSrcInfo()->ulColumnHost;

}

//

// Returns line number in unmodified host buffer (i.e. without extra scriptlet code added by ParseProcedureText --

// when e.g. we add extra code for event handlers, such as "function onclick(event)\n{\n").

//

ULONG FunctionProxy::GetLineNumberInHostBuffer(ULONG relativeLineNumber) const

{

ULONG lineNumber = relativeLineNumber;

if (lineNumber >= this->GetHostSrcInfo()->lnMinHost)

{

lineNumber -= this->GetHostSrcInfo()->lnMinHost;

}

// Note that '<' is still a valid case -- that would be the case for onclick scriptlet function itself (lineNumber == 0).

return lineNumber;

}

ULONG FunctionProxy::ComputeAbsoluteLineNumber(ULONG relativeLineNumber) const

{

// We add 1 because the line numbers start from 0.

return this->GetHostSrcInfo()->dlnHost + GetLineNumberInHostBuffer(relativeLineNumber) + 1;

}

ULONG FunctionProxy::ComputeAbsoluteColumnNumber(ULONG relativeLineNumber, ULONG relativeColumnNumber) const

{

if (this->GetLineNumberInHostBuffer(relativeLineNumber) == 0)

{

// The host column matters only for the first line.

return this->GetHostStartColumn() + relativeColumnNumber + 1;

}

// We add 1 because the column numbers start from 0.

return relativeColumnNumber + 1;

}

//

// Returns the line number of the function declaration in the source file.

//

ULONG

ParseableFunctionInfo::GetLineNumber() const

{

return this->ComputeAbsoluteLineNumber(this->m\_lineNumber);

}

//

// Returns the column number of the function declaration in the source file.

//

ULONG

ParseableFunctionInfo::GetColumnNumber() const

{

return ComputeAbsoluteColumnNumber(this->m\_lineNumber, m\_columnNumber);

}

LPCWSTR

ParseableFunctionInfo::GetSourceName() const

{

return GetSourceName(this->GetSourceContextInfo());

}

void

ParseableFunctionInfo::SetGrfscr(ulong grfscr)

{

this->m\_grfscr = grfscr;

}

ulong

ParseableFunctionInfo::GetGrfscr() const

{

return this->m\_grfscr;

}

ProxyEntryPointInfo\*

FunctionProxy::GetDefaultEntryPointInfo() const

{

return this->m\_defaultEntryPointInfo;

}

FunctionEntryPointInfo\*

FunctionBody::GetDefaultFunctionEntryPointInfo() const

{

Assert(((ProxyEntryPointInfo\*) this->defaultFunctionEntryPointInfo) == this->m\_defaultEntryPointInfo);

return this->defaultFunctionEntryPointInfo;

}

void

ParseableFunctionInfo::SetInParamsCount(ArgSlot newInParamCount)

{

AssertMsg(m\_inParamCount <= newInParamCount, "Cannot shrink register usage");

m\_inParamCount = newInParamCount;

if (newInParamCount <= 1)

{

SetHasImplicitArgIns(false);

}

}

ArgSlot

ParseableFunctionInfo::GetReportedInParamsCount() const

{

return m\_reportedInParamCount;

}

void

ParseableFunctionInfo::SetReportedInParamsCount(ArgSlot newInParamCount)

{

AssertMsg(m\_reportedInParamCount <= newInParamCount, "Cannot shrink register usage");

m\_reportedInParamCount = newInParamCount;

}

void

ParseableFunctionInfo::ResetInParams()

{

m\_inParamCount = 0;

m\_reportedInParamCount = 0;

}

const wchar\_t\*

ParseableFunctionInfo::GetDisplayName() const

{

return this->m\_displayName;

}

void ParseableFunctionInfo::BuildDeferredStubs(ParseNode \*pnodeFnc)

{

Assert(pnodeFnc->nop == knopFncDecl);

Recycler \*recycler = GetScriptContext()->GetRecycler();

this->deferredStubs = BuildDeferredStubTree(pnodeFnc, recycler);

}

FunctionProxyArray ParseableFunctionInfo::GetNestedFuncArray()

{

// The array is allocated as extra bytes past the end of the struct.

Assert(this->m\_nestedCount > 0);

return (FunctionProxyArray )((char\*)this + m\_derivedSize);

}

void ParseableFunctionInfo::SetNestedFunc(FunctionProxy\* nestedFunc, uint index, ulong flags)

{

AssertMsg(index < this->m\_nestedCount, "Trying to write past the nested func array");

FunctionProxyArray nested = this->GetNestedFuncArray();

nested[index] = nestedFunc;

if (nestedFunc)

{

nestedFunc->SetReferenceInParentFunction(GetNestedFuncReference(index));

if (!this->GetSourceContextInfo()->IsDynamic() && nestedFunc->IsDeferredParseFunction() && nestedFunc->GetParseableFunctionInfo()->GetIsDeclaration() && this->GetIsTopLevel() && !(flags & fscrEvalCode))

{

this->m\_utf8SourceInfo->TrackDeferredFunction(nestedFunc->GetLocalFunctionId(), nestedFunc->GetParseableFunctionInfo());

}

}

}

FunctionProxy\* ParseableFunctionInfo::GetNestedFunc(uint index)

{

return \*(GetNestedFuncReference(index));

}

FunctionProxyPtrPtr ParseableFunctionInfo::GetNestedFuncReference(uint index)

{

AssertMsg(index < this->m\_nestedCount, "Trying to write past the nested func array");

FunctionProxyArray nested = this->GetNestedFuncArray();

return &nested[index];

}

ParseableFunctionInfo\* ParseableFunctionInfo::GetNestedFunctionForExecution(uint index)

{

FunctionProxy\* currentNestedFunction = this->GetNestedFunc(index);

Assert(currentNestedFunction);

if (currentNestedFunction->IsDeferredDeserializeFunction())

{

currentNestedFunction = currentNestedFunction->EnsureDeserialized();

this->SetNestedFunc(currentNestedFunction, index, 0u);

}

return currentNestedFunction->GetParseableFunctionInfo();

}

void

FunctionProxy::UpdateFunctionBodyImpl(FunctionBody \* body)

{

Assert(functionBodyImpl == ((FunctionProxy\*) this));

Assert(!this->IsFunctionBody() || body == this);

this->functionBodyImpl = body;

this->attributes = (Attributes)(this->attributes & ~(DeferredParse | DeferredDeserialize));

this->UpdateReferenceInParentFunction(body);

}

void ParseableFunctionInfo::ClearNestedFunctionParentFunctionReference()

{

if (this->m\_nestedCount > 0)

{

// If the function is x-domain all the nested functions should also be marked as x-domain

FunctionProxyArray nested = this->GetNestedFuncArray();

for (uint i = 0; i < this->m\_nestedCount; ++i)

{

if (nested[i])

{

nested[i]->SetReferenceInParentFunction(nullptr);

}

}

}

}

//

// This method gets a function body for the purposes of execution

// It has an if within it to avoid making it a virtual- it's called from the interpreter

// It will cause the function info to get deserialized if it hasn't been deserialized

// already

//

ParseableFunctionInfo \* FunctionProxy::EnsureDeserialized()

{

FunctionProxy \* executionFunctionBody = this->functionBodyImpl;

if (executionFunctionBody == this && IsDeferredDeserializeFunction())

{

// No need to deserialize function body if scriptContext closed because we can't execute it.

// Bigger problem is the script engine might have released bytecode file mapping and we can't deserialize.

Assert(!m\_scriptContext->IsClosed());

executionFunctionBody = ((DeferDeserializeFunctionInfo\*) this)->Deserialize();

this->functionBodyImpl = executionFunctionBody;

Assert(executionFunctionBody->HasBody());

Assert(executionFunctionBody != this);

}

return (ParseableFunctionInfo \*)executionFunctionBody;

}

ScriptFunctionType \* FunctionProxy::GetDeferredPrototypeType() const

{

return deferredPrototypeType;

}

ScriptFunctionType \* FunctionProxy::EnsureDeferredPrototypeType()

{

Assert(this->GetFunctionProxy() == this);

return (deferredPrototypeType != nullptr)? deferredPrototypeType : AllocDeferredPrototypeType();

}

ScriptFunctionType \* FunctionProxy::AllocDeferredPrototypeType()

{

Assert(deferredPrototypeType == nullptr);

ScriptFunctionType \* type = ScriptFunctionType::New(this, true);

deferredPrototypeType = type;

return type;

}

JavascriptMethod FunctionProxy::GetDirectEntryPoint(ProxyEntryPointInfo\* entryPoint) const

{

Assert((JavascriptMethod)entryPoint->address != nullptr);

return (JavascriptMethod)entryPoint->address;

}

// Function object type list methods

template <typename Fn>

void FunctionProxy::MapFunctionObjectTypes(Fn func)

{

if (m\_functionObjectTypeList)

{

m\_functionObjectTypeList->Map([&] (int, FunctionTypeWeakRef\* typeWeakRef)

{

if (typeWeakRef)

{

DynamicType\* type = typeWeakRef->Get();

if (type)

{

func(type);

}

}

});

}

if (this->deferredPrototypeType)

{

func(this->deferredPrototypeType);

}

}

FunctionProxy::FunctionTypeWeakRefList\* FunctionProxy::EnsureFunctionObjectTypeList()

{

if (m\_functionObjectTypeList == nullptr)

{

Recycler\* recycler = this->GetScriptContext()->GetRecycler();

m\_functionObjectTypeList = RecyclerNew(recycler, FunctionTypeWeakRefList, recycler);

}

return m\_functionObjectTypeList;

}

void FunctionProxy::RegisterFunctionObjectType(DynamicType\* functionType)

{

FunctionTypeWeakRefList\* typeList = EnsureFunctionObjectTypeList();

Assert(functionType != deferredPrototypeType);

Recycler \* recycler = this->GetScriptContext()->GetRecycler();

FunctionTypeWeakRef\* weakRef = recycler->CreateWeakReferenceHandle(functionType);

typeList->SetAtFirstFreeSpot(weakRef);

OUTPUT\_TRACE(Js::ExpirableCollectPhase, L"Registered type 0x%p on function body %p, count = %d\n", functionType, this, typeList->Count());

}

void DeferDeserializeFunctionInfo::SetDisplayName(const wchar\_t\* displayName)

{

size\_t len = wcslen(displayName);

if (len > UINT\_MAX)

{

// Can't support display name that big

Js::Throw::OutOfMemory();

}

SetDisplayName(displayName, (uint)len, 0);

}

void DeferDeserializeFunctionInfo::SetDisplayName(const wchar\_t\* pszDisplayName, uint displayNameLength, uint displayShortNameOffset, SetDisplayNameFlags flags /\* default to None \*/)

{

this->m\_displayNameLength = displayNameLength;

this->m\_displayShortNameOffset = displayShortNameOffset;

FunctionProxy::SetDisplayName(pszDisplayName, &this->m\_displayName, displayNameLength, m\_scriptContext, flags);

}

LPCWSTR DeferDeserializeFunctionInfo::GetSourceInfo(int& lineNumber, int& columnNumber) const

{

// Read all the necessary information from the serialized byte code

int lineNumberField, columnNumberField;

bool m\_isEval, m\_isDynamicFunction;

ByteCodeSerializer::ReadSourceInfo(this, lineNumberField, columnNumberField, m\_isEval, m\_isDynamicFunction);

// Decode them

lineNumber = ComputeAbsoluteLineNumber(lineNumberField);

columnNumber = ComputeAbsoluteColumnNumber(lineNumberField, columnNumberField);

return Js::ParseableFunctionInfo::GetSourceName<SourceContextInfo\*>(this->GetSourceContextInfo(), m\_isEval, m\_isDynamicFunction);

}

void DeferDeserializeFunctionInfo::Finalize(bool isShutdown)

{

\_\_super::Finalize(isShutdown);

PERF\_COUNTER\_DEC(Code, DeferDeserializeFunctionProxy);

}

FunctionBody\* DeferDeserializeFunctionInfo::Deserialize()

{

if (functionBodyImpl == (FunctionBody\*) this)

{

FunctionBody \* body = ByteCodeSerializer::DeserializeFunction(this->m\_scriptContext, this);

this->Copy(body);

this->UpdateFunctionBodyImpl(body);

}

return GetFunctionBody();

}

//

// hrParse can be one of the following from deferred re-parse (check CompileScriptException::ProcessError):

// E\_OUTOFMEMORY

// E\_UNEXPECTED

// SCRIPT\_E\_RECORDED,

// with ei.scode: ERRnoMemory, VBSERR\_OutOfStack, E\_OUTOFMEMORY, E\_FAIL

// Any other ei.scode shouldn't appear in deferred re-parse.

//

// Map errors like OOM/SOE, return it and clean hrParse. Any other error remaining in hrParse is an internal error.

//

HRESULT ParseableFunctionInfo::MapDeferredReparseError(HRESULT& hrParse, const CompileScriptException& se)

{

HRESULT hrMapped = NO\_ERROR;

switch (hrParse)

{

case E\_OUTOFMEMORY:

hrMapped = E\_OUTOFMEMORY;

break;

case SCRIPT\_E\_RECORDED:

switch (se.ei.scode)

{

case ERRnoMemory:

case E\_OUTOFMEMORY:

case VBSERR\_OutOfMemory:

hrMapped = E\_OUTOFMEMORY;

break;

case VBSERR\_OutOfStack:

hrMapped = VBSERR\_OutOfStack;

break;

}

}

if (FAILED(hrMapped))

{

// If we have mapped error, clear hrParse. We'll throw error from hrMapped.

hrParse = NO\_ERROR;

}

return hrMapped;

}

FunctionBody\* ParseableFunctionInfo::Parse(ScriptFunction \*\* functionRef, bool isByteCodeDeserialization)

{

if ((functionBodyImpl != (FunctionBody\*) this) || !IsDeferredParseFunction())

{

// If not deferredparsed, the functionBodyImpl and this will be the same, just return the current functionBody.

Assert(GetFunctionBody()->IsFunctionParsed());

return GetFunctionBody();

}

BOOL fParsed = FALSE;

FunctionBody\* returnFunctionBody = nullptr;

ENTER\_PINNED\_SCOPE(Js::PropertyRecordList, propertyRecordList);

Recycler\* recycler = this->m\_scriptContext->GetRecycler();

propertyRecordList = RecyclerNew(recycler, Js::PropertyRecordList, recycler);

bool isDebugReparse = m\_scriptContext->IsInDebugOrSourceRundownMode() && !this->m\_utf8SourceInfo->GetIsLibraryCode();

bool isAsmJsReparse = false;

bool isReparse = isDebugReparse;

FunctionBody\* funcBody = nullptr;

// If m\_hasBeenParsed = true, one of the following things happened things happened:

// - We had multiple function objects which were all defer-parsed, but with the same function body and one of them

// got the body to be parsed before another was called

// - We are in debug mode and had our thunks switched to DeferParseThunk

// - This is an already parsed asm.js module, which has been invalidated at link time and must be reparsed as a non-asm.js function

if (!this->m\_hasBeenParsed)

{

funcBody = FunctionBody::NewFromRecycler(

this->m\_scriptContext,

this->m\_displayName,

this->m\_displayNameLength,

this->m\_displayShortNameOffset,

this->m\_nestedCount,

this->m\_utf8SourceInfo,

this->m\_functionNumber,

m\_utf8SourceInfo->GetSrcInfo()->sourceContextInfo->sourceContextId, /\* script id \*/

this->functionId, /\* function id \*/

propertyRecordList,

(Attributes)(this->GetAttributes() & ~(Attributes::DeferredDeserialize | Attributes::DeferredParse))

#ifdef PERF\_COUNTERS

, false /\* is function from deferred deserialized proxy \*/

#endif

);

this->Copy(funcBody);

PERF\_COUNTER\_DEC(Code, DeferedFunction);

if (!this->GetSourceContextInfo()->IsDynamic())

{

PHASE\_PRINT\_TESTTRACE1(Js::DeferParsePhase, L"TestTrace: Deferred function parsed - ID: %d; Display Name: %s; Length: %d; Nested Function Count: %d; Utf8SourceInfo: %d; Source Length: %d; Is Top Level: %s; Source Url: %s\n", m\_functionNumber, m\_displayName, this->m\_cchLength, this->GetNestedCount(), this->m\_utf8SourceInfo->GetSourceInfoId(), this->m\_utf8SourceInfo->GetCchLength(), this->GetIsTopLevel() ? L"True" : L"False", this->GetSourceContextInfo()->url);

}

else

{

PHASE\_PRINT\_TESTTRACE1(Js::DeferParsePhase, L"TestTrace: Deferred function parsed - ID: %d; Display Name: %s; Length: %d; Nested Function Count: %d; Utf8SourceInfo: %d; Source Length: %d\n; Is Top Level: %s;", m\_functionNumber, m\_displayName, this->m\_cchLength, this->GetNestedCount(), this->m\_utf8SourceInfo->GetSourceInfoId(), this->m\_utf8SourceInfo->GetCchLength(), this->GetIsTopLevel() ? L"True" : L"False");

}

if (!this->GetIsTopLevel() &&

!this->GetSourceContextInfo()->IsDynamic() &&

this->m\_scriptContext->DoUndeferGlobalFunctions())

{

this->m\_utf8SourceInfo->UndeferGlobalFunctions([this](JsUtil::SimpleDictionaryEntry<Js::LocalFunctionId, Js::ParseableFunctionInfo\*> func)

{

Js::ParseableFunctionInfo \*nextFunc = func.Value();

JavascriptExceptionObject\* pExceptionObject = nullptr;

if (nextFunc != nullptr && this != nextFunc)

{

try

{

nextFunc->Parse();

}

catch (OutOfMemoryException) {}

catch (StackOverflowException) {}

catch (JavascriptExceptionObject\* exceptionObject)

{

pExceptionObject = exceptionObject;

}

// Do not do anything with an OOM or SOE, returning true is fine, it will then be undeferred (or attempted to again when called)

if(pExceptionObject)

{

if(pExceptionObject != ThreadContext::GetContextForCurrentThread()->GetPendingOOMErrorObject() &&

pExceptionObject != ThreadContext::GetContextForCurrentThread()->GetPendingSOErrorObject())

{

throw pExceptionObject;

}

}

}

return true;

});

}

}

else

{

isAsmJsReparse = m\_isAsmjsMode && !isDebugReparse;

isReparse |= isAsmJsReparse;

funcBody = this->GetFunctionBody();

if (isReparse)

{

#if ENABLE\_DEBUG\_CONFIG\_OPTIONS

wchar\_t debugStringBuffer[MAX\_FUNCTION\_BODY\_DEBUG\_STRING\_SIZE];

#endif

#if DBG

Assert(

funcBody->IsReparsed()

|| m\_scriptContext->IsInDebugOrSourceRundownMode()

|| m\_isAsmjsMode);

#endif

OUTPUT\_TRACE(Js::DebuggerPhase, L"Full nested reparse of function: %s (%s)\n", funcBody->GetDisplayName(), funcBody->GetDebugNumberSet(debugStringBuffer));

if (funcBody->GetByteCode())

{

// The current function needs to be cleaned up before getting generated in the debug mode.

funcBody->CleanupToReparse();

}

}

}

// Note that we may be trying to re-gen an already-completed function. (This can happen, for instance,

// in the case of named function expressions inside "with" statements in compat mode.)

// In such a case, there's no work to do.

if (funcBody->GetByteCode() == nullptr)

{

#if ENABLE\_PROFILE\_INFO

Assert(!funcBody->HasExecutionDynamicProfileInfo());

#endif

// In debug mode, the eval code will be asked to recompile again.

AssertMsg(isDebugReparse || !(funcBody->GetGrfscr() & (fscrImplicitThis | fscrImplicitParents)),

"Deferred parsing of event handler body?");

// In debug or asm.js mode, the scriptlet will be asked to recompile again.

AssertMsg(isReparse || funcBody->GetGrfscr() & fscrGlobalCode || CONFIG\_FLAG(DeferNested), "Deferred parsing of non-global procedure?");

HRESULT hr = NO\_ERROR;

HRESULT hrParser = NO\_ERROR;

HRESULT hrParseCodeGen = NO\_ERROR;

BEGIN\_LEAVE\_SCRIPT\_INTERNAL(m\_scriptContext)

{

bool isCesu8 = m\_scriptContext->GetSource(funcBody->GetSourceIndex())->IsCesu8();

size\_t offset = this->StartOffset();

charcount\_t charOffset = this->StartInDocument();

size\_t length = this->LengthInBytes();

LPCUTF8 pszStart = this->GetStartOfDocument();

ulong grfscr = funcBody->GetGrfscr() | fscrDeferredFnc;

// For the global function we want to re-use the glo functionbody which is already created in the non-debug mode

if (!funcBody->GetIsGlobalFunc())

{

grfscr &= ~fscrGlobalCode;

}

if (!funcBody->GetIsDeclaration() && !funcBody->GetIsGlobalFunc()) // No refresh may reparse global function (e.g. eval code)

{

// Notify the parser that the top-level function was defined in an expression,

// (not a function declaration statement).

grfscr |= fscrDeferredFncExpression;

}

if (!CONFIG\_FLAG(DeferNested) || isDebugReparse || isAsmJsReparse)

{

grfscr &= ~fscrDeferFncParse; // Disable deferred parsing if not DeferNested, or doing a debug/asm.js re-parse

}

if (isReparse)

{

grfscr |= fscrNoAsmJs; // Disable asm.js when debugging or if linking failed

}

BEGIN\_TRANSLATE\_EXCEPTION\_TO\_HRESULT

{

CompileScriptException se;

Parser ps(m\_scriptContext, funcBody->GetIsStrictMode() ? TRUE : FALSE);

ParseNodePtr parseTree;

uint nextFunctionId = funcBody->GetLocalFunctionId();

hrParser = ps.ParseSourceWithOffset(&parseTree, pszStart, offset, length, charOffset, isCesu8, grfscr, &se,

&nextFunctionId, funcBody->GetRelativeLineNumber(), funcBody->GetSourceContextInfo(),

funcBody, isReparse);

Assert(FAILED(hrParser) || nextFunctionId == funcBody->deferredParseNextFunctionId || isReparse || isByteCodeDeserialization);

if (FAILED(hrParser))

{

hrParseCodeGen = MapDeferredReparseError(hrParser, se); // Map certain errors like OOM/SOE

AssertMsg(FAILED(hrParseCodeGen) && SUCCEEDED(hrParser), "Syntax errors should never be detected on deferred re-parse");

}

else

{

TRACE\_BYTECODE(L"\nDeferred parse %s\n", funcBody->GetDisplayName());

Js::AutoDynamicCodeReference dynamicFunctionReference(m\_scriptContext);

bool forceNoNative = isReparse ? this->GetScriptContext()->IsInterpreted() : false;

hrParseCodeGen = GenerateByteCode(parseTree, grfscr, m\_scriptContext,

funcBody->GetParseableFunctionInfoRef(), funcBody->GetSourceIndex(),

forceNoNative, &ps, &se, funcBody->GetScopeInfo(), functionRef);

if (se.ei.scode == JSERR\_AsmJsCompileError)

{

// if asm.js compilation failed, reparse without asm.js

m\_grfscr |= fscrNoAsmJs;

se.Clear();

return Parse(functionRef, isByteCodeDeserialization);

}

if (SUCCEEDED(hrParseCodeGen))

{

fParsed = TRUE;

}

else

{

Assert(hrParseCodeGen == SCRIPT\_E\_RECORDED);

hrParseCodeGen = se.ei.scode;

}

}

}

END\_TRANSLATE\_EXCEPTION\_TO\_HRESULT(hr);

}

END\_LEAVE\_SCRIPT\_INTERNAL(m\_scriptContext);

if (hr == E\_OUTOFMEMORY)

{

JavascriptError::ThrowOutOfMemoryError(m\_scriptContext);

}

else if(hr == VBSERR\_OutOfStack)

{

JavascriptError::ThrowStackOverflowError(m\_scriptContext);

}

else if(hr == E\_ABORT)

{

throw Js::ScriptAbortException();

}

else if(FAILED(hr))

{

throw Js::InternalErrorException();

}

Assert(hr == NO\_ERROR);

if (!SUCCEEDED(hrParser))

{

JavascriptError::ThrowError(m\_scriptContext, VBSERR\_InternalError);

}

else if (!SUCCEEDED(hrParseCodeGen))

{

/\*

\* VBSERR\_OutOfStack is of type kjstError but we throw a (more specific) StackOverflowError when a hard stack

\* overflow occurs. To keep the behavior consistent I'm special casing it here.

\*/

if (hrParseCodeGen == VBSERR\_OutOfStack)

{

JavascriptError::ThrowStackOverflowError(m\_scriptContext);

}

JavascriptError::MapAndThrowError(m\_scriptContext, hrParseCodeGen);

}

}

else

{

fParsed = FALSE;

}

if (fParsed == TRUE)

{

this->UpdateFunctionBodyImpl(funcBody);

this->m\_hasBeenParsed = true;

}

returnFunctionBody = GetFunctionBody();

LEAVE\_PINNED\_SCOPE();

return returnFunctionBody;

}

#ifndef TEMP\_DISABLE\_ASMJS

FunctionBody\* ParseableFunctionInfo::ParseAsmJs(Parser \* ps, \_\_out CompileScriptException \* se, \_\_out ParseNodePtr \* parseTree)

{

Assert(IsDeferredParseFunction());

Assert(m\_isAsmjsMode);

FunctionBody\* returnFunctionBody = nullptr;

ENTER\_PINNED\_SCOPE(Js::PropertyRecordList, propertyRecordList);

Recycler\* recycler = this->m\_scriptContext->GetRecycler();

propertyRecordList = RecyclerNew(recycler, Js::PropertyRecordList, recycler);

FunctionBody\* funcBody = nullptr;

funcBody = FunctionBody::NewFromRecycler(

this->m\_scriptContext,

this->m\_displayName,

this->m\_displayNameLength,

this->m\_displayShortNameOffset,

this->m\_nestedCount,

this->m\_utf8SourceInfo,

this->m\_functionNumber,

m\_utf8SourceInfo->GetSrcInfo()->sourceContextInfo->sourceContextId, /\* script id \*/

this->functionId, /\* function id \*/

propertyRecordList,

(Attributes)(this->GetAttributes() & ~(Attributes::DeferredDeserialize | Attributes::DeferredParse))

#ifdef PERF\_COUNTERS

, false /\* is function from deferred deserialized proxy \*/

#endif

);

this->Copy(funcBody);

PERF\_COUNTER\_DEC(Code, DeferedFunction);

if (!this->GetSourceContextInfo()->IsDynamic())

{

PHASE\_PRINT\_TESTTRACE1(Js::DeferParsePhase, L"TestTrace: Deferred function parsed - ID: %d; Display Name: %s; Length: %d; Nested Function Count: %d; Utf8SourceInfo: %d; Source Length: %d; Is Top Level: %s; Source Url: %s\n", m\_functionNumber, m\_displayName, this->m\_cchLength, this->GetNestedCount(), this->m\_utf8SourceInfo->GetSourceInfoId(), this->m\_utf8SourceInfo->GetCchLength(), this->GetIsTopLevel() ? L"True" : L"False", this->GetSourceContextInfo()->url);

}

else

{

PHASE\_PRINT\_TESTTRACE1(Js::DeferParsePhase, L"TestTrace: Deferred function parsed - ID: %d; Display Name: %s; Length: %d; Nested Function Count: %d; Utf8SourceInfo: %d; Source Length: %d\n; Is Top Level: %s;", m\_functionNumber, m\_displayName, this->m\_cchLength, this->GetNestedCount(), this->m\_utf8SourceInfo->GetSourceInfoId(), this->m\_utf8SourceInfo->GetCchLength(), this->GetIsTopLevel() ? L"True" : L"False");

}

#if ENABLE\_PROFILE\_INFO

Assert(!funcBody->HasExecutionDynamicProfileInfo());

#endif

HRESULT hrParser = NO\_ERROR;

HRESULT hrParseCodeGen = NO\_ERROR;

bool isCesu8 = m\_scriptContext->GetSource(funcBody->GetSourceIndex())->IsCesu8();

size\_t offset = this->StartOffset();

charcount\_t charOffset = this->StartInDocument();

size\_t length = this->LengthInBytes();

LPCUTF8 pszStart = this->GetStartOfDocument();

ulong grfscr = funcBody->GetGrfscr() | fscrDeferredFnc | fscrDeferredFncExpression;

uint nextFunctionId = funcBody->GetLocalFunctionId();

// if parser throws, it will be caught by function trying to bytecode gen the asm.js module, so don't need to catch/rethrow here

hrParser = ps->ParseSourceWithOffset(parseTree, pszStart, offset, length, charOffset, isCesu8, grfscr, se,

&nextFunctionId, funcBody->GetRelativeLineNumber(), funcBody->GetSourceContextInfo(),

funcBody, false);

Assert(FAILED(hrParser) || funcBody->deferredParseNextFunctionId == nextFunctionId);

if (FAILED(hrParser))

{

hrParseCodeGen = MapDeferredReparseError(hrParser, \*se); // Map certain errors like OOM/SOE

AssertMsg(FAILED(hrParseCodeGen) && SUCCEEDED(hrParser), "Syntax errors should never be detected on deferred re-parse");

}

if (!SUCCEEDED(hrParser))

{

Throw::InternalError();

}

else if (!SUCCEEDED(hrParseCodeGen))

{

if (hrParseCodeGen == VBSERR\_OutOfStack)

{

Throw::StackOverflow(m\_scriptContext, nullptr);

}

else

{

Assert(hrParseCodeGen == E\_OUTOFMEMORY);

Throw::OutOfMemory();

}

}

UpdateFunctionBodyImpl(funcBody);

m\_hasBeenParsed = true;

returnFunctionBody = GetFunctionBody();

LEAVE\_PINNED\_SCOPE();

return returnFunctionBody;

}

#endif

void ParseableFunctionInfo::Finalize(bool isShutdown)

{

\_\_super::Finalize(isShutdown);

if (!this->m\_hasBeenParsed)

{

if (!this->GetSourceContextInfo()->IsDynamic() && !this->GetIsTopLevel())

{

this->m\_utf8SourceInfo->StopTrackingDeferredFunction(this->GetLocalFunctionId());

}

PERF\_COUNTER\_DEC(Code, DeferedFunction);

}

}

bool ParseableFunctionInfo::IsFakeGlobalFunc(ulong flags) const

{

return GetIsGlobalFunc() && !(flags & fscrGlobalCode);

}

bool ParseableFunctionInfo::GetExternalDisplaySourceName(BSTR\* sourceName)

{

Assert(sourceName);

if (IsDynamicScript() && GetUtf8SourceInfo()->GetDebugDocumentName(sourceName))

{

return true;

}

\*sourceName = ::SysAllocString(GetSourceName());

return \*sourceName != nullptr;

}

const wchar\_t\* FunctionProxy::WrapWithBrackets(const wchar\_t\* name, charcount\_t sz, ScriptContext\* scriptContext)

{

wchar\_t \* wrappedName = RecyclerNewArrayLeaf(scriptContext->GetRecycler(), wchar\_t, sz + 3); //[]\0

wrappedName[0] = L'[';

wchar\_t \*next = wrappedName;

js\_wmemcpy\_s(++next, sz, name, sz);

wrappedName[sz + 1] = L']';

wrappedName[sz + 2] = L'\0';

return wrappedName;

}

const wchar\_t\* FunctionProxy::GetShortDisplayName(charcount\_t \* shortNameLength)

{

const wchar\_t\* name = this->GetDisplayName();

uint nameLength = this->GetDisplayNameLength();

if (name == nullptr)

{

\*shortNameLength = 0;

return Constants::Empty;

}

if (IsConstantFunctionName(name))

{

\*shortNameLength = nameLength;

return name;

}

uint shortNameOffset = this->GetShortDisplayNameOffset();

const wchar\_t \* shortName = name + shortNameOffset;

bool isBracketCase = shortNameOffset != 0 && name[shortNameOffset-1] == '[';

Assert(nameLength >= shortNameOffset);

\*shortNameLength = nameLength - shortNameOffset;

if (!isBracketCase)

{

return shortName;

}

Assert(name[nameLength - 1] == ']');

wchar\_t \* finalshorterName = RecyclerNewArrayLeaf(this->GetScriptContext()->GetRecycler(), wchar\_t, \*shortNameLength);

js\_wmemcpy\_s(finalshorterName, \*shortNameLength, shortName, \*shortNameLength - 1); // we don't want the last character in shorterName

finalshorterName[\*shortNameLength - 1] = L'\0';

\*shortNameLength = \*shortNameLength - 1;

return finalshorterName;

}

/\*static\*/

bool FunctionProxy::IsConstantFunctionName(const wchar\_t\* srcName)

{

if (srcName == Js::Constants::GlobalFunction ||

srcName == Js::Constants::AnonymousFunction ||

srcName == Js::Constants::GlobalCode ||

srcName == Js::Constants::Anonymous ||

srcName == Js::Constants::UnknownScriptCode ||

srcName == Js::Constants::FunctionCode)

{

return true;

}

return false;

}

/\*static \*/

/\*Return value: Whether the target value is a recycler pointer or not\*/

bool FunctionProxy::SetDisplayName(const wchar\_t\* srcName, const wchar\_t\*\* destName, uint displayNameLength, ScriptContext \* scriptContext, SetDisplayNameFlags flags /\* default to None \*/)

{

Assert(destName);

Assert(scriptContext);

if (srcName == nullptr)

{

\*destName = (L"");

return false;

}

else if (IsConstantFunctionName(srcName) || (flags & SetDisplayNameFlagsDontCopy) != 0)

{

\*destName = srcName;

return (flags & SetDisplayNameFlagsRecyclerAllocated) != 0; // Return true if array is recycler allocated

}

else

{

uint numCharacters = displayNameLength + 1;

Assert((flags & SetDisplayNameFlagsDontCopy) == 0);

\*destName = RecyclerNewArrayLeaf(scriptContext->GetRecycler(), wchar\_t, numCharacters);

js\_wmemcpy\_s((wchar\_t \*)\*destName, numCharacters, srcName, numCharacters);

((wchar\_t \*)(\*destName))[numCharacters - 1] = L'\0';

return true;

}

}

void FunctionProxy::SetDisplayName(const wchar\_t\* srcName, WriteBarrierPtr<const wchar\_t>\* destName, uint displayNameLength, ScriptContext \* scriptContext, SetDisplayNameFlags flags /\* default to None \*/)

{

const wchar\_t\* dest = nullptr;

bool targetIsRecyclerMemory = SetDisplayName(srcName, &dest, displayNameLength, scriptContext, flags);

if (targetIsRecyclerMemory)

{

\*destName = dest;

}

else

{

destName->NoWriteBarrierSet(dest);

}

}

void ParseableFunctionInfo::SetDisplayName(const wchar\_t\* pszDisplayName)

{

size\_t len = wcslen(pszDisplayName);

if (len > UINT\_MAX)

{

// Can't support display name that big

Js::Throw::OutOfMemory();

}

SetDisplayName(pszDisplayName, (uint)len, 0);

}

void ParseableFunctionInfo::SetDisplayName(const wchar\_t\* pszDisplayName, uint displayNameLength, uint displayShortNameOffset, SetDisplayNameFlags flags /\* default to None \*/)

{

this->m\_displayNameLength = displayNameLength;

this->m\_displayShortNameOffset = displayShortNameOffset;

FunctionProxy::SetDisplayName(pszDisplayName, &this->m\_displayName, displayNameLength, m\_scriptContext, flags);

}

// SourceInfo methods

FunctionBody::StatementMapList \* FunctionBody::GetStatementMaps() const

{

return this->pStatementMaps;

}

/\* static \*/ FunctionBody::StatementMap \* FunctionBody::GetNextNonSubexpressionStatementMap(StatementMapList \*statementMapList, int & startingAtIndex)

{

AssertMsg(statementMapList != nullptr, "Must have valid statementMapList to execute");

FunctionBody::StatementMap \*map = statementMapList->Item(startingAtIndex);

while (map->isSubexpression && startingAtIndex < statementMapList->Count() - 1)

{

map = statementMapList->Item(++startingAtIndex);

}

if (map->isSubexpression) // Didn't find any non inner maps

{

return nullptr;

}

return map;

}

/\* static \*/ FunctionBody::StatementMap \* FunctionBody::GetPrevNonSubexpressionStatementMap(StatementMapList \*statementMapList, int & startingAtIndex)

{

AssertMsg(statementMapList != nullptr, "Must have valid statementMapList to execute");

FunctionBody::StatementMap \*map = statementMapList->Item(startingAtIndex);

while (startingAtIndex && map->isSubexpression)

{

map = statementMapList->Item(--startingAtIndex);

}

if (map->isSubexpression) // Didn't find any non inner maps

{

return nullptr;

}

return map;

}

void ParseableFunctionInfo::CloneSourceInfo(ScriptContext\* scriptContext, const ParseableFunctionInfo& other, ScriptContext\* othersScriptContext, uint sourceIndex)

{

if (!m\_utf8SourceHasBeenSet)

{

this->m\_utf8SourceInfo = scriptContext->GetSource(sourceIndex);

this->m\_sourceIndex = sourceIndex;

this->m\_cchStartOffset = other.m\_cchStartOffset;

this->m\_cchLength = other.m\_cchLength;

this->m\_lineNumber = other.m\_lineNumber;

this->m\_columnNumber = other.m\_columnNumber;

this->m\_isEval = other.m\_isEval;

this->m\_isDynamicFunction = other.m\_isDynamicFunction;

this->m\_cbStartOffset = other.StartOffset();

this->m\_cbLength = other.LengthInBytes();

this->m\_utf8SourceHasBeenSet = true;

if (this->IsFunctionBody())

{

this->GetFunctionBody()->FinishSourceInfo();

}

}

#if DBG

else

{

AssertMsg(this->m\_cchStartOffset == other.m\_cchStartOffset, "Mismatched source character start offset");

AssertMsg(this->StartOffset() == other.StartOffset(), "Mismatched source start offset");

AssertMsg(this->m\_cchLength == other.m\_cchLength, "Mismatched source character length");

AssertMsg(this->LengthInBytes() == other.LengthInBytes(), "Mismatch source byte length");

AssertMsg(this->GetUtf8SourceInfo()->GetSourceHolder() == scriptContext->GetSource(this->m\_sourceIndex)->GetSourceHolder(),

"Mismatched source holder pointer");

AssertMsg(this->m\_isEval == other.m\_isEval, "Mismatched source type");

AssertMsg(this->m\_isDynamicFunction == other.m\_isDynamicFunction, "Mismatch source type");

}

#endif

}

void ParseableFunctionInfo::SetSourceInfo(uint sourceIndex, ParseNodePtr node, bool isEval, bool isDynamicFunction)

{

if (!m\_utf8SourceHasBeenSet)

{

this->m\_sourceIndex = sourceIndex;

this->m\_cchStartOffset = node->ichMin;

this->m\_cchLength = node->LengthInCodepoints();

this->m\_lineNumber = node->sxFnc.lineNumber;

this->m\_columnNumber = node->sxFnc.columnNumber;

this->m\_isEval = isEval;

this->m\_isDynamicFunction = isDynamicFunction;

// It would have been better if we detect and reject large source buffer eariler before parsing

size\_t cbMin = node->sxFnc.cbMin;

size\_t lengthInBytes = node->sxFnc.LengthInBytes();

if (cbMin > UINT\_MAX || lengthInBytes > UINT\_MAX)

{

Js::Throw::OutOfMemory();

}

this->m\_cbStartOffset = (uint)cbMin;

this->m\_cbLength = (uint)lengthInBytes;

Assert(this->m\_utf8SourceInfo != nullptr);

this->m\_utf8SourceHasBeenSet = true;

if (this->IsFunctionBody())

{

this->GetFunctionBody()->FinishSourceInfo();

}

}

#if DBG

else

{

AssertMsg(this->m\_sourceIndex == sourceIndex, "Mismatched source index");

if (!this->GetIsGlobalFunc())

{

// In the global function case with a @cc\_on, we modify some of these values so it might

// not match on reparse (see ParseableFunctionInfo::Parse()).

AssertMsg(this->StartOffset() == node->sxFnc.cbMin, "Mismatched source start offset");

AssertMsg(this->m\_cchStartOffset == node->ichMin, "Mismatched source character start offset");

AssertMsg(this->m\_cchLength == node->LengthInCodepoints(), "Mismatched source length");

AssertMsg(this->LengthInBytes() == node->sxFnc.LengthInBytes(), "Mismatched source encoded byte length");

}

AssertMsg(this->m\_isEval == isEval, "Mismatched source type");

AssertMsg(this->m\_isDynamicFunction == isDynamicFunction, "Mismatch source type");

}

#endif

#if DBG\_DUMP

if (PHASE\_TRACE1(Js::FunctionSourceInfoParsePhase))

{

if (this->HasBody())

{

FunctionProxy\* proxy = this->GetFunctionProxy();

if (proxy->IsFunctionBody())

{

FunctionBody\* functionBody = this->GetFunctionBody();

Assert( functionBody != nullptr );

functionBody->PrintStatementSourceLineFromStartOffset(functionBody->StartInDocument());

Output::Flush();

}

}

}

#endif

}

bool FunctionBody::Is(void\* ptr)

{

if(!ptr)

{

return false;

}

return VirtualTableInfo<FunctionBody>::HasVirtualTable(ptr);

}

bool FunctionBody::HasLineBreak() const

{

return this->HasLineBreak(this->StartOffset(), this->m\_cchStartOffset + this->m\_cchLength);

}

bool FunctionBody::HasLineBreak(charcount\_t start, charcount\_t end) const

{

if (start > end) return false;

charcount\_t cchLength = end - start;

if (start < this->m\_cchStartOffset || cchLength > this->m\_cchLength) return false;

LPCUTF8 src = this->GetSource(L"FunctionBody::HasLineBreak");

LPCUTF8 last = src + this->LengthInBytes();

size\_t offset = this->LengthInBytes() == this->m\_cchLength ?

start - this->m\_cchStartOffset :

utf8::CharacterIndexToByteIndex(src, this->LengthInBytes(), start - this->m\_cchStartOffset, utf8::doAllowThreeByteSurrogates);

src = src + offset;

utf8::DecodeOptions options = utf8::doAllowThreeByteSurrogates;

for (charcount\_t cch = cchLength; cch > 0; --cch)

{

switch (utf8::Decode(src, last, options))

{

case '\r':

case '\n':

case 0x2028:

case 0x2029:

return true;

}

}

return false;

}

FunctionBody::StatementMap\* FunctionBody::GetMatchingStatementMapFromByteCode(int byteCodeOffset, bool ignoreSubexpressions /\* = false \*/)

{

StatementMapList \* pStatementMaps = this->GetStatementMaps();

if (pStatementMaps)

{

Assert(m\_sourceInfo.pSpanSequence == nullptr);

for (int index = 0; index < pStatementMaps->Count(); index++)

{

FunctionBody::StatementMap\* pStatementMap = pStatementMaps->Item(index);

if (!(ignoreSubexpressions && pStatementMap->isSubexpression) && pStatementMap->byteCodeSpan.Includes(byteCodeOffset))

{

return pStatementMap;

}

}

}

return nullptr;

}

// Returns the StatementMap for the offset.

// 1. Current statementMap if bytecodeoffset falls within bytecode's span

// 2. Previous if the bytecodeoffset is in between previous's end to current's begin

FunctionBody::StatementMap\* FunctionBody::GetEnclosingStatementMapFromByteCode(int byteCodeOffset, bool ignoreSubexpressions /\* = false \*/)

{

int index = GetEnclosingStatementIndexFromByteCode(byteCodeOffset, ignoreSubexpressions);

if (index != -1)

{

return this->GetStatementMaps()->Item(index);

}

return nullptr;

}

// Returns the index of StatementMap for

// 1. Current statementMap if bytecodeoffset falls within bytecode's span

// 2. Previous if the bytecodeoffset is in between previous's end to current's begin

// 3. -1 of the failures.

int FunctionBody::GetEnclosingStatementIndexFromByteCode(int byteCodeOffset, bool ignoreSubexpressions /\* = false \*/)

{

StatementMapList \* pStatementMaps = this->GetStatementMaps();

if (pStatementMaps == nullptr)

{

// e.g. internal library.

return -1;

}

Assert(m\_sourceInfo.pSpanSequence == nullptr);

for (int index = 0; index < pStatementMaps->Count(); index++)

{

FunctionBody::StatementMap\* pStatementMap = pStatementMaps->Item(index);

if (!(ignoreSubexpressions && pStatementMap->isSubexpression) && pStatementMap->byteCodeSpan.Includes(byteCodeOffset))

{

return index;

}

else if (!pStatementMap->isSubexpression && byteCodeOffset < pStatementMap->byteCodeSpan.begin) // We always ignore sub expressions when checking if we went too far

{

return index > 0 ? index - 1 : 0;

}

}

return pStatementMaps->Count() - 1;

}

// In some cases in legacy mode, due to the state scriptContext->windowIdList, the parser might not detect an eval call in the first parse but do so in the reparse

// This fixes up the state at the start of reparse

void FunctionBody::SaveState(ParseNodePtr pnode)

{

Assert(!this->IsReparsed());

this->SetChildCallsEval(!!pnode->sxFnc.ChildCallsEval());

this->SetCallsEval(!!pnode->sxFnc.CallsEval());

this->SetHasReferenceableBuiltInArguments(!!pnode->sxFnc.HasReferenceableBuiltInArguments());

}

void FunctionBody::RestoreState(ParseNodePtr pnode)

{

Assert(this->IsReparsed());

#if ENABLE\_DEBUG\_CONFIG\_OPTIONS

wchar\_t debugStringBuffer[MAX\_FUNCTION\_BODY\_DEBUG\_STRING\_SIZE];

#endif

if(!!pnode->sxFnc.ChildCallsEval() != this->GetChildCallsEval())

{

OUTPUT\_VERBOSE\_TRACE(Js::DebuggerPhase, L"Child calls eval is different on debug reparse: %s(%s)\n", this->GetExternalDisplayName(), this->GetDebugNumberSet(debugStringBuffer));

}

if(!!pnode->sxFnc.CallsEval() != this->GetCallsEval())

{

OUTPUT\_VERBOSE\_TRACE(Js::DebuggerPhase, L"Calls eval is different on debug reparse: %s(%s)\n", this->GetExternalDisplayName(), this->GetDebugNumberSet(debugStringBuffer));

}

if(!!pnode->sxFnc.HasReferenceableBuiltInArguments() != this->HasReferenceableBuiltInArguments())

{

OUTPUT\_VERBOSE\_TRACE(Js::DebuggerPhase, L"Referencable Built in args is different on debug reparse: %s(%s)\n", this->GetExternalDisplayName(), this->GetDebugNumberSet(debugStringBuffer));

}

pnode->sxFnc.SetChildCallsEval(this->GetChildCallsEval());

pnode->sxFnc.SetCallsEval(this->GetCallsEval());

pnode->sxFnc.SetHasReferenceableBuiltInArguments(this->HasReferenceableBuiltInArguments());

}

// Retrieves statement map for given byte code offset.

// Parameters:

// - sourceOffset: byte code offset to get map for.

// - mapIndex: if not NULL, receives the index of found map.

FunctionBody::StatementMap\* FunctionBody::GetMatchingStatementMapFromSource(int sourceOffset, int\* pMapIndex /\* = nullptr \*/)

{

StatementMapList \* pStatementMaps = this->GetStatementMaps();

if (pStatementMaps && pStatementMaps->Count() > 0)

{

Assert(m\_sourceInfo.pSpanSequence == nullptr);

for (int index = pStatementMaps->Count() - 1; index >= 0; index--)

{

FunctionBody::StatementMap\* pStatementMap = pStatementMaps->Item(index);

if (!pStatementMap->isSubexpression && pStatementMap->sourceSpan.Includes(sourceOffset))

{

if (pMapIndex)

{

\*pMapIndex = index;

}

return pStatementMap;

}

}

}

if (pMapIndex)

{

\*pMapIndex = 0;

}

return nullptr;

}

//

// The function determine the line and column for a bytecode offset within the current script buffer.

//

bool FunctionBody::GetLineCharOffset(int byteCodeOffset, ULONG\* \_line, LONG\* \_charOffset, bool canAllocateLineCache /\*= true\*/)

{

Assert(!this->GetUtf8SourceInfo()->GetIsLibraryCode());

int startCharOfStatement = this->m\_cchStartOffset; // Default to the start of this function

if (m\_sourceInfo.pSpanSequence)

{

SmallSpanSequenceIter iter;

m\_sourceInfo.pSpanSequence->Reset(iter);

StatementData data;

if (m\_sourceInfo.pSpanSequence->GetMatchingStatementFromBytecode(byteCodeOffset, iter, data)

&& EndsAfter(data.sourceBegin))

{

startCharOfStatement = data.sourceBegin;

}

}

else

{

Js::FunctionBody::StatementMap\* map = this->GetEnclosingStatementMapFromByteCode(byteCodeOffset, false);

if (map && EndsAfter(map->sourceSpan.begin))

{

startCharOfStatement = map->sourceSpan.begin;

}

}

return this->GetLineCharOffsetFromStartChar(startCharOfStatement, \_line, \_charOffset, canAllocateLineCache);

}

bool FunctionBody::GetLineCharOffsetFromStartChar(int startCharOfStatement, ULONG\* \_line, LONG\* \_charOffset, bool canAllocateLineCache /\*= true\*/)

{

Assert(!this->GetUtf8SourceInfo()->GetIsLibraryCode());

// The following adjusts for where the script is within the document

ULONG line = this->GetHostStartLine();

charcount\_t column = 0;

ULONG lineCharOffset = 0;

charcount\_t lineByteOffset = 0;

if (startCharOfStatement > 0)

{

bool doSlowLookup = !canAllocateLineCache;

if (canAllocateLineCache)

{

HRESULT hr = m\_utf8SourceInfo->EnsureLineOffsetCacheNoThrow();

if (FAILED(hr))

{

if (hr != E\_OUTOFMEMORY)

{

Assert(hr == E\_ABORT); // The only other possible error we know about is ScriptAbort from QueryContinue.

return false;

}

// Clear the cache so it is not used.

this->m\_utf8SourceInfo->DeleteLineOffsetCache();

// We can try and do the slow lookup below

doSlowLookup = true;

}

}

charcount\_t cacheLine = 0;

this->m\_utf8SourceInfo->GetLineInfoForCharPosition(startCharOfStatement, &cacheLine, &column, &lineByteOffset, doSlowLookup);

// Update the tracking variables to jump to the line position (only need to jump if not on the first line).

if (cacheLine > 0)

{

line += cacheLine;

lineCharOffset = startCharOfStatement - column;

}

}

if (this->GetSourceContextInfo()->IsDynamic() && this->m\_isDynamicFunction)

{

line -= JavascriptFunction::numberLinesPrependedToAnonymousFunction;

}

if(\_line)

{

\*\_line = line;

}

if(\_charOffset)

{

\*\_charOffset = column;

// If we are at the beginning of the host code, adjust the offset based on the host provided offset

if (this->GetHostSrcInfo()->dlnHost == line)

{

\*\_charOffset += (LONG)this->GetHostStartColumn();

}

}

return true;

}

bool FunctionBody::GetStatementIndexAndLengthAt(int byteCodeOffset, UINT32\* statementIndex, UINT32\* statementLength)

{

Assert(statementIndex != nullptr);

Assert(statementLength != nullptr);

Assert(m\_scriptContext->IsInDebugMode());

StatementMap \* statement = GetEnclosingStatementMapFromByteCode(byteCodeOffset, false);

Assert(statement != nullptr);

// Bailout if we are unable to find a statement.

// We shouldn't be missing these when a debugger is attached but we don't want to AV on retail builds.

if (statement == nullptr)

{

return false;

}

Assert(m\_utf8SourceInfo);

const SRCINFO \* srcInfo = m\_utf8SourceInfo->GetSrcInfo();

// Offset from the beginning of the document minus any host-supplied source characters.

// Host supplied characters are inserted (for example) around onload:

// onload="foo('somestring', 0)" -> function onload(event).{.foo('somestring', 0).}

ULONG offsetFromDocumentBegin = srcInfo ? srcInfo->ulCharOffset - srcInfo->ichMinHost : 0;

\*statementIndex = statement->sourceSpan.Begin() + offsetFromDocumentBegin;

\*statementLength = statement->sourceSpan.End() - statement->sourceSpan.Begin();

return true;

}

void FunctionBody::RecordFrameDisplayRegister(RegSlot slot)

{

AssertMsg(slot != 0, "The assumption that the Frame Display Register cannot be at the 0 slot is wrong.");

SetFrameDisplayRegister(slot);

}

void FunctionBody::RecordObjectRegister(RegSlot slot)

{

AssertMsg(slot != 0, "The assumption that the Object Register cannot be at the 0 slot is wrong.");

SetObjectRegister(slot);

}

Js::RootObjectBase \* FunctionBody::GetRootObject() const

{

// Safe to be used by the JIT thread

Assert(this->m\_constTable != nullptr);

return (Js::RootObjectBase \*)this->m\_constTable[Js::FunctionBody::RootObjectRegSlot - FunctionBody::FirstRegSlot];

}

Js::RootObjectBase \* FunctionBody::LoadRootObject() const

{

if (this->GetModuleID() == kmodGlobal)

{

return JavascriptOperators::OP\_LdRoot(this->GetScriptContext());

}

return JavascriptOperators::GetModuleRoot(this->GetModuleID(), this->GetScriptContext());

}

#if ENABLE\_NATIVE\_CODEGEN

FunctionEntryPointInfo \* FunctionBody::GetEntryPointFromNativeAddress(DWORD\_PTR codeAddress)

{

FunctionEntryPointInfo \* entryPoint = nullptr;

this->MapEntryPoints([&entryPoint, &codeAddress](int index, FunctionEntryPointInfo \* currentEntryPoint)

{

// We need to do a second check for IsNativeCode because the entry point could be in the process of

// being recorded on the background thread

if (currentEntryPoint->IsInNativeAddressRange(codeAddress))

{

entryPoint = currentEntryPoint;

}

});

return entryPoint;

}

LoopEntryPointInfo \* FunctionBody::GetLoopEntryPointInfoFromNativeAddress(DWORD\_PTR codeAddress, uint loopNum) const

{

LoopEntryPointInfo \* entryPoint = nullptr;

LoopHeader \* loopHeader = this->GetLoopHeader(loopNum);

Assert(loopHeader);

loopHeader->MapEntryPoints([&](int index, LoopEntryPointInfo \* currentEntryPoint)

{

if (currentEntryPoint->IsCodeGenDone() &&

codeAddress >= currentEntryPoint->GetNativeAddress() &&

codeAddress < currentEntryPoint->GetNativeAddress() + currentEntryPoint->GetCodeSize())

{

entryPoint = currentEntryPoint;

}

});

return entryPoint;

}

int FunctionBody::GetStatementIndexFromNativeOffset(SmallSpanSequence \*pThrowSpanSequence, uint32 nativeOffset)

{

int statementIndex = -1;

if (pThrowSpanSequence)

{

SmallSpanSequenceIter iter;

StatementData tmpData;

if (pThrowSpanSequence->GetMatchingStatementFromBytecode(nativeOffset, iter, tmpData))

{

statementIndex = tmpData.sourceBegin; // sourceBegin represents statementIndex here

}

else

{

// If nativeOffset falls on the last span, GetMatchingStatement would miss it because SmallSpanSequence

// does not know about the last span end. Since we checked that codeAddress is within our range,

// we can safely consider it matches the last span.

statementIndex = iter.accumulatedSourceBegin;

}

}

return statementIndex;

}

int FunctionBody::GetStatementIndexFromNativeAddress(SmallSpanSequence \*pThrowSpanSequence, DWORD\_PTR codeAddress, DWORD\_PTR nativeBaseAddress)

{

uint32 nativeOffset = (uint32)(codeAddress - nativeBaseAddress);

return GetStatementIndexFromNativeOffset(pThrowSpanSequence, nativeOffset);

}

#endif

BOOL FunctionBody::GetMatchingStatementMap(StatementData &data, int statementIndex, FunctionBody \*inlinee)

{

SourceInfo \*si = &this->m\_sourceInfo;

if (inlinee)

{

si = &inlinee->m\_sourceInfo;

Assert(si);

}

if (statementIndex >= 0)

{

SmallSpanSequence \*pSpanSequence = si->pSpanSequence;

if (pSpanSequence)

{

SmallSpanSequenceIter iter;

pSpanSequence->Reset(iter);

if (pSpanSequence->Item(statementIndex, iter, data))

{

return TRUE;

}

}

else

{

StatementMapList\* pStatementMaps = GetStatementMaps();

Assert(pStatementMaps);

if (statementIndex >= pStatementMaps->Count())

{

return FALSE;

}

data.sourceBegin = pStatementMaps->Item(statementIndex)->sourceSpan.begin;

data.bytecodeBegin = pStatementMaps->Item(statementIndex)->byteCodeSpan.begin;

return TRUE;

}

}

return FALSE;

}

void FunctionBody::FindClosestStatements(long characterOffset, StatementLocation \*firstStatementLocation, StatementLocation \*secondStatementLocation)

{

auto statementMaps = this->GetStatementMaps();

if (statementMaps)

{

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

{

regex::Interval\* pSourceSpan = &(statementMaps->Item(i)->sourceSpan);

if (FunctionBody::IsDummyGlobalRetStatement(pSourceSpan))

{

// Workaround for handling global return, which is a empty range.

continue;

}

if (pSourceSpan->begin < characterOffset

&& (firstStatementLocation->function == nullptr || firstStatementLocation->statement.begin < pSourceSpan->begin))

{

firstStatementLocation->function = this;

firstStatementLocation->statement = \*pSourceSpan;

firstStatementLocation->bytecodeSpan = statementMaps->Item(i)->byteCodeSpan;

}

else if (pSourceSpan->begin >= characterOffset

&& (secondStatementLocation->function == nullptr || secondStatementLocation->statement.begin > pSourceSpan->begin))

{

secondStatementLocation->function = this;

secondStatementLocation->statement = \*pSourceSpan;

secondStatementLocation->bytecodeSpan = statementMaps->Item(i)->byteCodeSpan;

}

}

}

}

#if ENABLE\_NATIVE\_CODEGEN

BOOL FunctionBody::GetMatchingStatementMapFromNativeAddress(DWORD\_PTR codeAddress, StatementData &data, uint loopNum, FunctionBody \*inlinee /\* = nullptr \*/)

{

SmallSpanSequence \* spanSequence = nullptr;

DWORD\_PTR nativeBaseAddress = NULL;

EntryPointInfo \* entryPoint;

if (loopNum == -1)

{

entryPoint = GetEntryPointFromNativeAddress(codeAddress);

}

else

{

entryPoint = GetLoopEntryPointInfoFromNativeAddress(codeAddress, loopNum);

}

if (entryPoint != nullptr)

{

spanSequence = entryPoint->GetNativeThrowSpanSequence();

nativeBaseAddress = entryPoint->GetNativeAddress();

}

int statementIndex = GetStatementIndexFromNativeAddress(spanSequence, codeAddress, nativeBaseAddress);

return GetMatchingStatementMap(data, statementIndex, inlinee);

}

BOOL FunctionBody::GetMatchingStatementMapFromNativeOffset(DWORD\_PTR codeAddress, uint32 offset, StatementData &data, uint loopNum, FunctionBody \*inlinee /\* = nullptr \*/)

{

EntryPointInfo \* entryPoint;

if (loopNum == -1)

{

entryPoint = GetEntryPointFromNativeAddress(codeAddress);

}

else

{

entryPoint = GetLoopEntryPointInfoFromNativeAddress(codeAddress, loopNum);

}

SmallSpanSequence \*spanSequence = entryPoint ? entryPoint->GetNativeThrowSpanSequence() : nullptr;

int statementIndex = GetStatementIndexFromNativeOffset(spanSequence, offset);

return GetMatchingStatementMap(data, statementIndex, inlinee);

}

#endif

#if ENABLE\_PROFILE\_INFO

void FunctionBody::LoadDynamicProfileInfo()

{

SourceDynamicProfileManager \* sourceDynamicProfileManager = GetSourceContextInfo()->sourceDynamicProfileManager;

if (sourceDynamicProfileManager != nullptr)

{

this->dynamicProfileInfo = sourceDynamicProfileManager->GetDynamicProfileInfo(this);

#if DBG\_DUMP

if(this->dynamicProfileInfo)

{

if (Configuration::Global.flags.Dump.IsEnabled(DynamicProfilePhase, this->GetSourceContextId(), this->GetLocalFunctionId()))

{

Output::Print(L"Loaded:");

this->dynamicProfileInfo->Dump(this);

}

}

#endif

}

#ifdef DYNAMIC\_PROFILE\_MUTATOR

DynamicProfileMutator::Mutate(this);

#endif

}

bool FunctionBody::NeedEnsureDynamicProfileInfo() const

{

// Only need to ensure dynamic profile if we don't already have link up the dynamic profile info

// and dynamic profile collection is enabled

return

!this->m\_isFromNativeCodeModule &&

!this->m\_isAsmJsFunction &&

!this->HasExecutionDynamicProfileInfo() &&

DynamicProfileInfo::IsEnabled(this);

}

DynamicProfileInfo \* FunctionBody::EnsureDynamicProfileInfo()

{

if (this->NeedEnsureDynamicProfileInfo())

{

m\_scriptContext->AddDynamicProfileInfo(this, &this->dynamicProfileInfo);

Assert(!this->HasExecutionDynamicProfileInfo());

this->hasExecutionDynamicProfileInfo = true;

}

return this->dynamicProfileInfo;

}

DynamicProfileInfo\* FunctionBody::AllocateDynamicProfile()

{

return DynamicProfileInfo::New(m\_scriptContext->GetRecycler(), this);

}

#endif

BOOL FunctionBody::IsNativeOriginalEntryPoint() const

{

#if ENABLE\_NATIVE\_CODEGEN

return IsNativeFunctionAddr(this->GetScriptContext(), this->originalEntryPoint);

#else

return false;

#endif

}

bool FunctionBody::IsSimpleJitOriginalEntryPoint() const

{

const FunctionEntryPointInfo \*const simpleJitEntryPointInfo = GetSimpleJitEntryPointInfo();

return

simpleJitEntryPointInfo &&

reinterpret\_cast<Js::JavascriptMethod>(simpleJitEntryPointInfo->GetNativeAddress()) == originalEntryPoint;

}

void FunctionProxy::Finalize(bool isShutdown)

{

\_\_super::Finalize(isShutdown);

this->CleanupFunctionProxyCounters();

}

#if DBG

bool FunctionBody::HasValidSourceInfo()

{

SourceContextInfo\* sourceContextInfo;

if (m\_scriptContext->GetSourceContextInfoMap())

{

if(m\_scriptContext->GetSourceContextInfoMap()->TryGetValue(this->GetHostSourceContext(), &sourceContextInfo) &&

sourceContextInfo == this->GetSourceContextInfo())

{

return true;

}

}

Assert(this->IsDynamicScript());

if(m\_scriptContext->GetDynamicSourceContextInfoMap())

{

if(m\_scriptContext->GetDynamicSourceContextInfoMap()->TryGetValue(this->GetSourceContextInfo()->hash, &sourceContextInfo) &&

sourceContextInfo == this->GetSourceContextInfo())

{

return true;

}

}

// The SourceContextInfo will not be added to the dynamicSourceContextInfoMap, if they are host provided dynamic code. But they are valid source context info

if (this->GetSourceContextInfo()->isHostDynamicDocument)

{

return true;

}

return m\_scriptContext->IsNoContextSourceContextInfo(this->GetSourceContextInfo());

}

// originalEntryPoint: DefaultDeferredParsingThunk, DefaultDeferredDeserializeThunk, DefaultEntryThunk, dynamic interpreter thunk or native entry point

// directEntryPoint:

// if (!profiled) - DefaultDeferredParsingThunk, DefaultDeferredDeserializeThunk, DefaultEntryThunk, CheckCodeGenThunk,

// dynamic interpreter thunk, native entry point

// if (profiling) - ProfileDeferredParsingThunk, ProfileDeferredDeserializeThunk, ProfileEntryThunk, CheckCodeGenThunk

bool FunctionProxy::HasValidNonProfileEntryPoint() const

{

JavascriptMethod directEntryPoint = (JavascriptMethod)this->GetDefaultEntryPointInfo()->address;

JavascriptMethod originalEntryPoint = this->originalEntryPoint;

// Check the direct entry point to see if it is codegen thunk

// if it is not, the background codegen thread has updated both original entry point and direct entry point

// and they should still match, same as cases other then code gen

return IsIntermediateCodeGenThunk(directEntryPoint) || originalEntryPoint == directEntryPoint

#if ENABLE\_PROFILE\_INFO

|| (directEntryPoint == DynamicProfileInfo::EnsureDynamicProfileInfoThunk &&

this->IsFunctionBody() && this->GetFunctionBody()->IsNativeOriginalEntryPoint()

#ifdef ASMJS\_PLAT

|| (GetFunctionBody()->GetIsAsmJsFunction() && directEntryPoint == AsmJsDefaultEntryThunk)

|| (IsAsmJsCodeGenThunk(directEntryPoint))

#endif

);

#endif

;

}

bool FunctionProxy::HasValidProfileEntryPoint() const

{

JavascriptMethod directEntryPoint = (JavascriptMethod)this->GetDefaultEntryPointInfo()->address;

if (this->originalEntryPoint == DefaultDeferredParsingThunk)

{

return directEntryPoint == ProfileDeferredParsingThunk;

}

if (this->originalEntryPoint == DefaultDeferredDeserializeThunk)

{

return directEntryPoint == ProfileDeferredDeserializeThunk;

}

if (!this->IsFunctionBody())

{

return false;

}

#if ENABLE\_PROFILE\_INFO

FunctionBody \* functionBody = this->GetFunctionBody();

if (functionBody->IsInterpreterThunk() || functionBody->IsSimpleJitOriginalEntryPoint())

{

return directEntryPoint == ProfileEntryThunk || IsIntermediateCodeGenThunk(directEntryPoint);

}

#if ENABLE\_NATIVE\_CODEGEN

// In the profiler mode, the EnsureDynamicProfileInfoThunk is valid as we would be assigning to appropriate thunk when that thunk called.

return functionBody->IsNativeOriginalEntryPoint() &&

(directEntryPoint == DynamicProfileInfo::EnsureDynamicProfileInfoThunk || directEntryPoint == ProfileEntryThunk);

#endif

#else

return true;

#endif

}

bool FunctionProxy::HasValidEntryPoint() const

{

if (!m\_scriptContext->HadProfiled() &&

!(m\_scriptContext->IsInDebugMode() && m\_scriptContext->IsExceptionWrapperForBuiltInsEnabled()))

{

return this->HasValidNonProfileEntryPoint();

}

if (m\_scriptContext->IsProfiling())

{

return this->HasValidProfileEntryPoint();

}

return this->HasValidNonProfileEntryPoint() || this->HasValidProfileEntryPoint();

}

#endif

void ParseableFunctionInfo::SetDeferredParsingEntryPoint()

{

Assert(m\_scriptContext->DeferredParsingThunk == ProfileDeferredParsingThunk

|| m\_scriptContext->DeferredParsingThunk == DefaultDeferredParsingThunk);

this->SetEntryPoint(this->GetDefaultEntryPointInfo(), m\_scriptContext->DeferredParsingThunk);

originalEntryPoint = DefaultDeferredParsingThunk;

}

void ParseableFunctionInfo::SetInitialDefaultEntryPoint()

{

Assert(m\_scriptContext->CurrentThunk == ProfileEntryThunk || m\_scriptContext->CurrentThunk == DefaultEntryThunk);

Assert(originalEntryPoint == DefaultDeferredParsingThunk || originalEntryPoint == ProfileDeferredParsingThunk ||

originalEntryPoint == DefaultDeferredDeserializeThunk || originalEntryPoint == ProfileDeferredDeserializeThunk ||

originalEntryPoint == DefaultEntryThunk || originalEntryPoint == ProfileEntryThunk);

Assert(this->m\_defaultEntryPointInfo != nullptr);

// CONSIDER: we can optimize this to generate the dynamic interpreter thunk up front

// If we know that we are in the defer parsing thunk already

this->SetEntryPoint(this->GetDefaultEntryPointInfo(), m\_scriptContext->CurrentThunk);

this->originalEntryPoint = DefaultEntryThunk;

}

void FunctionBody::SetCheckCodeGenEntryPoint(FunctionEntryPointInfo\* entryPointInfo, JavascriptMethod entryPoint)

{

Assert(IsIntermediateCodeGenThunk(entryPoint));

Assert(

this->GetEntryPoint(entryPointInfo) == m\_scriptContext->CurrentThunk ||

(entryPointInfo == this->m\_defaultEntryPointInfo && this->IsInterpreterThunk()) ||

(

GetSimpleJitEntryPointInfo() &&

GetEntryPoint(entryPointInfo) == reinterpret\_cast<void \*>(GetSimpleJitEntryPointInfo()->GetNativeAddress())

));

this->SetEntryPoint(entryPointInfo, entryPoint);

}

#if DYNAMIC\_INTERPRETER\_THUNK

void FunctionBody::GenerateDynamicInterpreterThunk()

{

if (this->m\_dynamicInterpreterThunk == nullptr)

{

// NOTE: Etw rundown thread may be reading this->dynamicInterpreterThunk concurrently. We don't need to synchronize

// access as it is ok for etw rundown to get either null or updated new value.

if (m\_isAsmJsFunction)

{

this->originalEntryPoint = this->m\_scriptContext->GetNextDynamicAsmJsInterpreterThunk(&this->m\_dynamicInterpreterThunk);

}

else

{

this->originalEntryPoint = this->m\_scriptContext->GetNextDynamicInterpreterThunk(&this->m\_dynamicInterpreterThunk);

}

JS\_ETW(EtwTrace::LogMethodInterpreterThunkLoadEvent(this));

}

else

{

this->originalEntryPoint = (JavascriptMethod)InterpreterThunkEmitter::ConvertToEntryPoint(this->m\_dynamicInterpreterThunk);

}

}

JavascriptMethod FunctionBody::EnsureDynamicInterpreterThunk(FunctionEntryPointInfo\* entryPointInfo)

{

// This may be first call to the function, make sure we have dynamic profile info

//

// We need to ensure dynamic profile info even if we didn't generate a dynamic interpreter thunk

// This happens when we go through CheckCodeGen thunk, to DelayDynamicInterpreterThunk, to here

// but the background codegen thread updated the entry point with the native entry point.

this->EnsureDynamicProfileInfo();

Assert(HasValidEntryPoint());

if (InterpreterStackFrame::IsDelayDynamicInterpreterThunk(this->GetEntryPoint(entryPointInfo)))

{

// We are not doing code gen on this function, just change the entry point directly

Assert(InterpreterStackFrame::IsDelayDynamicInterpreterThunk(originalEntryPoint));

GenerateDynamicInterpreterThunk();

this->SetEntryPoint(entryPointInfo, originalEntryPoint);

}

else if (this->GetEntryPoint(entryPointInfo) == ProfileEntryThunk)

{

// We are not doing codegen on this function, just change the entry point directly

// Don't replace the profile entry thunk

Assert(InterpreterStackFrame::IsDelayDynamicInterpreterThunk(originalEntryPoint));

GenerateDynamicInterpreterThunk();

}

else if (InterpreterStackFrame::IsDelayDynamicInterpreterThunk(originalEntryPoint))

{

JsUtil::JobProcessor \* jobProcessor = this->GetScriptContext()->GetThreadContext()->GetJobProcessor();

if (jobProcessor->ProcessesInBackground())

{

JsUtil::BackgroundJobProcessor \* backgroundJobProcessor = static\_cast<JsUtil::BackgroundJobProcessor \*>(jobProcessor);

AutoCriticalSection autocs(backgroundJobProcessor->GetCriticalSection());

// Check again under lock

if (InterpreterStackFrame::IsDelayDynamicInterpreterThunk(originalEntryPoint))

{

// If the original entry point is DelayDynamicInterpreterThunk then there must be a version of this

// function being codegen'd.

Assert(IsIntermediateCodeGenThunk((JavascriptMethod)this->GetEntryPoint(this->GetDefaultEntryPointInfo())) || IsAsmJsCodeGenThunk((JavascriptMethod)this->GetEntryPoint(this->GetDefaultEntryPointInfo())));

GenerateDynamicInterpreterThunk();

}

}

else

{

// If the original entry point is DelayDynamicInterpreterThunk then there must be a version of this

// function being codegen'd.

Assert(IsIntermediateCodeGenThunk((JavascriptMethod)this->GetEntryPoint(this->GetDefaultEntryPointInfo())) || IsAsmJsCodeGenThunk((JavascriptMethod)this->GetEntryPoint(this->GetDefaultEntryPointInfo())));

GenerateDynamicInterpreterThunk();

}

}

return this->originalEntryPoint;

}

#endif

#if ENABLE\_NATIVE\_CODEGEN

void FunctionBody::SetNativeEntryPoint(FunctionEntryPointInfo\* entryPointInfo, JavascriptMethod originalEntryPoint, Var directEntryPoint)

{

if(entryPointInfo->nativeEntryPointProcessed)

{

return;

}

bool isAsmJs = this->GetIsAsmjsMode();

Assert(IsIntermediateCodeGenThunk((JavascriptMethod)entryPointInfo->address) || CONFIG\_FLAG(Prejit) || this->m\_isFromNativeCodeModule || isAsmJs);

entryPointInfo->EnsureIsReadyToCall();

// keep originalEntryPoint updated with the latest known good native entry point

if (entryPointInfo == this->GetDefaultEntryPointInfo())

{

this->originalEntryPoint = originalEntryPoint;

}

if (entryPointInfo->entryPointIndex == 0 && this->NeedEnsureDynamicProfileInfo())

{

entryPointInfo->address = DynamicProfileInfo::EnsureDynamicProfileInfoThunk;

}

else

{

entryPointInfo->address = directEntryPoint;

}

if (isAsmJs)

{

// release the old entrypointinfo if available

FunctionEntryPointInfo\* oldEntryPointInfo = entryPointInfo->GetOldFunctionEntryPointInfo();

if (oldEntryPointInfo)

{

this->GetScriptContext()->GetThreadContext()->QueueFreeOldEntryPointInfoIfInScript(oldEntryPointInfo);

oldEntryPointInfo = nullptr;

}

}

this->CaptureDynamicProfileState(entryPointInfo);

if(entryPointInfo->GetJitMode() == ExecutionMode::SimpleJit)

{

Assert(GetExecutionMode() == ExecutionMode::SimpleJit);

SetSimpleJitEntryPointInfo(entryPointInfo);

ResetSimpleJitCallCount();

}

else

{

Assert(entryPointInfo->GetJitMode() == ExecutionMode::FullJit);

Assert(isAsmJs || GetExecutionMode() == ExecutionMode::FullJit);

entryPointInfo->callsCount =

static\_cast<uint8>(

min(

static\_cast<uint>(static\_cast<uint8>(CONFIG\_FLAG(MinBailOutsBeforeRejit))) \*

(Js::FunctionEntryPointInfo::GetDecrCallCountPerBailout() - 1),

0xffu));

}

TraceExecutionMode();

if(entryPointInfo->GetJitMode() == ExecutionMode::SimpleJit)

{

Assert(GetExecutionMode() == ExecutionMode::SimpleJit);

SetSimpleJitEntryPointInfo(entryPointInfo);

ResetSimpleJitCallCount();

}

else

{

Assert(entryPointInfo->GetJitMode() == ExecutionMode::FullJit);

Assert(GetExecutionMode() == ExecutionMode::FullJit);

entryPointInfo->callsCount =

static\_cast<uint8>(

min(

static\_cast<uint>(static\_cast<uint8>(CONFIG\_FLAG(MinBailOutsBeforeRejit))) \*

(Js::FunctionEntryPointInfo::GetDecrCallCountPerBailout() - 1),

0xffu));

}

JS\_ETW(EtwTrace::LogMethodNativeLoadEvent(this, entryPointInfo));

#ifdef \_M\_ARM

// For ARM we need to make sure that pipeline is synchronized with memory/cache for newly jitted code.

\_InstructionSynchronizationBarrier();

#endif

entryPointInfo->nativeEntryPointProcessed = true;

}

void FunctionBody::DefaultSetNativeEntryPoint(FunctionEntryPointInfo\* entryPointInfo, FunctionBody \* functionBody, JavascriptMethod entryPoint)

{

Assert(functionBody->m\_scriptContext->CurrentThunk == DefaultEntryThunk);

functionBody->SetNativeEntryPoint(entryPointInfo, entryPoint, entryPoint);

}

void FunctionBody::ProfileSetNativeEntryPoint(FunctionEntryPointInfo\* entryPointInfo, FunctionBody \* functionBody, JavascriptMethod entryPoint)

{

Assert(functionBody->m\_scriptContext->CurrentThunk == ProfileEntryThunk);

functionBody->SetNativeEntryPoint(entryPointInfo, entryPoint, ProfileEntryThunk);

}

Js::JavascriptMethod FunctionBody::GetLoopBodyEntryPoint(Js::LoopHeader \* loopHeader, int entryPointIndex)

{

#if DBG

this->GetLoopNumber(loopHeader);

#endif

return (Js::JavascriptMethod)(loopHeader->GetEntryPointInfo(entryPointIndex)->address);

}

void FunctionBody::SetLoopBodyEntryPoint(Js::LoopHeader \* loopHeader, EntryPointInfo\* entryPointInfo, Js::JavascriptMethod entryPoint)

{

#if DBG\_DUMP

uint loopNum = this->GetLoopNumber(loopHeader);

if (PHASE\_TRACE1(Js::JITLoopBodyPhase))

{

DumpFunctionId(true);

Output::Print(L": %-20s LoopBody EntryPt Loop: %2d Address : %x\n", GetDisplayName(), loopNum, entryPoint);

Output::Flush();

}

#endif

Assert(((LoopEntryPointInfo\*) entryPointInfo)->loopHeader == loopHeader);

Assert(entryPointInfo->address == nullptr);

entryPointInfo->address = (void\*)entryPoint;

// reset the counter to 1 less than the threshold for TJLoopBody

if (loopHeader->GetCurrentEntryPointInfo()->GetIsAsmJSFunction())

{

loopHeader->interpretCount = entryPointInfo->GetFunctionBody()->GetLoopInterpretCount(loopHeader) - 1;

}

JS\_ETW(EtwTrace::LogLoopBodyLoadEvent(this, loopHeader, ((LoopEntryPointInfo\*) entryPointInfo)));

}

#endif

void FunctionBody::MarkScript(ByteBlock \*byteCodeBlock, ByteBlock\* auxBlock, ByteBlock\* auxContextBlock,

uint byteCodeCount, uint byteCodeInLoopCount, uint byteCodeWithoutLDACount)

{

CheckNotExecuting();

CheckEmpty();

#ifdef PERF\_COUNTERS

DWORD byteCodeSize = byteCodeBlock->GetLength()

+ (auxBlock? auxBlock->GetLength() : 0)

+ (auxContextBlock? auxContextBlock->GetLength() : 0);

PERF\_COUNTER\_ADD(Code, DynamicByteCodeSize, byteCodeSize);

#endif

m\_byteCodeCount = byteCodeCount;

m\_byteCodeInLoopCount = byteCodeInLoopCount;

m\_byteCodeWithoutLDACount = byteCodeWithoutLDACount;

InitializeExecutionModeAndLimits();

this->auxBlock = auxBlock;

this->auxContextBlock = auxContextBlock;

// Memory barrier needed here to make sure the background codegen thread's inliner

// gets all the assignment before it sees that the function has been parse

MemoryBarrier();

this->byteCodeBlock = byteCodeBlock;

PERF\_COUNTER\_ADD(Code, TotalByteCodeSize, byteCodeSize);

// If this is a defer parse function body, we would not have registered it

// on the function bodies list so we should register it now

if (!this->m\_isFuncRegistered)

{

this->m\_utf8SourceInfo->SetFunctionBody(this);

}

}

uint

FunctionBody::GetLoopNumber(LoopHeader const \* loopHeader) const

{

Assert(loopHeader >= this->loopHeaderArray);

uint loopNum = (uint)(loopHeader - this->loopHeaderArray);

Assert(loopNum < GetLoopCount());

return loopNum;

}

bool FunctionBody::InstallProbe(int offset)

{

if (offset < 0 || ((uint)offset + 1) >= byteCodeBlock->GetLength())

{

return false;

}

byte\* pbyteCodeBlockBuffer = this->byteCodeBlock->GetBuffer();

if(!GetProbeBackingBlock())

{

// The probe backing block is set on a different thread than the main thread

// The recycler doesn't like allocations from a different thread, so we allocate

// the backing byte code block in the arena

ArenaAllocator \*pArena = m\_scriptContext->AllocatorForDiagnostics();

AssertMem(pArena);

ByteBlock\* probeBackingBlock = ByteBlock::NewFromArena(pArena, pbyteCodeBlockBuffer, byteCodeBlock->GetLength());

SetProbeBackingBlock(probeBackingBlock);

}

// Make sure Break opcode only need one byte

Assert(OpCodeUtil::IsSmallEncodedOpcode(OpCode::Break));

#if ENABLE\_NATIVE\_CODEGEN

Assert(!OpCodeAttr::HasMultiSizeLayout(OpCode::Break));

#endif

\*(byte \*)(pbyteCodeBlockBuffer + offset) = (byte)OpCode::Break;

++m\_sourceInfo.m\_probeCount;

return true;

}

bool FunctionBody::UninstallProbe(int offset)

{

if (offset < 0 || ((uint)offset + 1) >= byteCodeBlock->GetLength())

{

return false;

}

byte\* pbyteCodeBlockBuffer = byteCodeBlock->GetBuffer();

Js::OpCode originalOpCode = ByteCodeReader::PeekByteOp(GetProbeBackingBlock()->GetBuffer() + offset);

\*(pbyteCodeBlockBuffer + offset) = (byte)originalOpCode;

--m\_sourceInfo.m\_probeCount;

AssertMsg(m\_sourceInfo.m\_probeCount >= 0, "Probe (Break Point) count became negative!");

return true;

}

bool FunctionBody::ProbeAtOffset(int offset, OpCode\* pOriginalOpcode)

{

if (!GetProbeBackingBlock())

{

return false;

}

if (offset < 0 || ((uint)offset + 1) >= this->byteCodeBlock->GetLength())

{

AssertMsg(false, "ProbeAtOffset called with out of bounds offset");

return false;

}

Js::OpCode runningOpCode = ByteCodeReader::PeekByteOp(this->byteCodeBlock->GetBuffer() + offset);

Js::OpCode originalOpcode = ByteCodeReader::PeekByteOp(GetProbeBackingBlock()->GetBuffer() + offset);

if ( runningOpCode != originalOpcode)

{

\*pOriginalOpcode = originalOpcode;

return true;

}

else

{

// e.g. inline break or a step hit and is checking for a bp

return false;

}

}

void FunctionBody::CloneByteCodeInto(ScriptContext \* scriptContext, FunctionBody \*newFunctionBody, uint sourceIndex)

{

((ParseableFunctionInfo\*) this)->CopyFunctionInfoInto(scriptContext, newFunctionBody, sourceIndex);

newFunctionBody->m\_constCount = this->m\_constCount;

newFunctionBody->m\_varCount = this->m\_varCount;

newFunctionBody->m\_outParamMaxDepth = this->m\_outParamMaxDepth;

newFunctionBody->m\_firstTmpReg = this->m\_firstTmpReg;

newFunctionBody->localClosureRegister = this->localClosureRegister;

newFunctionBody->localFrameDisplayRegister = this->localFrameDisplayRegister;

newFunctionBody->envRegister = this->envRegister;

newFunctionBody->thisRegisterForEventHandler = this->thisRegisterForEventHandler;

newFunctionBody->firstInnerScopeRegister = this->firstInnerScopeRegister;

newFunctionBody->funcExprScopeRegister = this->funcExprScopeRegister;

newFunctionBody->innerScopeCount = this->innerScopeCount;

newFunctionBody->hasCachedScopePropIds = this->hasCachedScopePropIds;

newFunctionBody->loopCount = this->loopCount;

newFunctionBody->profiledDivOrRemCount = this->profiledDivOrRemCount;

newFunctionBody->profiledSwitchCount = this->profiledSwitchCount;

newFunctionBody->profiledCallSiteCount = this->profiledCallSiteCount;

newFunctionBody->profiledArrayCallSiteCount = this->profiledArrayCallSiteCount;

newFunctionBody->profiledReturnTypeCount = this->profiledReturnTypeCount;

newFunctionBody->profiledLdElemCount = this->profiledLdElemCount;

newFunctionBody->profiledStElemCount = this->profiledStElemCount;

newFunctionBody->profiledSlotCount = this->profiledSlotCount;

newFunctionBody->flags = this->flags;

newFunctionBody->m\_isFuncRegistered = this->m\_isFuncRegistered;

newFunctionBody->m\_isFuncRegisteredToDiag = this->m\_isFuncRegisteredToDiag;

newFunctionBody->m\_hasBailoutInstrInJittedCode = this->m\_hasBailoutInstrInJittedCode;

newFunctionBody->m\_depth = this->m\_depth;

newFunctionBody->inlineDepth = 0;

newFunctionBody->recentlyBailedOutOfJittedLoopBody = false;

newFunctionBody->m\_pendingLoopHeaderRelease = this->m\_pendingLoopHeaderRelease;

newFunctionBody->m\_envDepth = this->m\_envDepth;

if (this->m\_constTable != nullptr)

{

this->CloneConstantTable(newFunctionBody);

}

newFunctionBody->cacheIdToPropertyIdMap = this->cacheIdToPropertyIdMap;

newFunctionBody->referencedPropertyIdMap = this->referencedPropertyIdMap;

newFunctionBody->propertyIdsForScopeSlotArray = this->propertyIdsForScopeSlotArray;

newFunctionBody->propertyIdOnRegSlotsContainer = this->propertyIdOnRegSlotsContainer;

newFunctionBody->scopeSlotArraySize = this->scopeSlotArraySize;

if (this->byteCodeBlock == nullptr)

{

newFunctionBody->SetDeferredParsingEntryPoint();

}

else

{

newFunctionBody->byteCodeBlock = this->byteCodeBlock->Clone(this->m\_scriptContext->GetRecycler());

newFunctionBody->isByteCodeDebugMode = this->isByteCodeDebugMode;

newFunctionBody->m\_byteCodeCount = this->m\_byteCodeCount;

newFunctionBody->m\_byteCodeWithoutLDACount = this->m\_byteCodeWithoutLDACount;

newFunctionBody->m\_byteCodeInLoopCount = this->m\_byteCodeInLoopCount;

#ifdef PERF\_COUNTERS

DWORD byteCodeSize = this->byteCodeBlock->GetLength();

#endif

if (this->auxBlock)

{

newFunctionBody->auxBlock = this->auxBlock->Clone(this->m\_scriptContext->GetRecycler());

#ifdef PERF\_COUNTERS

byteCodeSize += this->auxBlock->GetLength();

#endif

}

if (this->auxContextBlock)

{

newFunctionBody->auxContextBlock = this->auxContextBlock->Clone(scriptContext->GetRecycler(), scriptContext);

#ifdef PERF\_COUNTERS

byteCodeSize += this->auxContextBlock->GetLength();

#endif

}

if (this->GetProbeBackingBlock())

{

newFunctionBody->SetProbeBackingBlock(this->GetProbeBackingBlock()->Clone(scriptContext->GetRecycler()));

newFunctionBody->m\_sourceInfo.m\_probeCount = m\_sourceInfo.m\_probeCount;

}

#ifdef PERF\_COUNTERS

PERF\_COUNTER\_ADD(Code, DynamicByteCodeSize, byteCodeSize);

PERF\_COUNTER\_ADD(Code, TotalByteCodeSize, byteCodeSize);

#endif

newFunctionBody->SetFrameDisplayRegister(this->GetFrameDisplayRegister());

newFunctionBody->SetObjectRegister(this->GetObjectRegister());

StatementMapList \* pStatementMaps = this->GetStatementMaps();

if (pStatementMaps != nullptr)

{

Recycler\* recycler = newFunctionBody->GetScriptContext()->GetRecycler();

StatementMapList \* newStatementMaps = RecyclerNew(recycler, StatementMapList, recycler);

newFunctionBody->pStatementMaps = newStatementMaps;

pStatementMaps->Map([recycler, newStatementMaps](int index, FunctionBody::StatementMap\* oldStatementMap)

{

FunctionBody::StatementMap\* newStatementMap = StatementMap::New(recycler);

\*newStatementMap = \*oldStatementMap;

newStatementMaps->Add(newStatementMap);

});

}

if (this->m\_sourceInfo.pSpanSequence != nullptr)

{

// Span sequence is heap allocated

newFunctionBody->m\_sourceInfo.pSpanSequence = this->m\_sourceInfo.pSpanSequence->Clone();

}

Assert(newFunctionBody->GetDirectEntryPoint(newFunctionBody->GetDefaultEntryPointInfo()) == scriptContext->CurrentThunk);

Assert(newFunctionBody->IsInterpreterThunk());

}

// Create a new inline cache

newFunctionBody->inlineCacheCount = this->inlineCacheCount;

newFunctionBody->rootObjectLoadInlineCacheStart = this->rootObjectLoadInlineCacheStart;

newFunctionBody->rootObjectStoreInlineCacheStart = this->rootObjectStoreInlineCacheStart;

newFunctionBody->isInstInlineCacheCount = this->isInstInlineCacheCount;

newFunctionBody->referencedPropertyIdCount = this->referencedPropertyIdCount;

newFunctionBody->AllocateInlineCache();

newFunctionBody->objLiteralCount = this->objLiteralCount;

newFunctionBody->AllocateObjectLiteralTypeArray();

newFunctionBody->simpleJitEntryPointInfo = nullptr;

newFunctionBody->loopInterpreterLimit = loopInterpreterLimit;

newFunctionBody->ReinitializeExecutionModeAndLimits();

// Clone literal regexes

newFunctionBody->literalRegexCount = this->literalRegexCount;

newFunctionBody->AllocateLiteralRegexArray();

for(uint i = 0; i < this->literalRegexCount; ++i)

{

const auto literalRegex = this->literalRegexes[i];

if(!literalRegex)

{

Assert(!newFunctionBody->GetLiteralRegex(i));

continue;

}

const auto source = literalRegex->GetSource();

newFunctionBody->SetLiteralRegex(

i,

RegexHelper::CompileDynamic(

scriptContext,

source.GetBuffer(),

source.GetLength(),

literalRegex->GetFlags(),

true));

}

if (this->DoJITLoopBody())

{

newFunctionBody->AllocateLoopHeaders();

for (uint i = 0; i < this->GetLoopCount(); i++)

{

newFunctionBody->GetLoopHeader(i)->startOffset = GetLoopHeader(i)->startOffset;

newFunctionBody->GetLoopHeader(i)->endOffset = GetLoopHeader(i)->endOffset;

}

}

newFunctionBody->serializationIndex = this->serializationIndex;

newFunctionBody->m\_isFromNativeCodeModule = this->m\_isFromNativeCodeModule;

}

FunctionBody \*

FunctionBody::Clone(ScriptContext \* scriptContext, uint sourceIndex)

{

#if ENABLE\_NATIVE\_CODEGEN && defined(ENABLE\_PREJIT)

bool isNested = sourceIndex != Constants::InvalidSourceIndex;

#endif

Utf8SourceInfo\* sourceInfo = nullptr;

if(sourceIndex == Constants::InvalidSourceIndex)

{

// If we're copying a source info across script contexts, we need

// to create a copy of the Utf8SourceInfo (just the structure, not the source code itself)

// because a Utf8SourceInfo must reference only function bodies created within that script

// context

Utf8SourceInfo\* oldSourceInfo = GetUtf8SourceInfo();

SRCINFO\* srcInfo = GetHostSrcInfo()->Clone(scriptContext);

sourceInfo = scriptContext->CloneSourceCrossContext(oldSourceInfo, srcInfo);

sourceIndex = scriptContext->SaveSourceNoCopy(sourceInfo, oldSourceInfo->GetCchLength(), oldSourceInfo->GetIsCesu8());

}

else

{

sourceInfo = scriptContext->GetSource(sourceIndex);

}

FunctionBody \* newFunctionBody = FunctionBody::NewFromRecycler(scriptContext, this->GetDisplayName(), this->GetDisplayNameLength(),

this->GetShortDisplayNameOffset(), this->m\_nestedCount, sourceInfo, this->m\_functionNumber, this->m\_uScriptId,

this->GetLocalFunctionId(), this->m\_boundPropertyRecords,

this->GetAttributes()

#ifdef PERF\_COUNTERS

, false

#endif

);

if (this->m\_scopeInfo != nullptr)

{

newFunctionBody->SetScopeInfo(m\_scopeInfo->CloneFor(newFunctionBody));

}

newFunctionBody->CloneSourceInfo(scriptContext, (\*this), this->m\_scriptContext, sourceIndex);

CloneByteCodeInto(scriptContext, newFunctionBody, sourceIndex);

#if DBG

newFunctionBody->m\_iProfileSession = this->m\_iProfileSession;

newFunctionBody->deferredParseNextFunctionId = this->deferredParseNextFunctionId;

#endif

#if ENABLE\_PROFILE\_INFO

if (this->HasDynamicProfileInfo())

{

newFunctionBody->EnsureDynamicProfileInfo();

}

#endif

newFunctionBody->byteCodeCache = this->byteCodeCache;

#if ENABLE\_NATIVE\_CODEGEN

if (newFunctionBody->GetByteCode() && (IsIntermediateCodeGenThunk(this->GetOriginalEntryPoint())

|| IsNativeOriginalEntryPoint()))

{

#ifdef ENABLE\_PREJIT

if (Js::Configuration::Global.flags.Prejit)

{

if (!isNested)

{

GenerateAllFunctions(scriptContext->GetNativeCodeGenerator(), newFunctionBody);

}

}

else

#endif

{

GenerateFunction(scriptContext->GetNativeCodeGenerator(), newFunctionBody);

}

}

#endif

return newFunctionBody;

}

void FunctionBody::SetStackNestedFuncParent(FunctionBody \* parentFunctionBody)

{

Assert(this->stackNestedFuncParent == nullptr);

Assert(CanDoStackNestedFunc());

Assert(parentFunctionBody->DoStackNestedFunc());

this->stackNestedFuncParent = this->GetScriptContext()->GetRecycler()->CreateWeakReferenceHandle(parentFunctionBody);

}

FunctionBody \* FunctionBody::GetStackNestedFuncParent()

{

Assert(this->stackNestedFuncParent);

return this->stackNestedFuncParent->Get();

}

FunctionBody \* FunctionBody::GetAndClearStackNestedFuncParent()

{

if (this->stackNestedFuncParent)

{

FunctionBody \* parentFunctionBody = GetStackNestedFuncParent();

ClearStackNestedFuncParent();

return parentFunctionBody;

}

return nullptr;

}

void FunctionBody::ClearStackNestedFuncParent()

{

this->stackNestedFuncParent = nullptr;

}

ParseableFunctionInfo\* ParseableFunctionInfo::CopyFunctionInfoInto(ScriptContext \*scriptContext, Js::ParseableFunctionInfo\* newFunctionInfo, uint sourceIndex)

{

newFunctionInfo->m\_inParamCount = this->m\_inParamCount;

newFunctionInfo->m\_grfscr = this->m\_grfscr;

newFunctionInfo->m\_isDeclaration = this->m\_isDeclaration;

newFunctionInfo->m\_hasImplicitArgIns = this->m\_hasImplicitArgIns;

newFunctionInfo->m\_isAccessor = this->m\_isAccessor;

newFunctionInfo->m\_isGlobalFunc = this->m\_isGlobalFunc;

newFunctionInfo->m\_dontInline = this->m\_dontInline;

newFunctionInfo->m\_isTopLevel = this->m\_isTopLevel;

newFunctionInfo->m\_isPublicLibraryCode = this->m\_isPublicLibraryCode;

newFunctionInfo->scopeSlotArraySize = this->scopeSlotArraySize;

for (uint index = 0; index < this->m\_nestedCount; index++)

{

FunctionProxy\* proxy = this->GetNestedFunc(index);

if (proxy)

{

// Deserialize the proxy here if we have to

ParseableFunctionInfo\* body = proxy->EnsureDeserialized();

FunctionProxy\* newBody;

if (body->IsDeferredParseFunction())

{

newBody = body->Clone(scriptContext, sourceIndex);

}

else

{

newBody = body->GetFunctionBody()->Clone(scriptContext, sourceIndex);

}

// 0u is an empty value for the bit-mask 'flags', when initially parsing this is used to track defer-parse functions.

newFunctionInfo->SetNestedFunc(newBody, index, 0u);

}

else

{

// 0u is an empty value for the bit-mask 'flags', when initially parsing this is used to track defer-parse functions.

newFunctionInfo->SetNestedFunc(nullptr, index, 0u);

}

}

return newFunctionInfo;

}

ParseableFunctionInfo\* ParseableFunctionInfo::Clone(ScriptContext \*scriptContext, uint sourceIndex)

{

Utf8SourceInfo\* sourceInfo = nullptr;

if(sourceIndex == Constants::InvalidSourceIndex)

{

// If we're copying a source info across script contexts, we need

// to create a copy of the Utf8SourceInfo (just the structure, not the source code itself)

// because a Utf8SourceInfo must reference only function bodies created within that script

// context

Utf8SourceInfo\* oldSourceInfo = GetUtf8SourceInfo();

SRCINFO\* srcInfo = GetHostSrcInfo()->Clone(scriptContext);

sourceInfo = scriptContext->CloneSourceCrossContext(oldSourceInfo, srcInfo);

sourceIndex = scriptContext->SaveSourceNoCopy(sourceInfo, oldSourceInfo->GetCchLength(), oldSourceInfo->GetIsCesu8());

}

else

{

sourceInfo = scriptContext->GetSource(sourceIndex);

}

ParseableFunctionInfo\* newFunctionInfo = ParseableFunctionInfo::New(scriptContext, this->m\_nestedCount, this->GetLocalFunctionId(), sourceInfo, this->GetDisplayName(), this->GetDisplayNameLength(),

this->GetShortDisplayNameOffset(), this->m\_boundPropertyRecords, this->GetAttributes());

if (this->m\_scopeInfo != nullptr)

{

newFunctionInfo->SetScopeInfo(m\_scopeInfo->CloneFor(newFunctionInfo));

}

newFunctionInfo->CloneSourceInfo(scriptContext, (\*this), this->m\_scriptContext, sourceIndex);

CopyFunctionInfoInto(scriptContext, newFunctionInfo, sourceIndex);

#if DBG

newFunctionInfo->deferredParseNextFunctionId = this->deferredParseNextFunctionId;

#endif

return newFunctionInfo;

}

void FunctionBody::CreateCacheIdToPropertyIdMap(uint rootObjectLoadInlineCacheStart, uint rootObjectLoadMethodInlineCacheStart,

uint rootObjectStoreInlineCacheStart,

uint totalFieldAccessInlineCacheCount, uint isInstInlineCacheCount)

{

Assert(this->rootObjectLoadInlineCacheStart == 0);

Assert(this->rootObjectLoadMethodInlineCacheStart == 0);

Assert(this->rootObjectStoreInlineCacheStart == 0);

Assert(this->inlineCacheCount == 0);

Assert(this->isInstInlineCacheCount == 0);

this->rootObjectLoadInlineCacheStart = rootObjectLoadInlineCacheStart;

this->rootObjectLoadMethodInlineCacheStart = rootObjectLoadMethodInlineCacheStart;

this->rootObjectStoreInlineCacheStart = rootObjectStoreInlineCacheStart;

this->inlineCacheCount = totalFieldAccessInlineCacheCount;

this->isInstInlineCacheCount = isInstInlineCacheCount;

this->CreateCacheIdToPropertyIdMap();

}

void FunctionBody::CreateCacheIdToPropertyIdMap()

{

Assert(this->cacheIdToPropertyIdMap == nullptr);

Assert(this->inlineCaches == nullptr);

uint count = this->GetInlineCacheCount() ;

if (count!= 0)

{

this->cacheIdToPropertyIdMap =

RecyclerNewArrayLeaf(this->m\_scriptContext->GetRecycler(), PropertyId, count);

#if DBG

for (uint i = 0; i < count; i++)

{

this->cacheIdToPropertyIdMap[i] = Js::Constants::NoProperty;

}

#endif

}

}

#if DBG

void FunctionBody::VerifyCacheIdToPropertyIdMap()

{

uint count = this->GetInlineCacheCount();

for (uint i = 0; i < count; i++)

{

Assert(this->cacheIdToPropertyIdMap[i] != Js::Constants::NoProperty);

}

}

#endif

void FunctionBody::SetPropertyIdForCacheId(uint cacheId, PropertyId propertyId)

{

Assert(this->cacheIdToPropertyIdMap != nullptr);

Assert(cacheId < this->GetInlineCacheCount());

Assert(this->cacheIdToPropertyIdMap[cacheId] == Js::Constants::NoProperty);

this->cacheIdToPropertyIdMap[cacheId] = propertyId;

}

void FunctionBody::CreateReferencedPropertyIdMap(uint referencedPropertyIdCount)

{

this->referencedPropertyIdCount = referencedPropertyIdCount;

this->CreateReferencedPropertyIdMap();

}

void FunctionBody::CreateReferencedPropertyIdMap()

{

Assert(this->referencedPropertyIdMap == nullptr);

uint count = this->GetReferencedPropertyIdCount();

if (count!= 0)

{

this->referencedPropertyIdMap =

RecyclerNewArrayLeaf(this->m\_scriptContext->GetRecycler(), PropertyId, count);

#if DBG

for (uint i = 0; i < count; i++)

{

this->referencedPropertyIdMap[i] = Js::Constants::NoProperty;

}

#endif

}

}

#if DBG

void FunctionBody::VerifyReferencedPropertyIdMap()

{

uint count = this->GetReferencedPropertyIdCount();

for (uint i = 0; i < count; i++)

{

Assert(this->referencedPropertyIdMap[i] != Js::Constants::NoProperty);

}

}

#endif

PropertyId FunctionBody::GetReferencedPropertyId(uint index)

{

if (index < (uint)TotalNumberOfBuiltInProperties)

{

return index;

}

uint mapIndex = index - TotalNumberOfBuiltInProperties;

return GetReferencedPropertyIdWithMapIndex(mapIndex);

}

PropertyId FunctionBody::GetReferencedPropertyIdWithMapIndex(uint mapIndex)

{

Assert(this->referencedPropertyIdMap);

Assert(mapIndex < this->GetReferencedPropertyIdCount());

return this->referencedPropertyIdMap[mapIndex];

}

void FunctionBody::SetReferencedPropertyIdWithMapIndex(uint mapIndex, PropertyId propertyId)

{

Assert(propertyId >= TotalNumberOfBuiltInProperties);

Assert(mapIndex < this->GetReferencedPropertyIdCount());

Assert(this->referencedPropertyIdMap != nullptr);

Assert(this->referencedPropertyIdMap[mapIndex] == Js::Constants::NoProperty);

this->referencedPropertyIdMap[mapIndex] = propertyId;

}

void FunctionBody::CreateConstantTable()

{

Assert(this->m\_constTable == nullptr);

Assert(m\_constCount > FirstRegSlot);

this->m\_constTable = RecyclerNewArrayZ(this->m\_scriptContext->GetRecycler(), Var, m\_constCount);

// Initialize with the root object, which will always be recorded here.

Js::RootObjectBase \* rootObject = this->LoadRootObject();

if (rootObject)

{

this->RecordConstant(RootObjectRegSlot, rootObject);

}

else

{

Assert(false);

this->RecordConstant(RootObjectRegSlot, this->m\_scriptContext->GetLibrary()->GetUndefined());

}

}

void FunctionBody::RecordConstant(RegSlot location, Var var)

{

Assert(location < m\_constCount);

Assert(this->m\_constTable);

Assert(var != nullptr);

Assert(this->m\_constTable[location - FunctionBody::FirstRegSlot] == nullptr);

this->m\_constTable[location - FunctionBody::FirstRegSlot] = var;

}

void FunctionBody::RecordNullObject(RegSlot location)

{

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

Var nullObject = JavascriptOperators::OP\_LdNull(scriptContext);

this->RecordConstant(location, nullObject);

}

void FunctionBody::RecordUndefinedObject(RegSlot location)

{

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

Var undefObject = JavascriptOperators::OP\_LdUndef(scriptContext);

this->RecordConstant(location, undefObject);

}

void FunctionBody::RecordTrueObject(RegSlot location)

{

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

Var trueObject = JavascriptBoolean::OP\_LdTrue(scriptContext);

this->RecordConstant(location, trueObject);

}

void FunctionBody::RecordFalseObject(RegSlot location)

{

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

Var falseObject = JavascriptBoolean::OP\_LdFalse(scriptContext);

this->RecordConstant(location, falseObject);

}

void FunctionBody::RecordIntConstant(RegSlot location, unsigned int val)

{

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

Var intConst = JavascriptNumber::ToVar((int32)val, scriptContext);

this->RecordConstant(location, intConst);

}

void FunctionBody::RecordStrConstant(RegSlot location, LPCOLESTR psz, ulong cch)

{

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

PropertyRecord const \* propertyRecord;

scriptContext->FindPropertyRecord(psz, cch, &propertyRecord);

Var str;

if (propertyRecord == nullptr)

{

str = JavascriptString::NewCopyBuffer(psz, cch, scriptContext);

}

else

{

// If a particular string constant already has a propertyId, just create a property string for it

// as it might be likely that it is used for a property lookup

str = scriptContext->GetPropertyString(propertyRecord->GetPropertyId());

}

this->RecordConstant(location, str);

}

void FunctionBody::RecordFloatConstant(RegSlot location, double d)

{

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

Var floatConst = JavascriptNumber::ToVarIntCheck(d, scriptContext);

this->RecordConstant(location, floatConst);

}

void FunctionBody::RecordNullDisplayConstant(RegSlot location)

{

this->RecordConstant(location, (Js::Var)&Js::NullFrameDisplay);

}

void FunctionBody::RecordStrictNullDisplayConstant(RegSlot location)

{

this->RecordConstant(location, (Js::Var)&Js::StrictNullFrameDisplay);

}

void FunctionBody::InitConstantSlots(Var \*dstSlots)

{

// Initialize the given slots from the constant table.

Assert(m\_constCount > FunctionBody::FirstRegSlot);

js\_memcpy\_s(dstSlots, (m\_constCount - FunctionBody::FirstRegSlot) \* sizeof(Var), this->m\_constTable, (m\_constCount - FunctionBody::FirstRegSlot) \* sizeof(Var));

}

Var FunctionBody::GetConstantVar(RegSlot location)

{

Assert(this->m\_constTable);

Assert(location < m\_constCount);

Assert(location != 0);

return this->m\_constTable[location - FunctionBody::FirstRegSlot];

}

void FunctionBody::CloneConstantTable(FunctionBody \*newFunc)

{

// Creating the constant table initializes the root object.

newFunc->CreateConstantTable();

// Start walking the slots after the root object.

for (RegSlot reg = FunctionBody::RootObjectRegSlot + 1; reg < m\_constCount; reg++)

{

Var oldVar = this->GetConstantVar(reg);

Assert(oldVar != nullptr);

if (TaggedInt::Is(oldVar))

{

newFunc->RecordIntConstant(reg, TaggedInt::ToInt32(oldVar));

}

else if (oldVar == &Js::NullFrameDisplay)

{

newFunc->RecordNullDisplayConstant(reg);

}

else if (oldVar == &Js::StrictNullFrameDisplay)

{

newFunc->RecordStrictNullDisplayConstant(reg);

}

else

{

switch (JavascriptOperators::GetTypeId(oldVar))

{

case Js::TypeIds\_Undefined:

newFunc->RecordUndefinedObject(reg);

break;

case Js::TypeIds\_Null:

newFunc->RecordNullObject(reg);

break;

case Js::TypeIds\_Number:

newFunc->RecordFloatConstant(reg, JavascriptNumber::GetValue(oldVar));

break;

case Js::TypeIds\_String:

{

JavascriptString \*str = JavascriptString::FromVar(oldVar);

newFunc->RecordStrConstant(reg, str->GetSz(), str->GetLength());

break;

}

case Js::TypeIds\_ES5Array:

newFunc->RecordConstant(reg, oldVar);

break;

case Js::TypeIds\_Boolean:

if (Js::JavascriptBoolean::FromVar(oldVar)->GetValue())

{

newFunc->RecordTrueObject(reg);

}

else

{

newFunc->RecordFalseObject(reg);

}

break;

default:

AssertMsg(UNREACHED, "Unexpected object type in CloneConstantTable");

break;

}

}

}

}

#if DBG\_DUMP

void FunctionBody::Dump()

{

Js::ByteCodeDumper::Dump(this);

}

void FunctionBody::DumpScopes()

{

if(this->GetScopeObjectChain())

{

wchar\_t debugStringBuffer[MAX\_FUNCTION\_BODY\_DEBUG\_STRING\_SIZE];

Output::Print(L"%s (%s) :\n", this->GetDisplayName(), this->GetDebugNumberSet(debugStringBuffer));

this->GetScopeObjectChain()->pScopeChain->Map( [=] (uint index, DebuggerScope\* scope )

{

scope->Dump();

});

}

}

#if ENABLE\_NATIVE\_CODEGEN

void EntryPointInfo::DumpNativeOffsetMaps()

{

// Native Offsets

if (this->nativeOffsetMaps.Count() > 0)

{

Output::Print(L"Native Map: baseAddr: 0x%0Ix, size: 0x%0Ix\nstatementId, offset range, address range\n",

this->GetNativeAddress(),

this->GetCodeSize());

int count = this->nativeOffsetMaps.Count();

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

{

const NativeOffsetMap\* map = &this->nativeOffsetMaps.Item(i);

Output::Print(L"S%4d, (%5d, %5d) (0x%012Ix, 0x%012Ix)\n", map->statementIndex,

map->nativeOffsetSpan.begin,

map->nativeOffsetSpan.end,

map->nativeOffsetSpan.begin + this->GetNativeAddress(),

map->nativeOffsetSpan.end + this->GetNativeAddress());

}

}

}

#endif

void FunctionBody::DumpStatementMaps()

{

// Source Map to ByteCode

StatementMapList \* pStatementMaps = this->GetStatementMaps();

if (pStatementMaps)

{

Output::Print(L"Statement Map:\nstatementId, SourceSpan, ByteCodeSpan\n");

int count = pStatementMaps->Count();

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

{

StatementMap\* map = pStatementMaps->Item(i);

Output::Print(L"S%4d, (C%5d, C%5d) (B%5d, B%5d) Inner=%d\n", i,

map->sourceSpan.begin,

map->sourceSpan.end,

map->byteCodeSpan.begin,

map->byteCodeSpan.end,

map->isSubexpression);

}

}

}

#if ENABLE\_NATIVE\_CODEGEN

void EntryPointInfo::DumpNativeThrowSpanSequence()

{

// Native Throw Map

if (this->nativeThrowSpanSequence)

{

Output::Print(L"Native Throw Map: baseAddr: 0x%0Ix, size: 0x%Ix\nstatementId, offset range, address range\n",

this->GetNativeAddress(),

this->GetCodeSize());

int count = this->nativeThrowSpanSequence->Count();

SmallSpanSequenceIter iter;

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

{

StatementData data;

if (this->nativeThrowSpanSequence->Item(i, iter, data))

{

Output::Print(L"S%4d, (%5d -----) (0x%012Ix --------)\n", data.sourceBegin, // statementIndex

data.bytecodeBegin, // nativeOffset

data.bytecodeBegin + this->GetNativeAddress());

}

}

}

}

#endif

void FunctionBody::PrintStatementSourceLine(uint statementIndex)

{

const uint startOffset = GetStatementStartOffset(statementIndex);

// startOffset should only be 0 if statementIndex is 0, otherwise it is EOF and we should skip printing anything

if (startOffset != 0 || statementIndex == 0)

{

PrintStatementSourceLineFromStartOffset(startOffset);

}

}

void FunctionBody::PrintStatementSourceLineFromStartOffset(uint cchStartOffset)

{

ULONG line;

LONG col;

LPCUTF8 source = GetStartOfDocument(L"FunctionBody::PrintStatementSourceLineFromStartOffset");

Utf8SourceInfo\* sourceInfo = this->GetUtf8SourceInfo();

Assert(sourceInfo != nullptr);

LPCUTF8 sourceInfoSrc = sourceInfo->GetSource(L"FunctionBody::PrintStatementSourceLineFromStartOffset");

if(!sourceInfoSrc)

{

Assert(sourceInfo->GetIsLibraryCode());

return;

}

if( source != sourceInfoSrc )

{

Output::Print(L"\nDETECTED MISMATCH:\n");

Output::Print(L"GetUtf8SourceInfo()->GetSource(): 0x%08X: %.\*s ...\n", sourceInfo, 16, sourceInfo);

Output::Print(L"GetStartOfDocument(): 0x%08X: %.\*s ...\n", source, 16, source);

AssertMsg(false, "Non-matching start of document");

}

GetLineCharOffsetFromStartChar(cchStartOffset, &line, &col, false /\*canAllocateLineCache\*/);

WORD color = 0;

if (Js::Configuration::Global.flags.DumpLineNoInColor)

{

color = Output::SetConsoleForeground(12);

}

Output::Print(L"\n\n Line %3d: ", line + 1);

// Need to match up cchStartOffset to appropriate cbStartOffset given function's cbStartOffset and cchStartOffset

size\_t i = utf8::CharacterIndexToByteIndex(source, sourceInfo->GetCbLength(), cchStartOffset, this->m\_cbStartOffset, this->m\_cchStartOffset);

size\_t lastOffset = StartOffset() + LengthInBytes();

for (;i < lastOffset && source[i] != '\n' && source[i] != '\r'; i++)

{

Output::Print(L"%C", source[i]);

}

Output::Print(L"\n");

Output::Print(L" Col %4d:%s^\n", col + 1, ((col+1)<10000) ? L" " : L"");

if (color != 0)

{

Output::SetConsoleForeground(color);

}

}

#endif // DBG\_DUMP

/\*\*

\* Get the source code offset for the given <statementIndex>.

\*/

uint FunctionBody::GetStatementStartOffset(const uint statementIndex)

{

uint startOffset = 0;

if (statementIndex != Js::Constants::NoStatementIndex)

{

const Js::FunctionBody::SourceInfo \* sourceInfo = &(this->m\_sourceInfo);

if (sourceInfo->pSpanSequence != nullptr)

{

Js::SmallSpanSequenceIter iter;

sourceInfo->pSpanSequence->Reset(iter);

Js::StatementData data;

sourceInfo->pSpanSequence->Item(statementIndex, iter, data);

startOffset = data.sourceBegin;

}

else

{

int index = statementIndex;

Js::FunctionBody::StatementMap \* statementMap = GetNextNonSubexpressionStatementMap(GetStatementMaps(), index);

startOffset = statementMap->sourceSpan.Begin();

}

}

return startOffset;

}

#ifdef IR\_VIEWER

/\* BEGIN potentially reusable code \*/

/\*

This code could be reused for locating source code in a debugger or to

retrieve the text of source statements.

Currently this code is used to retrieve the text of a source code statement

in the IR\_VIEWER feature.

\*/

/\*\*

\* Given a statement's starting offset in the source code, calculate the beginning and end of a statement,

\* as well as the line and column number where the statement appears.

\*

\* @param startOffset (input) The offset into the source code where this statement begins.

\* @param sourceBegin (output) The beginning of the statement in the source string.

\* @param sourceEnd (output) The end of the statement in the source string.

\* @param line (output) The line number where the statement appeared in the source.

\* @param col (output) The column number where the statement appeared in the source.

\*/

void FunctionBody::GetSourceLineFromStartOffset(const uint startOffset, LPCUTF8 \*sourceBegin, LPCUTF8 \*sourceEnd,

ULONG \* line, LONG \* col)

{

//

// get source info

//

LPCUTF8 source = GetStartOfDocument(L"IR Viewer FunctionBody::GetSourceLineFromStartOffset");

Utf8SourceInfo\* sourceInfo = this->GetUtf8SourceInfo();

Assert(sourceInfo != nullptr);

LPCUTF8 sourceInfoSrc = sourceInfo->GetSource(L"IR Viewer FunctionBody::GetSourceLineFromStartOffset");

if (!sourceInfoSrc)

{

Assert(sourceInfo->GetIsLibraryCode());

return;

}

if (source != sourceInfoSrc)

{

Output::Print(L"\nDETECTED MISMATCH:\n");

Output::Print(L"GetUtf8SourceInfo()->GetSource(): 0x%08X: %.\*s ...\n", sourceInfo, 16, sourceInfo);

Output::Print(L"GetStartOfDocument(): 0x%08X: %.\*s ...\n", source, 16, source);

AssertMsg(false, "Non-matching start of document");

}

//

// calculate source line info

//

size\_t cbStartOffset = utf8::CharacterIndexToByteIndex(source, sourceInfo->GetCbLength(), (const charcount\_t)startOffset, (size\_t)this->m\_cbStartOffset, (charcount\_t)this->m\_cchStartOffset);

GetLineCharOffsetFromStartChar(startOffset, line, col);

size\_t lastOffset = StartOffset() + LengthInBytes();

size\_t i = 0;

for (i = cbStartOffset; i < lastOffset && source[i] != '\n' && source[i] != '\r'; i++)

{

// do nothing; scan until end of statement

}

size\_t cbEndOffset = i;

//

// return

//

\*sourceBegin = &source[cbStartOffset];

\*sourceEnd = &source[cbEndOffset];

}

/\*\*

\* Given a statement index and output parameters, calculate the beginning and end of a statement,

\* as well as the line and column number where the statement appears.

\*

\* @param statementIndex (input) The statement's index (as used by the StatementBoundary pragma).

\* @param sourceBegin (output) The beginning of the statement in the source string.

\* @param sourceEnd (output) The end of the statement in the source string.

\* @param line (output) The line number where the statement appeared in the source.

\* @param col (output) The column number where the statement appeared in the source.

\*/

void FunctionBody::GetStatementSourceInfo(const uint statementIndex, LPCUTF8 \*sourceBegin, LPCUTF8 \*sourceEnd,

ULONG \* line, LONG \* col)

{

const size\_t startOffset = GetStatementStartOffset(statementIndex);

// startOffset should only be 0 if statementIndex is 0, otherwise it is EOF and we should return empty string

if (startOffset != 0 || statementIndex == 0)

{

GetSourceLineFromStartOffset(startOffset, sourceBegin, sourceEnd, line, col);

}

else

{

\*sourceBegin = nullptr;

\*sourceEnd = nullptr;

\*line = 0;

\*col = 0;

return;

}

}

/\* END potentially reusable code \*/

#endif /\* IR\_VIEWER \*/

#ifdef IR\_VIEWER

Js::DynamicObject \* FunctionBody::GetIRDumpBaseObject()

{

if (!this->m\_irDumpBaseObject)

{

this->m\_irDumpBaseObject = this->m\_scriptContext->GetLibrary()->CreateObject();

}

return this->m\_irDumpBaseObject;

}

#endif /\* IR\_VIEWER \*/

#ifdef VTUNE\_PROFILING

#ifdef CDECL

#define ORIGINAL\_CDECL CDECL

#undef CDECL

#endif

// Not enabled in ChakraCore

#include "jitProfiling.h"

#ifdef ORIGINAL\_CDECL

#undef CDECL

#endif

#define CDECL ORIGINAL\_CDECL

int EntryPointInfo::GetNativeOffsetMapCount() const

{

return this->nativeOffsetMaps.Count();

}

uint EntryPointInfo::PopulateLineInfo(void\* pInfo, FunctionBody\* body)

{

LineNumberInfo\* pLineInfo = (LineNumberInfo\*)pInfo;

ULONG functionLineNumber = body->GetLineNumber();

pLineInfo[0].Offset = 0;

pLineInfo[0].LineNumber = functionLineNumber;

int lineNumber = 0;

int j = 1; // start with 1 since offset 0 has already been populated with function line number

int count = this->nativeOffsetMaps.Count();

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

{

const NativeOffsetMap\* map = &this->nativeOffsetMaps.Item(i);

uint32 statementIndex = map->statementIndex;

if (statementIndex == 0)

{

// statementIndex is 0, first line in the function, populate with function line number

pLineInfo[j].Offset = map->nativeOffsetSpan.begin;

pLineInfo[j].LineNumber = functionLineNumber;

j++;

}

lineNumber = body->GetSourceLineNumber(statementIndex);

if (lineNumber != 0)

{

pLineInfo[j].Offset = map->nativeOffsetSpan.end;

pLineInfo[j].LineNumber = lineNumber;

j++;

}

}

return j;

}

ULONG FunctionBody::GetSourceLineNumber(uint statementIndex)

{

ULONG line = 0;

if (statementIndex != Js::Constants::NoStatementIndex)

{

uint startOffset = GetStartOffset(statementIndex);

if (startOffset != 0 || statementIndex == 0)

{

GetLineCharOffsetFromStartChar(startOffset, &line, nullptr, false /\*canAllocateLineCache\*/);

line = line + 1;

}

}

return line;

}

uint FunctionBody::GetStartOffset(uint statementIndex) const

{

uint startOffset = 0;

const Js::FunctionBody::SourceInfo \* sourceInfo = &this->m\_sourceInfo;

if (sourceInfo->pSpanSequence != nullptr)

{

Js::SmallSpanSequenceIter iter;

sourceInfo->pSpanSequence->Reset(iter);

Js::StatementData data;

sourceInfo->pSpanSequence->Item(statementIndex, iter, data);

startOffset = data.sourceBegin;

}

else

{

int index = statementIndex;

Js::FunctionBody::StatementMap \* statementMap = GetNextNonSubexpressionStatementMap(GetStatementMaps(), index);

startOffset = statementMap->sourceSpan.Begin();

}

return startOffset;

}

#endif

void FunctionBody::SetIsNonUserCode(bool set)

{

// Mark current function as a non-user code, so that we can distinguish cases where exceptions are

// caught in non-user code (see ProbeContainer::HasAllowedForException).

SetFlags(set, Flags\_NonUserCode);

// Propagate setting for all functions in this scope (nested).

for (uint uIndex = 0; uIndex < this->m\_nestedCount; uIndex++)

{

Js::FunctionBody \* pBody = this->GetNestedFunc(uIndex)->GetFunctionBody();

if (pBody != nullptr)

{

pBody->SetIsNonUserCode(set);

}

}

}

void FunctionBody::InsertSymbolToRegSlotList(JsUtil::CharacterBuffer<WCHAR> const& propName, RegSlot reg, RegSlot totalRegsCount)

{

if (totalRegsCount > 0)

{

PropertyId propertyId = GetOrAddPropertyIdTracked(propName);

InsertSymbolToRegSlotList(reg, propertyId, totalRegsCount);

}

}

void FunctionBody::InsertSymbolToRegSlotList(RegSlot reg, PropertyId propertyId, RegSlot totalRegsCount)

{

if (totalRegsCount > 0)

{

if (this->propertyIdOnRegSlotsContainer == nullptr)

{

this->propertyIdOnRegSlotsContainer = PropertyIdOnRegSlotsContainer::New(m\_scriptContext->GetRecycler());

}

if (this->propertyIdOnRegSlotsContainer->propertyIdsForRegSlots == nullptr)

{

this->propertyIdOnRegSlotsContainer->CreateRegSlotsArray(m\_scriptContext->GetRecycler(), totalRegsCount);

}

Assert(this->propertyIdOnRegSlotsContainer != nullptr);

this->propertyIdOnRegSlotsContainer->Insert(reg, propertyId);

}

}

void FunctionBody::SetPropertyIdsOfFormals(PropertyIdArray \* formalArgs)

{

Assert(formalArgs);

if (this->propertyIdOnRegSlotsContainer == nullptr)

{

this->propertyIdOnRegSlotsContainer = PropertyIdOnRegSlotsContainer::New(m\_scriptContext->GetRecycler());

}

this->propertyIdOnRegSlotsContainer->SetFormalArgs(formalArgs);

}

HRESULT FunctionBody::RegisterFunction(BOOL fChangeMode, BOOL fOnlyCurrent)

{

if (!this->IsFunctionParsed())

{

return S\_OK;

}

HRESULT hr = this->ReportFunctionCompiled();

if (FAILED(hr))

{

return hr;

}

if (fChangeMode)

{

this->SetEntryToProfileMode();

}

if (!fOnlyCurrent)

{

for (uint uIndex = 0; uIndex < this->m\_nestedCount; uIndex++)

{

Js::ParseableFunctionInfo \* pBody = this->GetNestedFunctionForExecution(uIndex);

if (pBody == nullptr || !pBody->IsFunctionParsed())

{

continue;

}

hr = pBody->GetFunctionBody()->RegisterFunction(fChangeMode);

if (FAILED(hr))

{

break;

}

}

}

return hr;

}

HRESULT FunctionBody::ReportScriptCompiled()

{

AssertMsg(m\_scriptContext != nullptr, "Script Context is null when reporting function information");

PROFILER\_SCRIPT\_TYPE type = IsDynamicScript() ? PROFILER\_SCRIPT\_TYPE\_DYNAMIC : PROFILER\_SCRIPT\_TYPE\_USER;

IDebugDocumentContext \*pDebugDocumentContext = nullptr;

this->m\_scriptContext->GetDocumentContext(this->m\_scriptContext, this, &pDebugDocumentContext);

HRESULT hr = m\_scriptContext->OnScriptCompiled((PROFILER\_TOKEN) this->GetUtf8SourceInfo()->GetSourceInfoId(), type, pDebugDocumentContext);

RELEASEPTR(pDebugDocumentContext);

return hr;

}

HRESULT FunctionBody::ReportFunctionCompiled()

{

// Some assumptions by Logger interface.

// to send NULL as a name in case the name is anonymous and hint is anonymous code.

const wchar\_t \*pwszName = GetExternalDisplayName();

IDebugDocumentContext \*pDebugDocumentContext = nullptr;

this->m\_scriptContext->GetDocumentContext(this->m\_scriptContext, this, &pDebugDocumentContext);

SetHasFunctionCompiledSent(true);

HRESULT hr = m\_scriptContext->OnFunctionCompiled(m\_functionNumber, (PROFILER\_TOKEN) this->GetUtf8SourceInfo()->GetSourceInfoId(), pwszName, nullptr, pDebugDocumentContext);

RELEASEPTR(pDebugDocumentContext);

#if DBG

if (m\_iProfileSession >= m\_scriptContext->GetProfileSession())

{

OUTPUT\_TRACE\_DEBUGONLY(Js::ScriptProfilerPhase, L"FunctionBody::ReportFunctionCompiled, Duplicate compile event (%d < %d) for FunctionNumber : %d\n",

m\_iProfileSession, m\_scriptContext->GetProfileSession(), m\_functionNumber);

}

AssertMsg(m\_iProfileSession < m\_scriptContext->GetProfileSession(), "Duplicate compile event sent");

m\_iProfileSession = m\_scriptContext->GetProfileSession();

#endif

return hr;

}

void FunctionBody::SetEntryToProfileMode()

{

#if ENABLE\_NATIVE\_CODEGEN

AssertMsg(this->m\_scriptContext->CurrentThunk == ProfileEntryThunk, "ScriptContext not in profile mode");

#if DBG

AssertMsg(m\_iProfileSession == m\_scriptContext->GetProfileSession(), "Changing mode to profile for function that didn't send compile event");

#endif

// This is always done when bg thread is paused hence we don't need any kind of thread-synchronization at this point.

// Change entry points to Profile Thunk

// If the entrypoint is CodeGenOnDemand or CodeGen - then we don't change the entry points

ProxyEntryPointInfo\* defaultEntryPointInfo = this->GetDefaultEntryPointInfo();

if (!IsIntermediateCodeGenThunk((JavascriptMethod) defaultEntryPointInfo->address)

&& defaultEntryPointInfo->address != DynamicProfileInfo::EnsureDynamicProfileInfoThunk)

{

if (this->originalEntryPoint == DefaultDeferredParsingThunk)

{

defaultEntryPointInfo->address = ProfileDeferredParsingThunk;

}

else if (this->originalEntryPoint == DefaultDeferredDeserializeThunk)

{

defaultEntryPointInfo->address = ProfileDeferredDeserializeThunk;

}

else

{

defaultEntryPointInfo->address = ProfileEntryThunk;

}

}

// Update old entry points on the deferred prototype type so that they match current defaultEntryPointInfo.

// to make sure that new JavascriptFunction instances use profile thunk.

if (this->deferredPrototypeType)

{

this->deferredPrototypeType->SetEntryPoint((JavascriptMethod)this->GetDefaultEntryPointInfo()->address);

this->deferredPrototypeType->SetEntryPointInfo(this->GetDefaultEntryPointInfo());

}

#if DBG

if (!this->HasValidEntryPoint())

{

OUTPUT\_TRACE\_DEBUGONLY(Js::ScriptProfilerPhase, L"FunctionBody::SetEntryToProfileMode, Assert due to HasValidEntryPoint(), directEntrypoint : 0x%0IX, originalentrypoint : 0x%0IX\n",

(JavascriptMethod)this->GetDefaultEntryPointInfo()->address, this->originalEntryPoint);

AssertMsg(false, "Not a valid EntryPoint");

}

#endif

#endif //ENABLE\_NATIVE\_CODEGEN

}

#if DBG

void FunctionBody::MustBeInDebugMode()

{

Assert(m\_scriptContext->IsInDebugMode());

Assert(IsByteCodeDebugMode());

Assert(m\_sourceInfo.pSpanSequence == nullptr);

Assert(pStatementMaps != nullptr);

}

#endif

void FunctionBody::CleanupToReparse()

{

// The current function is already compiled. In order to prep this function to ready for debug mode, most of the previous information need to be thrown away.

// Clean up the nested functions

for (uint i = 0; i < m\_nestedCount; i++)

{

FunctionProxy\* proxy = GetNestedFunc(i);

if (proxy && proxy->IsFunctionBody())

{

proxy->GetFunctionBody()->CleanupToReparse();

}

}

CleanupRecyclerData(/\* isShutdown \*/ false, true /\* capture entry point cleanup stack trace \*/);

this->entryPoints->ClearAndZero();

#if DYNAMIC\_INTERPRETER\_THUNK

if (m\_isAsmJsFunction && m\_dynamicInterpreterThunk)

{

m\_scriptContext->ReleaseDynamicAsmJsInterpreterThunk((BYTE\*)this->m\_dynamicInterpreterThunk, true);

this->m\_dynamicInterpreterThunk = nullptr;

}

#endif

// Store the originalEntryPoint to restore it back immediately.

JavascriptMethod originalEntryPoint = this->originalEntryPoint;

this->CreateNewDefaultEntryPoint();

this->originalEntryPoint = originalEntryPoint;

if (this->m\_defaultEntryPointInfo)

{

this->GetDefaultFunctionEntryPointInfo()->entryPointIndex = 0;

}

this->auxBlock = nullptr;

this->auxContextBlock = nullptr;

this->byteCodeBlock = nullptr;

this->loopHeaderArray = nullptr;

this->m\_constTable = nullptr;

this->m\_scopeInfo = nullptr;

this->m\_codeGenRuntimeData = nullptr;

this->m\_codeGenGetSetRuntimeData = nullptr;

this->cacheIdToPropertyIdMap = nullptr;

this->referencedPropertyIdMap = nullptr;

this->literalRegexes = nullptr;

this->propertyIdsForScopeSlotArray = nullptr;

this->propertyIdOnRegSlotsContainer = nullptr;

this->pStatementMaps = nullptr;

this->profiledLdElemCount = 0;

this->profiledStElemCount = 0;

this->profiledCallSiteCount = 0;

this->profiledArrayCallSiteCount = 0;

this->profiledDivOrRemCount = 0;

this->profiledSwitchCount = 0;

this->profiledReturnTypeCount = 0;

this->profiledSlotCount = 0;

this->loopCount = 0;

this->m\_envDepth = (uint16)-1;

this->m\_byteCodeCount = 0;

this->m\_byteCodeWithoutLDACount = 0;

this->m\_byteCodeInLoopCount = 0;

this->functionBailOutRecord = nullptr;

#if ENABLE\_PROFILE\_INFO

this->dynamicProfileInfo = nullptr;

#endif

this->hasExecutionDynamicProfileInfo = false;

this->m\_firstTmpReg = Constants::NoRegister;

this->m\_varCount = 0;

this->m\_constCount = 0;

this->localClosureRegister = Constants::NoRegister;

this->localFrameDisplayRegister = Constants::NoRegister;

this->envRegister = Constants::NoRegister;

this->thisRegisterForEventHandler = Constants::NoRegister;

this->firstInnerScopeRegister = Constants::NoRegister;

this->funcExprScopeRegister = Constants::NoRegister;

this->innerScopeCount = 0;

this->hasCachedScopePropIds = false;

this->ResetObjectLiteralTypes();

this->inlineCacheCount = 0;

this->rootObjectLoadInlineCacheStart = 0;

this->rootObjectLoadMethodInlineCacheStart = 0;

this->rootObjectStoreInlineCacheStart = 0;

this->isInstInlineCacheCount = 0;

this->m\_inlineCachesOnFunctionObject = false;

this->referencedPropertyIdCount = 0;

#if ENABLE\_PROFILE\_INFO

this->polymorphicCallSiteInfoHead = nullptr;

#endif

this->interpretedCount = 0;

this->m\_hasDoneAllNonLocalReferenced = false;

this->debuggerScopeIndex = 0;

this->m\_utf8SourceInfo->DeleteLineOffsetCache();

// Reset to default.

this->flags = Flags\_HasNoExplicitReturnValue;

ResetInParams();

this->m\_isAsmjsMode = false;

this->m\_isAsmJsFunction = false;

this->m\_isAsmJsScheduledForFullJIT = false;

this->m\_asmJsTotalLoopCount = 0;

recentlyBailedOutOfJittedLoopBody = false;

loopInterpreterLimit = CONFIG\_FLAG(LoopInterpretCount);

ReinitializeExecutionModeAndLimits();

Assert(this->m\_sourceInfo.m\_probeCount == 0);

this->m\_sourceInfo.m\_probeBackingBlock = nullptr;

#if DBG

// This could be non-zero if the function threw exception before. Reset it.

this->m\_DEBUG\_executionCount = 0;

#endif

if (this->m\_sourceInfo.pSpanSequence != nullptr)

{

HeapDelete(this->m\_sourceInfo.pSpanSequence);

this->m\_sourceInfo.pSpanSequence = nullptr;

}

if (this->m\_sourceInfo.m\_auxStatementData != nullptr)

{

// This must be consistent with how we allocate the data for this and inner structures.

// We are using recycler, thus it's enough just to set to NULL.

Assert(m\_scriptContext->GetRecycler()->IsValidObject(m\_sourceInfo.m\_auxStatementData));

m\_sourceInfo.m\_auxStatementData = nullptr;

}

}

void FunctionBody::SetEntryToDeferParseForDebugger()

{

ProxyEntryPointInfo\* defaultEntryPointInfo = this->GetDefaultEntryPointInfo();

if (defaultEntryPointInfo->address != DefaultDeferredParsingThunk && defaultEntryPointInfo->address != ProfileDeferredParsingThunk)

{

// Just change the thunk, the cleanup will be done once the function gets called.

if (this->m\_scriptContext->CurrentThunk == ProfileEntryThunk)

{

defaultEntryPointInfo->address = ProfileDeferredParsingThunk;

}

else

{

defaultEntryPointInfo->address = DefaultDeferredParsingThunk;

}

this->originalEntryPoint = DefaultDeferredParsingThunk;

// Abandon the shared type so a new function will get a new one

this->deferredPrototypeType = nullptr;

this->attributes = (FunctionInfo::Attributes) (this->attributes | FunctionInfo::Attributes::DeferredParse);

}

// Set other state back to before parse as well

this->SetStackNestedFunc(false);

this->stackNestedFuncParent = nullptr;

this->SetReparsed(true);

#if DBG

wchar\_t debugStringBuffer[MAX\_FUNCTION\_BODY\_DEBUG\_STRING\_SIZE];

OUTPUT\_VERBOSE\_TRACE(Js::DebuggerPhase, L"Regenerate Due To Debug Mode: function %s (%s) from script context %p\n",

this->GetDisplayName(), this->GetDebugNumberSet(debugStringBuffer), m\_scriptContext);

#endif

}

//

// For library code all references to jitted entry points need to be removed

//

void FunctionBody::ResetEntryPoint()

{

if (this->entryPoints)

{

this->MapEntryPoints([] (int index, FunctionEntryPointInfo\* entryPoint)

{

if (nullptr != entryPoint)

{

// Finalize = Free up work item if it hasn't been released yet + entry point clean up

// isShutdown is false because cleanup is called only in the !isShutdown case

entryPoint->Finalize(/\*isShutdown\*/ false);

}

});

this->MapLoopHeaders([] (uint loopNumber, LoopHeader\* header)

{

header->MapEntryPoints([] (int index, LoopEntryPointInfo\* entryPoint)

{

entryPoint->Cleanup(/\*isShutdown\*/ false, true /\* capture cleanup stack \*/);

});

});

}

this->entryPoints->ClearAndZero();

this->CreateNewDefaultEntryPoint();

this->originalEntryPoint = DefaultEntryThunk;

m\_defaultEntryPointInfo->address = m\_scriptContext->CurrentThunk;

if (this->deferredPrototypeType)

{

// Update old entry points on the deferred prototype type,

// as they may point to old native code gen regions which age gone now.

this->deferredPrototypeType->SetEntryPoint((JavascriptMethod)this->GetDefaultEntryPointInfo()->address);

this->deferredPrototypeType->SetEntryPointInfo(this->GetDefaultEntryPointInfo());

}

ReinitializeExecutionModeAndLimits();

}

void FunctionBody::AddDeferParseAttribute()

{

this->attributes = (FunctionInfo::Attributes) (this->attributes | DeferredParse);

}

void FunctionBody::RemoveDeferParseAttribute()

{

this->attributes = (FunctionInfo::Attributes) (this->attributes & (~DeferredParse));

}

Js::DebuggerScope \* FunctionBody::GetDiagCatchScopeObjectAt(int byteCodeOffset)

{

if (GetScopeObjectChain())

{

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

{

Js::DebuggerScope \*debuggerScope = GetScopeObjectChain()->pScopeChain->Item(i);

Assert(debuggerScope);

if (debuggerScope->IsCatchScope() && debuggerScope->IsOffsetInScope(byteCodeOffset))

{

return debuggerScope;

}

}

}

return nullptr;

}

ushort SmallSpanSequence::GetDiff(int current, int prev)

{

int diff = current - prev;

if ((diff) < SHRT\_MIN || (diff) >= SHRT\_MAX)

{

diff = SHRT\_MAX;

if (!this->pActualOffsetList)

{

this->pActualOffsetList = JsUtil::GrowingUint32HeapArray::Create(4);

}

this->pActualOffsetList->Add(current);

}

return (ushort)diff;

}

// Get Values of the beginning of the statement at particular index.

BOOL SmallSpanSequence::GetRangeAt(int index, SmallSpanSequenceIter &iter, int \* pCountOfMissed, StatementData & data)

{

Assert((uint32)index < pStatementBuffer->Count());

SmallSpan span(pStatementBuffer->ItemInBuffer((uint32)index));

int countOfMissed = 0;

if ((short)span.sourceBegin == SHRT\_MAX)

{

// Look in ActualOffset store

Assert(this->pActualOffsetList);

Assert(this->pActualOffsetList->Count() > 0);

Assert(this->pActualOffsetList->Count() > (uint32)iter.indexOfActualOffset);

data.sourceBegin = this->pActualOffsetList->ItemInBuffer((uint32)iter.indexOfActualOffset);

countOfMissed++;

}

else

{

data.sourceBegin = iter.accumulatedSourceBegin + (short)span.sourceBegin;

}

if (span.bytecodeBegin == SHRT\_MAX)

{

// Look in ActualOffset store

Assert(this->pActualOffsetList);

Assert(this->pActualOffsetList->Count() > 0);

Assert(this->pActualOffsetList->Count() > (uint32)(iter.indexOfActualOffset + countOfMissed));

data.bytecodeBegin = this->pActualOffsetList->ItemInBuffer((uint32)iter.indexOfActualOffset + countOfMissed);

countOfMissed++;

}

else

{

data.bytecodeBegin = iter.accumulatedBytecodeBegin + span.bytecodeBegin;

}

if (pCountOfMissed)

{

\*pCountOfMissed = countOfMissed;

}

return TRUE;

}

void SmallSpanSequence::Reset(SmallSpanSequenceIter &iter)

{

iter.accumulatedIndex = 0;

iter.accumulatedSourceBegin = baseValue;

iter.accumulatedBytecodeBegin = 0;

iter.indexOfActualOffset = 0;

}

BOOL SmallSpanSequence::GetMatchingStatementFromBytecode(int bytecode, SmallSpanSequenceIter &iter, StatementData & data)

{

if (Count() > 0 && bytecode >= 0)

{

// Support only in forward direction

if (bytecode < iter.accumulatedBytecodeBegin

|| iter.accumulatedIndex <= 0 || (uint32)iter.accumulatedIndex >= Count())

{

// re-initialize the accumulators

Reset(iter);

}

while ((uint32)iter.accumulatedIndex < Count())

{

int countOfMissed = 0;

if (!GetRangeAt(iter.accumulatedIndex, iter, &countOfMissed, data))

{

Assert(FALSE);

break;

}

if (data.bytecodeBegin >= bytecode)

{

if (data.bytecodeBegin > bytecode)

{

// Not exactly at the current bytecode, so it falls in between previous statement.

data.sourceBegin = iter.accumulatedSourceBegin;

data.bytecodeBegin = iter.accumulatedBytecodeBegin;

}

return TRUE;

}

// Look for the next

iter.accumulatedSourceBegin = data.sourceBegin;

iter.accumulatedBytecodeBegin = data.bytecodeBegin;

iter.accumulatedIndex++;

if (countOfMissed)

{

iter.indexOfActualOffset += countOfMissed;

}

}

if (iter.accumulatedIndex != -1)

{

// Give the last one.

Assert(data.bytecodeBegin < bytecode);

return TRUE;

}

}

// Failed to give the correct one, init to default

iter.accumulatedIndex = -1;

return FALSE;

}

BOOL SmallSpanSequence::Item(int index, SmallSpanSequenceIter &iter, StatementData & data)

{

if (!pStatementBuffer || (uint32)index >= pStatementBuffer->Count())

{

return FALSE;

}

if (iter.accumulatedIndex <= 0 || iter.accumulatedIndex > index)

{

Reset(iter);

}

while (iter.accumulatedIndex <= index)

{

Assert((uint32)iter.accumulatedIndex < pStatementBuffer->Count());

int countOfMissed = 0;

if (!GetRangeAt(iter.accumulatedIndex, iter, &countOfMissed, data))

{

Assert(FALSE);

break;

}

// We store the next index

iter.accumulatedSourceBegin = data.sourceBegin;

iter.accumulatedBytecodeBegin = data.bytecodeBegin;

iter.accumulatedIndex++;

if (countOfMissed)

{

iter.indexOfActualOffset += countOfMissed;

}

if ((iter.accumulatedIndex - 1) == index)

{

return TRUE;

}

}

return FALSE;

}

BOOL SmallSpanSequence::Seek(int index, StatementData & data)

{

// This method will not alter any state of the variables, so this will just do plain search

// from the beginning to look for that index.

SmallSpanSequenceIter iter;

Reset(iter);

return Item(index, iter, data);

}

SmallSpanSequence \* SmallSpanSequence::Clone()

{

SmallSpanSequence \*pNewSequence = HeapNew(SmallSpanSequence);

pNewSequence->baseValue = baseValue;

if (pStatementBuffer)

{

pNewSequence->pStatementBuffer = pStatementBuffer->Clone();

}

if (pActualOffsetList)

{

pNewSequence->pActualOffsetList = pActualOffsetList->Clone();

}

return pNewSequence;

}

PropertyIdOnRegSlotsContainer \* PropertyIdOnRegSlotsContainer::New(Recycler \* recycler)

{

return RecyclerNew(recycler, PropertyIdOnRegSlotsContainer);

}

PropertyIdOnRegSlotsContainer::PropertyIdOnRegSlotsContainer()

: propertyIdsForRegSlots(nullptr), length(0), propertyIdsForFormalArgs(nullptr)

{

}

void PropertyIdOnRegSlotsContainer::CreateRegSlotsArray(Recycler \* recycler, uint \_length)

{

Assert(propertyIdsForRegSlots == nullptr);

propertyIdsForRegSlots = RecyclerNewArrayLeafZ(recycler, PropertyId, \_length);

length = \_length;

}

void PropertyIdOnRegSlotsContainer::SetFormalArgs(PropertyIdArray \* formalArgs)

{

propertyIdsForFormalArgs = formalArgs;

}

//

// Helper methods for PropertyIdOnRegSlotsContainer

void PropertyIdOnRegSlotsContainer::Insert(RegSlot reg, PropertyId propId)

{

//

// Reg is being used as an index;

Assert(propertyIdsForRegSlots);

Assert(reg < length);

//

// the current reg is unaccounted for const reg count. while fetching calculate the actual regslot value.

Assert(propertyIdsForRegSlots[reg] == 0 || propertyIdsForRegSlots[reg] == propId);

propertyIdsForRegSlots[reg] = propId;

}

void PropertyIdOnRegSlotsContainer::FetchItemAt(uint index, FunctionBody \*pFuncBody, \_\_out PropertyId \*pPropId, \_\_out RegSlot \*pRegSlot)

{

Assert(index < length);

Assert(pPropId);

Assert(pRegSlot);

Assert(pFuncBody);

\*pPropId = propertyIdsForRegSlots[index];

\*pRegSlot = pFuncBody->MapRegSlot(index);

}

bool PropertyIdOnRegSlotsContainer::IsRegSlotFormal(RegSlot reg)

{

if (propertyIdsForFormalArgs != nullptr && reg < length)

{

PropertyId propId = propertyIdsForRegSlots[reg];

for (uint32 i = 0; i < propertyIdsForFormalArgs->count; i++)

{

if (propertyIdsForFormalArgs->elements[i] == propId)

{

return true;

}

}

}

return false;

}

ScopeType FrameDisplay::GetScopeType(void\* scope)

{

if(Js::ActivationObject::Is(scope))

{

return ScopeType\_ActivationObject;

}

if(Js::ScopeSlots::Is(scope))

{

return ScopeType\_SlotArray;

}

return ScopeType\_WithScope;

}

// DebuggerScope

// Get the sibling for the current debugger scope.

DebuggerScope \* DebuggerScope::GetSiblingScope(RegSlot location, FunctionBody \*functionBody)

{

bool isBlockSlotOrObject = scopeType == Js::DiagExtraScopesType::DiagBlockScopeInSlot || scopeType == Js::DiagExtraScopesType::DiagBlockScopeInObject;

bool isCatchSlotOrObject = scopeType == Js::DiagExtraScopesType::DiagCatchScopeInSlot || scopeType == Js::DiagExtraScopesType::DiagCatchScopeInObject;

// This is expected to be called only when the current scope is either slot or activation object.

Assert(isBlockSlotOrObject || isCatchSlotOrObject);

if (siblingScope == nullptr)

{

// If the sibling isn't there, attempt to retrieve it if we're reparsing or create it anew if this is the first parse.

siblingScope = functionBody->RecordStartScopeObject(isBlockSlotOrObject ? Js::DiagExtraScopesType::DiagBlockScopeDirect : Js::DiagExtraScopesType::DiagCatchScopeDirect, GetStart(), location);

}

return siblingScope;

}

// Adds a new property to be tracked in the debugger scope.

// location - The slot array index or register slot location of where the property is stored.

// propertyId - The property ID of the property.

// flags - Flags that help describe the property.

void DebuggerScope::AddProperty(RegSlot location, Js::PropertyId propertyId, DebuggerScopePropertyFlags flags)

{

DebuggerScopeProperty scopeProperty;

scopeProperty.location = location;

scopeProperty.propId = propertyId;

// This offset is uninitialized until the property is initialized (with a ld opcode, for example).

scopeProperty.byteCodeInitializationOffset = Constants::InvalidByteCodeOffset;

scopeProperty.flags = flags;

// Delay allocate the property list so we don't take up memory if there are no properties in this scope.

// Scopes are created during non-debug mode as well so we want to keep them as small as possible.

this->EnsurePropertyListIsAllocated();

// The property doesn't exist yet, so add it.

this->scopeProperties->Add(scopeProperty);

}

bool DebuggerScope::GetPropertyIndex(Js::PropertyId propertyId, int& index)

{

if (!this->HasProperties())

{

index = -1;

return false;

}

bool found = this->scopeProperties->MapUntil( [&](int i, const DebuggerScopeProperty& scopeProperty) {

if(scopeProperty.propId == propertyId)

{

index = scopeProperty.location;

return true;

}

return false;

});

if(!found)

{

return false;

}

return true;

}

#if DBG

void DebuggerScope::Dump()

{

int indent = (GetScopeDepth() - 1) \* 4;

Output::Print(indent, L"Begin scope: Address: %p Type: %s Location: %d Sibling: %p Range: [%d, %d]\n ", this, GetDebuggerScopeTypeString(scopeType), scopeLocation, this->siblingScope, range.begin, range.end);

if (this->HasProperties())

{

this->scopeProperties->Map( [=] (int i, Js::DebuggerScopeProperty& scopeProperty) {

Output::Print(indent, L"%s(%d) Location: %d Const: %s Initialized: %d\n", ThreadContext::GetContextForCurrentThread()->GetPropertyName(scopeProperty.propId)->GetBuffer(),

scopeProperty.propId, scopeProperty.location, scopeProperty.IsConst() ? L"true": L"false", scopeProperty.byteCodeInitializationOffset);

});

}

Output::Print(L"\n");

}

// Returns the debugger scope type in string format.

PCWSTR DebuggerScope::GetDebuggerScopeTypeString(DiagExtraScopesType scopeType)

{

switch (scopeType)

{

case DiagExtraScopesType::DiagBlockScopeDirect:

return L"DiagBlockScopeDirect";

case DiagExtraScopesType::DiagBlockScopeInObject:

return L"DiagBlockScopeInObject";

case DiagExtraScopesType::DiagBlockScopeInSlot:

return L"DiagBlockScopeInSlot";

case DiagExtraScopesType::DiagBlockScopeRangeEnd:

return L"DiagBlockScopeRangeEnd";

case DiagExtraScopesType::DiagCatchScopeDirect:

return L"DiagCatchScopeDirect";

case DiagExtraScopesType::DiagCatchScopeInObject:

return L"DiagCatchScopeInObject";

case DiagExtraScopesType::DiagCatchScopeInSlot:

return L"DiagCatchScopeInSlot";

case DiagExtraScopesType::DiagUnknownScope:

return L"DiagUnknownScope";

case DiagExtraScopesType::DiagWithScope:

return L"DiagWithScope";

default:

AssertMsg(false, "Missing a debug scope type.");

return L"";

}

}

#endif

// Updates the current offset of where the property is first initialized. This is used to

// detect whether or not a property is in a dead zone when broken in the debugger.

// location - The slot array index or register slot location of where the property is stored.

// propertyId - The property ID of the property.

// byteCodeOffset - The offset to set the initialization point at.

// isFunctionDeclaration - Whether or not the property is a function declaration or not. Used for verification.

// <returns> - True if the property was found and updated for the current scope, else false.

bool DebuggerScope::UpdatePropertyInitializationOffset(

RegSlot location,

Js::PropertyId propertyId,

int byteCodeOffset,

bool isFunctionDeclaration /\*= false\*/)

{

if (UpdatePropertyInitializationOffsetInternal(location, propertyId, byteCodeOffset, isFunctionDeclaration))

{

return true;

}

if (siblingScope != nullptr && siblingScope->UpdatePropertyInitializationOffsetInternal(location, propertyId, byteCodeOffset, isFunctionDeclaration))

{

return true;

}

return false;

}

bool DebuggerScope::UpdatePropertyInitializationOffsetInternal(

RegSlot location,

Js::PropertyId propertyId,

int byteCodeOffset,

bool isFunctionDeclaration /\*= false\*/)

{

if (scopeProperties == nullptr)

{

return false;

}

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

{

DebuggerScopeProperty propertyItem = scopeProperties->Item(i);

if (propertyItem.propId == propertyId && propertyItem.location == location)

{

if (propertyItem.byteCodeInitializationOffset == Constants::InvalidByteCodeOffset)

{

propertyItem.byteCodeInitializationOffset = byteCodeOffset;

scopeProperties->SetExistingItem(i, propertyItem);

}

#if DBG

else

{

// If the bytecode initialization offset is not Constants::InvalidByteCodeOffset,

// it means we have two or more functions declared in the same scope with the same name

// and one has already been marked. We track each location with a property entry

// on the debugging side (when calling DebuggerScope::AddProperty()) as opposed to scanning

// and checking if the property already exists each time we add in order to avoid duplicates.

AssertMsg(isFunctionDeclaration, "Only function declarations can be defined more than once in the same scope with the same name.");

AssertMsg(propertyItem.byteCodeInitializationOffset == byteCodeOffset, "The bytecode offset for all function declarations should be identical for this scope.");

}

#endif // DBG

return true;

}

}

return false;

}

// Updates the debugger scopes fields due to a regeneration of bytecode (happens during debugger attach or detach, for

// example).

void DebuggerScope::UpdateDueToByteCodeRegeneration(DiagExtraScopesType scopeType, int start, RegSlot scopeLocation)

{

#if DBG

if (this->scopeType != Js::DiagUnknownScope)

{

// If the scope is unknown, it was deserialized without a scope type. Otherwise, it should not have changed.

// The scope type can change on a re-parse in certain scenarios related to eval detection in legacy mode -> Winblue: 272122

AssertMsg(this->scopeType == scopeType, "The debugger scope type should not have changed when generating bytecode again.");

}

#endif // DBG

this->scopeType = scopeType;

this->SetBegin(start);

if(this->scopeProperties)

{

this->scopeProperties->Clear();

}

// Reset the scope location as it may have changed during bytecode generation from the last run.

this->SetScopeLocation(scopeLocation);

if (siblingScope)

{

// If we had a sibling scope during initial parsing, clear it now so that it will be reset

// when it is retrieved during this bytecode generation pass, in GetSiblingScope().

// GetSiblingScope() will ensure that the FunctionBody currentDebuggerScopeIndex value is

// updated accordingly to account for future scopes coming after the sibling.

// Calling of GetSiblingScope() will happen when register properties are added to this scope

// via TrackRegisterPropertyForDebugger().

siblingScope = nullptr;

}

}

void DebuggerScope::UpdatePropertiesInForInOrOfCollectionScope()

{

if (this->scopeProperties != nullptr)

{

this->scopeProperties->All([&](Js::DebuggerScopeProperty& propertyItem)

{

propertyItem.flags |= DebuggerScopePropertyFlags\_ForInOrOfCollection;

return true;

});

}

}

void DebuggerScope::EnsurePropertyListIsAllocated()

{

if (this->scopeProperties == nullptr)

{

this->scopeProperties = RecyclerNew(this->recycler, DebuggerScopePropertyList, this->recycler);

}

}

// Checks if the passed in ByteCodeGenerator offset is in this scope's being/end range.

bool DebuggerScope::IsOffsetInScope(int offset) const

{

Assert(this->range.end != -1);

return this->range.Includes(offset);

}

// Determines if the DebuggerScope contains a property with the passed in ID and

// location in the internal property list.

// propertyId - The ID of the property to search for.

// location - The slot array index or register to search for.

// outScopeProperty - Optional parameter that will return the property, if found.

bool DebuggerScope::Contains(Js::PropertyId propertyId, RegSlot location) const

{

DebuggerScopeProperty tempProperty;

return TryGetProperty(propertyId, location, &tempProperty);

}

// Gets whether or not the scope is a block scope (non-catch or with).

bool DebuggerScope::IsBlockScope() const

{

AssertMsg(this->scopeType != Js::DiagBlockScopeRangeEnd, "Debugger scope type should never be set to range end - only reserved for marking the end of a scope (not persisted).");

return this->scopeType == Js::DiagBlockScopeDirect

|| this->scopeType == Js::DiagBlockScopeInObject

|| this->scopeType == Js::DiagBlockScopeInSlot

|| this->scopeType == Js::DiagBlockScopeRangeEnd;

}

// Gets whether or not the scope is a catch block scope.

bool DebuggerScope::IsCatchScope() const

{

return this->scopeType == Js::DiagCatchScopeDirect

|| this->scopeType == Js::DiagCatchScopeInObject

|| this->scopeType == Js::DiagCatchScopeInSlot;

}

// Gets whether or not the scope is a with block scope.

bool DebuggerScope::IsWithScope() const

{

return this->scopeType == Js::DiagWithScope;

}

// Gets whether or not the scope is a slot array scope.

bool DebuggerScope::IsSlotScope() const

{

return this->scopeType == Js::DiagBlockScopeInSlot

|| this->scopeType == Js::DiagCatchScopeInSlot;

}

// Gets whether or not the scope has any properties in it.

bool DebuggerScope::HasProperties() const

{

return this->scopeProperties && this->scopeProperties->Count() > 0;

}

// Checks if this scope is an ancestor of the passed in scope.

bool DebuggerScope::IsAncestorOf(const DebuggerScope\* potentialChildScope)

{

if (potentialChildScope == nullptr)

{

// If the child scope is null, it represents the global scope which

// cannot be a child of anything.

return false;

}

const DebuggerScope\* currentScope = potentialChildScope;

while (currentScope)

{

if (currentScope->GetParentScope() == this)

{

return true;

}

currentScope = currentScope->GetParentScope();

}

return false;

}

// Checks if all properties of the scope are currently in a dead zone given the specified offset.

bool DebuggerScope::AreAllPropertiesInDeadZone(int byteCodeOffset) const

{

if (!this->HasProperties())

{

return false;

}

return this->scopeProperties->All([&](Js::DebuggerScopeProperty& propertyItem)

{

return propertyItem.IsInDeadZone(byteCodeOffset);

});

}

// Attempts to get the specified property. Returns true if the property was copied to the structure; false otherwise.

bool DebuggerScope::TryGetProperty(Js::PropertyId propertyId, RegSlot location, DebuggerScopeProperty\* outScopeProperty) const

{

Assert(outScopeProperty);

if (scopeProperties == nullptr)

{

return false;

}

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

{

DebuggerScopeProperty propertyItem = scopeProperties->Item(i);

if (propertyItem.propId == propertyId && propertyItem.location == location)

{

\*outScopeProperty = propertyItem;

return true;

}

}

return false;

}

bool DebuggerScope::TryGetValidProperty(Js::PropertyId propertyId, RegSlot location, int offset, DebuggerScopeProperty\* outScopeProperty, bool\* isInDeadZone) const

{

if (TryGetProperty(propertyId, location, outScopeProperty))

{

if (IsOffsetInScope(offset))

{

if (isInDeadZone != nullptr)

{

\*isInDeadZone = outScopeProperty->IsInDeadZone(offset);

}

return true;

}

}

return false;

}

void DebuggerScope::SetBegin(int begin)

{

range.begin = begin;

if (siblingScope != nullptr)

{

siblingScope->SetBegin(begin);

}

}

void DebuggerScope::SetEnd(int end)

{

range.end = end;

if (siblingScope != nullptr)

{

siblingScope->SetEnd(end);

}

}

// Finds the common ancestor scope between this scope and the passed in scope.

// Returns nullptr if the scopes are part of different trees.

DebuggerScope\* DebuggerScope::FindCommonAncestor(DebuggerScope\* debuggerScope)

{

AnalysisAssert(debuggerScope);

if (this == debuggerScope)

{

return debuggerScope;

}

if (this->IsAncestorOf(debuggerScope))

{

return this;

}

if (debuggerScope->IsAncestorOf(this))

{

return debuggerScope;

}

DebuggerScope\* firstNode = this;

DebuggerScope\* secondNode = debuggerScope;

int firstDepth = firstNode->GetScopeDepth();

int secondDepth = secondNode->GetScopeDepth();

// Calculate the depth difference in order to bring the deep node up to the sibling

// level of the shorter node.

int depthDifference = abs(firstDepth - secondDepth);

DebuggerScope\*& nodeToBringUp = firstDepth > secondDepth ? firstNode : secondNode;

while (depthDifference > 0)

{

AnalysisAssert(nodeToBringUp);

nodeToBringUp = nodeToBringUp->GetParentScope();

--depthDifference;

}

// Move up the tree and see where the nodes meet.

while (firstNode && secondNode)

{

if (firstNode == secondNode)

{

return firstNode;

}

firstNode = firstNode->GetParentScope();

secondNode = secondNode->GetParentScope();

}

// The nodes are not part of the same scope tree.

return nullptr;

}

// Gets the depth of the scope in the parent link tree.

int DebuggerScope::GetScopeDepth() const

{

int depth = 0;

const DebuggerScope\* currentDebuggerScope = this;

while (currentDebuggerScope)

{

currentDebuggerScope = currentDebuggerScope->GetParentScope();

++depth;

}

return depth;

}

bool ScopeObjectChain::TryGetDebuggerScopePropertyInfo(PropertyId propertyId, RegSlot location, int offset, bool\* isPropertyInDebuggerScope, bool \*isConst, bool\* isInDeadZone)

{

Assert(pScopeChain);

Assert(isPropertyInDebuggerScope);

Assert(isConst);

\*isPropertyInDebuggerScope = false;

\*isConst = false;

// Search through each block scope until we find the current scope. If the register was found

// in any of the scopes going down until we reach the scope of the debug break, then it's in scope.

// if found but not in the scope, the out param will be updated (since it is actually a let or const), so that caller can make a call accordingly.

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

{

Js::DebuggerScope \*debuggerScope = pScopeChain->Item(i);

DebuggerScopeProperty debuggerScopeProperty;

if (debuggerScope->TryGetProperty(propertyId, location, &debuggerScopeProperty))

{

bool isOffsetInScope = debuggerScope->IsOffsetInScope(offset);

// For the Object scope, all the properties will have the same location (-1) so they can match. Use further check below to determine the propertyInDebuggerScope

\*isPropertyInDebuggerScope = isOffsetInScope || !debuggerScope->IsBlockObjectScope();

if (isOffsetInScope)

{

if (isInDeadZone != nullptr)

{

\*isInDeadZone = debuggerScopeProperty.IsInDeadZone(offset);

}

\*isConst = debuggerScopeProperty.IsConst();

return true;

}

}

}

return false;

}

void FunctionBody::AllocateInlineCache()

{

Assert(this->inlineCaches == nullptr);

uint isInstInlineCacheStart = this->GetInlineCacheCount();

uint totalCacheCount = isInstInlineCacheStart + isInstInlineCacheCount;

if (totalCacheCount != 0)

{

// Root object inline cache are not leaf

void \*\* inlineCaches = RecyclerNewArrayZ(this->m\_scriptContext->GetRecycler(),

void\*, totalCacheCount);

#if DBG

this->m\_inlineCacheTypes = RecyclerNewArrayLeafZ(this->m\_scriptContext->GetRecycler(),

byte, totalCacheCount);

#endif

uint i = 0;

uint plainInlineCacheEnd = rootObjectLoadInlineCacheStart;

\_\_analysis\_assume(plainInlineCacheEnd <= totalCacheCount);

for (; i < plainInlineCacheEnd; i++)

{

inlineCaches[i] = AllocatorNewZ(InlineCacheAllocator,

this->m\_scriptContext->GetInlineCacheAllocator(), InlineCache);

}

Js::RootObjectBase \* rootObject = this->GetRootObject();

ThreadContext \* threadContext = this->GetScriptContext()->GetThreadContext();

uint rootObjectLoadInlineCacheEnd = rootObjectLoadMethodInlineCacheStart;

\_\_analysis\_assume(rootObjectLoadInlineCacheEnd <= totalCacheCount);

for (; i < rootObjectLoadInlineCacheEnd; i++)

{

inlineCaches[i] = rootObject->GetInlineCache(

threadContext->GetPropertyName(this->GetPropertyIdFromCacheId(i)), false, false);

}

uint rootObjectLoadMethodInlineCacheEnd = rootObjectStoreInlineCacheStart;

\_\_analysis\_assume(rootObjectLoadMethodInlineCacheEnd <= totalCacheCount);

for (; i < rootObjectLoadMethodInlineCacheEnd; i++)

{

inlineCaches[i] = rootObject->GetInlineCache(

threadContext->GetPropertyName(this->GetPropertyIdFromCacheId(i)), true, false);

}

uint rootObjectStoreInlineCacheEnd = isInstInlineCacheStart;

\_\_analysis\_assume(rootObjectStoreInlineCacheEnd <= totalCacheCount);

for (; i < rootObjectStoreInlineCacheEnd; i++)

{

#pragma prefast(suppress:6386, "The analysis assume didn't help prefast figure out this is in range")

inlineCaches[i] = rootObject->GetInlineCache(

threadContext->GetPropertyName(this->GetPropertyIdFromCacheId(i)), false, true);

}

for (; i < totalCacheCount; i++)

{

inlineCaches[i] = AllocatorNewStructZ(IsInstInlineCacheAllocator,

this->m\_scriptContext->GetIsInstInlineCacheAllocator(), IsInstInlineCache);

}

#if DBG

this->m\_inlineCacheTypes = RecyclerNewArrayLeafZ(this->m\_scriptContext->GetRecycler(),

byte, totalCacheCount);

#endif

this->inlineCaches = inlineCaches;

}

}

InlineCache \*FunctionBody::GetInlineCache(uint index)

{

Assert(this->inlineCaches != nullptr);

Assert(index < this->GetInlineCacheCount());

#if DBG

Assert(this->m\_inlineCacheTypes[index] == InlineCacheTypeNone ||

this->m\_inlineCacheTypes[index] == InlineCacheTypeInlineCache);

this->m\_inlineCacheTypes[index] = InlineCacheTypeInlineCache;

#endif

return reinterpret\_cast<InlineCache \*>(this->inlineCaches[index]);

}

bool FunctionBody::CanFunctionObjectHaveInlineCaches()

{

if (this->DoStackNestedFunc() || this->IsGenerator())

{

return false;

}

uint totalCacheCount = this->GetInlineCacheCount() + this->GetIsInstInlineCacheCount();

if (PHASE\_FORCE(Js::ScriptFunctionWithInlineCachePhase, this) && totalCacheCount > 0)

{

return true;

}

// Only have inline caches on function object for possible inlining candidates.

// Since we don't know the size of the top function, check against the maximum possible inline threshold

// Negative inline byte code size threshold will disable inline cache on function object.

const int byteCodeSizeThreshold = CONFIG\_FLAG(InlineThreshold) + CONFIG\_FLAG(InlineThresholdAdjustCountInSmallFunction);

if (byteCodeSizeThreshold < 0 || this->GetByteCodeWithoutLDACount() > (uint)byteCodeSizeThreshold)

{

return false;

}

// Negative FuncObjectInlineCacheThreshold will disable inline cache on function object.

if (CONFIG\_FLAG(FuncObjectInlineCacheThreshold) < 0 || totalCacheCount > (uint)CONFIG\_FLAG(FuncObjectInlineCacheThreshold) || totalCacheCount == 0)

{

return false;

}

return true;

}

void\*\* FunctionBody::GetInlineCaches()

{

return this->inlineCaches;

}

#if DBG

byte\* FunctionBody::GetInlineCacheTypes()

{

return this->m\_inlineCacheTypes;

}

#endif

IsInstInlineCache \*FunctionBody::GetIsInstInlineCache(uint index)

{

Assert(this->inlineCaches != nullptr);

Assert(index < GetIsInstInlineCacheCount());

index += this->GetInlineCacheCount();

#if DBG

Assert(this->m\_inlineCacheTypes[index] == InlineCacheTypeNone ||

this->m\_inlineCacheTypes[index] == InlineCacheTypeIsInst);

this->m\_inlineCacheTypes[index] = InlineCacheTypeIsInst;

#endif

return reinterpret\_cast<IsInstInlineCache \*>(this->inlineCaches[index]);

}

PolymorphicInlineCache \* FunctionBody::GetPolymorphicInlineCache(uint index)

{

return this->polymorphicInlineCaches.GetInlineCache(this, index);

}

PolymorphicInlineCache \* FunctionBody::CreateNewPolymorphicInlineCache(uint index, PropertyId propertyId, InlineCache \* inlineCache)

{

Assert(GetPolymorphicInlineCache(index) == nullptr);

// Only create polymorphic inline caches for non-root inline cache indexes

if (index < rootObjectLoadInlineCacheStart

#if DBG

&& !PHASE\_OFF1(Js::PolymorphicInlineCachePhase)

#endif

)

{

PolymorphicInlineCache \* polymorphicInlineCache = CreatePolymorphicInlineCache(index, PolymorphicInlineCache::GetInitialSize());

#ifdef ENABLE\_DEBUG\_CONFIG\_OPTIONS

if (PHASE\_VERBOSE\_TRACE1(Js::PolymorphicInlineCachePhase))

{

this->DumpFullFunctionName();

Output::Print(L": New PIC, index = %d, size = %d\n", index, PolymorphicInlineCache::GetInitialSize());

}

#endif

#if PHASE\_PRINT\_INTRUSIVE\_TESTTRACE1

wchar\_t debugStringBuffer[MAX\_FUNCTION\_BODY\_DEBUG\_STRING\_SIZE];

#endif

PHASE\_PRINT\_INTRUSIVE\_TESTTRACE1(

Js::PolymorphicInlineCachePhase,

L"TestTrace PIC: New, Function %s (%s), 0x%x, index = %d, size = %d\n", this->GetDisplayName(), this->GetDebugNumberSet(debugStringBuffer), polymorphicInlineCache, index, PolymorphicInlineCache::GetInitialSize());

uint indexInPolymorphicCache = polymorphicInlineCache->GetInlineCacheIndexForType(inlineCache->GetType());

inlineCache->CopyTo(propertyId, m\_scriptContext, &(polymorphicInlineCache->GetInlineCaches()[indexInPolymorphicCache]));

polymorphicInlineCache->UpdateInlineCachesFillInfo(indexInPolymorphicCache, true /\*set\*/);

return polymorphicInlineCache;

}

return nullptr;

}

PolymorphicInlineCache \* FunctionBody::CreateBiggerPolymorphicInlineCache(uint index, PropertyId propertyId)

{

PolymorphicInlineCache \* polymorphicInlineCache = GetPolymorphicInlineCache(index);

Assert(polymorphicInlineCache && polymorphicInlineCache->CanAllocateBigger());

uint16 polymorphicInlineCacheSize = polymorphicInlineCache->GetSize();

uint16 newPolymorphicInlineCacheSize = PolymorphicInlineCache::GetNextSize(polymorphicInlineCacheSize);

Assert(newPolymorphicInlineCacheSize > polymorphicInlineCacheSize);

PolymorphicInlineCache \* newPolymorphicInlineCache = CreatePolymorphicInlineCache(index, newPolymorphicInlineCacheSize);

polymorphicInlineCache->CopyTo(propertyId, m\_scriptContext, newPolymorphicInlineCache);

#ifdef ENABLE\_DEBUG\_CONFIG\_OPTIONS

if (PHASE\_VERBOSE\_TRACE1(Js::PolymorphicInlineCachePhase))

{

this->DumpFullFunctionName();

Output::Print(L": Bigger PIC, index = %d, oldSize = %d, newSize = %d\n", index, polymorphicInlineCacheSize, newPolymorphicInlineCacheSize);

}

#endif

#if PHASE\_PRINT\_INTRUSIVE\_TESTTRACE1

wchar\_t debugStringBuffer[MAX\_FUNCTION\_BODY\_DEBUG\_STRING\_SIZE];

#endif

PHASE\_PRINT\_INTRUSIVE\_TESTTRACE1(

Js::PolymorphicInlineCachePhase,

L"TestTrace PIC: Bigger, Function %s (%s), 0x%x, index = %d, size = %d\n", this->GetDisplayName(), this->GetDebugNumberSet(debugStringBuffer), newPolymorphicInlineCache, index, newPolymorphicInlineCacheSize);

return newPolymorphicInlineCache;

}

void FunctionBody::ResetInlineCaches()

{

isInstInlineCacheCount = inlineCacheCount = rootObjectLoadInlineCacheStart = rootObjectStoreInlineCacheStart = 0;

this->inlineCaches = nullptr;

this->polymorphicInlineCaches.Reset();

}

PolymorphicInlineCache \* FunctionBody::CreatePolymorphicInlineCache(uint index, uint16 size)

{

Recycler \* recycler = this->m\_scriptContext->GetRecycler();

PolymorphicInlineCache \* newPolymorphicInlineCache = PolymorphicInlineCache::New(size, this);

this->polymorphicInlineCaches.SetInlineCache(recycler, this, index, newPolymorphicInlineCache);

return newPolymorphicInlineCache;

}

uint FunctionBody::NewObjectLiteral()

{

Assert(objLiteralTypes == nullptr);

return objLiteralCount++;

}

DynamicType \*\* FunctionBody::GetObjectLiteralTypeRef(uint index)

{

Assert(index < objLiteralCount);

Assert(objLiteralTypes != nullptr);

return objLiteralTypes + index;

}

void FunctionBody::AllocateObjectLiteralTypeArray()

{

Assert(objLiteralTypes == nullptr);

if (objLiteralCount == 0)

{

return;

}

objLiteralTypes = RecyclerNewArrayZ(this->GetScriptContext()->GetRecycler(), DynamicType \*, objLiteralCount);

}

uint FunctionBody::NewLiteralRegex()

{

Assert(!this->literalRegexes);

return literalRegexCount++;

}

uint FunctionBody::GetLiteralRegexCount() const

{

return literalRegexCount;

}

void FunctionBody::AllocateLiteralRegexArray()

{

Assert(!this->literalRegexes);

if (literalRegexCount == 0)

{

return;

}

this->literalRegexes =

RecyclerNewArrayZ(m\_scriptContext->GetRecycler(), UnifiedRegex::RegexPattern \*, literalRegexCount);

}

#ifndef TEMP\_DISABLE\_ASMJS

AsmJsFunctionInfo\* FunctionBody::AllocateAsmJsFunctionInfo()

{

Assert( !this->asmJsFunctionInfo );

this->asmJsFunctionInfo = RecyclerNew( m\_scriptContext->GetRecycler(), AsmJsFunctionInfo );

return this->asmJsFunctionInfo;

}

AsmJsModuleInfo\* FunctionBody::AllocateAsmJsModuleInfo()

{

Assert( !this->asmJsModuleInfo );

Recycler\* rec = m\_scriptContext->GetRecycler();

this->asmJsModuleInfo = RecyclerNew( rec, AsmJsModuleInfo, rec );

return this->asmJsModuleInfo;

}

#endif

UnifiedRegex::RegexPattern \*FunctionBody::GetLiteralRegex(const uint index)

{

Assert(index < literalRegexCount);

Assert(this->literalRegexes);

return this->literalRegexes[index];

}

void FunctionBody::SetLiteralRegex(const uint index, UnifiedRegex::RegexPattern \*const pattern)

{

Assert(index < literalRegexCount);

Assert(this->literalRegexes);

if (this->literalRegexes[index] && this->literalRegexes[index] == pattern)

{

return;

}

Assert(!this->literalRegexes[index]);

this->literalRegexes[index] = pattern;

}

void FunctionBody::ResetObjectLiteralTypes()

{

this->objLiteralTypes = nullptr;

this->objLiteralCount = 0;

}

void FunctionBody::ResetLiteralRegexes()

{

literalRegexCount = 0;

this->literalRegexes = nullptr;

}

void FunctionBody::ResetProfileIds()

{

#if ENABLE\_PROFILE\_INFO

Assert(!HasDynamicProfileInfo()); // profile data relies on the profile ID counts; it should not have been created yet

Assert(!this->m\_codeGenRuntimeData); // relies on 'profiledCallSiteCount'

profiledCallSiteCount = 0;

profiledArrayCallSiteCount = 0;

profiledReturnTypeCount = 0;

profiledSlotCount = 0;

profiledLdElemCount = 0;

profiledStElemCount = 0;

#endif

}

void FunctionBody::ResetByteCodeGenState()

{

// Byte code generation failed for this function. Revert any intermediate state being tracked in the function body, in

// case byte code generation is attempted again for this function body.

ResetInlineCaches();

ResetObjectLiteralTypes();

ResetLiteralRegexes();

ResetLoops();

ResetProfileIds();

m\_firstTmpReg = Constants::NoRegister;

localClosureRegister = Constants::NoRegister;

localFrameDisplayRegister = Constants::NoRegister;

envRegister = Constants::NoRegister;

thisRegisterForEventHandler = Constants::NoRegister;

firstInnerScopeRegister = Constants::NoRegister;

funcExprScopeRegister = Constants::NoRegister;

innerScopeCount = 0;

hasCachedScopePropIds = false;

m\_constCount = 0;

this->m\_constTable = nullptr;

this->byteCodeBlock = nullptr;

// There is other state that is set by the byte code generator but the state should be the same each time byte code

// generation is done for the function, so it doesn't need to be reverted

}

void FunctionBody::ResetByteCodeGenVisitState()

{

// This function body is about to be visited by the byte code generator after defer-parsing it. Since the previous visit

// pass may have failed, we need to restore state that is tracked on the function body by the visit pass.

ResetLiteralRegexes();

}

#if ENABLE\_NATIVE\_CODEGEN

const FunctionCodeGenRuntimeData \*FunctionBody::GetInlineeCodeGenRuntimeData(const ProfileId profiledCallSiteId) const

{

Assert(profiledCallSiteId < profiledCallSiteCount);

return this->m\_codeGenRuntimeData ? this->m\_codeGenRuntimeData[profiledCallSiteId] : nullptr;

}

const FunctionCodeGenRuntimeData \*FunctionBody::GetInlineeCodeGenRuntimeDataForTargetInlinee(const ProfileId profiledCallSiteId, Js::FunctionBody \*inlineeFuncBody) const

{

Assert(profiledCallSiteId < profiledCallSiteCount);

if (!this->m\_codeGenRuntimeData)

{

return nullptr;

}

const FunctionCodeGenRuntimeData \*runtimeData = this->m\_codeGenRuntimeData[profiledCallSiteId];

while (runtimeData && runtimeData->GetFunctionBody() != inlineeFuncBody)

{

runtimeData = runtimeData->GetNext();

}

return runtimeData;

}

FunctionCodeGenRuntimeData \*FunctionBody::EnsureInlineeCodeGenRuntimeData(

Recycler \*const recycler,

\_\_in\_range(0, profiledCallSiteCount - 1) const ProfileId profiledCallSiteId,

FunctionBody \*const inlinee)

{

Assert(recycler);

Assert(profiledCallSiteId < profiledCallSiteCount);

Assert(inlinee);

if(!this->m\_codeGenRuntimeData)

{

const auto codeGenRuntimeData = RecyclerNewArrayZ(recycler, FunctionCodeGenRuntimeData \*, profiledCallSiteCount);

this->m\_codeGenRuntimeData = codeGenRuntimeData;

}

const auto inlineeData = this->m\_codeGenRuntimeData[profiledCallSiteId];

if(!inlineeData)

{

return this->m\_codeGenRuntimeData[profiledCallSiteId] = RecyclerNew(recycler, FunctionCodeGenRuntimeData, inlinee);

}

// Find the right code gen runtime data

FunctionCodeGenRuntimeData \*next = inlineeData;

while(next && (next->GetFunctionBody() != inlinee))

{

next = next->GetNext();

}

if (next)

{

return next;

}

FunctionCodeGenRuntimeData \*runtimeData = RecyclerNew(recycler, FunctionCodeGenRuntimeData, inlinee);

runtimeData->SetupRuntimeDataChain(inlineeData);

return this->m\_codeGenRuntimeData[profiledCallSiteId] = runtimeData;

}

const FunctionCodeGenRuntimeData \*FunctionBody::GetLdFldInlineeCodeGenRuntimeData(const uint inlineCacheIndex) const

{

Assert(inlineCacheIndex < inlineCacheCount);

return this->m\_codeGenGetSetRuntimeData ? this->m\_codeGenGetSetRuntimeData[inlineCacheIndex] : nullptr;

}

FunctionCodeGenRuntimeData \*FunctionBody::EnsureLdFldInlineeCodeGenRuntimeData(

Recycler \*const recycler,

\_\_in\_range(0, this->inlineCacheCount - 1) const uint inlineCacheIndex,

FunctionBody \*const inlinee)

{

Assert(recycler);

Assert(inlineCacheIndex < this->GetInlineCacheCount());

Assert(inlinee);

if(!this->m\_codeGenGetSetRuntimeData)

{

const auto codeGenRuntimeData = RecyclerNewArrayZ(recycler, FunctionCodeGenRuntimeData \*, this->GetInlineCacheCount());

this->m\_codeGenGetSetRuntimeData = codeGenRuntimeData;

}

const auto inlineeData = this->m\_codeGenGetSetRuntimeData[inlineCacheIndex];

if(inlineeData)

{

return inlineeData;

}

return this->m\_codeGenGetSetRuntimeData[inlineCacheIndex] = RecyclerNew(recycler, FunctionCodeGenRuntimeData, inlinee);

}

#endif

void FunctionBody::AllocateLoopHeaders()

{

Assert(this->loopHeaderArray == nullptr);

if (loopCount != 0)

{

this->loopHeaderArray = RecyclerNewArrayZ(this->m\_scriptContext->GetRecycler(), LoopHeader, loopCount);

for (uint i = 0; i < loopCount; i++)

{

this->loopHeaderArray[i].Init(this);

}

}

}

void FunctionBody::ReleaseLoopHeaders()

{

#if ENABLE\_NATIVE\_CODEGEN

this->MapLoopHeaders([](uint loopNumber, LoopHeader \* loopHeader)

{

loopHeader->ReleaseEntryPoints();

});

#endif

}

void FunctionBody::ResetLoops()

{

loopCount = 0;

this->loopHeaderArray = nullptr;

}

void FunctionBody::RestoreOldDefaultEntryPoint(FunctionEntryPointInfo\* oldEntryPointInfo,

JavascriptMethod oldOriginalEntryPoint,

FunctionEntryPointInfo\* newEntryPointInfo)

{

Assert(newEntryPointInfo);

this->SetDefaultFunctionEntryPointInfo(oldEntryPointInfo, oldOriginalEntryPoint);

this->entryPoints->RemoveAt(newEntryPointInfo->entryPointIndex);

}

FunctionEntryPointInfo\* FunctionBody::CreateNewDefaultEntryPoint()

{

Recycler \*const recycler = this->m\_scriptContext->GetRecycler();

const JavascriptMethod currentThunk = m\_scriptContext->CurrentThunk;

void\* validationCookie = nullptr;

#if ENABLE\_NATIVE\_CODEGEN

validationCookie = (void\*)m\_scriptContext->GetNativeCodeGenerator();

#endif

FunctionEntryPointInfo \*const entryPointInfo =

RecyclerNewFinalized(

recycler,

FunctionEntryPointInfo,

this,

currentThunk,

m\_scriptContext->GetThreadContext(),

validationCookie);

AddEntryPointToEntryPointList(entryPointInfo);

{

// Allocations in this region may trigger expiry and cause unexpected changes to state

AUTO\_NO\_EXCEPTION\_REGION;

FunctionEntryPointInfo \*const simpleJitEntryPointInfo = GetSimpleJitEntryPointInfo();

Js::JavascriptMethod originalEntryPoint, directEntryPoint;

if(simpleJitEntryPointInfo && GetExecutionMode() == ExecutionMode::FullJit)

{

directEntryPoint =

originalEntryPoint = reinterpret\_cast<Js::JavascriptMethod>(simpleJitEntryPointInfo->GetNativeAddress());

}

else

{

#if DYNAMIC\_INTERPRETER\_THUNK

// If the dynamic interpreter thunk hasn't been created yet, then the entry point can be set to

// the default entry point. Otherwise, since the new default entry point is being created to

// move back to the interpreter, the original entry point is going to be the dynamic interpreter thunk

originalEntryPoint =

m\_dynamicInterpreterThunk

? static\_cast<JavascriptMethod>(InterpreterThunkEmitter::ConvertToEntryPoint(m\_dynamicInterpreterThunk))

: DefaultEntryThunk;

#else

originalEntryPoint = DefaultEntryThunk;

#endif

directEntryPoint = currentThunk == DefaultEntryThunk ? originalEntryPoint : currentThunk;

}

entryPointInfo->address = directEntryPoint;

SetDefaultFunctionEntryPointInfo(entryPointInfo, originalEntryPoint);

}

return entryPointInfo;

}

LoopHeader \*FunctionBody::GetLoopHeader(uint index) const

{

Assert(this->loopHeaderArray != nullptr);

Assert(index < loopCount);

return &this->loopHeaderArray[index];

}

FunctionEntryPointInfo \*FunctionBody::GetSimpleJitEntryPointInfo() const

{

return simpleJitEntryPointInfo;

}

void FunctionBody::SetSimpleJitEntryPointInfo(FunctionEntryPointInfo \*const entryPointInfo)

{

simpleJitEntryPointInfo = entryPointInfo;

}

void FunctionBody::VerifyExecutionMode(const ExecutionMode executionMode) const

{

#if DBG

Assert(initializedExecutionModeAndLimits);

Assert(executionMode < ExecutionMode::Count);

switch(executionMode)

{

case ExecutionMode::Interpreter:

Assert(!DoInterpreterProfile());

break;

case ExecutionMode::AutoProfilingInterpreter:

Assert(DoInterpreterProfile());

Assert(DoInterpreterAutoProfile());

break;

case ExecutionMode::ProfilingInterpreter:

Assert(DoInterpreterProfile());

break;

case ExecutionMode::SimpleJit:

Assert(DoSimpleJit());

break;

case ExecutionMode::FullJit:

Assert(DoFullJit());

break;

default:

Assert(false);

\_\_assume(false);

}

#endif

}

ExecutionMode FunctionBody::GetDefaultInterpreterExecutionMode() const

{

if(!DoInterpreterProfile())

{

VerifyExecutionMode(ExecutionMode::Interpreter);

return ExecutionMode::Interpreter;

}

if(DoInterpreterAutoProfile())

{

VerifyExecutionMode(ExecutionMode::AutoProfilingInterpreter);

return ExecutionMode::AutoProfilingInterpreter;

}

VerifyExecutionMode(ExecutionMode::ProfilingInterpreter);

return ExecutionMode::ProfilingInterpreter;

}

ExecutionMode FunctionBody::GetExecutionMode() const

{

VerifyExecutionMode(executionMode);

return executionMode;

}

ExecutionMode FunctionBody::GetInterpreterExecutionMode(const bool isPostBailout)

{

Assert(initializedExecutionModeAndLimits);

if(isPostBailout && DoInterpreterProfile())

{

return ExecutionMode::ProfilingInterpreter;

}

switch(GetExecutionMode())

{

case ExecutionMode::Interpreter:

case ExecutionMode::AutoProfilingInterpreter:

case ExecutionMode::ProfilingInterpreter:

return GetExecutionMode();

case ExecutionMode::SimpleJit:

if(IsNewSimpleJit())

{

return GetDefaultInterpreterExecutionMode();

}

// fall through

case ExecutionMode::FullJit:

{

const ExecutionMode executionMode =

DoInterpreterProfile() ? ExecutionMode::ProfilingInterpreter : ExecutionMode::Interpreter;

VerifyExecutionMode(executionMode);

return executionMode;

}

default:

Assert(false);

\_\_assume(false);

}

}

void FunctionBody::SetExecutionMode(const ExecutionMode executionMode)

{

VerifyExecutionMode(executionMode);

this->executionMode = executionMode;

}

bool FunctionBody::IsInterpreterExecutionMode() const

{

return GetExecutionMode() <= ExecutionMode::ProfilingInterpreter;

}

bool FunctionBody::TryTransitionToNextExecutionMode()

{

Assert(initializedExecutionModeAndLimits);

switch(GetExecutionMode())

{

case ExecutionMode::Interpreter:

if(interpretedCount < interpreterLimit)

{

VerifyExecutionMode(GetExecutionMode());

return false;

}

CommitExecutedIterations(interpreterLimit, interpreterLimit);

goto TransitionToFullJit;

TransitionToAutoProfilingInterpreter:

if(autoProfilingInterpreter0Limit != 0 || autoProfilingInterpreter1Limit != 0)

{

SetExecutionMode(ExecutionMode::AutoProfilingInterpreter);

interpretedCount = 0;

return true;

}

goto TransitionFromAutoProfilingInterpreter;

case ExecutionMode::AutoProfilingInterpreter:

{

uint16 &autoProfilingInterpreterLimit =

autoProfilingInterpreter0Limit == 0 && profilingInterpreter0Limit == 0

? autoProfilingInterpreter1Limit

: autoProfilingInterpreter0Limit;

if(interpretedCount < autoProfilingInterpreterLimit)

{

VerifyExecutionMode(GetExecutionMode());

return false;

}

CommitExecutedIterations(autoProfilingInterpreterLimit, autoProfilingInterpreterLimit);

// fall through

}

TransitionFromAutoProfilingInterpreter:

Assert(autoProfilingInterpreter0Limit == 0 || autoProfilingInterpreter1Limit == 0);

if(profilingInterpreter0Limit == 0 && autoProfilingInterpreter1Limit == 0)

{

goto TransitionToSimpleJit;

}

// fall through

TransitionToProfilingInterpreter:

if(profilingInterpreter0Limit != 0 || profilingInterpreter1Limit != 0)

{

SetExecutionMode(ExecutionMode::ProfilingInterpreter);

interpretedCount = 0;

return true;

}

goto TransitionFromProfilingInterpreter;

case ExecutionMode::ProfilingInterpreter:

{

uint16 &profilingInterpreterLimit =

profilingInterpreter0Limit == 0 && autoProfilingInterpreter1Limit == 0 && simpleJitLimit == 0

? profilingInterpreter1Limit

: profilingInterpreter0Limit;

if(interpretedCount < profilingInterpreterLimit)

{

VerifyExecutionMode(GetExecutionMode());

return false;

}

CommitExecutedIterations(profilingInterpreterLimit, profilingInterpreterLimit);

// fall through

}

TransitionFromProfilingInterpreter:

Assert(profilingInterpreter0Limit == 0 || profilingInterpreter1Limit == 0);

if(autoProfilingInterpreter1Limit == 0 && simpleJitLimit == 0 && profilingInterpreter1Limit == 0)

{

goto TransitionToFullJit;

}

goto TransitionToAutoProfilingInterpreter;

TransitionToSimpleJit:

if(simpleJitLimit != 0)

{

SetExecutionMode(ExecutionMode::SimpleJit);

// Zero the interpreted count here too, so that we can determine how many interpreter iterations ran

// while waiting for simple JIT

interpretedCount = 0;

return true;

}

goto TransitionToProfilingInterpreter;

case ExecutionMode::SimpleJit:

{

FunctionEntryPointInfo \*const simpleJitEntryPointInfo = GetSimpleJitEntryPointInfo();

if(!simpleJitEntryPointInfo || simpleJitEntryPointInfo->callsCount != 0)

{

VerifyExecutionMode(GetExecutionMode());

return false;

}

CommitExecutedIterations(simpleJitLimit, simpleJitLimit);

goto TransitionToProfilingInterpreter;

}

TransitionToFullJit:

if(DoFullJit())

{

SetExecutionMode(ExecutionMode::FullJit);

return true;

}

// fall through

case ExecutionMode::FullJit:

VerifyExecutionMode(GetExecutionMode());

return false;

default:

Assert(false);

\_\_assume(false);

}

}

void FunctionBody::TryTransitionToNextInterpreterExecutionMode()

{

Assert(IsInterpreterExecutionMode());

TryTransitionToNextExecutionMode();

SetExecutionMode(GetInterpreterExecutionMode(false));

}

void FunctionBody::SetIsSpeculativeJitCandidate()

{

// This function is a candidate for speculative JIT. Ensure that it is profiled immediately by transitioning out of the

// auto-profiling interpreter mode.

if(GetExecutionMode() != ExecutionMode::AutoProfilingInterpreter || GetProfiledIterations() != 0)

{

return;

}

TraceExecutionMode("IsSpeculativeJitCandidate (before)");

if(autoProfilingInterpreter0Limit != 0)

{

(profilingInterpreter0Limit == 0 ? profilingInterpreter0Limit : autoProfilingInterpreter1Limit) +=

autoProfilingInterpreter0Limit;

autoProfilingInterpreter0Limit = 0;

}

else if(profilingInterpreter0Limit == 0)

{

profilingInterpreter0Limit += autoProfilingInterpreter1Limit;

autoProfilingInterpreter1Limit = 0;

}

TraceExecutionMode("IsSpeculativeJitCandidate");

TryTransitionToNextInterpreterExecutionMode();

}

bool FunctionBody::TryTransitionToJitExecutionMode()

{

const ExecutionMode previousExecutionMode = GetExecutionMode();

TryTransitionToNextExecutionMode();

switch(GetExecutionMode())

{

case ExecutionMode::SimpleJit:

break;

case ExecutionMode::FullJit:

if(fullJitRequeueThreshold == 0)

{

break;

}

--fullJitRequeueThreshold;

return false;

default:

return false;

}

if(GetExecutionMode() != previousExecutionMode)

{

TraceExecutionMode();

}

return true;

}

void FunctionBody::TransitionToSimpleJitExecutionMode()

{

CommitExecutedIterations();

interpreterLimit = 0;

autoProfilingInterpreter0Limit = 0;

profilingInterpreter0Limit = 0;

autoProfilingInterpreter1Limit = 0;

fullJitThreshold = simpleJitLimit + profilingInterpreter1Limit;

VerifyExecutionModeLimits();

SetExecutionMode(ExecutionMode::SimpleJit);

}

void FunctionBody::TransitionToFullJitExecutionMode()

{

CommitExecutedIterations();

interpreterLimit = 0;

autoProfilingInterpreter0Limit = 0;

profilingInterpreter0Limit = 0;

autoProfilingInterpreter1Limit = 0;

simpleJitLimit = 0;

profilingInterpreter1Limit = 0;

fullJitThreshold = 0;

VerifyExecutionModeLimits();

SetExecutionMode(ExecutionMode::FullJit);

}

void FunctionBody::VerifyExecutionModeLimits()

{

Assert(initializedExecutionModeAndLimits);

Assert(

(

interpreterLimit +

autoProfilingInterpreter0Limit +

profilingInterpreter0Limit +

autoProfilingInterpreter1Limit +

simpleJitLimit +

profilingInterpreter1Limit

) == fullJitThreshold);

}

void FunctionBody::InitializeExecutionModeAndLimits()

{

DebugOnly(initializedExecutionModeAndLimits = true);

const ConfigFlagsTable &configFlags = Configuration::Global.flags;

interpreterLimit = 0;

autoProfilingInterpreter0Limit = static\_cast<uint16>(configFlags.AutoProfilingInterpreter0Limit);

profilingInterpreter0Limit = static\_cast<uint16>(configFlags.ProfilingInterpreter0Limit);

autoProfilingInterpreter1Limit = static\_cast<uint16>(configFlags.AutoProfilingInterpreter1Limit);

simpleJitLimit = static\_cast<uint16>(configFlags.SimpleJitLimit);

profilingInterpreter1Limit = static\_cast<uint16>(configFlags.ProfilingInterpreter1Limit);

// Based on which execution modes are disabled, calculate the number of additional iterations that need to be covered by

// the execution mode that will scale with the full JIT threshold

uint16 scale = 0;

const bool doInterpreterProfile = DoInterpreterProfile();

if(!doInterpreterProfile)

{

scale +=

autoProfilingInterpreter0Limit +

profilingInterpreter0Limit +

autoProfilingInterpreter1Limit +

profilingInterpreter1Limit;

autoProfilingInterpreter0Limit = 0;

profilingInterpreter0Limit = 0;

autoProfilingInterpreter1Limit = 0;

profilingInterpreter1Limit = 0;

}

else if(!DoInterpreterAutoProfile())

{

scale += autoProfilingInterpreter0Limit + autoProfilingInterpreter1Limit;

autoProfilingInterpreter0Limit = 0;

autoProfilingInterpreter1Limit = 0;

if(!IsNewSimpleJit())

{

simpleJitLimit += profilingInterpreter0Limit;

profilingInterpreter0Limit = 0;

}

}

if(!DoSimpleJit())

{

if(!IsNewSimpleJit() && doInterpreterProfile)

{

// The old simple JIT is off, but since it does profiling, it will be replaced with the profiling interpreter

profilingInterpreter1Limit += simpleJitLimit;

}

else

{

scale += simpleJitLimit;

}

simpleJitLimit = 0;

}

if(!DoFullJit())

{

scale += profilingInterpreter1Limit;

profilingInterpreter1Limit = 0;

}

uint16 fullJitThreshold =

static\_cast<uint16>(

configFlags.AutoProfilingInterpreter0Limit +

configFlags.ProfilingInterpreter0Limit +

configFlags.AutoProfilingInterpreter1Limit +

configFlags.SimpleJitLimit +

configFlags.ProfilingInterpreter1Limit);

if(!configFlags.EnforceExecutionModeLimits)

{

/\*

Scale the full JIT threshold based on some heuristics:

- If the % of code in loops is > 50, scale by 1

- Byte-code size of code outside loops

- If the size is < 50, scale by 1.2

- If the size is < 100, scale by 1.4

- If the size is >= 100, scale by 1.6

\*/

const uint loopPercentage = GetByteCodeInLoopCount() \* 100 / max(1u, GetByteCodeCount());

if(loopPercentage <= 50)

{

const uint straightLineSize = GetByteCodeCount() - GetByteCodeInLoopCount();

double fullJitDelayMultiplier;

if(straightLineSize < 50)

{

fullJitDelayMultiplier = 1.2;

}

else if(straightLineSize < 100)

{

fullJitDelayMultiplier = 1.4;

}

else

{

fullJitDelayMultiplier = 1.6;

}

const uint16 newFullJitThreshold = static\_cast<uint16>(fullJitThreshold \* fullJitDelayMultiplier);

scale += newFullJitThreshold - fullJitThreshold;

fullJitThreshold = newFullJitThreshold;

}

}

Assert(fullJitThreshold >= scale);

this->fullJitThreshold = fullJitThreshold - scale;

interpretedCount = 0;

SetExecutionMode(GetDefaultInterpreterExecutionMode());

SetFullJitThreshold(fullJitThreshold);

TryTransitionToNextInterpreterExecutionMode();

}

void FunctionBody::ReinitializeExecutionModeAndLimits()

{

wasCalledFromLoop = false;

fullJitRequeueThreshold = 0;

committedProfiledIterations = 0;

InitializeExecutionModeAndLimits();

}

void FunctionBody::SetFullJitThreshold(const uint16 newFullJitThreshold, const bool skipSimpleJit)

{

Assert(initializedExecutionModeAndLimits);

Assert(GetExecutionMode() != ExecutionMode::FullJit);

int scale = newFullJitThreshold - fullJitThreshold;

if(scale == 0)

{

VerifyExecutionModeLimits();

return;

}

fullJitThreshold = newFullJitThreshold;

const auto ScaleLimit = [&](uint16 &limit) -> bool

{

Assert(scale != 0);

const int limitScale = max(-static\_cast<int>(limit), scale);

const int newLimit = limit + limitScale;

Assert(static\_cast<int>(static\_cast<uint16>(newLimit)) == newLimit);

limit = static\_cast<uint16>(newLimit);

scale -= limitScale;

Assert(limit == 0 || scale == 0);

if(&limit == simpleJitLimit.AddressOf())

{

FunctionEntryPointInfo \*const simpleJitEntryPointInfo = GetSimpleJitEntryPointInfo();

if(GetDefaultFunctionEntryPointInfo() == simpleJitEntryPointInfo)

{

Assert(GetExecutionMode() == ExecutionMode::SimpleJit);

const int newSimpleJitCallCount = max(0, simpleJitEntryPointInfo->callsCount + limitScale);

Assert(static\_cast<int>(static\_cast<uint16>(newSimpleJitCallCount)) == newSimpleJitCallCount);

SetSimpleJitCallCount(static\_cast<uint16>(newSimpleJitCallCount));

}

}

return scale == 0;

};

/\*

Determine which execution mode's limit scales with the full JIT threshold, in order of preference:

- New simple JIT

- Auto-profiling interpreter 1

- Auto-profiling interpreter 0

- Interpreter

- Profiling interpreter 0 (when using old simple JIT)

- Old simple JIT

- Profiling interpreter 1

- Profiling interpreter 0 (when using new simple JIT)

\*/

const bool doSimpleJit = DoSimpleJit();

const bool doInterpreterProfile = DoInterpreterProfile();

const bool fullyScaled =

IsNewSimpleJit() && doSimpleJit && ScaleLimit(simpleJitLimit) ||

(

doInterpreterProfile

? DoInterpreterAutoProfile() &&

(ScaleLimit(autoProfilingInterpreter1Limit) || ScaleLimit(autoProfilingInterpreter0Limit))

: ScaleLimit(interpreterLimit)

) ||

(

IsNewSimpleJit()

? doInterpreterProfile &&

(ScaleLimit(profilingInterpreter1Limit) || ScaleLimit(profilingInterpreter0Limit))

: doInterpreterProfile && ScaleLimit(profilingInterpreter0Limit) ||

doSimpleJit && ScaleLimit(simpleJitLimit) ||

doInterpreterProfile && ScaleLimit(profilingInterpreter1Limit)

);

Assert(fullyScaled);

Assert(scale == 0);

if(GetExecutionMode() != ExecutionMode::SimpleJit)

{

Assert(IsInterpreterExecutionMode());

if(simpleJitLimit != 0 &&

(skipSimpleJit || simpleJitLimit < DEFAULT\_CONFIG\_MinSimpleJitIterations) &&

!PHASE\_FORCE(Phase::SimpleJitPhase, this))

{

// Simple JIT code has not yet been generated, and was either requested to be skipped, or the limit was scaled

// down too much. Skip simple JIT by moving any remaining iterations to an equivalent interpreter execution

// mode.

(IsNewSimpleJit() ? autoProfilingInterpreter1Limit : profilingInterpreter1Limit) += simpleJitLimit;

simpleJitLimit = 0;

TryTransitionToNextInterpreterExecutionMode();

}

}

VerifyExecutionModeLimits();

}

void FunctionBody::CommitExecutedIterations()

{

Assert(initializedExecutionModeAndLimits);

switch(GetExecutionMode())

{

case ExecutionMode::Interpreter:

CommitExecutedIterations(interpreterLimit, interpretedCount);

break;

case ExecutionMode::AutoProfilingInterpreter:

CommitExecutedIterations(

autoProfilingInterpreter0Limit == 0 && profilingInterpreter0Limit == 0

? autoProfilingInterpreter1Limit

: autoProfilingInterpreter0Limit,

interpretedCount);

break;

case ExecutionMode::ProfilingInterpreter:

CommitExecutedIterations(

GetSimpleJitEntryPointInfo()

? profilingInterpreter1Limit

: profilingInterpreter0Limit,

interpretedCount);

break;

case ExecutionMode::SimpleJit:

CommitExecutedIterations(simpleJitLimit, GetSimpleJitExecutedIterations());

break;

case ExecutionMode::FullJit:

break;

default:

Assert(false);

\_\_assume(false);

}

}

void FunctionBody::CommitExecutedIterations(uint16 &limit, const uint executedIterations)

{

Assert(initializedExecutionModeAndLimits);

Assert(

&limit == interpreterLimit.AddressOf() ||

&limit == autoProfilingInterpreter0Limit.AddressOf() ||

&limit == profilingInterpreter0Limit.AddressOf() ||

&limit == autoProfilingInterpreter1Limit.AddressOf() ||

&limit == simpleJitLimit.AddressOf() ||

&limit == profilingInterpreter1Limit.AddressOf());

const uint16 clampedExecutedIterations = executedIterations >= limit ? limit : static\_cast<uint16>(executedIterations);

Assert(fullJitThreshold >= clampedExecutedIterations);

fullJitThreshold -= clampedExecutedIterations;

limit -= clampedExecutedIterations;

VerifyExecutionModeLimits();

if(&limit == profilingInterpreter0Limit.AddressOf() ||

!IsNewSimpleJit() && &limit == simpleJitLimit.AddressOf() ||

&limit == profilingInterpreter1Limit.AddressOf())

{

const uint16 newCommittedProfiledIterations = committedProfiledIterations + clampedExecutedIterations;

committedProfiledIterations =

newCommittedProfiledIterations >= committedProfiledIterations ? newCommittedProfiledIterations : MAXUINT16;

}

}

uint16 FunctionBody::GetSimpleJitExecutedIterations() const

{

Assert(initializedExecutionModeAndLimits);

Assert(GetExecutionMode() == ExecutionMode::SimpleJit);

FunctionEntryPointInfo \*const simpleJitEntryPointInfo = GetSimpleJitEntryPointInfo();

if(!simpleJitEntryPointInfo)

{

return 0;

}

// Simple JIT counts down and transitions on overflow

const uint8 callCount = simpleJitEntryPointInfo->callsCount;

Assert(simpleJitLimit == 0 ? callCount == 0 : simpleJitLimit > callCount);

return callCount == 0 ? simpleJitLimit : simpleJitLimit - callCount - 1;

}

void FunctionBody::ResetSimpleJitLimitAndCallCount()

{

Assert(initializedExecutionModeAndLimits);

Assert(GetExecutionMode() == ExecutionMode::SimpleJit);

Assert(GetDefaultFunctionEntryPointInfo() == GetSimpleJitEntryPointInfo());

const uint16 simpleJitNewLimit = static\_cast<uint8>(Configuration::Global.flags.SimpleJitLimit);

Assert(simpleJitNewLimit == Configuration::Global.flags.SimpleJitLimit);

if(simpleJitLimit < simpleJitNewLimit)

{

fullJitThreshold += simpleJitNewLimit - simpleJitLimit;

simpleJitLimit = simpleJitNewLimit;

}

interpretedCount = 0;

ResetSimpleJitCallCount();

}

void FunctionBody::SetSimpleJitCallCount(const uint16 simpleJitLimit) const

{

Assert(GetExecutionMode() == ExecutionMode::SimpleJit);

Assert(GetDefaultFunctionEntryPointInfo() == GetSimpleJitEntryPointInfo());

// Simple JIT counts down and transitions on overflow

const uint8 limit = static\_cast<uint8>(min(0xffui16, simpleJitLimit));

GetSimpleJitEntryPointInfo()->callsCount = limit == 0 ? 0 : limit - 1;

}

void FunctionBody::ResetSimpleJitCallCount()

{

SetSimpleJitCallCount(

simpleJitLimit > interpretedCount

? simpleJitLimit - static\_cast<uint16>(interpretedCount)

: 0ui16);

}

uint16 FunctionBody::GetProfiledIterations() const

{

Assert(initializedExecutionModeAndLimits);

uint16 profiledIterations = committedProfiledIterations;

switch(GetExecutionMode())

{

case ExecutionMode::ProfilingInterpreter:

{

const uint16 clampedInterpretedCount =

interpretedCount <= MAXUINT16

? static\_cast<uint16>(interpretedCount)

: MAXUINT16;

const uint16 newProfiledIterations = profiledIterations + clampedInterpretedCount;

profiledIterations = newProfiledIterations >= profiledIterations ? newProfiledIterations : MAXUINT16;

break;

}

case ExecutionMode::SimpleJit:

if(!IsNewSimpleJit())

{

const uint16 newProfiledIterations = profiledIterations + GetSimpleJitExecutedIterations();

profiledIterations = newProfiledIterations >= profiledIterations ? newProfiledIterations : MAXUINT16;

}

break;

}

return profiledIterations;

}

void FunctionBody::OnFullJitDequeued(const FunctionEntryPointInfo \*const entryPointInfo)

{

Assert(initializedExecutionModeAndLimits);

Assert(GetExecutionMode() == ExecutionMode::FullJit);

Assert(entryPointInfo);

if(entryPointInfo != GetDefaultFunctionEntryPointInfo())

{

return;

}

// Re-queue the full JIT work item after this many iterations

fullJitRequeueThreshold = static\_cast<uint16>(DEFAULT\_CONFIG\_FullJitRequeueThreshold);

}

void FunctionBody::TraceExecutionMode(const char \*const eventDescription) const

{

Assert(initializedExecutionModeAndLimits);

if(PHASE\_TRACE(Phase::ExecutionModePhase, this))

{

DoTraceExecutionMode(eventDescription);

}

}

void FunctionBody::TraceInterpreterExecutionMode() const

{

Assert(initializedExecutionModeAndLimits);

if(!PHASE\_TRACE(Phase::ExecutionModePhase, this))

{

return;

}

switch(GetExecutionMode())

{

case ExecutionMode::Interpreter:

case ExecutionMode::AutoProfilingInterpreter:

case ExecutionMode::ProfilingInterpreter:

DoTraceExecutionMode(nullptr);

break;

}

}

void FunctionBody::DoTraceExecutionMode(const char \*const eventDescription) const

{

Assert(PHASE\_TRACE(Phase::ExecutionModePhase, this));

Assert(initializedExecutionModeAndLimits);

wchar\_t functionIdString[MAX\_FUNCTION\_BODY\_DEBUG\_STRING\_SIZE];

Output::Print(

L"ExecutionMode - "

L"function: %s (%s), "

L"mode: %S, "

L"size: %u, "

L"limits: %hu.%hu.%hu.%hu.%hu = %hu",

GetDisplayName(),

GetDebugNumberSet(functionIdString),

ExecutionModeName(executionMode),

GetByteCodeCount(),

interpreterLimit + autoProfilingInterpreter0Limit,

profilingInterpreter0Limit,

autoProfilingInterpreter1Limit,

simpleJitLimit,

profilingInterpreter1Limit,

fullJitThreshold);

if(eventDescription)

{

Output::Print(L", event: %S", eventDescription);

}

Output::Print(L"\n");

Output::Flush();

}

bool FunctionBody::IsNewSimpleJit()

{

return CONFIG\_FLAG(NewSimpleJit);

}

bool FunctionBody::DoSimpleJit() const

{

return

!PHASE\_OFF(Js::SimpleJitPhase, this) &&

!GetScriptContext()->GetConfig()->IsNoNative() &&

!GetScriptContext()->IsInDebugMode() &&

DoInterpreterProfile() &&

(!IsNewSimpleJit() || DoInterpreterAutoProfile()) &&

!IsGenerator(); // Generator JIT requires bailout which SimpleJit cannot do since it skips GlobOpt

}

bool FunctionBody::DoSimpleJitDynamicProfile() const

{

Assert(DoSimpleJit());

return !PHASE\_OFF(Js::SimpleJitDynamicProfilePhase, this) && !IsNewSimpleJit();

}

bool FunctionBody::DoInterpreterProfile() const

{

#if ENABLE\_PROFILE\_INFO

// Switch off profiling is asmJsFunction

if (this->GetIsAsmJsFunction())

{

return false;

}

else

{

return !PHASE\_OFF(InterpreterProfilePhase, this) && DynamicProfileInfo::IsEnabled(this);

}

#else

return false;

#endif

}

bool FunctionBody::DoInterpreterAutoProfile() const

{

Assert(DoInterpreterProfile());

return !PHASE\_OFF(InterpreterAutoProfilePhase, this) && !GetScriptContext()->IsInDebugMode();

}

bool FunctionBody::WasCalledFromLoop() const

{

return wasCalledFromLoop;

}

void FunctionBody::SetWasCalledFromLoop()

{

if(wasCalledFromLoop)

{

return;

}

wasCalledFromLoop = true;

if(Configuration::Global.flags.EnforceExecutionModeLimits)

{

if(PHASE\_TRACE(Phase::ExecutionModePhase, this))

{

CommitExecutedIterations();

TraceExecutionMode("WasCalledFromLoop (before)");

}

}

else

{

// This function is likely going to be called frequently since it's called from a loop. Reduce the full JIT

// threshold to realize the full JIT perf benefit sooner.

CommitExecutedIterations();

TraceExecutionMode("WasCalledFromLoop (before)");

if(fullJitThreshold > 1)

{

SetFullJitThreshold(fullJitThreshold / 2, !IsNewSimpleJit());

}

}

{

// Reduce the loop interpreter limit too, for the same reasons as above

const uint oldLoopInterpreterLimit = loopInterpreterLimit;

const uint newLoopInterpreterLimit = GetReducedLoopInterpretCount();

Assert(newLoopInterpreterLimit <= oldLoopInterpreterLimit);

loopInterpreterLimit = newLoopInterpreterLimit;

// Adjust loop headers' interpret counts to ensure that loops will still be profiled a number of times before

// loop bodies are jitted

const uint oldLoopProfileThreshold = GetLoopProfileThreshold(oldLoopInterpreterLimit);

const uint newLoopProfileThreshold = GetLoopProfileThreshold(newLoopInterpreterLimit);

MapLoopHeaders([=](const uint index, LoopHeader \*const loopHeader)

{

const uint interpretedCount = loopHeader->interpretCount;

if(interpretedCount <= newLoopProfileThreshold || interpretedCount >= oldLoopInterpreterLimit)

{

// The loop hasn't been profiled yet and wouldn't have started profiling even with the new profile

// threshold, or it has already been profiled the necessary minimum number of times based on the old limit

return;

}

if(interpretedCount <= oldLoopProfileThreshold)

{

// The loop hasn't been profiled yet, but would have started profiling with the new profile threshold. Start

// profiling on the next iteration.

loopHeader->interpretCount = newLoopProfileThreshold;

return;

}

// The loop has been profiled some already. Preserve the number of profiled iterations.

loopHeader->interpretCount = newLoopProfileThreshold + (interpretedCount - oldLoopProfileThreshold);

});

}

TraceExecutionMode("WasCalledFromLoop");

}

bool FunctionBody::RecentlyBailedOutOfJittedLoopBody() const

{

return recentlyBailedOutOfJittedLoopBody;

}

void FunctionBody::SetRecentlyBailedOutOfJittedLoopBody(const bool value)

{

recentlyBailedOutOfJittedLoopBody = value;

}

uint16 FunctionBody::GetMinProfileIterations()

{

return

static\_cast<uint>(

IsNewSimpleJit()

? DEFAULT\_CONFIG\_MinProfileIterations

: DEFAULT\_CONFIG\_MinProfileIterations\_OldSimpleJit);

}

uint16 FunctionBody::GetMinFunctionProfileIterations()

{

return GetMinProfileIterations();

}

uint FunctionBody::GetMinLoopProfileIterations(const uint loopInterpreterLimit)

{

return min(static\_cast<uint>(GetMinProfileIterations()), loopInterpreterLimit);

}

uint FunctionBody::GetLoopProfileThreshold(const uint loopInterpreterLimit) const

{

return

DoInterpreterProfile()

? DoInterpreterAutoProfile()

? loopInterpreterLimit - GetMinLoopProfileIterations(loopInterpreterLimit)

: 0

: static\_cast<uint>(-1);

}

uint FunctionBody::GetReducedLoopInterpretCount()

{

const uint loopInterpretCount = CONFIG\_FLAG(LoopInterpretCount);

if(CONFIG\_ISENABLED(LoopInterpretCountFlag))

{

return loopInterpretCount;

}

return max(loopInterpretCount / 3, GetMinLoopProfileIterations(loopInterpretCount));

}

uint FunctionBody::GetLoopInterpretCount(LoopHeader\* loopHeader) const

{

if(loopHeader->isNested)

{

Assert(loopInterpreterLimit >= GetReducedLoopInterpretCount());

return GetReducedLoopInterpretCount();

}

return loopInterpreterLimit;

}

bool FunctionBody::DoObjectHeaderInlining()

{

return !PHASE\_OFF1(ObjectHeaderInliningPhase);

}

bool FunctionBody::DoObjectHeaderInliningForConstructors()

{

return !PHASE\_OFF1(ObjectHeaderInliningForConstructorsPhase) && DoObjectHeaderInlining();

}

bool FunctionBody::DoObjectHeaderInliningForConstructor(const uint32 inlineSlotCapacity)

{

return inlineSlotCapacity == 0 ? DoObjectHeaderInliningForEmptyObjects() : DoObjectHeaderInliningForConstructors();

}

bool FunctionBody::DoObjectHeaderInliningForObjectLiterals()

{

return !PHASE\_OFF1(ObjectHeaderInliningForObjectLiteralsPhase) && DoObjectHeaderInlining();

}

bool FunctionBody::DoObjectHeaderInliningForObjectLiteral(const uint32 inlineSlotCapacity)

{

return

inlineSlotCapacity == 0

? DoObjectHeaderInliningForEmptyObjects()

: DoObjectHeaderInliningForObjectLiterals() &&

inlineSlotCapacity <= static\_cast<uint32>(MaxPreInitializedObjectHeaderInlinedTypeInlineSlotCount);

}

bool FunctionBody::DoObjectHeaderInliningForObjectLiteral(

const PropertyIdArray \*const propIds,

ScriptContext \*const scriptContext)

{

Assert(propIds);

Assert(scriptContext);

return

DoObjectHeaderInliningForObjectLiteral(propIds->count) &&

PathTypeHandlerBase::UsePathTypeHandlerForObjectLiteral(propIds, scriptContext);

}

bool FunctionBody::DoObjectHeaderInliningForEmptyObjects()

{

#pragma prefast(suppress:6237, "(<zero> && <expression>) is always zero. <expression> is never evaluated and might have side effects.")

return PHASE\_ON1(ObjectHeaderInliningForEmptyObjectsPhase) && DoObjectHeaderInlining();

}

void FunctionBody::Finalize(bool isShutdown)

{

#if ENABLE\_DEBUG\_CONFIG\_OPTIONS

if (Js::Configuration::Global.flags.Instrument.IsEnabled(Js::LinearScanPhase, this->GetSourceContextId(), this->GetLocalFunctionId()))

{

this->DumpRegStats(this);

}

#endif

this->Cleanup(isShutdown);

this->CleanupSourceInfo(isShutdown);

this->ClearNestedFunctionParentFunctionReference();

this->CleanupFunctionProxyCounters();

}

void FunctionBody::CleanupSourceInfo(bool isScriptContextClosing)

{

Assert(this->cleanedUp);

if (!sourceInfoCleanedUp)

{

if (GetIsFuncRegistered() && !isScriptContextClosing)

{

// If our function is registered, then there must

// be a Utf8SourceInfo pinned by it.

Assert(this->m\_utf8SourceInfo);

this->m\_utf8SourceInfo->RemoveFunctionBody(this);

}

if (this->m\_sourceInfo.pSpanSequence != nullptr)

{

HeapDelete(this->m\_sourceInfo.pSpanSequence);

this->m\_sourceInfo.pSpanSequence = nullptr;

}

sourceInfoCleanedUp = true;

}

}

template<bool IsScriptContextShutdown>

void FunctionBody::CleanUpInlineCaches()

{

uint unregisteredInlineCacheCount = 0;

if (nullptr != this->inlineCaches)

{

// Inline caches are in this order

// plain inline cache

// root object load inline cache

// root object store inline cache

// isInst inline cache

// The inlineCacheCount includes all but isInst inline cache

uint i = 0;

uint plainInlineCacheEnd = GetRootObjectLoadInlineCacheStart();

for (; i < plainInlineCacheEnd; i++)

{

if (nullptr != this->inlineCaches[i])

{

InlineCache\* inlineCache = (InlineCache\*)this->inlineCaches[i];

if (IsScriptContextShutdown)

{

memset(inlineCache, 0, sizeof(InlineCache));

}

else

{

if (inlineCache->RemoveFromInvalidationList())

{

unregisteredInlineCacheCount++;

}

AllocatorDelete(InlineCacheAllocator, this->m\_scriptContext->GetInlineCacheAllocator(), inlineCache);

}

}

}

RootObjectBase \* rootObjectBase = this->GetRootObject();

uint rootObjectLoadInlineCacheEnd = GetRootObjectLoadMethodInlineCacheStart();

for (; i < rootObjectLoadInlineCacheEnd; i++)

{

if (nullptr != this->inlineCaches[i])

{

InlineCache\* inlineCache = (InlineCache\*)this->inlineCaches[i];

if (IsScriptContextShutdown)

{

memset(inlineCache, 0, sizeof(InlineCache));

}

else

{

// A single root object inline caches for a given property is shared by all functions. It is ref counted

// and doesn't get released to the allocator until there are no more outstanding references. Thus we don't need

// to (and, in fact, cannot) remove it from the invalidation list here. Instead, we'll do it in ReleaseInlineCache

// when there are no more outstanding references.

rootObjectBase->ReleaseInlineCache(this->GetPropertyIdFromCacheId(i), false, false, IsScriptContextShutdown);

}

}

}

uint rootObjectLoadMethodInlineCacheEnd = GetRootObjectStoreInlineCacheStart();

for (; i < rootObjectLoadMethodInlineCacheEnd; i++)

{

if (nullptr != this->inlineCaches[i])

{

InlineCache\* inlineCache = (InlineCache\*)this->inlineCaches[i];

if (IsScriptContextShutdown)

{

memset(inlineCache, 0, sizeof(InlineCache));

}

else

{

// A single root object inline caches for a given property is shared by all functions. It is ref counted

// and doesn't get released to the allocator until there are no more outstanding references. Thus we don't need

// to (and, in fact, cannot) remove it from the invalidation list here. Instead, we'll do it in ReleaseInlineCache

// when there are no more outstanding references.

rootObjectBase->ReleaseInlineCache(this->GetPropertyIdFromCacheId(i), true, false, IsScriptContextShutdown);

}

}

}

uint rootObjectStoreInlineCacheEnd = this->GetInlineCacheCount();

for (; i < rootObjectStoreInlineCacheEnd; i++)

{

if (nullptr != this->inlineCaches[i])

{

InlineCache\* inlineCache = (InlineCache\*)this->inlineCaches[i];

if (IsScriptContextShutdown)

{

memset(inlineCache, 0, sizeof(InlineCache));

}

else

{

// A single root object inline caches for a given property is shared by all functions. It is ref counted

// and doesn't get released to the allocator until there are no more outstanding references. Thus we don't need

// to (and, in fact, cannot) remove it from the invalidation list here. Instead, we'll do it in ReleaseInlineCache

// when there are no more outstanding references.

rootObjectBase->ReleaseInlineCache(this->GetPropertyIdFromCacheId(i), false, true, IsScriptContextShutdown);

}

}

}

uint totalCacheCount = inlineCacheCount + GetIsInstInlineCacheCount();

for (; i < totalCacheCount; i++)

{

if (nullptr != this->inlineCaches[i])

{

IsInstInlineCache\* inlineCache = (IsInstInlineCache\*)this->inlineCaches[i];

if (IsScriptContextShutdown)

{

memset(inlineCache, 0, sizeof(IsInstInlineCache));

}

else

{

AllocatorDelete(IsInstInlineCacheAllocator, this->m\_scriptContext->GetIsInstInlineCacheAllocator(), inlineCache);

}

}

}

this->inlineCaches = nullptr;

}

if (nullptr != this->m\_codeGenRuntimeData)

{

for (ProfileId i = 0; i < this->profiledCallSiteCount; i++)

{

const FunctionCodeGenRuntimeData\* runtimeData = this->m\_codeGenRuntimeData[i];

if (nullptr != runtimeData)

{

runtimeData->MapInlineCaches([&](InlineCache\* inlineCache)

{

if (nullptr != inlineCache)

{

if (IsScriptContextShutdown)

{

memset(inlineCache, 0, sizeof(InlineCache));

}

else

{

if (inlineCache->RemoveFromInvalidationList())

{

unregisteredInlineCacheCount++;

}

AllocatorDelete(InlineCacheAllocator, this->m\_scriptContext->GetInlineCacheAllocator(), inlineCache);

}

}

});

}

}

}

if (nullptr != this->m\_codeGenGetSetRuntimeData)

{

for (uint i = 0; i < this->GetInlineCacheCount(); i++)

{

const FunctionCodeGenRuntimeData\* runtimeData = this->m\_codeGenGetSetRuntimeData[i];

if (nullptr != runtimeData)

{

runtimeData->MapInlineCaches([&](InlineCache\* inlineCache)

{

if (nullptr != inlineCache)

{

if (IsScriptContextShutdown)

{

memset(inlineCache, 0, sizeof(InlineCache));

}

else

{

if (inlineCache->RemoveFromInvalidationList())

{

unregisteredInlineCacheCount++;

}

AllocatorDelete(InlineCacheAllocator, this->m\_scriptContext->GetInlineCacheAllocator(), inlineCache);

}

}

});

}

}

}

if (!IsScriptContextShutdown)

{

ThreadContext\* threadContext = this->m\_scriptContext->GetThreadContext();

if (unregisteredInlineCacheCount > 0)

{

threadContext->NotifyInlineCacheBatchUnregistered(unregisteredInlineCacheCount);

}

}

while (polymorphicInlineCachesHead)

{

polymorphicInlineCachesHead->Finalize(IsScriptContextShutdown);

}

polymorphicInlineCaches.Reset();

}

void FunctionBody::CleanupRecyclerData(bool isShutdown, bool doEntryPointCleanupCaptureStack)

{

// If we're not shutting down (i.e closing the script context), we need to remove our inline caches from

// thread context's invalidation lists, and release memory back to the arena. During script context shutdown,

// we leave everything in place, because the inline cache arena will stay alive until script context is destroyed

// (i.e it's destructor has been called) and thus the invalidation lists are safe to keep references to caches from this

// script context. We will, however, zero all inline caches so that we don't have to process them on subsequent

// collections, which may still happen from other script contexts.

if (isShutdown)

{

CleanUpInlineCaches<true>();

}

else

{

CleanUpInlineCaches<false>();

}

if (this->entryPoints)

{

#if defined(ENABLE\_DEBUG\_CONFIG\_OPTIONS) && !(DBG)

// On fretest builds, capture the stack only if the FreTestDiagMode switch is on

doEntryPointCleanupCaptureStack = doEntryPointCleanupCaptureStack && Js::Configuration::Global.flags.FreTestDiagMode;

#endif

this->MapEntryPoints([=](int index, FunctionEntryPointInfo\* entryPoint)

{

if (nullptr != entryPoint)

{

// Finalize = Free up work item if it hasn't been released yet + entry point clean up

// isShutdown is false because cleanup is called only in the !isShutdown case

entryPoint->Finalize(isShutdown);

#ifdef ENABLE\_DEBUG\_CONFIG\_OPTIONS

// Do this separately since calling EntryPoint::Finalize doesn't capture the stack trace

// and in some calls to CleanupRecyclerData, we do want the stack trace captured.

if (doEntryPointCleanupCaptureStack)

{

entryPoint->CaptureCleanupStackTrace();

}

#endif

}

});

this->MapLoopHeaders([=](uint loopNumber, LoopHeader\* header)

{

bool shuttingDown = isShutdown;

header->MapEntryPoints([=](int index, LoopEntryPointInfo\* entryPoint)

{

entryPoint->Cleanup(shuttingDown, doEntryPointCleanupCaptureStack);

});

});

}

#ifdef PERF\_COUNTERS

this->CleanupPerfCounter();

#endif

}

//

// Removes all references of the function body and causes clean up of entry points.

// If the cleanup has already occurred before this would be a no-op.

//

void FunctionBody::Cleanup(bool isScriptContextClosing)

{

if (cleanedUp)

{

return;

}

CleanupRecyclerData(isScriptContextClosing, false /\* capture entry point cleanup stack trace \*/);

this->ResetObjectLiteralTypes();

// Manually clear these values to break any circular references

// that might prevent the script context from being disposed

this->auxBlock = nullptr;

this->auxContextBlock = nullptr;

this->byteCodeBlock = nullptr;

this->entryPoints = nullptr;

this->loopHeaderArray = nullptr;

this->m\_constTable = nullptr;

this->m\_codeGenRuntimeData = nullptr;

this->m\_codeGenGetSetRuntimeData = nullptr;

this->inlineCaches = nullptr;

this->polymorphicInlineCaches.Reset();

this->polymorphicInlineCachesHead = nullptr;

this->cacheIdToPropertyIdMap = nullptr;

this->referencedPropertyIdMap = nullptr;

this->literalRegexes = nullptr;

this->propertyIdsForScopeSlotArray = nullptr;

this->propertyIdOnRegSlotsContainer = nullptr;

#if DYNAMIC\_INTERPRETER\_THUNK

if (this->HasInterpreterThunkGenerated())

{

JS\_ETW(EtwTrace::LogMethodInterpreterThunkUnloadEvent(this));

if (!isScriptContextClosing)

{

if (m\_isAsmJsFunction)

{

m\_scriptContext->ReleaseDynamicAsmJsInterpreterThunk((BYTE\*)this->m\_dynamicInterpreterThunk, /\*addtoFreeList\*/!isScriptContextClosing);

}

else

{

m\_scriptContext->ReleaseDynamicInterpreterThunk((BYTE\*)this->m\_dynamicInterpreterThunk, /\*addtoFreeList\*/!isScriptContextClosing);

}

}

}

#endif

#if ENABLE\_PROFILE\_INFO

this->polymorphicCallSiteInfoHead = nullptr;

#endif

this->cleanedUp = true;

}

#ifdef PERF\_COUNTERS

void FunctionBody::CleanupPerfCounter()

{

// We might not have the byte code block yet if we defer parsed.

DWORD byteCodeSize = (this->byteCodeBlock? this->byteCodeBlock->GetLength() : 0)

+ (this->auxBlock? this->auxBlock->GetLength() : 0)

+ (this->auxContextBlock? this->auxContextBlock->GetLength() : 0);

PERF\_COUNTER\_SUB(Code, DynamicByteCodeSize, byteCodeSize);

if (this->m\_isDeserializedFunction)

{

PERF\_COUNTER\_DEC(Code, DeserializedFunctionBody);

}

PERF\_COUNTER\_SUB(Code, TotalByteCodeSize, byteCodeSize);

}

#endif

void FunctionBody::CaptureDynamicProfileState(FunctionEntryPointInfo\* entryPointInfo)

{

// DisableJIT-TODO: Move this to be under if DYNAMIC\_PROFILE

#if ENABLE\_NATIVE\_CODEGEN

// (See also the FunctionBody member written in CaptureDymamicProfileState.)

this->savedPolymorphicCacheState = entryPointInfo->GetPendingPolymorphicCacheState();

this->savedInlinerVersion = entryPointInfo->GetPendingInlinerVersion();

this->savedImplicitCallsFlags = entryPointInfo->GetPendingImplicitCallFlags();

#endif

}

#if ENABLE\_NATIVE\_CODEGEN

BYTE FunctionBody::GetSavedInlinerVersion() const

{

Assert(this->dynamicProfileInfo != nullptr);

return this->savedInlinerVersion;

}

uint32 FunctionBody::GetSavedPolymorphicCacheState() const

{

Assert(this->dynamicProfileInfo != nullptr);

return this->savedPolymorphicCacheState;

}

#endif

void FunctionBody::SetHasHotLoop()

{

if(hasHotLoop)

{

return;

}

hasHotLoop = true;

if(Configuration::Global.flags.EnforceExecutionModeLimits)

{

return;

}

CommitExecutedIterations();

TraceExecutionMode("HasHotLoop (before)");

if(fullJitThreshold > 1)

{

SetFullJitThreshold(1, true);

}

TraceExecutionMode("HasHotLoop");

}

bool FunctionBody::IsInlineApplyDisabled()

{

return this->disableInlineApply;

}

void FunctionBody::SetDisableInlineApply(bool set)

{

this->disableInlineApply = set;

}

void FunctionBody::InitDisableInlineApply()

{

SetDisableInlineApply(this->functionId != Js::Constants::NoFunctionId && PHASE\_OFF(Js::InlinePhase, this) || PHASE\_OFF(Js::InlineApplyPhase, this));

}

bool FunctionBody::CheckCalleeContextForInlining(FunctionProxy\* calleeFunctionProxy)

{

if (this->GetScriptContext() == calleeFunctionProxy->GetScriptContext())

{

if (this->GetHostSourceContext() == calleeFunctionProxy->GetHostSourceContext() &&

this->GetSecondaryHostSourceContext() == calleeFunctionProxy->GetSecondaryHostSourceContext())

{

return true;

}

}

return false;

}

#if ENABLE\_NATIVE\_CODEGEN

ImplicitCallFlags FunctionBody::GetSavedImplicitCallsFlags() const

{

Assert(this->dynamicProfileInfo != nullptr);

return this->savedImplicitCallsFlags;

}

bool FunctionBody::HasNonBuiltInCallee()

{

for (ProfileId i = 0; i < profiledCallSiteCount; i++)

{

Assert(HasDynamicProfileInfo());

bool ctor;

bool isPolymorphic;

FunctionInfo \*info = dynamicProfileInfo->GetCallSiteInfo(this, i, &ctor, &isPolymorphic);

if (info == nullptr || info->HasBody())

{

return true;

}

}

return false;

}

#endif

void FunctionBody::CheckAndRegisterFuncToDiag(ScriptContext \*scriptContext)

{

// We will register function if, this is not host managed and it was not registered before.

if (GetHostSourceContext() == Js::Constants::NoHostSourceContext

&& !m\_isFuncRegisteredToDiag

&& !scriptContext->GetDebugContext()->GetProbeContainer()->IsContextRegistered(GetSecondaryHostSourceContext()))

{

FunctionBody \*pFunc = scriptContext->GetDebugContext()->GetProbeContainer()->GetGlobalFunc(scriptContext, GetSecondaryHostSourceContext());

if (pFunc)

{

// Existing behavior here is to ignore the OOM and since RegisterFuncToDiag

// can throw now, we simply ignore the OOM here

try

{

// Register the function to the PDM as eval code (the debugger app will show file as 'eval code')

pFunc->RegisterFuncToDiag(scriptContext, Constants::EvalCode);

}

catch (Js::OutOfMemoryException)

{

}

scriptContext->GetDebugContext()->GetProbeContainer()->RegisterContextToDiag(GetSecondaryHostSourceContext(), scriptContext->AllocatorForDiagnostics());

m\_isFuncRegisteredToDiag = true;

}

}

else

{

m\_isFuncRegisteredToDiag = true;

}

}

DebuggerScope\* FunctionBody::RecordStartScopeObject(DiagExtraScopesType scopeType, int start, RegSlot scopeLocation, int\* index)

{

Recycler\* recycler = m\_scriptContext->GetRecycler();

if (!GetScopeObjectChain())

{

SetScopeObjectChain(RecyclerNew(recycler, ScopeObjectChain, recycler));

}

// Check if we need to create the scope object or if it already exists from a previous bytecode

// generator pass.

DebuggerScope\* debuggerScope = nullptr;

int currentDebuggerScopeIndex = this->GetNextDebuggerScopeIndex();

if (!this->TryGetDebuggerScopeAt(currentDebuggerScopeIndex, debuggerScope))

{

// Create a new debugger scope.

debuggerScope = AddScopeObject(scopeType, start, scopeLocation);

}

else

{

debuggerScope->UpdateDueToByteCodeRegeneration(scopeType, start, scopeLocation);

}

if(index)

{

\*index = currentDebuggerScopeIndex;

}

return debuggerScope;

}

void FunctionBody::RecordEndScopeObject(DebuggerScope\* currentScope, int end)

{

AssertMsg(currentScope, "No current debugger scope passed in.");

currentScope->SetEnd(end);

}

DebuggerScope \* FunctionBody::AddScopeObject(DiagExtraScopesType scopeType, int start, RegSlot scopeLocation)

{

Assert(GetScopeObjectChain());

DebuggerScope \*scopeObject = RecyclerNew(m\_scriptContext->GetRecycler(), DebuggerScope, m\_scriptContext->GetRecycler(), scopeType, scopeLocation, start);

GetScopeObjectChain()->pScopeChain->Add(scopeObject);

return scopeObject;

}

// Tries to retrieve the debugger scope at the specified index. If the index is out of range, nullptr

// is returned.

bool FunctionBody::TryGetDebuggerScopeAt(int index, DebuggerScope\*& debuggerScope)

{

AssertMsg(this->GetScopeObjectChain(), "TryGetDebuggerScopeAt should only be called with a valid scope chain in place.");

Assert(index >= 0);

const Js::ScopeObjectChain::ScopeObjectChainList\* scopeChain = this->GetScopeObjectChain()->pScopeChain;

if (index < scopeChain->Count())

{

debuggerScope = scopeChain->Item(index);

return true;

}

return false;

}

#if DYNAMIC\_INTERPRETER\_THUNK

DWORD FunctionBody::GetDynamicInterpreterThunkSize() const

{

return InterpreterThunkEmitter::ThunkSize;

}

#endif

#ifdef ENABLE\_DEBUG\_CONFIG\_OPTIONS

void

FunctionBody::DumpFullFunctionName()

{

wchar\_t debugStringBuffer[MAX\_FUNCTION\_BODY\_DEBUG\_STRING\_SIZE];

Output::Print(L"Function %s (%s)", this->GetDisplayName(), this->GetDebugNumberSet(debugStringBuffer));

}

void FunctionBody::DumpFunctionId(bool pad)

{

uint sourceContextId = this->GetSourceContextInfo()->sourceContextId;

if (sourceContextId == Js::Constants::NoSourceContext)

{

if (this->IsDynamicScript())

{

Output::Print(pad? L"Dy.%-3d" : L"Dyn#%d", this->GetLocalFunctionId());

}

else

{

// Function from LoadFile

Output::Print(pad? L"%-5d" : L"#%d", this->GetLocalFunctionId());

}

}

else

{

Output::Print(pad? L"%2d.%-3d" : L"#%d.%d", sourceContextId, this->GetLocalFunctionId());

}

}

#endif

void FunctionBody::EnsureAuxStatementData()

{

if (m\_sourceInfo.m\_auxStatementData == nullptr)

{

Recycler\* recycler = m\_scriptContext->GetRecycler();

// Note: allocating must be consistent with clean up in CleanupToReparse.

m\_sourceInfo.m\_auxStatementData = RecyclerNew(recycler, AuxStatementData);

}

}

/\*static\*/

void FunctionBody::GetShortNameFromUrl(\_\_in LPCWSTR pchUrl, \_Out\_writes\_z\_(cchBuffer) LPWSTR pchShortName, \_\_in size\_t cchBuffer)

{

LPCWSTR pchFile = wcsrchr(pchUrl, L'/');

if (pchFile == nullptr)

{

pchFile = wcsrchr(pchUrl, L'\\');

}

LPCWSTR pchToCopy = pchUrl;

if (pchFile != nullptr)

{

pchToCopy = pchFile + 1;

}

wcscpy\_s(pchShortName, cchBuffer, pchToCopy);

}

FunctionBody::StatementAdjustmentRecordList\* FunctionBody::GetStatementAdjustmentRecords()

{

if (m\_sourceInfo.m\_auxStatementData)

{

return m\_sourceInfo.m\_auxStatementData->m\_statementAdjustmentRecords;

}

return nullptr;

}

FunctionBody::CrossFrameEntryExitRecordList\* FunctionBody::GetCrossFrameEntryExitRecords()

{

if (m\_sourceInfo.m\_auxStatementData)

{

return m\_sourceInfo.m\_auxStatementData->m\_crossFrameBlockEntryExisRecords;

}

return nullptr;

}

void FunctionBody::RecordCrossFrameEntryExitRecord(uint byteCodeOffset, bool isEnterBlock)

{

this->EnsureAuxStatementData();

Recycler\* recycler = this->m\_scriptContext->GetRecycler();

if (this->GetCrossFrameEntryExitRecords() == nullptr)

{

m\_sourceInfo.m\_auxStatementData->m\_crossFrameBlockEntryExisRecords = RecyclerNew(recycler, CrossFrameEntryExitRecordList, recycler);

}

Assert(this->GetCrossFrameEntryExitRecords());

CrossFrameEntryExitRecord record(byteCodeOffset, isEnterBlock);

this->GetCrossFrameEntryExitRecords()->Add(record); // Will copy stack value and put the copy into the container.

}

FunctionBody::AuxStatementData::AuxStatementData() : m\_statementAdjustmentRecords(nullptr), m\_crossFrameBlockEntryExisRecords(nullptr)

{

}

FunctionBody::StatementAdjustmentRecord::StatementAdjustmentRecord() :

m\_byteCodeOffset((uint)Constants::InvalidOffset), m\_adjustmentType(SAT\_None)

{

}

FunctionBody::StatementAdjustmentRecord::StatementAdjustmentRecord(StatementAdjustmentType type, int byteCodeOffset) :

m\_adjustmentType(type), m\_byteCodeOffset(byteCodeOffset)

{

Assert(SAT\_None <= type && type <= SAT\_All);

}

FunctionBody::StatementAdjustmentRecord::StatementAdjustmentRecord(const StatementAdjustmentRecord& other) :

m\_byteCodeOffset(other.m\_byteCodeOffset), m\_adjustmentType(other.m\_adjustmentType)

{

}

uint FunctionBody::StatementAdjustmentRecord::GetByteCodeOffset()

{

Assert(m\_byteCodeOffset != Constants::InvalidOffset);

return m\_byteCodeOffset;

}

FunctionBody::StatementAdjustmentType FunctionBody::StatementAdjustmentRecord::GetAdjustmentType()

{

Assert(this->m\_adjustmentType != SAT\_None);

return m\_adjustmentType;

}

FunctionBody::CrossFrameEntryExitRecord::CrossFrameEntryExitRecord() :

m\_byteCodeOffset((uint)Constants::InvalidOffset), m\_isEnterBlock(false)

{

}

FunctionBody::CrossFrameEntryExitRecord::CrossFrameEntryExitRecord(uint byteCodeOffset, bool isEnterBlock) :

m\_byteCodeOffset(byteCodeOffset), m\_isEnterBlock(isEnterBlock)

{

}

FunctionBody::CrossFrameEntryExitRecord::CrossFrameEntryExitRecord(const CrossFrameEntryExitRecord& other) :

m\_byteCodeOffset(other.m\_byteCodeOffset), m\_isEnterBlock(other.m\_isEnterBlock)

{

}

uint FunctionBody::CrossFrameEntryExitRecord::GetByteCodeOffset() const

{

Assert(m\_byteCodeOffset != Constants::InvalidOffset);

return m\_byteCodeOffset;

}

bool FunctionBody::CrossFrameEntryExitRecord::GetIsEnterBlock()

{

return m\_isEnterBlock;

}

PolymorphicInlineCacheInfo \* EntryPointPolymorphicInlineCacheInfo::GetInlineeInfo(FunctionBody \* inlineeFunctionBody)

{

SListBase<PolymorphicInlineCacheInfo\*>::Iterator iter(&inlineeInfo);

while (iter.Next())

{

PolymorphicInlineCacheInfo \* info = iter.Data();

if (info->GetFunctionBody() == inlineeFunctionBody)

{

return info;

}

}

return nullptr;

}

PolymorphicInlineCacheInfo \* EntryPointPolymorphicInlineCacheInfo::EnsureInlineeInfo(Recycler \* recycler, FunctionBody \* inlineeFunctionBody)

{

PolymorphicInlineCacheInfo \* info = GetInlineeInfo(inlineeFunctionBody);

if (!info)

{

info = RecyclerNew(recycler, PolymorphicInlineCacheInfo, inlineeFunctionBody);

inlineeInfo.Prepend(recycler, info);

}

return info;

}

void EntryPointPolymorphicInlineCacheInfo::SetPolymorphicInlineCache(FunctionBody \* functionBody, uint index, PolymorphicInlineCache \* polymorphicInlineCache, bool isInlinee, byte polyCacheUtil)

{

if (!isInlinee)

{

SetPolymorphicInlineCache(&selfInfo, functionBody, index, polymorphicInlineCache, polyCacheUtil);

Assert(functionBody == selfInfo.GetFunctionBody());

}

else

{

SetPolymorphicInlineCache(EnsureInlineeInfo(functionBody->GetScriptContext()->GetRecycler(), functionBody), functionBody, index, polymorphicInlineCache, polyCacheUtil);

Assert(functionBody == GetInlineeInfo(functionBody)->GetFunctionBody());

}

}

void EntryPointPolymorphicInlineCacheInfo::SetPolymorphicInlineCache(PolymorphicInlineCacheInfo \* polymorphicInlineCacheInfo, FunctionBody \* functionBody, uint index, PolymorphicInlineCache \* polymorphicInlineCache, byte polyCacheUtil)

{

polymorphicInlineCacheInfo->GetPolymorphicInlineCaches()->SetInlineCache(functionBody->GetScriptContext()->GetRecycler(), functionBody, index, polymorphicInlineCache);

polymorphicInlineCacheInfo->GetUtilArray()->SetUtil(functionBody, index, polyCacheUtil);

}

void PolymorphicCacheUtilizationArray::SetUtil(Js::FunctionBody\* functionBody, uint index, byte util)

{

Assert(functionBody);

Assert(index < functionBody->GetInlineCacheCount());

EnsureUtilArray(functionBody->GetScriptContext()->GetRecycler(), functionBody);

this->utilArray[index] = util;

}

byte PolymorphicCacheUtilizationArray::GetUtil(Js::FunctionBody\* functionBody, uint index)

{

Assert(index < functionBody->GetInlineCacheCount());

return this->utilArray[index];

}

void PolymorphicCacheUtilizationArray::EnsureUtilArray(Recycler \* const recycler, Js::FunctionBody \* functionBody)

{

Assert(recycler);

Assert(functionBody);

Assert(functionBody->GetInlineCacheCount() != 0);

if(this->utilArray)

{

return;

}

this->utilArray = RecyclerNewArrayZ(recycler, byte, functionBody->GetInlineCacheCount());

}

#if ENABLE\_NATIVE\_CODEGEN

void EntryPointInfo::AddWeakFuncRef(RecyclerWeakReference<FunctionBody> \*weakFuncRef, Recycler \*recycler)

{

Assert(this->state == CodeGenPending);

this->weakFuncRefSet = this->EnsureWeakFuncRefSet(recycler);

this->weakFuncRefSet->AddNew(weakFuncRef);

}

EntryPointInfo::WeakFuncRefSet \*

EntryPointInfo::EnsureWeakFuncRefSet(Recycler \*recycler)

{

if (this->weakFuncRefSet == nullptr)

{

this->weakFuncRefSet = RecyclerNew(recycler, WeakFuncRefSet, recycler);

}

return this->weakFuncRefSet;

}

void EntryPointInfo::EnsureIsReadyToCall()

{

ProcessJitTransferData();

}

void EntryPointInfo::ProcessJitTransferData()

{

Assert(!IsCleanedUp());

if (GetJitTransferData() != nullptr && GetJitTransferData()->GetIsReady())

{

class AutoCleanup

{

EntryPointInfo \*entryPointInfo;

public:

AutoCleanup(EntryPointInfo \*entryPointInfo) : entryPointInfo(entryPointInfo)

{

}

void Done()

{

entryPointInfo = nullptr;

}

~AutoCleanup()

{

if (entryPointInfo)

{

entryPointInfo->OnNativeCodeInstallFailure();

}

}

} autoCleanup(this);

ScriptContext\* scriptContext = GetScriptContext();

PinTypeRefs(scriptContext);

InstallGuards(scriptContext);

FreeJitTransferData();

autoCleanup.Done();

}

}

EntryPointInfo::JitTransferData\* EntryPointInfo::EnsureJitTransferData(Recycler\* recycler)

{

if (this->jitTransferData == nullptr)

{

this->jitTransferData = RecyclerNew(recycler, EntryPointInfo::JitTransferData);

}

return this->jitTransferData;

}

#ifdef FIELD\_ACCESS\_STATS

FieldAccessStats\* EntryPointInfo::EnsureFieldAccessStats(Recycler\* recycler)

{

if (this->fieldAccessStats == nullptr)

{

this->fieldAccessStats = RecyclerNew(recycler, FieldAccessStats);

}

return this->fieldAccessStats;

}

#endif

void EntryPointInfo::JitTransferData::AddJitTimeTypeRef(void\* typeRef, Recycler\* recycler)

{

Assert(typeRef != nullptr);

EnsureJitTimeTypeRefs(recycler);

this->jitTimeTypeRefs->AddNew(typeRef);

}

void EntryPointInfo::JitTransferData::EnsureJitTimeTypeRefs(Recycler\* recycler)

{

if (this->jitTimeTypeRefs == nullptr)

{

this->jitTimeTypeRefs = RecyclerNew(recycler, TypeRefSet, recycler);

}

}

void EntryPointInfo::PinTypeRefs(ScriptContext\* scriptContext)

{

Assert(this->jitTransferData != nullptr && this->jitTransferData->GetIsReady());

Recycler\* recycler = scriptContext->GetRecycler();

if (this->jitTransferData->GetRuntimeTypeRefs() != nullptr)

{

// Copy pinned types from a heap allocated array created on the background thread

// to a recycler allocated array which will live as long as this EntryPointInfo.

// The original heap allocated array will be freed at the end of NativeCodeGenerator::CheckCodeGenDone

void\*\* jitPinnedTypeRefs = this->jitTransferData->GetRuntimeTypeRefs();

size\_t jitPinnedTypeRefCount = this->jitTransferData->GetRuntimeTypeRefCount();

this->runtimeTypeRefs = RecyclerNewArray(recycler, void\*, jitPinnedTypeRefCount + 1);

js\_memcpy\_s(this->runtimeTypeRefs, jitPinnedTypeRefCount \* sizeof(void\*), jitPinnedTypeRefs, jitPinnedTypeRefCount \* sizeof(void\*));

this->runtimeTypeRefs[jitPinnedTypeRefCount] = nullptr;

}

}

void EntryPointInfo::InstallGuards(ScriptContext\* scriptContext)

{

Assert(this->jitTransferData != nullptr && this->jitTransferData->GetIsReady());

Assert(this->equivalentTypeCacheCount == 0 && this->equivalentTypeCaches == nullptr);

Assert(this->propertyGuardCount == 0 && this->propertyGuardWeakRefs == nullptr);

class AutoCleanup

{

EntryPointInfo \*entryPointInfo;

public:

AutoCleanup(EntryPointInfo \*entryPointInfo) : entryPointInfo(entryPointInfo)

{

}

void Done()

{

entryPointInfo = nullptr;

}

~AutoCleanup()

{

if (entryPointInfo)

{

entryPointInfo->equivalentTypeCacheCount = 0;

entryPointInfo->equivalentTypeCaches = nullptr;

entryPointInfo->propertyGuardCount = 0;

entryPointInfo->propertyGuardWeakRefs = nullptr;

entryPointInfo->UnregisterEquivalentTypeCaches();

}

}

} autoCleanup(this);

for (int i = 0; i < this->jitTransferData->lazyBailoutPropertyCount; i++)

{

Assert(this->jitTransferData->lazyBailoutProperties != nullptr);

Js::PropertyId propertyId = this->jitTransferData->lazyBailoutProperties[i];

Js::PropertyGuard\* sharedPropertyGuard;

bool hasSharedPropertyGuard = TryGetSharedPropertyGuard(propertyId, sharedPropertyGuard);

Assert(hasSharedPropertyGuard);

bool isValid = hasSharedPropertyGuard ? sharedPropertyGuard->IsValid() : false;

if (isValid)

{

scriptContext->GetThreadContext()->RegisterLazyBailout(propertyId, this);

}

else

{

OUTPUT\_TRACE2(Js::LazyBailoutPhase, this->GetFunctionBody(), L"Lazy bailout - Invalidation due to property: %s \n", scriptContext->GetPropertyName(propertyId)->GetBuffer());

this->Invalidate(true);

return;

}

}

if (this->jitTransferData->equivalentTypeGuardCount > 0)

{

Assert(this->jitTransferData->equivalentTypeGuards != nullptr);

Recycler\* recycler = scriptContext->GetRecycler();

int guardCount = this->jitTransferData->equivalentTypeGuardCount;

JitEquivalentTypeGuard\*\* guards = this->jitTransferData->equivalentTypeGuards;

// Create an array of equivalent type caches on the entry point info to ensure they are kept

// alive for the lifetime of the entry point.

this->equivalentTypeCacheCount = guardCount;

// No need to zero-initialize, since we will populate all data slots.

// We used to let the recycler scan the types in the cache, but we no longer do. See

// ThreadContext::ClearEquivalentTypeCaches for an explanation.

this->equivalentTypeCaches = RecyclerNewArrayLeafZ(recycler, EquivalentTypeCache, guardCount);

this->RegisterEquivalentTypeCaches();

EquivalentTypeCache\* cache = this->equivalentTypeCaches;

for (JitEquivalentTypeGuard\*\* guard = guards; guard < guards + guardCount; guard++)

{

EquivalentTypeCache\* oldCache = (\*guard)->GetCache();

// Copy the contents of the heap-allocated cache to the recycler-allocated version to make sure the types are

// kept alive. Allow the properties pointer to refer to the heap-allocated arrays. It will stay alive as long

// as the entry point is alive, and property entries contain no pointers to other recycler allocated objects.

(\*cache) = (\*oldCache);

// Set the recycler-allocated cache on the (heap-allocated) guard.

(\*guard)->SetCache(cache);

cache++;

}

}

// The propertyGuardsByPropertyId structure is temporary and serves only to register the type guards for the correct

// properties. If we've done code gen for this EntryPointInfo, typePropertyGuardsByPropertyId will have been used and nulled out.

if (this->jitTransferData->propertyGuardsByPropertyId != nullptr)

{

this->propertyGuardCount = this->jitTransferData->propertyGuardCount;

this->propertyGuardWeakRefs = RecyclerNewArrayZ(scriptContext->GetRecycler(), FakePropertyGuardWeakReference\*, this->propertyGuardCount);

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

Js::TypeGuardTransferEntry\* entry = this->jitTransferData->propertyGuardsByPropertyId;

while (entry->propertyId != Js::Constants::NoProperty)

{

Js::PropertyId propertyId = entry->propertyId;

Js::PropertyGuard\* sharedPropertyGuard;

// We use the shared guard created during work item creation to ensure that the condition we assumed didn't change while

// we were JIT-ing. If we don't have a shared property guard for this property then we must not need to protect it,

// because it exists on the instance. Unfortunately, this means that if we have a bug and fail to create a shared

// guard for some property during work item creation, we won't find out about it here.

bool isNeeded = TryGetSharedPropertyGuard(propertyId, sharedPropertyGuard);

bool isValid = isNeeded ? sharedPropertyGuard->IsValid() : false;

int entryGuardIndex = 0;

while (entry->guards[entryGuardIndex] != nullptr)

{

if (isNeeded)

{

Js::JitIndexedPropertyGuard\* guard = entry->guards[entryGuardIndex];

int guardIndex = guard->GetIndex();

Assert(guardIndex >= 0 && guardIndex < this->propertyGuardCount);

// We use the shared guard here to make sure the conditions we assumed didn't change while we were JIT-ing.

// If they did, we proactively invalidate the guard here, so that we bail out if we try to call this code.

if (isValid)

{

auto propertyGuardWeakRef = this->propertyGuardWeakRefs[guardIndex];

if (propertyGuardWeakRef == nullptr)

{

propertyGuardWeakRef = Js::FakePropertyGuardWeakReference::New(scriptContext->GetRecycler(), guard);

this->propertyGuardWeakRefs[guardIndex] = propertyGuardWeakRef;

}

Assert(propertyGuardWeakRef->Get() == guard);

threadContext->RegisterUniquePropertyGuard(propertyId, propertyGuardWeakRef);

}

else

{

guard->Invalidate();

}

}

entryGuardIndex++;

}

entry = reinterpret\_cast<Js::TypeGuardTransferEntry\*>(&entry->guards[++entryGuardIndex]);

}

}

// The ctorCacheGuardsByPropertyId structure is temporary and serves only to register the constructor cache guards for the correct

// properties. If we've done code gen for this EntryPointInfo, ctorCacheGuardsByPropertyId will have been used and nulled out.

// Unlike type property guards, constructor cache guards use the live constructor caches associated with function objects. These are

// recycler allocated and are kept alive by the constructorCaches field, where they were inserted during work item creation.

if (this->jitTransferData->ctorCacheGuardsByPropertyId != nullptr)

{

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

Js::CtorCacheGuardTransferEntry\* entry = this->jitTransferData->ctorCacheGuardsByPropertyId;

while (entry->propertyId != Js::Constants::NoProperty)

{

Js::PropertyId propertyId = entry->propertyId;

Js::PropertyGuard\* sharedPropertyGuard;

// We use the shared guard created during work item creation to ensure that the condition we assumed didn't change while

// we were JIT-ing. If we don't have a shared property guard for this property then we must not need to protect it,

// because it exists on the instance. Unfortunately, this means that if we have a bug and fail to create a shared

// guard for some property during work item creation, we won't find out about it here.

bool isNeeded = TryGetSharedPropertyGuard(propertyId, sharedPropertyGuard);

bool isValid = isNeeded ? sharedPropertyGuard->IsValid() : false;

int entryCacheIndex = 0;

while (entry->caches[entryCacheIndex] != nullptr)

{

if (isNeeded)

{

Js::ConstructorCache\* cache = entry->caches[entryCacheIndex];

// We use the shared cache here to make sure the conditions we assumed didn't change while we were JIT-ing.

// If they did, we proactively invalidate the cache here, so that we bail out if we try to call this code.

if (isValid)

{

threadContext->RegisterConstructorCache(propertyId, cache);

}

else

{

cache->InvalidateAsGuard();

}

}

entryCacheIndex++;

}

entry = reinterpret\_cast<Js::CtorCacheGuardTransferEntry\*>(&entry->caches[++entryCacheIndex]);

}

}

if (PHASE\_ON(Js::FailNativeCodeInstallPhase, this->GetFunctionBody()))

{

Js::Throw::OutOfMemory();

}

autoCleanup.Done();

}

PropertyGuard\* EntryPointInfo::RegisterSharedPropertyGuard(Js::PropertyId propertyId, ScriptContext\* scriptContext)

{

if (this->sharedPropertyGuards == nullptr)

{

Recycler\* recycler = scriptContext->GetRecycler();

this->sharedPropertyGuards = RecyclerNew(recycler, SharedPropertyGuardDictionary, recycler);

}

PropertyGuard\* guard;

if (!this->sharedPropertyGuards->TryGetValue(propertyId, &guard))

{

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

guard = threadContext->RegisterSharedPropertyGuard(propertyId);

this->sharedPropertyGuards->Add(propertyId, guard);

}

return guard;

}

bool EntryPointInfo::HasSharedPropertyGuard(Js::PropertyId propertyId)

{

return this->sharedPropertyGuards != nullptr ? this->sharedPropertyGuards->ContainsKey(propertyId) : false;

}

bool EntryPointInfo::TryGetSharedPropertyGuard(Js::PropertyId propertyId, Js::PropertyGuard\*& guard)

{

return this->sharedPropertyGuards != nullptr ? this->sharedPropertyGuards->TryGetValue(propertyId, &guard) : false;

}

void EntryPointInfo::RecordTypeGuards(int typeGuardCount, TypeGuardTransferEntry\* typeGuardTransferRecord, size\_t typeGuardTransferPlusSize)

{

Assert(this->jitTransferData != nullptr);

this->jitTransferData->propertyGuardCount = typeGuardCount;

this->jitTransferData->propertyGuardsByPropertyId = typeGuardTransferRecord;

this->jitTransferData->propertyGuardsByPropertyIdPlusSize = typeGuardTransferPlusSize;

}

void EntryPointInfo::RecordCtorCacheGuards(CtorCacheGuardTransferEntry\* ctorCacheTransferRecord, size\_t ctorCacheTransferPlusSize)

{

Assert(this->jitTransferData != nullptr);

this->jitTransferData->ctorCacheGuardsByPropertyId = ctorCacheTransferRecord;

this->jitTransferData->ctorCacheGuardsByPropertyIdPlusSize = ctorCacheTransferPlusSize;

}

void EntryPointInfo::FreePropertyGuards()

{

// While typePropertyGuardWeakRefs are allocated via NativeCodeData::Allocator and will be automatically freed to the heap,

// we must zero out the fake weak references so that property guard invalidation doesn't access freed memory.

if (this->propertyGuardWeakRefs != nullptr)

{

for (int i = 0; i < this->propertyGuardCount; i++)

{

if (this->propertyGuardWeakRefs[i] != nullptr)

{

this->propertyGuardWeakRefs[i]->Zero();

}

}

this->propertyGuardCount = 0;

this->propertyGuardWeakRefs = nullptr;

}

}

void EntryPointInfo::RecordBailOutMap(JsUtil::List<LazyBailOutRecord, ArenaAllocator>\* bailoutMap)

{

Assert(this->bailoutRecordMap == nullptr);

this->bailoutRecordMap = HeapNew(BailOutRecordMap, &HeapAllocator::Instance);

this->bailoutRecordMap->Copy(bailoutMap);

}

void EntryPointInfo::RecordInlineeFrameMap(JsUtil::List<NativeOffsetInlineeFramePair, ArenaAllocator>\* tempInlineeFrameMap)

{

Assert(this->inlineeFrameMap == nullptr);

if (tempInlineeFrameMap->Count() > 0)

{

this->inlineeFrameMap = HeapNew(InlineeFrameMap, &HeapAllocator::Instance);

this->inlineeFrameMap->Copy(tempInlineeFrameMap);

}

}

InlineeFrameRecord\* EntryPointInfo::FindInlineeFrame(void\* returnAddress)

{

if (this->inlineeFrameMap == nullptr)

{

return nullptr;

}

size\_t offset = (size\_t)((BYTE\*)returnAddress - (BYTE\*)this->GetNativeAddress());

int index = this->inlineeFrameMap->BinarySearch([=](const NativeOffsetInlineeFramePair& pair, int index) {

if (pair.offset >= offset)

{

if (index == 0 || index > 0 && this->inlineeFrameMap->Item(index - 1).offset < offset)

{

return 0;

}

else

{

return 1;

}

}

return -1;

});

if (index == -1)

{

return nullptr;

}

return this->inlineeFrameMap->Item(index).record;

}

void EntryPointInfo::DoLazyBailout(BYTE\*\* addressOfInstructionPointer, Js::FunctionBody\* functionBody, const PropertyRecord\* propertyRecord)

{

BYTE\* instructionPointer = \*addressOfInstructionPointer;

Assert(instructionPointer > (BYTE\*)this->nativeAddress && instructionPointer < ((BYTE\*)this->nativeAddress + this->codeSize));

size\_t offset = instructionPointer - (BYTE\*)this->nativeAddress;

LazyBailOutRecord record;

int found = this->bailoutRecordMap->BinarySearch([=](const LazyBailOutRecord& record, int index)

{

// find the closest entry which is greater than the current offset.

if (record.offset >= offset)

{

if (index == 0 || index > 0 && this->bailoutRecordMap->Item(index - 1).offset < offset)

{

return 0;

}

else

{

return 1;

}

}

return -1;

});

if (found != -1)

{

LazyBailOutRecord& record = this->bailoutRecordMap->Item(found);

\*addressOfInstructionPointer = record.instructionPointer;

record.SetBailOutKind();

if (PHASE\_TRACE1(Js::LazyBailoutPhase))

{

Output::Print(L"On stack lazy bailout. Property: %s Old IP: 0x%x New IP: 0x%x ", propertyRecord->GetBuffer(), instructionPointer, record.instructionPointer);

#if DBG

record.Dump(functionBody);

#endif

Output::Print(L"\n");

}

}

else

{

AssertMsg(false, "Lazy Bailout address mapping missing");

}

}

void EntryPointInfo::FreeJitTransferData()

{

JitTransferData\* jitTransferData = this->jitTransferData;

this->jitTransferData = nullptr;

if (jitTransferData != nullptr)

{

// This dictionary is recycler allocated so it doesn't need to be explicitly freed.

jitTransferData->jitTimeTypeRefs = nullptr;

if (jitTransferData->lazyBailoutProperties != nullptr)

{

HeapDeleteArray(jitTransferData->lazyBailoutPropertyCount, jitTransferData->lazyBailoutProperties);

jitTransferData->lazyBailoutProperties = nullptr;

}

// All structures below are heap allocated and need to be freed explicitly.

if (jitTransferData->runtimeTypeRefs != nullptr)

{

HeapDeleteArray(jitTransferData->runtimeTypeRefCount, jitTransferData->runtimeTypeRefs);

jitTransferData->runtimeTypeRefs = nullptr;

}

jitTransferData->runtimeTypeRefCount = 0;

if (jitTransferData->propertyGuardsByPropertyId != nullptr)

{

HeapDeletePlus(jitTransferData->propertyGuardsByPropertyIdPlusSize, jitTransferData->propertyGuardsByPropertyId);

jitTransferData->propertyGuardsByPropertyId = nullptr;

}

jitTransferData->propertyGuardCount = 0;

jitTransferData->propertyGuardsByPropertyIdPlusSize = 0;

if (jitTransferData->ctorCacheGuardsByPropertyId != nullptr)

{

HeapDeletePlus(jitTransferData->ctorCacheGuardsByPropertyIdPlusSize, jitTransferData->ctorCacheGuardsByPropertyId);

jitTransferData->ctorCacheGuardsByPropertyId = nullptr;

}

jitTransferData->ctorCacheGuardsByPropertyIdPlusSize = 0;

if (jitTransferData->equivalentTypeGuards != nullptr)

{

HeapDeleteArray(jitTransferData->equivalentTypeGuardCount, jitTransferData->equivalentTypeGuards);

jitTransferData->equivalentTypeGuards = nullptr;

}

jitTransferData->equivalentTypeGuardCount = 0;

if (jitTransferData->data != nullptr)

{

HeapDelete(jitTransferData->data);

jitTransferData->data = nullptr;

}

jitTransferData = nullptr;

}

}

void EntryPointInfo::RegisterEquivalentTypeCaches()

{

Assert(this->registeredEquivalentTypeCacheRef == nullptr);

this->registeredEquivalentTypeCacheRef =

GetScriptContext()->GetThreadContext()->RegisterEquivalentTypeCacheEntryPoint(this);

}

void EntryPointInfo::UnregisterEquivalentTypeCaches()

{

if (this->registeredEquivalentTypeCacheRef != nullptr)

{

ScriptContext \*scriptContext = GetScriptContext();

if (scriptContext != nullptr)

{

scriptContext->GetThreadContext()->UnregisterEquivalentTypeCacheEntryPoint(

this->registeredEquivalentTypeCacheRef);

}

this->registeredEquivalentTypeCacheRef = nullptr;

}

}

bool EntryPointInfo::ClearEquivalentTypeCaches()

{

Assert(this->equivalentTypeCaches != nullptr);

Assert(this->equivalentTypeCacheCount > 0);

bool isAnyCacheLive = false;

Recycler \*recycler = GetScriptContext()->GetRecycler();

for (EquivalentTypeCache \*cache = this->equivalentTypeCaches;

cache < this->equivalentTypeCaches + this->equivalentTypeCacheCount;

cache++)

{

bool isCacheLive = cache->ClearUnusedTypes(recycler);

if (isCacheLive)

{

isAnyCacheLive = true;

}

}

if (!isAnyCacheLive)

{

// The caller must take care of unregistering this entry point. We may be in the middle of

// walking the list of registered entry points.

this->equivalentTypeCaches = nullptr;

this->equivalentTypeCacheCount = 0;

this->registeredEquivalentTypeCacheRef = nullptr;

}

return isAnyCacheLive;

}

bool EquivalentTypeCache::ClearUnusedTypes(Recycler \*recycler)

{

bool isAnyTypeLive = false;

Assert(this->guard);

if (this->guard->IsValid())

{

Type \*type = reinterpret\_cast<Type\*>(this->guard->GetValue());

if (!recycler->IsObjectMarked(type))

{

this->guard->Invalidate();

}

else

{

isAnyTypeLive = true;

}

}

for (int i = 0; i < EQUIVALENT\_TYPE\_CACHE\_SIZE; i++)

{

Type \*type = this->types[i];

if (type != nullptr)

{

if (!recycler->IsObjectMarked(type))

{

this->types[i] = nullptr;

}

else

{

isAnyTypeLive = true;

}

}

}

return isAnyTypeLive;

}

void EntryPointInfo::RegisterConstructorCache(Js::ConstructorCache\* constructorCache, Recycler\* recycler)

{

Assert(constructorCache != nullptr);

if (!this->constructorCaches)

{

this->constructorCaches = RecyclerNew(recycler, ConstructorCacheList, recycler);

}

this->constructorCaches->Prepend(constructorCache);

}

#endif

#ifdef ENABLE\_DEBUG\_CONFIG\_OPTIONS

void EntryPointInfo::CaptureCleanupStackTrace()

{

if (this->cleanupStack != nullptr)

{

this->cleanupStack->Delete(&NoCheckHeapAllocator::Instance);

this->cleanupStack = nullptr;

}

this->cleanupStack = StackBackTrace::Capture(&NoCheckHeapAllocator::Instance);

}

#endif

void EntryPointInfo::Finalize(bool isShutdown)

{

\_\_super::Finalize(isShutdown);

if (!isShutdown)

{

ReleasePendingWorkItem();

}

#if ENABLE\_DEBUG\_CONFIG\_OPTIONS

this->SetCleanupReason(CleanupReason::CleanUpForFinalize);

#endif

this->Cleanup(isShutdown, false);

#if DBG

if (this->cleanupStack != nullptr)

{

this->cleanupStack->Delete(&NoCheckHeapAllocator::Instance);

this->cleanupStack = nullptr;

}

#endif

this->library = nullptr;

}

#if ENABLE\_NATIVE\_CODEGEN

EntryPointPolymorphicInlineCacheInfo \* EntryPointInfo::EnsurePolymorphicInlineCacheInfo(Recycler \* recycler, FunctionBody \* functionBody)

{

if (!polymorphicInlineCacheInfo)

{

polymorphicInlineCacheInfo = RecyclerNew(recycler, EntryPointPolymorphicInlineCacheInfo, functionBody);

}

return polymorphicInlineCacheInfo;

}

#endif

void EntryPointInfo::Cleanup(bool isShutdown, bool captureCleanupStack)

{

if (this->GetState() != CleanedUp)

{

this->OnCleanup(isShutdown);

#if ENABLE\_NATIVE\_CODEGEN

FreeJitTransferData();

if (this->bailoutRecordMap != nullptr)

{

HeapDelete(this->bailoutRecordMap);

bailoutRecordMap = nullptr;

}

if (this->sharedPropertyGuards != nullptr)

{

sharedPropertyGuards->Clear();

sharedPropertyGuards = nullptr;

}

FreePropertyGuards();

if (this->equivalentTypeCaches != nullptr)

{

this->UnregisterEquivalentTypeCaches();

this->equivalentTypeCacheCount = 0;

this->equivalentTypeCaches = nullptr;

}

if (this->constructorCaches != nullptr)

{

this->constructorCaches->Clear();

}

#endif

// This is how we set the CleanedUp state

this->workItem = nullptr;

this->nativeAddress = nullptr;

#if ENABLE\_NATIVE\_CODEGEN

this->weakFuncRefSet = nullptr;

this->runtimeTypeRefs = nullptr;

#endif

this->codeSize = -1;

this->library = nullptr;

#if ENABLE\_NATIVE\_CODEGEN

DeleteNativeCodeData(this->data);

this->data = nullptr;

this->numberChunks = nullptr;

#endif

this->state = CleanedUp;

#if ENABLE\_DEBUG\_CONFIG\_OPTIONS

#if !DBG

captureCleanupStack = captureCleanupStack && Js::Configuration::Global.flags.FreTestDiagMode;

#endif

if (captureCleanupStack)

{

this->CaptureCleanupStackTrace();

}

#endif

#if ENABLE\_NATIVE\_CODEGEN

if (nullptr != this->nativeThrowSpanSequence)

{

HeapDelete(this->nativeThrowSpanSequence);

this->nativeThrowSpanSequence = nullptr;

}

this->polymorphicInlineCacheInfo = nullptr;

#endif

#if DBG\_DUMP | defined(VTUNE\_PROFILING)

this->nativeOffsetMaps.Reset();

#endif

}

}

void EntryPointInfo::Reset(bool resetStateToNotScheduled)

{

Assert(this->GetState() != CleanedUp);

this->nativeAddress = nullptr;

this->workItem = nullptr;

#if ENABLE\_NATIVE\_CODEGEN

if (nullptr != this->nativeThrowSpanSequence)

{

HeapDelete(this->nativeThrowSpanSequence);

this->nativeThrowSpanSequence = nullptr;

}

#endif

this->codeSize = 0;

#if ENABLE\_NATIVE\_CODEGEN

this->weakFuncRefSet = nullptr;

this->sharedPropertyGuards = nullptr;

FreePropertyGuards();

FreeJitTransferData();

if (this->data != nullptr)

{

DeleteNativeCodeData(this->data);

this->data = nullptr;

}

#endif

// Set the state to NotScheduled only if the call to Reset is not because of JIT cap being reached

if (resetStateToNotScheduled)

{

this->state = NotScheduled;

}

}

#if ENABLE\_NATIVE\_CODEGEN

void EntryPointInfo::ResetOnNativeCodeInstallFailure()

{

// Reset the entry point without attempting to create a new default and GenerateFunction on it.

// Do this for LoopEntryPointInfo or if we throw during FunctionEntryPointInfo::Invalidate.

this->Reset(true);

Assert(this->address != nullptr);

FreeNativeCodeGenAllocation(GetScriptContext(), this->address);

this->address = nullptr;

}

#endif

#ifdef PERF\_COUNTERS

void FunctionEntryPointInfo::OnRecorded()

{

PERF\_COUNTER\_ADD(Code, TotalNativeCodeSize, GetCodeSize());

PERF\_COUNTER\_ADD(Code, FunctionNativeCodeSize, GetCodeSize());

PERF\_COUNTER\_ADD(Code, DynamicNativeCodeSize, GetCodeSize());

}

#endif

FunctionEntryPointInfo::FunctionEntryPointInfo(FunctionProxy \* functionProxy, void \* address, ThreadContext\* context, void\* cookie) :

EntryPointInfo(address, functionProxy->GetScriptContext()->GetLibrary(), cookie, context),

localVarSlotsOffset(Js::Constants::InvalidOffset),

localVarChangedOffset(Js::Constants::InvalidOffset),

callsCount(0),

jitMode(ExecutionMode::Interpreter),

nativeEntryPointProcessed(false),

functionProxy(functionProxy),

nextEntryPoint(nullptr),

mIsTemplatizedJitMode(false)

{

}

#ifndef TEMP\_DISABLE\_ASMJS

void FunctionEntryPointInfo::SetOldFunctionEntryPointInfo(FunctionEntryPointInfo\* entrypointInfo)

{

Assert(this->GetIsAsmJSFunction());

Assert(entrypointInfo);

mOldFunctionEntryPointInfo = entrypointInfo;

};

FunctionEntryPointInfo\* FunctionEntryPointInfo::GetOldFunctionEntryPointInfo()const

{

Assert(this->GetIsAsmJSFunction());

return mOldFunctionEntryPointInfo;

};

void FunctionEntryPointInfo::SetIsTJMode(bool value)

{

Assert(this->GetIsAsmJSFunction());

mIsTemplatizedJitMode = value;

}

bool FunctionEntryPointInfo::GetIsTJMode()const

{

return mIsTemplatizedJitMode;

};

#endif

//End AsmJS Support

#if ENABLE\_NATIVE\_CODEGEN

ExecutionMode FunctionEntryPointInfo::GetJitMode() const

{

return jitMode;

}

void FunctionEntryPointInfo::SetJitMode(const ExecutionMode jitMode)

{

Assert(jitMode == ExecutionMode::SimpleJit || jitMode == ExecutionMode::FullJit);

this->jitMode = jitMode;

}

#endif

void FunctionEntryPointInfo::ReleasePendingWorkItem()

{

// Do this outside of Cleanup since cleanup can be called from the background thread

// We remove any work items corresponding to the function body being reclaimed

// so that the background thread doesn't try to use them. ScriptContext != null => this

// is a function entry point

// In general this is not needed for loop bodies since loop bodies aren't in the low priority

// queue, they should be jitted before the entry point is finalized

if (!this->IsNotScheduled() && !this->IsCleanedUp())

{

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

// On ARM machines, order of writes is not guaranteed while reading data from another processor

// So we need to have a memory barrier here in order to make sure that the work item is consistent

MemoryBarrier();

#endif

CodeGenWorkItem\* workItem = this->GetWorkItem();

if (workItem != nullptr)

{

Assert(this->library != nullptr);

#if ENABLE\_NATIVE\_CODEGEN

TryReleaseNonHiPriWorkItem(this->library->GetScriptContext(), workItem);

#endif

}

}

}

FunctionBody \*FunctionEntryPointInfo::GetFunctionBody() const

{

return functionProxy->GetFunctionBody();

}

void FunctionEntryPointInfo::OnCleanup(bool isShutdown)

{

if (this->IsCodeGenDone())

{

Assert(this->functionProxy->HasBody());

#if ENABLE\_NATIVE\_CODEGEN

if (nullptr != this->inlineeFrameMap)

{

HeapDelete(this->inlineeFrameMap);

this->inlineeFrameMap = nullptr;

}

#endif

if(nativeEntryPointProcessed)

{

JS\_ETW(EtwTrace::LogMethodNativeUnloadEvent(this->functionProxy->GetFunctionBody(), this));

}

FunctionBody\* functionBody = this->functionProxy->GetFunctionBody();

#ifndef TEMP\_DISABLE\_ASMJS

if (this->GetIsTJMode())

{

// release LoopHeaders here if the entrypointInfo is TJ

this->GetFunctionBody()->ReleaseLoopHeaders();

}

#endif

if(functionBody->GetSimpleJitEntryPointInfo() == this)

{

functionBody->SetSimpleJitEntryPointInfo(nullptr);

}

// If we're shutting down, the script context might be gone

if (!isShutdown)

{

ScriptContext\* scriptContext = this->functionProxy->GetScriptContext();

void\* currentCookie = nullptr;

#if ENABLE\_NATIVE\_CODEGEN

// In the debugger case, we might call cleanup after the native code gen that

// allocated this entry point has already shutdown. In that case, the validation

// check below should fail and we should not try to free this entry point

// since it's already been freed

NativeCodeGenerator\* currentNativeCodegen = scriptContext->GetNativeCodeGenerator();

Assert(this->validationCookie != nullptr);

currentCookie = (void\*)currentNativeCodegen;

#endif

if (validationCookie == currentCookie)

{

scriptContext->FreeFunctionEntryPoint((Js::JavascriptMethod)this->GetNativeAddress());

}

}

#ifdef PERF\_COUNTERS

PERF\_COUNTER\_SUB(Code, TotalNativeCodeSize, GetCodeSize());

PERF\_COUNTER\_SUB(Code, FunctionNativeCodeSize, GetCodeSize());

PERF\_COUNTER\_SUB(Code, DynamicNativeCodeSize, GetCodeSize());

#endif

}

this->functionProxy = nullptr;

}

#if ENABLE\_NATIVE\_CODEGEN

void FunctionEntryPointInfo::OnNativeCodeInstallFailure()

{

this->Invalidate(false);

#if ENABLE\_DEBUG\_CONFIG\_OPTIONS

this->SetCleanupReason(CleanupReason::NativeCodeInstallFailure);

#endif

this->Cleanup(false, true /\* capture cleanup stack \*/);

}

void FunctionEntryPointInfo::EnterExpirableCollectMode()

{

this->lastCallsCount = this->callsCount;

// For code that is not jitted yet we don't want to expire since there is nothing to free here

if (this->IsCodeGenPending())

{

this->SetIsObjectUsed();

}

}

void FunctionEntryPointInfo::Invalidate(bool prolongEntryPoint)

{

Assert(!this->functionProxy->IsDeferred());

FunctionBody\* functionBody = this->functionProxy->GetFunctionBody();

Assert(this != functionBody->GetSimpleJitEntryPointInfo());

// We may have got here following OOM in ProcessJitTransferData. Free any data we have

// to reduce the chance of another OOM below.

this->FreeJitTransferData();

FunctionEntryPointInfo\* entryPoint = functionBody->GetDefaultFunctionEntryPointInfo();

if (entryPoint->IsCodeGenPending())

{

OUTPUT\_TRACE(Js::LazyBailoutPhase, L"Skipping creating new entrypoint as one is already pending\n");

}

else

{

class AutoCleanup

{

EntryPointInfo \*entryPointInfo;

public:

AutoCleanup(EntryPointInfo \*entryPointInfo) : entryPointInfo(entryPointInfo)

{

}

void Done()

{

entryPointInfo = nullptr;

}

~AutoCleanup()

{

if (entryPointInfo)

{

entryPointInfo->ResetOnNativeCodeInstallFailure();

}

}

} autoCleanup(this);

entryPoint = functionBody->CreateNewDefaultEntryPoint();

GenerateFunction(functionBody->GetScriptContext()->GetNativeCodeGenerator(), functionBody, /\*function\*/ nullptr);

autoCleanup.Done();

}

this->functionProxy->MapFunctionObjectTypes([&](DynamicType\* type)

{

Assert(type->GetTypeId() == TypeIds\_Function);

ScriptFunctionType\* functionType = (ScriptFunctionType\*)type;

if (functionType->GetEntryPointInfo() == this)

{

functionType->SetEntryPointInfo(entryPoint);

functionType->SetEntryPoint(this->functionProxy->GetDirectEntryPoint(entryPoint));

}

});

if (!prolongEntryPoint)

{

ThreadContext\* threadContext = this->functionProxy->GetScriptContext()->GetThreadContext();

threadContext->QueueFreeOldEntryPointInfoIfInScript(this);

}

}

void FunctionEntryPointInfo::Expire()

{

if (this->lastCallsCount != this->callsCount || !this->nativeEntryPointProcessed || this->IsCleanedUp())

{

return;

}

ThreadContext\* threadContext = this->functionProxy->GetScriptContext()->GetThreadContext();

Assert(!this->functionProxy->IsDeferred());

FunctionBody\* functionBody = this->functionProxy->GetFunctionBody();

FunctionEntryPointInfo \*simpleJitEntryPointInfo = functionBody->GetSimpleJitEntryPointInfo();

const bool expiringSimpleJitEntryPointInfo = simpleJitEntryPointInfo == this;

if(expiringSimpleJitEntryPointInfo)

{

if(functionBody->GetExecutionMode() != ExecutionMode::FullJit)

{

// Don't expire simple JIT code until the transition to full JIT

return;

}

simpleJitEntryPointInfo = nullptr;

functionBody->SetSimpleJitEntryPointInfo(nullptr);

}

try

{

AUTO\_NESTED\_HANDLED\_EXCEPTION\_TYPE(ExceptionType\_OutOfMemory);

FunctionEntryPointInfo\* newEntryPoint = nullptr;

FunctionEntryPointInfo \*const defaultEntryPointInfo = functionBody->GetDefaultFunctionEntryPointInfo();

if(this == defaultEntryPointInfo)

{

if(simpleJitEntryPointInfo)

{

newEntryPoint = simpleJitEntryPointInfo;

functionBody->SetDefaultFunctionEntryPointInfo(

simpleJitEntryPointInfo,

reinterpret\_cast<JavascriptMethod>(newEntryPoint->GetNativeAddress()));

functionBody->SetExecutionMode(ExecutionMode::SimpleJit);

functionBody->ResetSimpleJitLimitAndCallCount();

}

#ifdef ASMJS\_PLAT

else if (functionBody->GetIsAsmJsFunction())

{

// the new entrypoint will be set to interpreter

newEntryPoint = functionBody->CreateNewDefaultEntryPoint();

newEntryPoint->SetIsAsmJSFunction(true);

newEntryPoint->address = AsmJsDefaultEntryThunk;

newEntryPoint->SetModuleAddress(GetModuleAddress());

functionBody->SetIsAsmJsFullJitScheduled(false);

functionBody->SetExecutionMode(functionBody->GetDefaultInterpreterExecutionMode());

this->functionProxy->SetOriginalEntryPoint(AsmJsDefaultEntryThunk);

}

#endif

else

{

newEntryPoint = functionBody->CreateNewDefaultEntryPoint();

functionBody->SetExecutionMode(functionBody->GetDefaultInterpreterExecutionMode());

}

functionBody->TraceExecutionMode("JitCodeExpired");

}

else

{

newEntryPoint = defaultEntryPointInfo;

}

OUTPUT\_TRACE(Js::ExpirableCollectPhase, L"Expiring 0x%p\n", this);

this->functionProxy->MapFunctionObjectTypes([&] (DynamicType\* type)

{

Assert(type->GetTypeId() == TypeIds\_Function);

ScriptFunctionType\* functionType = (ScriptFunctionType\*) type;

if (functionType->GetEntryPointInfo() == this)

{

OUTPUT\_TRACE(Js::ExpirableCollectPhase, L"Type 0x%p uses this entry point- switching to default entry point\n", this);

functionType->SetEntryPointInfo(newEntryPoint);

// we are allowed to replace the entry point on the type only if it's

// directly using the jitted code or a type is referencing this entry point

// but the entry point hasn't been called since the codegen thunk was installed on it

if (functionType->GetEntryPoint() == functionProxy->GetDirectEntryPoint(this) || IsIntermediateCodeGenThunk(functionType->GetEntryPoint()))

{

functionType->SetEntryPoint(this->functionProxy->GetDirectEntryPoint(newEntryPoint));

}

}

});

if(expiringSimpleJitEntryPointInfo)

{

// We could have just created a new entry point info that is using the simple JIT code. An allocation may have

// triggered shortly after, resulting in expiring the simple JIT entry point info. Update any entry point infos

// that are using the simple JIT code, and update the original entry point as necessary as well.

const JavascriptMethod newOriginalEntryPoint =

functionBody->GetDynamicInterpreterEntryPoint()

? static\_cast<JavascriptMethod>(

InterpreterThunkEmitter::ConvertToEntryPoint(functionBody->GetDynamicInterpreterEntryPoint()))

: DefaultEntryThunk;

const JavascriptMethod currentThunk = functionBody->GetScriptContext()->CurrentThunk;

const JavascriptMethod newDirectEntryPoint =

currentThunk == DefaultEntryThunk ? newOriginalEntryPoint : currentThunk;

const JavascriptMethod simpleJitNativeAddress = reinterpret\_cast<JavascriptMethod>(GetNativeAddress());

functionBody->MapEntryPoints([&](const int entryPointIndex, FunctionEntryPointInfo \*const entryPointInfo)

{

if(entryPointInfo != this && entryPointInfo->address == simpleJitNativeAddress)

{

entryPointInfo->address = newDirectEntryPoint;

}

});

if(functionBody->GetOriginalEntryPoint\_Unchecked() == simpleJitNativeAddress)

{

functionBody->SetOriginalEntryPoint(newOriginalEntryPoint);

functionBody->VerifyOriginalEntryPoint();

}

}

threadContext->QueueFreeOldEntryPointInfoIfInScript(this);

}

catch (Js::OutOfMemoryException)

{

// If we can't allocate a new entry point, skip expiring this object

if(expiringSimpleJitEntryPointInfo)

{

simpleJitEntryPointInfo = this;

functionBody->SetSimpleJitEntryPointInfo(this);

}

}

}

#endif

#ifdef PERF\_COUNTERS

void LoopEntryPointInfo::OnRecorded()

{

PERF\_COUNTER\_ADD(Code, TotalNativeCodeSize, GetCodeSize());

PERF\_COUNTER\_ADD(Code, LoopNativeCodeSize, GetCodeSize());

PERF\_COUNTER\_ADD(Code, DynamicNativeCodeSize, GetCodeSize());

}

#endif

FunctionBody \*LoopEntryPointInfo::GetFunctionBody() const

{

return loopHeader->functionBody;

}

//End AsmJs Support

void LoopEntryPointInfo::OnCleanup(bool isShutdown)

{

#ifndef TEMP\_DISABLE\_ASMJS

if (this->IsCodeGenDone() && !this->GetIsTJMode())

#else

if (this->IsCodeGenDone())

#endif

{

JS\_ETW(EtwTrace::LogLoopBodyUnloadEvent(this->loopHeader->functionBody, this->loopHeader, this));

#if ENABLE\_NATIVE\_CODEGEN

if (nullptr != this->inlineeFrameMap)

{

HeapDelete(this->inlineeFrameMap);

this->inlineeFrameMap = nullptr;

}

#endif

if (!isShutdown)

{

void\* currentCookie = nullptr;

ScriptContext\* scriptContext = this->loopHeader->functionBody->GetScriptContext();

#if ENABLE\_NATIVE\_CODEGEN

// In the debugger case, we might call cleanup after the native code gen that

// allocated this entry point has already shutdown. In that case, the validation

// check below should fail and we should not try to free this entry point

// since it's already been freed

NativeCodeGenerator\* currentNativeCodegen = scriptContext->GetNativeCodeGenerator();

Assert(this->validationCookie != nullptr);

currentCookie = (void\*)currentNativeCodegen;

#endif

if (validationCookie == currentCookie)

{

scriptContext->FreeLoopBody((Js::JavascriptMethod)this->GetNativeAddress());

}

}

#ifdef PERF\_COUNTERS

PERF\_COUNTER\_SUB(Code, TotalNativeCodeSize, GetCodeSize());

PERF\_COUNTER\_SUB(Code, LoopNativeCodeSize, GetCodeSize());

PERF\_COUNTER\_SUB(Code, DynamicNativeCodeSize, GetCodeSize());

#endif

}

}

#if ENABLE\_NATIVE\_CODEGEN

void LoopEntryPointInfo::OnNativeCodeInstallFailure()

{

this->ResetOnNativeCodeInstallFailure();

}

#endif

void LoopHeader::Init( FunctionBody \* functionBody )

{

// DisableJIT-TODO: Should this entire class be ifdefed out?

#if ENABLE\_NATIVE\_CODEGEN

this->functionBody = functionBody;

Recycler\* recycler = functionBody->GetScriptContext()->GetRecycler();

// Sync entryPoints changes to etw rundown lock

auto syncObj = functionBody->GetScriptContext()->GetThreadContext()->GetEtwRundownCriticalSection();

this->entryPoints = RecyclerNew(recycler, LoopEntryPointList, recycler, syncObj);

this->CreateEntryPoint();

#endif

}

#if ENABLE\_NATIVE\_CODEGEN

int LoopHeader::CreateEntryPoint()

{

ScriptContext\* scriptContext = this->functionBody->GetScriptContext();

Recycler\* recycler = scriptContext->GetRecycler();

LoopEntryPointInfo\* entryPoint = RecyclerNew(recycler, LoopEntryPointInfo, this, scriptContext->GetLibrary(), scriptContext->GetNativeCodeGenerator());

return this->entryPoints->Add(entryPoint);

}

void LoopHeader::ReleaseEntryPoints()

{

for (int iEntryPoint = 0; iEntryPoint < this->entryPoints->Count(); iEntryPoint++)

{

LoopEntryPointInfo \* entryPoint = this->entryPoints->Item(iEntryPoint);

if (entryPoint != nullptr && entryPoint->IsCodeGenDone())

{

// ReleaseEntryPoints is not called during recycler shutdown scenarios

// We also don't capture the cleanup stack since we've not seen cleanup bugs affect

// loop entry points so far. We can pass true here if this is no longer the case.

entryPoint->Cleanup(false /\* isShutdown \*/, false /\* capture cleanup stack \*/);

this->entryPoints->Item(iEntryPoint, nullptr);

}

}

}

#endif

#if ENABLE\_DEBUG\_CONFIG\_OPTIONS

void FunctionBody::DumpRegStats(FunctionBody \*funcBody)

{

if (funcBody->callCountStats == 0)

{

return;

}

uint loads = funcBody->regAllocLoadCount;

uint stores = funcBody->regAllocStoreCount;

if (Js::Configuration::Global.flags.NormalizeStats)

{

loads /= this->callCountStats;

stores /= this->callCountStats;

}

funcBody->DumpFullFunctionName();

Output::SkipToColumn(55);

Output::Print(L"Calls:%6d Loads:%9d Stores:%9d Total refs:%9d\n", this->callCountStats,

loads, stores, loads + stores);

}

#endif

Js::RegSlot FunctionBody::GetRestParamRegSlot()

{

Js::RegSlot dstRegSlot = GetConstantCount();

if (GetHasImplicitArgIns())

{

dstRegSlot += GetInParamsCount() - 1;

}

return dstRegSlot;

}

uint FunctionBody::GetNumberOfRecursiveCallSites()

{

uint recursiveInlineSpan = 0;

uint recursiveCallSiteInlineInfo = 0;

#if ENABLE\_PROFILE\_INFO

if (this->HasDynamicProfileInfo())

{

recursiveCallSiteInlineInfo = this->dynamicProfileInfo->GetRecursiveInlineInfo();

}

#endif

while (recursiveCallSiteInlineInfo)

{

recursiveInlineSpan += (recursiveCallSiteInlineInfo & 1);

recursiveCallSiteInlineInfo >>= 1;

}

return recursiveInlineSpan;

}

bool FunctionBody::CanInlineRecursively(uint depth, bool tryAggressive)

{

uint recursiveInlineSpan = this->GetNumberOfRecursiveCallSites();

uint minRecursiveInlineDepth = (uint)CONFIG\_FLAG(RecursiveInlineDepthMin);

if (recursiveInlineSpan != this->GetProfiledCallSiteCount() || tryAggressive == false)

{

return depth < minRecursiveInlineDepth;

}

uint maxRecursiveInlineDepth = (uint)CONFIG\_FLAG(RecursiveInlineDepthMax);

uint maxRecursiveBytecodeBudget = (uint)CONFIG\_FLAG(RecursiveInlineThreshold);

uint numberOfAllowedFuncs = maxRecursiveBytecodeBudget / this->m\_byteCodeWithoutLDACount;

uint maxDepth;

if (recursiveInlineSpan == 1)

{

maxDepth = numberOfAllowedFuncs;

}

else

{

maxDepth = (uint)ceil(log((double)((double)numberOfAllowedFuncs) / log((double)recursiveInlineSpan)));

}

maxDepth = maxDepth < minRecursiveInlineDepth ? minRecursiveInlineDepth : maxDepth;

maxDepth = maxDepth < maxRecursiveInlineDepth ? maxDepth : maxRecursiveInlineDepth;

return depth < maxDepth;

}

static const wchar\_t LoopWStr[] = L"Loop";

size\_t FunctionBody::GetLoopBodyName(uint loopNumber, \_Out\_writes\_opt\_z\_(size) wchar\_t\* nameBuffer, \_In\_ size\_t size)

{

const wchar\_t\* functionName = this->GetExternalDisplayName();

size\_t length = wcslen(functionName) + /\*length of largest int32\*/ 10 + \_countof(LoopWStr) + /\*null\*/ 1;

if (size < length || nameBuffer == nullptr)

{

return length;

}

int charsWritten = swprintf\_s(nameBuffer, length, L"%s%s%u", functionName, LoopWStr, loopNumber + 1);

Assert(charsWritten != -1);

return charsWritten + /\*nullptr\*/ 1;

}

}

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

// 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

struct CodeGenWorkItem;

class SourceContextInfo;

class FunctionBailOutRecord;

struct DeferredFunctionStub;

#ifdef DYNAMIC\_PROFILE\_MUTATOR

class DynamicProfileMutator;

class DynamicProfileMutatorImpl;

#endif

#define MAX\_FUNCTION\_BODY\_DEBUG\_STRING\_SIZE 42 //11\*3+8+1

namespace Js

{

#pragma region Class Forward Declarations

class ByteCodeBufferReader;

class ByteCodeBufferBuilder;

class ByteCodeCache;

class ScopeInfo;

class SmallSpanSequence;

struct StatementLocation;

class SmallSpanSequenceIter;

struct StatementData;

struct PropertyIdOnRegSlotsContainer;

struct InlineCache;

struct PolymorphicInlineCache;

struct IsInstInlineCache;

class ScopeObjectChain;

class EntryPointInfo;

class FunctionProxy;

class ParseableFunctionInfo;

class FunctionBody;

class DebuggerScopeProperty;

class DebuggerScope;

class FunctionEntryPointInfo;

#ifndef TEMP\_DISABLE\_ASMJS

class AsmJsFunctionInfo;

class AmsJsModuleInfo;

#endif

class ArrayBuffer;

class FunctionCodeGenRuntimeData;

#pragma endregion

typedef JsUtil::BaseDictionary<Js::PropertyId, const Js::PropertyRecord\*, RecyclerNonLeafAllocator, PowerOf2SizePolicy, DefaultComparer, JsUtil::SimpleDictionaryEntry> PropertyRecordList;

typedef JsUtil::BaseHashSet<void\*, Recycler, PowerOf2SizePolicy> TypeRefSet;

// Definition of scopes such as With, Catch and Block which will be used further in the debugger for additional look-ups.

enum DiagExtraScopesType

{

DiagUnknownScope, // Unknown scope set when deserializing bytecode and the scope is not yet known.

DiagWithScope, // With scope.

DiagCatchScopeDirect, // Catch scope in regslot

DiagCatchScopeInSlot, // Catch scope in slot array

DiagCatchScopeInObject, // Catch scope in scope object

DiagBlockScopeDirect, // Block scope in regslot

DiagBlockScopeInSlot, // Block scope in slot array

DiagBlockScopeInObject, // Block scope in activation object

DiagBlockScopeRangeEnd, // Used to end a block scope range.

};

class PropertyGuard

{

friend class PropertyGuardValidator;

private:

intptr\_t value;

public:

static PropertyGuard\* New(Recycler\* recycler) { return RecyclerNewLeaf(recycler, Js::PropertyGuard); }

PropertyGuard() : value(1) {}

PropertyGuard(intptr\_t value) : value(value) { Assert(this->value != 0); }

inline static size\_t const GetSizeOfValue() { return sizeof(((PropertyGuard\*)0)->value); }

inline static size\_t const GetOffsetOfValue() { return offsetof(PropertyGuard, value); }

intptr\_t GetValue() const { return this->value; }

bool IsValid() { return this->value != 0; }

void SetValue(intptr\_t value) { Assert(value != 0); this->value = value; }

intptr\_t const\* GetAddressOfValue() { return &this->value; }

void Invalidate() { this->value = 0; }

};

class PropertyGuardValidator

{

// Required by EquivalentTypeGuard::SetType.

CompileAssert(offsetof(PropertyGuard, value) == 0);

CompileAssert(offsetof(ConstructorCache, guard.value) == offsetof(PropertyGuard, value));

};

class JitIndexedPropertyGuard : public Js::PropertyGuard

{

private:

int index;

public:

JitIndexedPropertyGuard(intptr\_t value, int index):

Js::PropertyGuard(value), index(index) {}

int GetIndex() const { return this->index; }

};

class JitTypePropertyGuard : public Js::JitIndexedPropertyGuard

{

public:

JitTypePropertyGuard(Js::Type\* type, int index):

JitIndexedPropertyGuard(reinterpret\_cast<intptr\_t>(type), index) {}

Js::Type\* GetType() const { return reinterpret\_cast<Js::Type\*>(this->GetValue()); }

};

struct TypeGuardTransferEntry

{

PropertyId propertyId;

JitIndexedPropertyGuard\* guards[0];

TypeGuardTransferEntry(): propertyId(Js::Constants::NoProperty) {}

};

class FakePropertyGuardWeakReference: public RecyclerWeakReference<Js::PropertyGuard>

{

public:

static FakePropertyGuardWeakReference\* New(Recycler\* recycler, Js::PropertyGuard\* guard)

{

Assert(guard != nullptr);

return RecyclerNewLeaf(recycler, Js::FakePropertyGuardWeakReference, guard);

}

FakePropertyGuardWeakReference(const Js::PropertyGuard\* guard)

{

this->strongRef = (char\*)guard;

this->strongRefHeapBlock = &CollectedRecyclerWeakRefHeapBlock::Instance;

}

void Zero()

{

Assert(this->strongRef != nullptr);

this->strongRef = nullptr;

}

};

struct CtorCacheGuardTransferEntry

{

PropertyId propertyId;

ConstructorCache\* caches[0];

CtorCacheGuardTransferEntry(): propertyId(Js::Constants::NoProperty) {}

};

#define EQUIVALENT\_TYPE\_CACHE\_SIZE (8)

struct EquivalentTypeCache

{

Js::Type\* types[EQUIVALENT\_TYPE\_CACHE\_SIZE];

PropertyGuard \*guard;

TypeEquivalenceRecord record;

uint nextEvictionVictim;

bool isLoadedFromProto;

bool hasFixedValue;

EquivalentTypeCache(): nextEvictionVictim(EQUIVALENT\_TYPE\_CACHE\_SIZE) {}

bool ClearUnusedTypes(Recycler \*recycler);

void SetGuard(PropertyGuard \*theGuard) { this->guard = theGuard; }

void SetIsLoadedFromProto() { this->isLoadedFromProto = true; }

bool IsLoadedFromProto() const { return this->isLoadedFromProto; }

void SetHasFixedValue() { this->hasFixedValue = true; }

bool HasFixedValue() const { return this->hasFixedValue; }

};

class JitEquivalentTypeGuard : public JitIndexedPropertyGuard

{

// This pointer is allocated from background thread first, and then transferred to recycler,

// so as to keep the cached types alive.

EquivalentTypeCache\* cache;

uint32 objTypeSpecFldId;

public:

JitEquivalentTypeGuard(Type\* type, int index, uint32 objTypeSpecFldId):

JitIndexedPropertyGuard(reinterpret\_cast<intptr\_t>(type), index), cache(nullptr), objTypeSpecFldId(objTypeSpecFldId) {}

Js::Type\* GetType() const { return reinterpret\_cast<Js::Type\*>(this->GetValue()); }

void SetType(const Js::Type\* type)

{

this->SetValue(reinterpret\_cast<intptr\_t>(type));

}

uint32 GetObjTypeSpecFldId() const

{

return this->objTypeSpecFldId;

}

Js::EquivalentTypeCache\* GetCache() const

{

return this->cache;

}

void SetCache(Js::EquivalentTypeCache\* cache)

{

this->cache = cache;

}

};

#pragma region Inline Cache Info class declarations

class PolymorphicCacheUtilizationArray

{

private:

byte \*utilArray;

public:

PolymorphicCacheUtilizationArray()

: utilArray(nullptr)

{

}

void EnsureUtilArray(Recycler \* const recycler, Js::FunctionBody \* functionBody);

void SetUtil(Js::FunctionBody\* functionBody, uint index, byte util);

byte GetUtil(Js::FunctionBody\* functionBody, uint index);

};

class PolymorphicInlineCacheInfo sealed

{

private:

InlineCachePointerArray<PolymorphicInlineCache> polymorphicInlineCaches;

PolymorphicCacheUtilizationArray polymorphicCacheUtilizationArray;

FunctionBody \* functionBody;

public:

PolymorphicInlineCacheInfo(FunctionBody \* functionBody)

: functionBody(functionBody)

{

}

InlineCachePointerArray<PolymorphicInlineCache> \* GetPolymorphicInlineCaches() { return &polymorphicInlineCaches; }

PolymorphicCacheUtilizationArray \* GetUtilArray() { return &polymorphicCacheUtilizationArray; }

FunctionBody \* GetFunctionBody() { return functionBody; }

};

class EntryPointPolymorphicInlineCacheInfo sealed

{

private:

PolymorphicInlineCacheInfo selfInfo;

SListBase<PolymorphicInlineCacheInfo\*> inlineeInfo;

static void SetPolymorphicInlineCache(PolymorphicInlineCacheInfo \* polymorphicInlineCacheInfo, FunctionBody \* functionBody, uint index, PolymorphicInlineCache \* polymorphicInlineCache, byte polyCacheUtil);

public:

EntryPointPolymorphicInlineCacheInfo(FunctionBody \* functionBody)

: selfInfo(functionBody)

{

}

PolymorphicInlineCacheInfo \* GetSelfInfo() { return &selfInfo; }

PolymorphicInlineCacheInfo \* EnsureInlineeInfo(Recycler \* recycler, FunctionBody \* inlineeFunctionBody);

PolymorphicInlineCacheInfo \* GetInlineeInfo(FunctionBody \* inlineeFunctionBody);

void SetPolymorphicInlineCache(FunctionBody \* functionBody, uint index, PolymorphicInlineCache \* polymorphicInlineCache, bool isInlinee, byte polyCacheUtil);

template <class Fn>

void MapInlinees(Fn fn)

{

SListBase<PolymorphicInlineCacheInfo\*>::Iterator iter(&inlineeInfo);

while (iter.Next())

{

fn(iter.Data());

}

}

};

#pragma endregion

#ifdef FIELD\_ACCESS\_STATS

struct FieldAccessStats

{

uint totalInlineCacheCount;

uint noInfoInlineCacheCount;

uint monoInlineCacheCount;

uint emptyMonoInlineCacheCount;

uint polyInlineCacheCount;

uint nullPolyInlineCacheCount;

uint emptyPolyInlineCacheCount;

uint ignoredPolyInlineCacheCount;

uint highUtilPolyInlineCacheCount;

uint lowUtilPolyInlineCacheCount;

uint equivPolyInlineCacheCount;

uint nonEquivPolyInlineCacheCount;

uint disabledPolyInlineCacheCount;

uint clonedMonoInlineCacheCount;

uint clonedPolyInlineCacheCount;

FieldAccessStats() :

totalInlineCacheCount(0), noInfoInlineCacheCount(0), monoInlineCacheCount(0), emptyMonoInlineCacheCount(0),

polyInlineCacheCount(0), nullPolyInlineCacheCount(0), emptyPolyInlineCacheCount(0), ignoredPolyInlineCacheCount(0),

highUtilPolyInlineCacheCount(0), lowUtilPolyInlineCacheCount(0),

equivPolyInlineCacheCount(0), nonEquivPolyInlineCacheCount(0), disabledPolyInlineCacheCount(0),

clonedMonoInlineCacheCount(0), clonedPolyInlineCacheCount(0) {}

void Add(FieldAccessStats\* other);

};

typedef FieldAccessStats\* FieldAccessStatsPtr;

#else

typedef void\* FieldAccessStatsPtr;

#endif

#pragma region Entry point class declarations

class ProxyEntryPointInfo: public ExpirableObject

{

public:

// These are public because we don't manage them nor their consistency;

// the user of this class does.

void \* address;

ProxyEntryPointInfo(void\* address, ThreadContext\* context = nullptr):

ExpirableObject(context),

address(address)

{

}

static DWORD GetAddressOffset() { return offsetof(ProxyEntryPointInfo, address); }

virtual void Expire()

{

AssertMsg(false, "Expire called on object that doesn't support expiration");

}

virtual void EnterExpirableCollectMode()

{

AssertMsg(false, "EnterExpirableCollectMode called on object that doesn't support expiration");

}

virtual bool IsFunctionEntryPointInfo() const { return false; }

};

// Not thread safe.

// Note that instances of this class are read from and written to from the

// main and JIT threads.

class EntryPointInfo : public ProxyEntryPointInfo

{

private:

enum State : BYTE

{

NotScheduled, // code gen has not been scheduled

CodeGenPending, // code gen job has been scheduled

CodeGenQueued, // code gen has been queued and all the code gen data has been gathered.

CodeGenRecorded, // backend completed, but job still pending

CodeGenDone, // code gen job successfully completed

JITCapReached, // workitem created but JIT cap reached

PendingCleanup, // workitem needs to be cleaned up but couldn't for some reason- it'll be cleaned up at the next opportunity

CleanedUp // the entry point has been cleaned up

};

#if ENABLE\_NATIVE\_CODEGEN

class JitTransferData

{

friend EntryPointInfo;

private:

TypeRefSet\* jitTimeTypeRefs;

void\*\* runtimeTypeRefs;

int runtimeTypeRefCount;

int propertyGuardCount;

// This is a dynamically sized array of dynamically sized TypeGuardTransferEntries. It's heap allocated by the JIT

// thread and lives until entry point is installed, at which point it is explicitly freed.

TypeGuardTransferEntry\* propertyGuardsByPropertyId;

size\_t propertyGuardsByPropertyIdPlusSize;

// This is a dynamically sized array of dynamically sized CtorCacheGuardTransferEntry. It's heap allocated by the JIT

// thread and lives until entry point is installed, at which point it is explicitly freed.

CtorCacheGuardTransferEntry\* ctorCacheGuardsByPropertyId;

size\_t ctorCacheGuardsByPropertyIdPlusSize;

int equivalentTypeGuardCount;

int lazyBailoutPropertyCount;

// This is a dynamically sized array of JitEquivalentTypeGuards. It's heap allocated by the JIT thread and lives

// until entry point is installed, at which point it is explicitly freed. We need it during installation so as to

// swap the cache associated with each guard from the heap to the recycler (so the types in the cache are kept alive).

JitEquivalentTypeGuard\*\* equivalentTypeGuards;

Js::PropertyId\* lazyBailoutProperties;

NativeCodeData\* data;

bool falseReferencePreventionBit;

bool isReady;

public:

JitTransferData():

jitTimeTypeRefs(nullptr), runtimeTypeRefCount(0), runtimeTypeRefs(nullptr),

propertyGuardCount(0), propertyGuardsByPropertyId(nullptr), propertyGuardsByPropertyIdPlusSize(0),

ctorCacheGuardsByPropertyId(nullptr), ctorCacheGuardsByPropertyIdPlusSize(0),

equivalentTypeGuardCount(0), equivalentTypeGuards(nullptr), data(nullptr),

falseReferencePreventionBit(true), isReady(false), lazyBailoutProperties(nullptr), lazyBailoutPropertyCount(0){}

void AddJitTimeTypeRef(void\* typeRef, Recycler\* recycler);

int GetRuntimeTypeRefCount() { return this->runtimeTypeRefCount; }

void\*\* GetRuntimeTypeRefs() { return this->runtimeTypeRefs; }

void SetRuntimeTypeRefs(void\*\* runtimeTypeRefs, int count) { this->runtimeTypeRefs = runtimeTypeRefs; this->runtimeTypeRefCount = count; }

JitEquivalentTypeGuard\*\* GetEquivalentTypeGuards() const { return this->equivalentTypeGuards; }

void SetEquivalentTypeGuards(JitEquivalentTypeGuard\*\* guards, int count)

{

this->equivalentTypeGuardCount = count;

this->equivalentTypeGuards = guards;

}

void SetLazyBailoutProperties(Js::PropertyId\* properties, int count)

{

this->lazyBailoutProperties = properties;

this->lazyBailoutPropertyCount = count;

}

bool GetIsReady() { return this->isReady; }

void SetIsReady() { this->isReady = true; }

private:

void EnsureJitTimeTypeRefs(Recycler\* recycler);

};

NativeCodeData \* data;

CodeGenNumberChunk \* numberChunks;

SmallSpanSequence \*nativeThrowSpanSequence;

typedef JsUtil::BaseHashSet<RecyclerWeakReference<FunctionBody>\*, Recycler, PowerOf2SizePolicy> WeakFuncRefSet;

WeakFuncRefSet \*weakFuncRefSet;

// Need to keep strong references to the guards here so they don't get collected while the entry point is alive.

typedef JsUtil::BaseDictionary<Js::PropertyId, PropertyGuard\*, Recycler, PowerOf2SizePolicy> SharedPropertyGuardDictionary;

SharedPropertyGuardDictionary\* sharedPropertyGuards;

typedef JsUtil::List<LazyBailOutRecord, HeapAllocator> BailOutRecordMap;

BailOutRecordMap\* bailoutRecordMap;

// This array holds fake weak references to type property guards. We need it to zero out the weak references when the

// entry point is finalized and the guards are about to be freed. Otherwise, if one of the guards was to be invalidated

// from the thread context, we would AV trying to access freed memory. Note that the guards themselves are allocated by

// NativeCodeData::Allocator and are kept alive by the data field. The weak references are recycler allocated, and so

// the array must be recycler allocated also, so that the recycler doesn't collect the weak references.

FakePropertyGuardWeakReference\*\* propertyGuardWeakRefs;

EquivalentTypeCache\* equivalentTypeCaches;

EntryPointInfo \*\* registeredEquivalentTypeCacheRef;

int propertyGuardCount;

int equivalentTypeCacheCount;

#endif

CodeGenWorkItem \* workItem;

void \* nativeAddress;

ptrdiff\_t codeSize;

bool isAsmJsFunction; // true if entrypoint is for asmjs function

uintptr mModuleAddress; //asm Module address

#ifdef FIELD\_ACCESS\_STATS

FieldAccessStatsPtr fieldAccessStats;

#endif

protected:

JavascriptLibrary\* library;

#if ENABLE\_NATIVE\_CODEGEN

typedef JsUtil::List<NativeOffsetInlineeFramePair, HeapAllocator> InlineeFrameMap;

InlineeFrameMap\* inlineeFrameMap;

#endif

#ifdef ENABLE\_DEBUG\_CONFIG\_OPTIONS

StackBackTrace\* cleanupStack;

#endif

public:

enum CleanupReason

{

NotCleanedUp,

CodeGenFailedOOM,

CodeGenFailedStackOverflow,

CodeGenFailedAborted,

NativeCodeInstallFailure,

CleanUpForFinalize

};

uint frameHeight;

private:

#if ENABLE\_NATIVE\_CODEGEN

typedef SListCounted<ConstructorCache\*, Recycler> ConstructorCacheList;

ConstructorCacheList\* constructorCaches;

EntryPointPolymorphicInlineCacheInfo \* polymorphicInlineCacheInfo;

// This field holds any recycler allocated references that must be kept alive until

// we install the entry point. It is freed at that point, so anything that must survive

// until the EntryPointInfo itself goes away, must be copied somewhere else.

JitTransferData\* jitTransferData;

// If we pin types this array contains strong references to types, otherwise it holds weak references.

void \*\*runtimeTypeRefs;

uint32 pendingPolymorphicCacheState;

#endif

State state; // Single state member so users can query state w/o a lock

#if ENABLE\_DEBUG\_CONFIG\_OPTIONS

CleanupReason cleanupReason;

#endif

BYTE pendingInlinerVersion;

bool isLoopBody;

bool hasJittedStackClosure;

#if ENABLE\_NATIVE\_CODEGEN

ImplicitCallFlags pendingImplicitCallFlags;

#endif

public:

virtual void Finalize(bool isShutdown) override;

virtual bool IsFunctionEntryPointInfo() const override { return true; }

protected:

EntryPointInfo(void\* address, JavascriptLibrary\* library, void\* validationCookie, ThreadContext\* context = nullptr, bool isLoopBody = false) :

ProxyEntryPointInfo(address, context),

#if ENABLE\_NATIVE\_CODEGEN

nativeThrowSpanSequence(nullptr), workItem(nullptr), weakFuncRefSet(nullptr),

jitTransferData(nullptr), sharedPropertyGuards(nullptr), propertyGuardCount(0), propertyGuardWeakRefs(nullptr),

equivalentTypeCacheCount(0), equivalentTypeCaches(nullptr), constructorCaches(nullptr), state(NotScheduled), data(nullptr),

numberChunks(nullptr), polymorphicInlineCacheInfo(nullptr), runtimeTypeRefs(nullptr),

isLoopBody(isLoopBody), hasJittedStackClosure(false), registeredEquivalentTypeCacheRef(nullptr), bailoutRecordMap(nullptr),

#endif

library(library), codeSize(0), nativeAddress(nullptr), isAsmJsFunction(false), validationCookie(validationCookie)

#ifdef ENABLE\_DEBUG\_CONFIG\_OPTIONS

, cleanupStack(nullptr)

, cleanupReason(NotCleanedUp)

#endif

#if DBG\_DUMP | defined(VTUNE\_PROFILING)

, nativeOffsetMaps(&HeapAllocator::Instance)

#endif

#ifdef FIELD\_ACCESS\_STATS

, fieldAccessStats(nullptr)

#endif

{}

virtual void ReleasePendingWorkItem() {};

virtual void OnCleanup(bool isShutdown) = 0;

#ifdef PERF\_COUNTERS

virtual void OnRecorded() = 0;

#endif

private:

State GetState() const

{

Assert(this->state >= NotScheduled && this->state <= CleanedUp);

return this->state;

}

public:

ScriptContext\* GetScriptContext();

virtual FunctionBody \*GetFunctionBody() const = 0;

#if ENABLE\_NATIVE\_CODEGEN

EntryPointPolymorphicInlineCacheInfo \* EnsurePolymorphicInlineCacheInfo(Recycler \* recycler, FunctionBody \* functionBody);

EntryPointPolymorphicInlineCacheInfo \* GetPolymorphicInlineCacheInfo() { return polymorphicInlineCacheInfo; }

JitTransferData\* GetJitTransferData() { return this->jitTransferData; }

JitTransferData\* EnsureJitTransferData(Recycler\* recycler);

#ifdef FIELD\_ACCESS\_STATS

FieldAccessStats\* GetFieldAccessStats() { return this->fieldAccessStats; }

FieldAccessStats\* EnsureFieldAccessStats(Recycler\* recycler);

#endif

void PinTypeRefs(ScriptContext\* scriptContext);

void InstallGuards(ScriptContext\* scriptContext);

#endif

void Cleanup(bool isShutdown, bool captureCleanupStack);

#ifdef ENABLE\_DEBUG\_CONFIG\_OPTIONS

void CaptureCleanupStackTrace();

#endif

bool IsNotScheduled() const

{

return this->GetState() == NotScheduled;

}

bool IsCodeGenPending() const

{

return this->GetState() == CodeGenPending;

}

bool IsNativeCode() const

{

#if ENABLE\_NATIVE\_CODEGEN

return this->GetState() == CodeGenRecorded ||

this->GetState() == CodeGenDone;

#else

return false;

#endif

}

bool IsCodeGenDone() const

{

#if ENABLE\_NATIVE\_CODEGEN

return this->GetState() == CodeGenDone;

#else

return false;

#endif

}

bool IsCodeGenQueued() const

{

#if ENABLE\_NATIVE\_CODEGEN

return this->GetState() == CodeGenQueued;

#else

return false;

#endif

}

bool IsJITCapReached() const

{

#if ENABLE\_NATIVE\_CODEGEN

return this->GetState() == JITCapReached;

#else

return false;

#endif

}

bool IsCleanedUp() const

{

return this->GetState() == CleanedUp;

}

bool IsPendingCleanup() const

{

return this->GetState() == PendingCleanup;

}

void SetPendingCleanup()

{

this->state = PendingCleanup;

}

#if ENABLE\_DEBUG\_CONFIG\_OPTIONS

void SetCleanupReason(CleanupReason reason)

{

this->cleanupReason = reason;

}

#endif

bool IsLoopBody() const

{

return this->isLoopBody;

}

#if ENABLE\_NATIVE\_CODEGEN

bool HasJittedStackClosure() const

{

return this->hasJittedStackClosure;

}

void SetHasJittedStackClosure()

{

this->hasJittedStackClosure = true;

}

#endif

#ifndef TEMP\_DISABLE\_ASMJS

void SetModuleAddress(uintptr moduleAddress)

{

Assert(this->GetIsAsmJSFunction());

Assert(moduleAddress);

mModuleAddress = moduleAddress;

}

uintptr GetModuleAddress()const

{

Assert(this->GetIsAsmJSFunction());

Assert(mModuleAddress); // module address should not be null

return mModuleAddress;

}

#endif

void Reset(bool resetStateToNotScheduled = true);

#if ENABLE\_NATIVE\_CODEGEN

void SetCodeGenPending(CodeGenWorkItem \* workItem)

{

Assert(this->GetState() == NotScheduled || this->GetState() == CleanedUp);

Assert(workItem != nullptr);

this->workItem = workItem;

this->state = CodeGenPending;

}

void SetCodeGenPending()

{

Assert(this->GetState() == CodeGenQueued);

this->state = CodeGenPending;

}

void SetCodeGenQueued()

{

Assert(this->GetState() == CodeGenPending);

this->state = CodeGenQueued;

}

void RevertToNotScheduled()

{

Assert(this->GetState() == CodeGenPending);

Assert(this->workItem != nullptr);

this->workItem = nullptr;

this->state = NotScheduled;

}

void SetCodeGenPendingWithStackAllocatedWorkItem()

{

Assert(this->GetState() == NotScheduled || this->GetState() == CleanedUp);

this->workItem = nullptr;

this->state = CodeGenPending;

}

void SetCodeGenRecorded(void \* nativeAddress, ptrdiff\_t codeSize,

NativeCodeData \* data, NativeCodeData \* transferData, CodeGenNumberChunk \* numberChunks)

{

Assert(this->GetState() == CodeGenQueued);

Assert(nativeAddress != nullptr);

Assert(codeSize > 0);

Assert(this->jitTransferData != nullptr || transferData == nullptr);

this->nativeAddress = (void \*)nativeAddress;

this->codeSize = codeSize;

this->data = data;

if (transferData != nullptr)

{

this->jitTransferData->data = transferData;

}

this->numberChunks = numberChunks;

this->state = CodeGenRecorded;

#ifdef PERF\_COUNTERS

this->OnRecorded();

#endif

}

void SetCodeGenDone()

{

Assert(this->GetState() == CodeGenRecorded);

this->state = CodeGenDone;

this->workItem = nullptr;

}

void SetJITCapReached()

{

Assert(this->GetState() == CodeGenQueued);

this->state = JITCapReached;

this->workItem = nullptr;

}

SmallSpanSequence\* GetNativeThrowSpanSequence() const

{

Assert(this->GetState() != NotScheduled);

Assert(this->GetState() != CleanedUp);

return nativeThrowSpanSequence;

}

void SetNativeThrowSpanSequence(SmallSpanSequence\* seq)

{

Assert(this->GetState() == CodeGenQueued);

Assert(this->nativeThrowSpanSequence == nullptr);

nativeThrowSpanSequence = seq;

}

bool IsInNativeAddressRange(DWORD\_PTR codeAddress) {

return (IsNativeCode() &&

codeAddress >= GetNativeAddress() &&

codeAddress < GetNativeAddress() + GetCodeSize());

}

#endif

DWORD\_PTR GetNativeAddress() const

{

// need the assert to skip for asmjsFunction as nativeAddress can be interpreter too for asmjs

Assert(this->GetState() == CodeGenRecorded || this->GetState() == CodeGenDone || this->isAsmJsFunction);

return (DWORD\_PTR)this->nativeAddress;

}

ptrdiff\_t GetCodeSize() const

{

Assert(this->GetState() == CodeGenRecorded || this->GetState() == CodeGenDone);

return codeSize;

}

CodeGenWorkItem \* GetWorkItem() const

{

State state = this->GetState();

Assert(state != NotScheduled || this->workItem == nullptr);

Assert(state == CleanedUp && this->workItem == nullptr ||

state != CleanedUp);

if (state == PendingCleanup)

{

return nullptr;

}

return this->workItem;

}

#ifndef TEMP\_DISABLE\_ASMJS

// set code size, used by TJ to set the code size

void SetCodeSize(ptrdiff\_t size)

{

Assert(isAsmJsFunction);

this->codeSize = size;

}

void SetNativeAddress(void\* address)

{

Assert(isAsmJsFunction);

this->nativeAddress = address;

}

void SetIsAsmJSFunction(bool value)

{

this->isAsmJsFunction = value;

}

#endif

bool GetIsAsmJSFunction()const

{

return this->isAsmJsFunction;

}

#ifndef TEMP\_DISABLE\_ASMJS

void SetTJCodeGenDone()

{

Assert(isAsmJsFunction);

this->state = CodeGenDone;

this->workItem = nullptr;

}

#endif

#if ENABLE\_NATIVE\_CODEGEN

void AddWeakFuncRef(RecyclerWeakReference<FunctionBody> \*weakFuncRef, Recycler \*recycler);

WeakFuncRefSet \*EnsureWeakFuncRefSet(Recycler \*recycler);

void EnsureIsReadyToCall();

void ProcessJitTransferData();

void ResetOnNativeCodeInstallFailure();

virtual void OnNativeCodeInstallFailure() = 0;

Js::PropertyGuard\* RegisterSharedPropertyGuard(Js::PropertyId propertyId, ScriptContext\* scriptContext);

bool HasSharedPropertyGuards() { return this->sharedPropertyGuards != nullptr; }

bool HasSharedPropertyGuard(Js::PropertyId propertyId);

bool TryGetSharedPropertyGuard(Js::PropertyId propertyId, Js::PropertyGuard\*& guard);

void RecordTypeGuards(int propertyGuardCount, TypeGuardTransferEntry\* typeGuardTransferRecord, size\_t typeGuardTransferPlusSize);

void RecordCtorCacheGuards(CtorCacheGuardTransferEntry\* ctorCacheTransferRecord, size\_t ctorCacheTransferPlusSize);

void FreePropertyGuards();

void FreeJitTransferData();

void RegisterEquivalentTypeCaches();

void UnregisterEquivalentTypeCaches();

bool ClearEquivalentTypeCaches();

void RegisterConstructorCache(Js::ConstructorCache\* constructorCache, Recycler\* recycler);

uint GetConstructorCacheCount() const { return this->constructorCaches != nullptr ? this->constructorCaches->Count() : 0; }

uint32 GetPendingPolymorphicCacheState() const { return this->pendingPolymorphicCacheState; }

void SetPendingPolymorphicCacheState(uint32 state) { this->pendingPolymorphicCacheState = state; }

BYTE GetPendingInlinerVersion() const { return this->pendingInlinerVersion; }

void SetPendingInlinerVersion(BYTE version) { this->pendingInlinerVersion = version; }

ImplicitCallFlags GetPendingImplicitCallFlags() const { return this->pendingImplicitCallFlags; }

void SetPendingImplicitCallFlags(ImplicitCallFlags flags) { this->pendingImplicitCallFlags = flags; }

virtual void Invalidate(bool prolongEntryPoint) { Assert(false); }

void RecordBailOutMap(JsUtil::List<LazyBailOutRecord, ArenaAllocator>\* bailoutMap);

void RecordInlineeFrameMap(JsUtil::List<NativeOffsetInlineeFramePair, ArenaAllocator>\* tempInlineeFrameMap);

InlineeFrameRecord\* FindInlineeFrame(void\* returnAddress);

bool HasInlinees() { return this->frameHeight > 0; }

void DoLazyBailout(BYTE\*\* addressOfeturnAddress, Js::FunctionBody\* functionBody, const PropertyRecord\* propertyRecord);

#endif

#if DBG\_DUMP

public:

#else if defined(VTUNE\_PROFILING)

private:

#endif

#if DBG\_DUMP | defined(VTUNE\_PROFILING)

// NativeOffsetMap is public for DBG\_DUMP, private for VTUNE\_PROFILING

struct NativeOffsetMap

{

uint32 statementIndex;

regex::Interval nativeOffsetSpan;

};

private:

JsUtil::List<NativeOffsetMap, HeapAllocator> nativeOffsetMaps;

public:

void RecordNativeMap(uint32 offset, uint32 statementIndex);

int GetNativeOffsetMapCount() const;

#endif

#if DBG\_DUMP && ENABLE\_NATIVE\_CODEGEN

void DumpNativeOffsetMaps();

void DumpNativeThrowSpanSequence();

NativeOffsetMap\* GetNativeOffsetMap(int index)

{

Assert(index >= 0);

Assert(index < GetNativeOffsetMapCount());

return &nativeOffsetMaps.Item(index);

}

#endif

#ifdef VTUNE\_PROFILING

public:

uint PopulateLineInfo(void\* pLineInfo, FunctionBody\* body);

#endif

protected:

void\* validationCookie;

};

class FunctionEntryPointInfo : public EntryPointInfo

{

public:

FunctionProxy \* functionProxy;

FunctionEntryPointInfo\* nextEntryPoint;

// The offset on the native stack, from which the locals are located (Populated at RegAlloc phase). Used for debug purpose.

int32 localVarSlotsOffset;

// The offset which stores that any of the locals are changed from the debugger.

int32 localVarChangedOffset;

uint entryPointIndex;

uint8 callsCount;

uint8 lastCallsCount;

bool nativeEntryPointProcessed;

private:

ExecutionMode jitMode;

FunctionEntryPointInfo\* mOldFunctionEntryPointInfo; // strong ref to oldEntryPointInfo(Int or TJ) in asm to ensure we don't collect it before JIT is completed

bool mIsTemplatizedJitMode; // true only if in TJ mode, used only for debugging

public:

static const uint8 GetDecrCallCountPerBailout()

{

return (100 / (uint8)CONFIG\_FLAG(RejitRatioLimit)) + 1;

}

FunctionEntryPointInfo(FunctionProxy \* functionInfo, void \* address, ThreadContext\* context, void\* validationCookie);

#ifndef TEMP\_DISABLE\_ASMJS

//AsmJS Support

void SetOldFunctionEntryPointInfo(FunctionEntryPointInfo\* entrypointInfo);

FunctionEntryPointInfo\* GetOldFunctionEntryPointInfo()const;

void SetIsTJMode(bool value);

bool GetIsTJMode()const;

//End AsmJS Support

#endif

virtual FunctionBody \*GetFunctionBody() const override;

#if ENABLE\_NATIVE\_CODEGEN

ExecutionMode GetJitMode() const;

void SetJitMode(const ExecutionMode jitMode);

virtual void Invalidate(bool prolongEntryPoint) override;

virtual void Expire() override;

virtual void EnterExpirableCollectMode() override;

virtual void OnNativeCodeInstallFailure() override;

#endif

virtual void OnCleanup(bool isShutdown) override;

virtual void ReleasePendingWorkItem() override;

#ifdef PERF\_COUNTERS

virtual void OnRecorded() override;

#endif

};

class LoopEntryPointInfo : public EntryPointInfo

{

public:

LoopHeader\* loopHeader;

LoopEntryPointInfo(LoopHeader\* loopHeader, Js::JavascriptLibrary\* library, void\* validationCookie) :

loopHeader(loopHeader), mIsTemplatizedJitMode(false),EntryPointInfo(nullptr, library, validationCookie, /\*threadContext\*/ nullptr, /\*isLoopBody\*/ true)

#ifdef BGJIT\_STATS

,used(false)

#endif

{ }

virtual FunctionBody \*GetFunctionBody() const override;

virtual void OnCleanup(bool isShutdown) override;

#if ENABLE\_NATIVE\_CODEGEN

virtual void OnNativeCodeInstallFailure() override;

#endif

#ifndef TEMP\_DISABLE\_ASMJS

void SetIsTJMode(bool value)

{

Assert(this->GetIsAsmJSFunction());

mIsTemplatizedJitMode = value;

}

bool GetIsTJMode()const

{

return mIsTemplatizedJitMode;

};

#endif

#ifdef PERF\_COUNTERS

virtual void OnRecorded() override;

#endif

#ifdef BGJIT\_STATS

bool IsUsed() const

{

return this->used;

}

void MarkAsUsed()

{

this->used = true;

}

#endif

private:

#ifdef BGJIT\_STATS

bool used;

#endif

bool mIsTemplatizedJitMode;

};

typedef RecyclerWeakReference<FunctionEntryPointInfo> FunctionEntryPointWeakRef;

typedef SynchronizableList<FunctionEntryPointWeakRef\*, JsUtil::List<FunctionEntryPointWeakRef\*>> FunctionEntryPointList;

typedef SynchronizableList<LoopEntryPointInfo\*, JsUtil::List<LoopEntryPointInfo\*>> LoopEntryPointList;

#pragma endregion

struct LoopHeader

{

private:

LoopEntryPointList\* entryPoints;

public:

uint startOffset;

uint endOffset;

uint interpretCount;

uint profiledLoopCounter;

bool isNested;

bool isInTry;

FunctionBody \* functionBody;

#if DBG\_DUMP

uint nativeCount;

#endif

static const uint NoLoop = (uint)-1;

static const uint GetOffsetOfProfiledLoopCounter() { return offsetof(LoopHeader, profiledLoopCounter); }

static const uint GetOffsetOfInterpretCount() { return offsetof(LoopHeader, interpretCount); }

bool Contains(Js::LoopHeader \* loopHeader) const

{

return (this->startOffset <= loopHeader->startOffset && loopHeader->endOffset <= this->endOffset);

}

bool Contains(uint offset) const

{

return this->startOffset <= offset && offset < this->endOffset;

}

void \* GetCurrentEntryPoint() const

{

LoopEntryPointInfo \* entryPoint = GetCurrentEntryPointInfo();

if (entryPoint != nullptr)

{

return this->entryPoints->Item(this->GetCurrentEntryPointIndex())->address;

}

return nullptr;

}

LoopEntryPointInfo \* GetCurrentEntryPointInfo() const

{

Assert(this->entryPoints->Count() > 0);

return this->entryPoints->Item(this->GetCurrentEntryPointIndex());

}

uint GetByteCodeCount()

{

return (endOffset - startOffset);

}

int GetCurrentEntryPointIndex() const

{

return this->entryPoints->Count() - 1;

}

LoopEntryPointInfo \* GetEntryPointInfo(int index) const

{

return this->entryPoints->Item(index);

}

template <class Fn>

void MapEntryPoints(Fn fn) const

{

if (this->entryPoints) // ETW rundown may call this before entryPoints initialization

{

this->entryPoints->Map([&](int index, LoopEntryPointInfo \* entryPoint)

{

if (entryPoint != nullptr)

{

fn(index, entryPoint);

}

});

}

}

template <class DebugSite, class Fn>

HRESULT MapEntryPoints(DebugSite site, Fn fn) const // external debugging version

{

return Map(site, this->entryPoints, [&](int index, LoopEntryPointInfo \* entryPoint)

{

if (entryPoint != nullptr)

{

fn(index, entryPoint);

}

});

}

void Init(FunctionBody \* functionBody);

#if ENABLE\_NATIVE\_CODEGEN

int CreateEntryPoint();

void ReleaseEntryPoints();

#endif

void ResetInterpreterCount()

{

this->interpretCount = 0;

}

void ResetProfiledLoopCounter()

{

this->profiledLoopCounter = 0;

}

};

class FunctionProxy;

#ifdef RECYCLER\_WRITE\_BARRIER

CompileAssert(sizeof(WriteBarrierPtr<FunctionProxy>) == sizeof(FunctionProxy\*));

typedef WriteBarrierPtr<FunctionProxy>\* FunctionProxyArray;

typedef WriteBarrierPtr<FunctionProxy>\* FunctionProxyPtrPtr;

#else

typedef FunctionProxy\*\* FunctionProxyArray;

typedef FunctionProxy\*\* FunctionProxyPtrPtr;

#endif

//

// FunctionProxy represents a user defined function

// This could be either from a source file or the byte code cache

// The function need not have been compiled yet- it could be parsed or compiled

// at a later time

//

class FunctionProxy : public FunctionInfo

{

protected:

FunctionProxy(JavascriptMethod entryPoint, Attributes attributes, int nestedCount, int derivedSize,

LocalFunctionId functionId, ScriptContext\* scriptContext, Utf8SourceInfo\* utf8SourceInfo, uint functionNumber);

DEFINE\_VTABLE\_CTOR\_NO\_REGISTER(FunctionProxy, FunctionInfo);

public:

enum SetDisplayNameFlags

{

SetDisplayNameFlagsNone = 0,

SetDisplayNameFlagsDontCopy = 1,

SetDisplayNameFlagsRecyclerAllocated = 2

};

typedef RecyclerWeakReference<DynamicType> FunctionTypeWeakRef;

typedef JsUtil::List<FunctionTypeWeakRef\*, Recycler, false, WeakRefFreeListedRemovePolicy> FunctionTypeWeakRefList;

uint32 GetSourceContextId() const;

wchar\_t\* GetDebugNumberSet(wchar(&bufferToWriteTo)[MAX\_FUNCTION\_BODY\_DEBUG\_STRING\_SIZE]) const;

bool GetIsTopLevel() { return m\_isTopLevel; }

void SetIsTopLevel(bool set) { m\_isTopLevel = set; }

bool GetIsAnonymousFunction() const { return this->GetDisplayName() == Js::Constants::AnonymousFunction; }

uint GetNestedCount() const { return m\_nestedCount; }

void Copy(FunctionProxy\* other);

ParseableFunctionInfo\* EnsureDeserialized();

ScriptContext\* GetScriptContext() const;

Utf8SourceInfo\* GetUtf8SourceInfo() const { return this->m\_utf8SourceInfo; }

void SetReferenceInParentFunction(FunctionProxyPtrPtr reference);

void UpdateReferenceInParentFunction(FunctionProxy\* newFunctionInfo);

DWORD\_PTR GetSecondaryHostSourceContext() const;

DWORD\_PTR GetHostSourceContext() const;

SourceContextInfo \* GetSourceContextInfo() const;

SRCINFO const \* GetHostSrcInfo() const;

uint GetFunctionNumber() const { return m\_functionNumber; }

virtual void Finalize(bool isShutdown) override;

void UpdateFunctionBodyImpl(FunctionBody\* body);

bool IsFunctionBody() const;

ProxyEntryPointInfo\* GetDefaultEntryPointInfo() const;

ScriptFunctionType \* GetDeferredPrototypeType() const;

ScriptFunctionType \* EnsureDeferredPrototypeType();

JavascriptMethod GetDirectEntryPoint(ProxyEntryPointInfo\* entryPoint) const;

// Function object type list methods

FunctionTypeWeakRefList\* EnsureFunctionObjectTypeList();

void RegisterFunctionObjectType(DynamicType\* functionType);

template <typename Fn>

void MapFunctionObjectTypes(Fn func);

static uint GetOffsetOfDeferredPrototypeType() { return offsetof(Js::FunctionProxy, deferredPrototypeType); }

static Js::ScriptFunctionType \* EnsureFunctionProxyDeferredPrototypeType(FunctionProxy \* proxy)

{

return proxy->EnsureDeferredPrototypeType();

}

void SetIsPublicLibraryCode() { m\_isPublicLibraryCode = true; }

bool IsPublicLibraryCode() const { return m\_isPublicLibraryCode; }

#if DBG

bool HasValidEntryPoint() const;

bool HasValidProfileEntryPoint() const;

bool HasValidNonProfileEntryPoint() const;

#endif

virtual void SetDisplayName(const wchar\_t\* displayName, uint displayNameLength, uint displayShortNameOffset, SetDisplayNameFlags flags = SetDisplayNameFlagsNone) = 0;

virtual const wchar\_t\* GetDisplayName() const = 0;

virtual uint GetDisplayNameLength() const = 0;

virtual uint GetShortDisplayNameOffset() const = 0;

static const wchar\_t\* WrapWithBrackets(const wchar\_t\* name, charcount\_t sz, ScriptContext\* scriptContext);

// Used only in the library function stringify (toString, DiagGetValueString).

// If we need more often to give the short name, we should create a member variable which points to the short name

// this is also now being used for function.name.

const wchar\_t\* GetShortDisplayName(charcount\_t \* shortNameLength);

bool IsJitLoopBodyPhaseEnabled() const

{

// Consider: Allow JitLoopBody in generator functions for loops that do not yield.

return !PHASE\_OFF(JITLoopBodyPhase, this) && DoFullJit() && !this->IsGenerator();

}

bool IsJitLoopBodyPhaseForced() const

{

return

IsJitLoopBodyPhaseEnabled() &&

(

PHASE\_FORCE(JITLoopBodyPhase, this)

#ifdef ENABLE\_PREJIT

|| Configuration::Global.flags.Prejit

#endif

);

}

ULONG GetHostStartLine() const;

ULONG GetHostStartColumn() const;

bool DoFullJit() const

{

return !PHASE\_OFF(FullJitPhase, this);

}

protected:

// Static method(s)

static void SetDisplayName(const wchar\_t\* srcName, WriteBarrierPtr<const wchar\_t>\* destName, uint displayNameLength, ScriptContext \* scriptContext, SetDisplayNameFlags flags = SetDisplayNameFlagsNone);

static bool SetDisplayName(const wchar\_t\* srcName, const wchar\_t\*\* destName, uint displayNameLength, ScriptContext \* scriptContext, SetDisplayNameFlags flags = SetDisplayNameFlagsNone);

static bool IsConstantFunctionName(const wchar\_t\* srcName);

protected:

NoWriteBarrierPtr<ScriptContext> m\_scriptContext; // Memory context for this function body

WriteBarrierPtr<Utf8SourceInfo> m\_utf8SourceInfo;

// WriteBarrier-TODO: Consider changing this to NoWriteBarrierPtr, and skip tagging- also, tagging is likely unnecessary since that pointer in question is likely not resolvable

FunctionProxyPtrPtr m\_referenceInParentFunction; // Reference to nested function reference to this function in the parent function body (tagged to not be actual reference)

WriteBarrierPtr<ScriptFunctionType> deferredPrototypeType;

// Script function types not including the deferred prototype type

WriteBarrierPtr<FunctionTypeWeakRefList> m\_functionObjectTypeList;

WriteBarrierPtr<ProxyEntryPointInfo> m\_defaultEntryPointInfo; // The default entry point info for the function proxy

NoWriteBarrierField<uint> m\_derivedSize;

NoWriteBarrierField<uint> m\_functionNumber; // Per thread global function number

#define DEFINE\_FUNCTION\_PROXY\_FIELDS 1

#define CURRENT\_ACCESS\_MODIFIER protected:

#include "SerializableFunctionFields.h"

bool m\_isTopLevel : 1; // Indicates that this function is top-level function, currently being used in script profiler and debugger

bool m\_isPublicLibraryCode: 1; // Indicates this function is public boundary library code that should be visible in JS stack

void CleanupFunctionProxyCounters()

{

PERF\_COUNTER\_DEC(Code, TotalFunction);

}

ULONG ComputeAbsoluteLineNumber(ULONG relativeLineNumber) const;

ULONG ComputeAbsoluteColumnNumber(ULONG relativeLineNumber, ULONG relativeColumnNumber) const;

ULONG GetLineNumberInHostBuffer(ULONG relativeLineNumber) const;

private:

ScriptFunctionType \* AllocDeferredPrototypeType();

};

// Represents a function from the byte code cache which will

// be deserialized upon use

class DeferDeserializeFunctionInfo: public FunctionProxy

{

friend struct ByteCodeSerializer;

private:

DeferDeserializeFunctionInfo(int nestedFunctionCount, LocalFunctionId functionId, ByteCodeCache\* byteCodeCache, const byte\* serializedFunction, Utf8SourceInfo\* sourceInfo, ScriptContext\* scriptContext, uint functionNumber, const wchar\_t\* displayName, uint displayNameLength, uint displayShortNameOffset, NativeModule \*nativeModule, Attributes attributes);

public:

static DeferDeserializeFunctionInfo\* New(ScriptContext\* scriptContext, int nestedFunctionCount, LocalFunctionId functionId, ByteCodeCache\* byteCodeCache, const byte\* serializedFunction, Utf8SourceInfo\* utf8SourceInfo, const wchar\_t\* displayName, uint displayNameLength, uint displayShortNameOffset, NativeModule \*nativeModule, Attributes attributes);

virtual void Finalize(bool isShutdown) override;

FunctionBody\* Deserialize();

virtual const wchar\_t\* GetDisplayName() const override;

void SetDisplayName(const wchar\_t\* displayName);

virtual void SetDisplayName(const wchar\_t\* displayName, uint displayNameLength, uint displayShortNameOffset, SetDisplayNameFlags flags = SetDisplayNameFlagsNone) override;

virtual uint GetDisplayNameLength() const { return m\_displayNameLength; }

virtual uint GetShortDisplayNameOffset() const { return m\_displayShortNameOffset; }

LPCWSTR GetSourceInfo(int& lineNumber, int& columnNumber) const;

private:

const byte\* m\_functionBytes;

ByteCodeCache\* m\_cache;

const wchar\_t \* m\_displayName; // Optional name

uint m\_displayNameLength;

uint m\_displayShortNameOffset;

NativeModule \*m\_nativeModule;

};

class ParseableFunctionInfo: public FunctionProxy

{

friend class ByteCodeBufferReader;

protected:

ParseableFunctionInfo(JavascriptMethod method, int nestedFunctionCount, int derivedSize, LocalFunctionId functionId, Utf8SourceInfo\* sourceInfo, ScriptContext\* scriptContext, uint functionNumber, const wchar\_t\* displayName, uint m\_displayNameLength, uint displayShortNameOffset, Attributes attributes, Js::PropertyRecordList\* propertyRecordList);

public:

static ParseableFunctionInfo\* New(ScriptContext\* scriptContext, int nestedFunctionCount, LocalFunctionId functionId, Utf8SourceInfo\* utf8SourceInfo, const wchar\_t\* displayName, uint m\_displayNameLength, uint displayShortNameOffset, Js::PropertyRecordList\* propertyRecordList, Attributes attributes);

DEFINE\_VTABLE\_CTOR\_NO\_REGISTER(ParseableFunctionInfo, FunctionProxy);

FunctionBody\* Parse(ScriptFunction \*\* functionRef = nullptr, bool isByteCodeDeserialization = false);

#ifndef TEMP\_DISABLE\_ASMJS

FunctionBody\* ParseAsmJs(Parser \* p, \_\_out CompileScriptException \* se, \_\_out ParseNodePtr \* ptree);

#endif

virtual uint GetDisplayNameLength() const { return m\_displayNameLength; }

virtual uint GetShortDisplayNameOffset() const { return m\_displayShortNameOffset; }

bool GetIsDeclaration() const { return m\_isDeclaration; }

void SetIsDeclaration(const bool is) { m\_isDeclaration = is; }

bool GetIsAccessor() const { return m\_isAccessor; }

void SetIsAccessor(const bool is) { m\_isAccessor = is; }

bool GetIsGlobalFunc() const { return m\_isGlobalFunc; }

void SetIsStaticNameFunction(const bool is) { m\_isStaticNameFunction = is; }

bool GetIsStaticNameFunction() const { return m\_isStaticNameFunction; }

void SetIsNamedFunctionExpression(const bool is) { m\_isNamedFunctionExpression = is; }

bool GetIsNamedFunctionExpression() const { return m\_isNamedFunctionExpression; }

void SetIsNameIdentifierRef (const bool is) { m\_isNameIdentifierRef = is; }

bool GetIsNameIdentifierRef () const { return m\_isNameIdentifierRef ; }

// Fake global ->

// 1) new Function code's global code

// 2) global code generated from the reparsing deferred parse function

bool IsFakeGlobalFunc(ulong flags) const;

void SetIsGlobalFunc(bool is) { m\_isGlobalFunc = is; }

bool GetIsStrictMode() const { return m\_isStrictMode; }

void SetIsStrictMode() { m\_isStrictMode = true; }

bool GetIsAsmjsMode() const { return m\_isAsmjsMode; }

void SetIsAsmjsMode(bool value)

{

m\_isAsmjsMode = value;

#if DBG

if (value)

{

m\_wasEverAsmjsMode = true;

}

#endif

}

bool GetHasImplicitArgIns() { return m\_hasImplicitArgIns; }

void SetHasImplicitArgIns(bool has) { m\_hasImplicitArgIns = has; }

ulong GetGrfscr() const;

void SetGrfscr(ulong grfscr);

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

///

/// ParseableFunctionInfo::GetInParamsCount

///

/// GetInParamsCount() returns the number of "in parameters" that have

/// currently been declared for this function:

/// - If this is "RegSlot\_VariableCount", the function takes a variable number

/// of parameters.

///

/// Consider: Change to store type information about parameters- names, type,

/// direction, etc.

///

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

ArgSlot GetInParamsCount() const { return m\_inParamCount; }

void SetInParamsCount(ArgSlot newInParamCount);

ArgSlot GetReportedInParamsCount() const;

void SetReportedInParamsCount(ArgSlot newReportedInParamCount);

void ResetInParams();

ScopeInfo\* GetScopeInfo() const { return this->m\_scopeInfo; }

void SetScopeInfo(ScopeInfo\* scopeInfo) { this->m\_scopeInfo = scopeInfo; }

PropertyId GetOrAddPropertyIdTracked(JsUtil::CharacterBuffer<WCHAR> const& propName);

bool IsTrackedPropertyId(PropertyId pid);

Js::PropertyRecordList\* GetBoundPropertyRecords() { return this->m\_boundPropertyRecords; }

void SetBoundPropertyRecords(Js::PropertyRecordList\* boundPropertyRecords)

{

Assert(this->m\_boundPropertyRecords == nullptr);

this->m\_boundPropertyRecords = boundPropertyRecords;

}

void ClearBoundPropertyRecords()

{

this->m\_boundPropertyRecords = nullptr;

}

ParseableFunctionInfo\* Clone(ScriptContext \*scriptContext, uint sourceIndex = Js::Constants::InvalidSourceIndex);

ParseableFunctionInfo\* CopyFunctionInfoInto(ScriptContext \*scriptContext, Js::ParseableFunctionInfo\* functionInfo, uint sourceIndex = Js::Constants::InvalidSourceIndex);

void CloneSourceInfo(ScriptContext\* scriptContext, const ParseableFunctionInfo& other, ScriptContext\* othersScriptContext, uint sourceIndex);

void SetInitialDefaultEntryPoint();

void SetDeferredParsingEntryPoint();

void SetEntryPoint(ProxyEntryPointInfo\* entryPoint, Js::JavascriptMethod address) {

entryPoint->address = address;

}

bool IsDynamicScript() const;

uint LengthInBytes() const { return m\_cbLength; }

uint StartOffset() const;

ULONG GetLineNumber() const;

ULONG GetColumnNumber() const;

template <class T>

LPCWSTR GetSourceName(const T& sourceContextInfo) const;

template <class T>

static LPCWSTR GetSourceName(const T& sourceContextInfo, bool m\_isEval, bool m\_isDynamicFunction);

LPCWSTR GetSourceName() const;

ULONG GetRelativeLineNumber() const { return m\_lineNumber; }

ULONG GetRelativeColumnNumber() const { return m\_columnNumber; }

uint GetSourceIndex() const;

LPCUTF8 GetSource(const wchar\_t\* reason = nullptr) const;

charcount\_t LengthInChars() const { return m\_cchLength; }

charcount\_t StartInDocument() const;

bool IsEval() const { return m\_isEval; }

bool IsDynamicFunction() const;

bool GetDontInline() { return m\_dontInline; }

void SetDontInline(bool is) { m\_dontInline = is; }

LPCUTF8 GetStartOfDocument(const wchar\_t\* reason = nullptr) const;

bool IsReparsed() const { return m\_reparsed; }

void SetReparsed(bool set) { m\_reparsed = set; }

bool IsByteCodeDebugMode() { return isByteCodeDebugMode; }

void SetIsByteCodeDebugMode(bool set) { isByteCodeDebugMode = set; }

bool GetExternalDisplaySourceName(BSTR\* sourceName);

void SetDoBackendArgumentsOptimization(bool set)

{

if (m\_doBackendArgumentsOptimization)

{

m\_doBackendArgumentsOptimization = set;

}

}

bool GetDoBackendArgumentsOptimization()

{

return m\_doBackendArgumentsOptimization;

}

bool IsFunctionParsed()

{

return !IsDeferredParseFunction() || m\_hasBeenParsed;

}

void SetFunctionParsed(bool hasBeenParsed)

{

m\_hasBeenParsed = hasBeenParsed;

}

void SetSourceInfo(uint sourceIndex, ParseNodePtr node, bool isEval, bool isDynamicFunction);

void Copy(FunctionBody\* other);

const wchar\_t\* GetExternalDisplayName() const;

//

// Algorithm to retrieve a function body's external display name. Template supports both

// local FunctionBody and ScriptDAC (debugging) scenarios.

//

template <class T>

static const wchar\_t\* GetExternalDisplayName(const T\* funcBody)

{

Assert(funcBody != nullptr);

Assert(funcBody->GetDisplayName() != nullptr);

return funcBody->GetDisplayName();

}

virtual const wchar\_t\* GetDisplayName() const override;

void SetDisplayName(const wchar\_t\* displayName);

virtual void SetDisplayName(const wchar\_t\* displayName, uint displayNameLength, uint displayShortNameOffset, SetDisplayNameFlags flags = SetDisplayNameFlagsNone) override;

virtual void Finalize(bool isShutdown) override;

Var GetCachedSourceString()

{

return cachedSourceString;

}

void SetCachedSourceString(Var sourceString)

{

Assert(this->cachedSourceString == nullptr);

this->cachedSourceString = sourceString;

}

FunctionProxyArray GetNestedFuncArray();

FunctionProxy\* GetNestedFunc(uint index);

FunctionProxyPtrPtr GetNestedFuncReference(uint index);

ParseableFunctionInfo\* GetNestedFunctionForExecution(uint index);

void SetNestedFunc(FunctionProxy\* nestedFunc, uint index, ulong flags);

void ClearNestedFunctionParentFunctionReference();

void SetCapturesThis() { attributes = (Attributes)(attributes | Attributes::CapturesThis); }

bool GetCapturesThis() { return (attributes & Attributes::CapturesThis) != 0; }

void BuildDeferredStubs(ParseNode \*pnodeFnc);

DeferredFunctionStub \*GetDeferredStubs() const { return this->deferredStubs; }

void SetDeferredStubs(DeferredFunctionStub \*stub) { this->deferredStubs = stub; }

void RegisterFuncToDiag(ScriptContext \* scriptContext, wchar\_t const \* pszTitle);

protected:

static HRESULT MapDeferredReparseError(HRESULT& hrParse, const CompileScriptException& se);

bool m\_hasBeenParsed : 1; // Has function body been parsed- true for actual function bodies, false for deferparse

bool m\_isDeclaration : 1;

bool m\_isAccessor : 1; // Function is a property getter or setter

bool m\_isStaticNameFunction : 1;

bool m\_isNamedFunctionExpression : 1;

bool m\_isNameIdentifierRef : 1;

bool m\_isClassMember : 1;

bool m\_isStrictMode : 1;

bool m\_isAsmjsMode : 1;

bool m\_isAsmJsFunction : 1;

bool m\_isGlobalFunc : 1;

bool m\_doBackendArgumentsOptimization :1;

bool m\_isEval : 1; // Source code is in 'eval'

bool m\_isDynamicFunction : 1; // Source code is in 'Function'

bool m\_hasImplicitArgIns : 1;

bool m\_dontInline : 1; // Used by the JIT's inliner

// Indicates if the function has been reparsed for debug attach/detach scenario.

bool m\_reparsed : 1;

bool isByteCodeDebugMode : 1; // Whether last time generated bytecode was generated for debug mode.

// This field is not required for deferred parsing but because our thunks can't handle offsets > 128 bytes

// yet, leaving this here for now. We can look at optimizing the function info and function proxy structures some

// more and also fix our thunks to handle 8 bit offsets

NoWriteBarrierField<bool> m\_utf8SourceHasBeenSet; // start of UTF8-encoded source

NoWriteBarrierField<uint> m\_sourceIndex; // index into the scriptContext's list of saved sources

#if DYNAMIC\_INTERPRETER\_THUNK

void\* m\_dynamicInterpreterThunk; // Unique 'thunk' for every interpreted function - used for ETW symbol decoding.

#endif

NoWriteBarrierField<uint> m\_cbStartOffset; // pUtf8Source is this many bytes from the start of the scriptContext's source buffer.

// This is generally the same as m\_cchStartOffset unless the buffer has a BOM

#define DEFINE\_PARSEABLE\_FUNCTION\_INFO\_FIELDS 1

#define CURRENT\_ACCESS\_MODIFIER protected:

#include "SerializableFunctionFields.h"

ULONG m\_lineNumber;

ULONG m\_columnNumber;

WriteBarrierPtr<const wchar\_t> m\_displayName; // Optional name

uint m\_displayNameLength;

uint m\_displayShortNameOffset;

WriteBarrierPtr<ScopeInfo> m\_scopeInfo;

WriteBarrierPtr<PropertyRecordList> m\_boundPropertyRecords;

WriteBarrierVar cachedSourceString;

WriteBarrierPtr<DeferredFunctionStub> deferredStubs;

public:

#if DBG

bool m\_wasEverAsmjsMode; // has m\_isAsmjsMode ever been true

NoWriteBarrierField<Js::LocalFunctionId> deferredParseNextFunctionId;

#endif

#if DBG

NoWriteBarrierField<UINT> scopeObjectSize; // If the scope is an activation object - its size

#endif

};

//

// Algorithm to retrieve a function body's source name (url). Template supports both

// local FunctionBody and ScriptDAC (debugging) scenarios.

//

template <class T>

LPCWSTR ParseableFunctionInfo::GetSourceName(const T& sourceContextInfo) const

{

return GetSourceName<T>(sourceContextInfo, this->m\_isEval, this->m\_isDynamicFunction);

}

template <class T>

LPCWSTR ParseableFunctionInfo::GetSourceName(const T& sourceContextInfo, bool m\_isEval, bool m\_isDynamicFunction)

{

if (sourceContextInfo->IsDynamic())

{

if (m\_isEval)

{

return Constants::EvalCode;

}

else if (m\_isDynamicFunction)

{

return Constants::FunctionCode;

}

else

{

return Constants::UnknownScriptCode;

}

}

else

{

return sourceContextInfo->url;

}

}

class FunctionBody : public ParseableFunctionInfo

{

DEFINE\_VTABLE\_CTOR\_NO\_REGISTER(FunctionBody, ParseableFunctionInfo);

friend class ByteCodeBufferBuilder;

friend class ByteCodeBufferReader;

public:

// same as MachDouble, used in the Func.h

static const uint DIAGLOCALSLOTSIZE = 8;

struct StatementMap

{

StatementMap() : isSubexpression(false) {}

static StatementMap \* New(Recycler\* recycler)

{

return RecyclerNew(recycler, StatementMap);

}

regex::Interval sourceSpan;

regex::Interval byteCodeSpan;

bool isSubexpression;

};

// The type of StatementAdjustmentRecord.

// A bitmask that can be OR'ed of multiple values of the enum.

enum StatementAdjustmentType : ushort

{

SAT\_None = 0,

// Specifies an adjustment for next statement when going from current to next.

// Used for transitioning from current stmt to next during normal control-flow,

// such as offset of Br after if-block when there is else block present,

// when throw happens inside if and we ignore exceptions (next statement in the list

// would be 'else' but we need to pass flow control to Br target rather than entering 'else').

SAT\_FromCurrentToNext = 0x01,

// Specifies an adjustment for beginning of next statement.

// If there is adjustment record, the statement following it starts at specified offset and not at offset specified in statementMap.

// Used for set next statement from arbitrary location.

SAT\_NextStatementStart = 0x02,

SAT\_All = SAT\_FromCurrentToNext | SAT\_NextStatementStart

};

class StatementAdjustmentRecord

{

uint m\_byteCodeOffset;

StatementAdjustmentType m\_adjustmentType;

public:

StatementAdjustmentRecord();

StatementAdjustmentRecord(StatementAdjustmentType type, int byteCodeOffset);

StatementAdjustmentRecord(const StatementAdjustmentRecord& other);

uint GetByteCodeOffset();

StatementAdjustmentType GetAdjustmentType();

};

// Offset and entry/exit of a block that must be processed in new interpreter frame rather than current.

// Used for try and catch blocks.

class CrossFrameEntryExitRecord

{

uint m\_byteCodeOffset;

// true means enter, false means exit.

bool m\_isEnterBlock;

public:

CrossFrameEntryExitRecord();

CrossFrameEntryExitRecord(uint byteCodeOffset, bool isEnterBlock);

CrossFrameEntryExitRecord(const CrossFrameEntryExitRecord& other);

uint GetByteCodeOffset() const;

bool GetIsEnterBlock();

};

typedef JsUtil::List<Js::FunctionBody::StatementMap\*> StatementMapList;

// Note: isLeaf = true template param below means that recycler should not be used to dispose the items.

typedef JsUtil::List<StatementAdjustmentRecord, Recycler, /\* isLeaf = \*/ true> StatementAdjustmentRecordList;

typedef JsUtil::List<CrossFrameEntryExitRecord, Recycler, /\* isLeaf = \*/ true> CrossFrameEntryExitRecordList;

// Contains recorded at bytecode generation time information about statements and try-catch blocks.

// Used by debugger.

struct AuxStatementData

{

// Contains statement adjustment data:

// For given bytecode, following statement needs an adjustment, see StatementAdjustmentType for details.

StatementAdjustmentRecordList\* m\_statementAdjustmentRecords;

// Contain data about entry/exit of blocks that cause processing in different interpreter stack frame, such as try or catch.

CrossFrameEntryExitRecordList\* m\_crossFrameBlockEntryExisRecords;

AuxStatementData();

};

class SourceInfo

{

friend class RemoteFunctionBody;

friend class ByteCodeBufferReader;

friend class ByteCodeBufferBuilder;

public:

SmallSpanSequence \* pSpanSequence;

RegSlot frameDisplayRegister; // this register slot cannot be 0 so we use that sentinel value to indicate invalid

RegSlot objectRegister; // this register slot cannot be 0 so we use that sentinel value to indicate invalid

WriteBarrierPtr<ScopeObjectChain> pScopeObjectChain;

WriteBarrierPtr<ByteBlock> m\_probeBackingBlock; // NULL if no Probes, otherwise a copy of the unmodified the byte-codeblock //Delay

int32 m\_probeCount; // The number of installed probes (such as breakpoints).

// List of bytecode offset for the Branch bytecode.

WriteBarrierPtr<AuxStatementData> m\_auxStatementData;

SourceInfo():

frameDisplayRegister(0),

objectRegister(0),

pScopeObjectChain(nullptr),

m\_probeBackingBlock(nullptr),

m\_probeCount(0),

m\_auxStatementData(nullptr),

pSpanSequence(nullptr)

{

}

};

private:

WriteBarrierPtr<ByteBlock> auxBlock; // Optional auxiliary information

WriteBarrierPtr<ByteBlock> auxContextBlock; // Optional auxiliary context specific information

WriteBarrierPtr<ByteBlock> byteCodeBlock; // Function byte-code for script functions

WriteBarrierPtr<FunctionEntryPointList> entryPoints;

WriteBarrierPtr<Js::LoopHeader> loopHeaderArray;

WriteBarrierPtr<Var> m\_constTable;

WriteBarrierPtr<FunctionCodeGenRuntimeData\*> m\_codeGenRuntimeData;

WriteBarrierPtr<FunctionCodeGenRuntimeData\*> m\_codeGenGetSetRuntimeData;

WriteBarrierPtr<void\*> inlineCaches;

InlineCachePointerArray<PolymorphicInlineCache> polymorphicInlineCaches; // Contains the latest polymorphic inline caches

WriteBarrierPtr<PolymorphicInlineCache> polymorphicInlineCachesHead; // DList of all polymorphic inline caches that aren't finalized yet

WriteBarrierPtr<PropertyId> cacheIdToPropertyIdMap;

WriteBarrierPtr<PropertyId> referencedPropertyIdMap;

WriteBarrierPtr<UnifiedRegex::RegexPattern\*>literalRegexes;

WriteBarrierPtr<AsmJsFunctionInfo> asmJsFunctionInfo;

WriteBarrierPtr<AsmJsModuleInfo> asmJsModuleInfo;

// For SourceInfo

WriteBarrierPtr<PropertyId> propertyIdsForScopeSlotArray;

// This is used for showing locals for the current frame.

WriteBarrierPtr<PropertyIdOnRegSlotsContainer> propertyIdOnRegSlotsContainer;

#if DBG

#define InlineCacheTypeNone 0x00

#define InlineCacheTypeInlineCache 0x01

#define InlineCacheTypeIsInst 0x02

WriteBarrierPtr<byte> m\_inlineCacheTypes;

#endif

WriteBarrierPtr<StatementMapList> pStatementMaps;

public:

static DWORD GetConstTableOffset() { return offsetof(FunctionBody, m\_constTable); }

static DWORD GetAuxiliaryDataOffset() { return offsetof(FunctionBody, auxBlock); }

static DWORD GetAuxiliaryContextDataOffset() { return offsetof(FunctionBody, auxContextBlock); }

static DWORD GetObjLiteralTypesOffset() { return offsetof(FunctionBody, objLiteralTypes); }

static DWORD GetInlineCachesOffset() { return offsetof(FunctionBody, inlineCaches); }

static DWORD GetInlineCacheCountOffset() { return offsetof(FunctionBody, inlineCacheCount); }

static DWORD GetLiteralRegexesOffset() { return offsetof(FunctionBody, literalRegexes); }

static DWORD GetDerivedSizeOffset() { return offsetof(FunctionBody, m\_derivedSize); }

static DWORD GetReferencedPropertyIdMapOffset() { return offsetof(FunctionBody, referencedPropertyIdMap); }

static DWORD GetCacheIdToPropertyIdMapOffset() { return offsetof(FunctionBody, cacheIdToPropertyIdMap); }

static DWORD GetAsmJsTotalLoopCountOffset(){ return offsetof(FunctionBody, m\_asmJsTotalLoopCount); }

#if DBG

int m\_DEBUG\_executionCount; // Count of outstanding on InterpreterStackFrame

bool m\_nativeEntryPointIsInterpreterThunk; // NativeEntry entry point is in fact InterpreterThunk.

// Set by bgjit in OutOfMemory scenario during codegen.

#endif

#if ENABLE\_DEBUG\_CONFIG\_OPTIONS

NoWriteBarrierField<uint> regAllocStoreCount;

NoWriteBarrierField<uint> regAllocLoadCount;

NoWriteBarrierField<uint> callCountStats;

#endif

NoWriteBarrierField<uint> interpretedCount;

NoWriteBarrierField<uint> loopInterpreterLimit;

// >>>>>>WARNING! WARNING!<<<<<<<<<<

//

// If you add compile-time attributes to this set, be sure to add them to the attributes that are

// copied in FunctionBody::Clone

//

SourceInfo m\_sourceInfo; // position of the source

// Data needed by profiler:

NoWriteBarrierField<uint> m\_uScriptId; // Delay //Script Block it belongs to. This is function no. of the global function created by engine for each block

#if DBG

NoWriteBarrierField<int> m\_iProfileSession; // Script profile session the meta data of this function is reported to.

#endif // DEBUG

// R0 is reserved for the return value, R1 for the root object

static const RegSlot ReturnValueRegSlot = 0;

static const RegSlot RootObjectRegSlot = 1;

static const RegSlot FirstRegSlot = 1;

// This value be set on the stack (on a particular offset), when the frame value got changed.

static const int LocalsChangeDirtyValue = 1;

enum FunctionBodyFlags : byte

{

Flags\_None = 0x00,

Flags\_StackNestedFunc = 0x01,

Flags\_HasOrParentHasArguments = 0x02,

Flags\_HasTry = 0x04,

Flags\_HasThis = 0x08,

Flags\_NonUserCode = 0x10,

Flags\_HasOnlyThisStatements = 0x20,

Flags\_HasNoExplicitReturnValue = 0x40, // Returns undefined, i.e. has no return statements or return with no expression

Flags\_HasRestParameter = 0x80

};

#define DEFINE\_FUNCTION\_BODY\_FIELDS 1

#define CURRENT\_ACCESS\_MODIFIER public:

#include "SerializableFunctionFields.h"

private:

bool m\_tag : 1; // Used to tag the low bit to prevent possible GC false references

bool m\_nativeEntryPointUsed : 1; // Code might have been generated but not yet used.

bool hasDoneLoopBodyCodeGen : 1; // Code generated for loop body, but not necessary available to execute yet.

bool m\_isFuncRegistered : 1;

bool m\_isFuncRegisteredToDiag : 1; // Mentions the function's context is registered with diagprobe.

bool funcEscapes : 1;

bool m\_hasBailoutInstrInJittedCode : 1; // Indicates whether function has bailout instructions. Valid only if hasDoneCodeGen is true

bool m\_pendingLoopHeaderRelease : 1; // Indicates whether loop headers need to be released

bool hasExecutionDynamicProfileInfo : 1;

bool cleanedUp: 1;

bool sourceInfoCleanedUp: 1;

bool dontRethunkAfterBailout : 1;

bool disableInlineApply : 1;

bool disableInlineSpread : 1;

bool hasHotLoop: 1;

bool wasCalledFromLoop : 1;

bool hasNestedLoop : 1;

bool recentlyBailedOutOfJittedLoopBody : 1;

bool m\_firstFunctionObject: 1;

bool m\_inlineCachesOnFunctionObject: 1;

// Used for the debug re-parse. Saves state of function on the first parse, and restores it on a reparse. The state below is either dependent on

// the state of the script context, or on other factors like whether it was defer parsed or not.

bool m\_hasSetIsObject : 1;

// Used for the debug purpose, this info will be stored (in the non-debug mode), when a function has all locals marked as non-local-referenced.

// So when we got to no-refresh debug mode, and try to re-use the same function body we can then enforce all locals to be non-local-referenced.

bool m\_hasAllNonLocalReferenced : 1;

bool m\_hasFunExprNameReference : 1;

bool m\_ChildCallsEval : 1;

bool m\_CallsEval : 1;

bool m\_hasReferenceableBuiltInArguments : 1;

// Used in the debug purpose. This is to avoid setting all locals to non-local-referenced, multiple times for each child function.

bool m\_hasDoneAllNonLocalReferenced : 1;

// Used by the script profiler, once the function compiled is sent this will be set to true.

bool m\_hasFunctionCompiledSent : 1;

bool m\_isFromNativeCodeModule : 1;

bool m\_isPartialDeserializedFunction : 1;

bool m\_isAsmJsScheduledForFullJIT : 1;

#ifdef PERF\_COUNTERS

bool m\_isDeserializedFunction : 1;

#endif

#if DBG

// Indicates that nested functions can be allocated on the stack (but may not be)

bool m\_canDoStackNestedFunc : 1;

#endif

#if DBG

bool initializedExecutionModeAndLimits : 1;

#endif

#ifdef IR\_VIEWER

// whether IR Dump is enabled for this function (used by parseIR)

bool m\_isIRDumpEnabled : 1;

WriteBarrierPtr<Js::DynamicObject> m\_irDumpBaseObject;

#endif /\* IR\_VIEWER \*/

NoWriteBarrierField<uint8> bailOnMisingProfileCount;

NoWriteBarrierField<uint8> bailOnMisingProfileRejitCount;

NoWriteBarrierField<byte> inlineDepth; // Used by inlining to avoid recursively inlining functions excessively

NoWriteBarrierField<ExecutionMode> executionMode;

NoWriteBarrierField<uint16> interpreterLimit;

NoWriteBarrierField<uint16> autoProfilingInterpreter0Limit;

NoWriteBarrierField<uint16> profilingInterpreter0Limit;

NoWriteBarrierField<uint16> autoProfilingInterpreter1Limit;

NoWriteBarrierField<uint16> simpleJitLimit;

NoWriteBarrierField<uint16> profilingInterpreter1Limit;

NoWriteBarrierField<uint16> fullJitThreshold;

NoWriteBarrierField<uint16> fullJitRequeueThreshold;

NoWriteBarrierField<uint16> committedProfiledIterations;

NoWriteBarrierField<uint> m\_depth; // Indicates how many times the function has been entered (so increases by one on each recursive call, decreases by one when we're done)

WriteBarrierPtr<RecyclerWeakReference<FunctionBody>> stackNestedFuncParent;

// >>>>>>WARNING! WARNING!<<<<<<<<<<

//

// If you add compile-time attributes to the above set, be sure to add them to the attributes that are

// copied in FunctionBody::Clone

//

NoWriteBarrierPtr<Js::ByteCodeCache> byteCodeCache; // Not GC allocated so naked pointer

NoWriteBarrierField<int> serializationIndex;

#ifdef ENABLE\_DEBUG\_CONFIG\_OPTIONS

static bool shareInlineCaches;

#endif

WriteBarrierPtr<DynamicType\*> objLiteralTypes;

WriteBarrierPtr<FunctionEntryPointInfo> defaultFunctionEntryPointInfo;

WriteBarrierPtr<FunctionEntryPointInfo> simpleJitEntryPointInfo;

#if ENABLE\_PROFILE\_INFO

WriteBarrierPtr<DynamicProfileInfo> dynamicProfileInfo;

WriteBarrierPtr<PolymorphicCallSiteInfo> polymorphicCallSiteInfoHead;

#endif

FunctionBailOutRecord \* functionBailOutRecord;

// select dynamic profile info saved off when we codegen and later

// used for rejit decisions (see bailout.cpp)

NoWriteBarrierField<BYTE> savedInlinerVersion;

#if ENABLE\_NATIVE\_CODEGEN

NoWriteBarrierField<ImplicitCallFlags> savedImplicitCallsFlags;

#endif

// State of inline caches (polymorphic vs. monomorphic) reflected in

// last jitted version of this func.

NoWriteBarrierField<uint32> savedPolymorphicCacheState;

// Used to track where we are when adding debugger scopes to the scope chain

// in order to avoid re-adding existing entries.

NoWriteBarrierField<int> debuggerScopeIndex;

FunctionBody(ScriptContext\* scriptContext, const wchar\_t\* displayName, uint displayNameLength, uint displayShortNameOffset, uint nestedCount, Utf8SourceInfo\* sourceInfo,

uint uFunctionNumber, uint uScriptId, Js::LocalFunctionId functionId, Js::PropertyRecordList\* propRecordList, Attributes attributes

#ifdef PERF\_COUNTERS

, bool isDeserializedFunction = false

#endif

);

void SetNativeEntryPoint(FunctionEntryPointInfo\* entryPointInfo, JavascriptMethod originalEntryPoint, Var directEntryPoint);

#if DYNAMIC\_INTERPRETER\_THUNK

void GenerateDynamicInterpreterThunk();

#endif

void CloneByteCodeInto(ScriptContext \* scriptContext, FunctionBody \*newFunctionBody, uint sourceIndex);

void \* GetEntryPoint(ProxyEntryPointInfo\* entryPoint) const { return entryPoint->address; }

void CaptureDynamicProfileState(FunctionEntryPointInfo\* entryPointInfo);

#if ENABLE\_DEBUG\_CONFIG\_OPTIONS

void DumpRegStats(FunctionBody \*funcBody);

#endif

// Gets the next index for tracking debugger scopes (increments the internal counter as well).

int GetNextDebuggerScopeIndex();

public:

FunctionBody(ByteCodeCache\* cache, Utf8SourceInfo\* sourceInfo, ScriptContext\* scriptContext):

ParseableFunctionInfo((JavascriptMethod) nullptr, 0, 0, (LocalFunctionId) 0, sourceInfo, scriptContext, 0, nullptr, 0, 0, None, nullptr)

{

// Dummy constructor- does nothing

// Must be stack allocated

// Used during deferred bytecode serialization

}

static FunctionBody \* NewFromRecycler(Js::ScriptContext \* scriptContext, const wchar\_t \* displayName, uint displayNameLength, uint displayShortNameOffset, uint nestedCount,

Utf8SourceInfo\* sourceInfo, uint uScriptId, Js::LocalFunctionId functionId, Js::PropertyRecordList\* boundPropertyRecords, Attributes attributes

#ifdef PERF\_COUNTERS

, bool isDeserializedFunction

#endif

);

static FunctionBody \* NewFromRecycler(Js::ScriptContext \* scriptContext, const wchar\_t \* displayName, uint displayNameLength, uint displayShortNameOffset, uint nestedCount,

Utf8SourceInfo\* sourceInfo, uint uFunctionNumber, uint uScriptId, Js::LocalFunctionId functionId, Js::PropertyRecordList\* boundPropertyRecords, Attributes attributes

#ifdef PERF\_COUNTERS

, bool isDeserializedFunction

#endif

);

FunctionEntryPointInfo \* GetEntryPointInfo(int index) const;

FunctionEntryPointInfo \* TryGetEntryPointInfo(int index) const;

Js::RootObjectBase \* LoadRootObject() const;

Js::RootObjectBase \* GetRootObject() const;

ByteBlock\* GetAuxiliaryData();

ByteBlock\* GetAuxiliaryContextData();

ByteBlock\* GetByteCode();

ByteBlock\* GetOriginalByteCode(); // Returns original bytecode without probes (such as BPs).

const Js::ByteCodeCache \* GetByteCodeCache() const;

void SetSerializationIndex(int index) { Assert(serializationIndex == -1 && index != -1); serializationIndex = index; }

const int GetSerializationIndex() const;

uint GetByteCodeCount() const { return m\_byteCodeCount; }

uint GetByteCodeWithoutLDACount() const { return m\_byteCodeWithoutLDACount; }

uint GetByteCodeInLoopCount() const { return m\_byteCodeInLoopCount; }

uint16 GetEnvDepth() const { return m\_envDepth; }

void SetEnvDepth(uint16 depth) { m\_envDepth = depth; }

RegSlot GetEnvReg() const { return envRegister; }

void SetEnvReg(RegSlot reg) { Assert(envRegister == Constants::NoRegister); envRegister = this->MapRegSlot(reg); }

RegSlot GetThisRegForEventHandler() const { return thisRegisterForEventHandler; }

void SetThisRegForEventHandler(RegSlot reg) { Assert(thisRegisterForEventHandler == Constants::NoRegister); thisRegisterForEventHandler = this->MapRegSlot(reg); }

bool HasScopeObject() const { return hasScopeObject; }

void SetHasScopeObject(bool has) { hasScopeObject = has; }

void SetLocalClosureReg(RegSlot reg) { Assert(localClosureRegister == Constants::NoRegister); localClosureRegister = this->MapRegSlot(reg); }

RegSlot GetLocalClosureReg() const { return localClosureRegister; }

void SetLocalFrameDisplayReg(RegSlot reg) { Assert(localFrameDisplayRegister == Constants::NoRegister); localFrameDisplayRegister = this->MapRegSlot(reg); }

RegSlot GetLocalFrameDisplayReg() const { return localFrameDisplayRegister; /\*localClosureRegister == Constants::NoRegister ? Constants::NoRegister : localClosureRegister + 1;\*/ }

RegSlot FirstInnerScopeReg() const { Assert(firstInnerScopeRegister != Constants::NoRegister); return firstInnerScopeRegister; }

void SetFirstInnerScopeReg(RegSlot reg) { Assert(reg != Constants::NoRegister); firstInnerScopeRegister = this->MapRegSlot(reg); }

RegSlot GetFuncExprScopeReg() const { return funcExprScopeRegister; }

void SetFuncExprScopeReg(RegSlot reg) { Assert(reg != Constants::NoRegister); funcExprScopeRegister = this->MapRegSlot(reg); }

uint GetInnerScopeCount() const { return innerScopeCount; }

void SetInnerScopeCount(uint count) { innerScopeCount = count; }

bool HasCachedScopePropIds() const { return hasCachedScopePropIds; }

void SetHasCachedScopePropIds(bool has) { hasCachedScopePropIds = has; }

size\_t GetLoopBodyName(uint loopNumber, \_Out\_writes\_opt\_z\_(sizeInChars) WCHAR\* displayName, \_In\_ size\_t sizeInChars);

void AllocateLoopHeaders();

void ReleaseLoopHeaders();

Js::LoopHeader \* GetLoopHeader(uint index) const;

Js::Var GetLoopHeaderArrayPtr() const

{

Assert(this->loopHeaderArray != nullptr);

return this->loopHeaderArray;

}

#ifndef TEMP\_DISABLE\_ASMJS

void SetIsAsmJsFullJitScheduled(bool val){ m\_isAsmJsScheduledForFullJIT = val; }

bool GetIsAsmJsFullJitScheduled(){ return m\_isAsmJsScheduledForFullJIT; }

uint32 GetAsmJSTotalLoopCount() const

{

return m\_asmJsTotalLoopCount;

}

void SetIsAsmJsFunction(bool isAsmJsFunction)

{

m\_isAsmJsFunction = isAsmJsFunction;

}

#endif

const bool GetIsAsmJsFunction() const

{

return m\_isAsmJsFunction;

}

#ifndef TEMP\_DISABLE\_ASMJS

bool IsHotAsmJsLoop()

{

// Negative MinTemplatizedJitLoopRunCount treats all loops as hot asm loop

if (CONFIG\_FLAG(MinTemplatizedJitLoopRunCount) < 0 || m\_asmJsTotalLoopCount > static\_cast<uint>(CONFIG\_FLAG(MinTemplatizedJitLoopRunCount)))

{

return true;

}

return false;

}

#endif

private:

void ResetLoops();

public:

static bool Is(void\* ptr);

uint GetScriptId() const { return m\_uScriptId; }

void\* GetAddressOfScriptId() const

{

return (void\*)&m\_uScriptId;

}

uint8 \*GetCallsCountAddress(EntryPointInfo\* info) const

{

FunctionEntryPointInfo\* entryPoint = (FunctionEntryPointInfo\*) info;

return &entryPoint->callsCount;

}

FunctionEntryPointInfo\* GetDefaultFunctionEntryPointInfo() const;

void SetDefaultFunctionEntryPointInfo(FunctionEntryPointInfo\* entryPointInfo, const JavascriptMethod originalEntryPoint);

FunctionEntryPointInfo \*GetSimpleJitEntryPointInfo() const;

void SetSimpleJitEntryPointInfo(FunctionEntryPointInfo \*const entryPointInfo);

private:

void VerifyExecutionMode(const ExecutionMode executionMode) const;

public:

ExecutionMode GetDefaultInterpreterExecutionMode() const;

ExecutionMode GetExecutionMode() const;

ExecutionMode GetInterpreterExecutionMode(const bool isPostBailout);

void SetExecutionMode(const ExecutionMode executionMode);

private:

bool IsInterpreterExecutionMode() const;

public:

bool TryTransitionToNextExecutionMode();

void TryTransitionToNextInterpreterExecutionMode();

void SetIsSpeculativeJitCandidate();

bool TryTransitionToJitExecutionMode();

void TransitionToSimpleJitExecutionMode();

void TransitionToFullJitExecutionMode();

private:

void VerifyExecutionModeLimits();

void InitializeExecutionModeAndLimits();

public:

void ReinitializeExecutionModeAndLimits();

private:

void SetFullJitThreshold(const uint16 newFullJitThreshold, const bool skipSimpleJit = false);

void CommitExecutedIterations();

void CommitExecutedIterations(uint16 &limit, const uint executedIterations);

private:

uint16 GetSimpleJitExecutedIterations() const;

public:

void ResetSimpleJitLimitAndCallCount();

private:

void SetSimpleJitCallCount(const uint16 simpleJitLimit) const;

void ResetSimpleJitCallCount();

public:

uint16 GetProfiledIterations() const;

public:

void OnFullJitDequeued(const FunctionEntryPointInfo \*const entryPointInfo);

public:

void TraceExecutionMode(const char \*const eventDescription = nullptr) const;

void TraceInterpreterExecutionMode() const;

private:

void DoTraceExecutionMode(const char \*const eventDescription) const;

public:

static bool IsNewSimpleJit();

bool DoSimpleJit() const;

bool DoSimpleJitDynamicProfile() const;

private:

bool DoInterpreterProfile() const;

bool DoInterpreterAutoProfile() const;

public:

bool WasCalledFromLoop() const;

void SetWasCalledFromLoop();

public:

bool RecentlyBailedOutOfJittedLoopBody() const;

void SetRecentlyBailedOutOfJittedLoopBody(const bool value);

private:

static uint16 GetMinProfileIterations();

public:

static uint16 GetMinFunctionProfileIterations();

private:

static uint GetMinLoopProfileIterations(const uint loopInterpreterLimit);

public:

uint GetLoopProfileThreshold(const uint loopInterpreterLimit) const;

private:

static uint GetReducedLoopInterpretCount();

public:

uint GetLoopInterpretCount(LoopHeader\* loopHeader) const;

private:

static bool DoObjectHeaderInlining();

static bool DoObjectHeaderInliningForConstructors();

public:

static bool DoObjectHeaderInliningForConstructor(const uint32 inlineSlotCapacity);

private:

static bool DoObjectHeaderInliningForObjectLiterals();

public:

static bool DoObjectHeaderInliningForObjectLiteral(const uint32 inlineSlotCapacity);

static bool DoObjectHeaderInliningForObjectLiteral(const PropertyIdArray \*const propIds, ScriptContext \*const scriptContext);

static bool DoObjectHeaderInliningForEmptyObjects();

public:

#if DBG

int GetProfileSession() { return m\_iProfileSession; }

#endif

virtual void Finalize(bool isShutdown) override;

void Cleanup(bool isScriptContextClosing);

void CleanupSourceInfo(bool isScriptContextClosing);

template<bool IsScriptContextShutdown>

void CleanUpInlineCaches();

void CleanupRecyclerData(bool isRecyclerShutdown, bool doEntryPointCleanupCaptureStack);

#ifdef PERF\_COUNTERS

void CleanupPerfCounter();

#endif

virtual void Dispose(bool isShutdown) override { }

bool HasRejit() const

{

if(this->entryPoints)

{

return this->entryPoints->Count() > 1;

}

return false;

}

#pragma region SourceInfo Methods

void CopySourceInfo(ParseableFunctionInfo\* originalFunctionInfo);

void FinishSourceInfo();

RegSlot GetFrameDisplayRegister() const;

void SetFrameDisplayRegister(RegSlot frameDisplayRegister);

RegSlot GetObjectRegister() const;

void SetObjectRegister(RegSlot objectRegister);

bool HasObjectRegister() const { return GetObjectRegister() != 0; }

ScopeObjectChain \*GetScopeObjectChain() const;

void SetScopeObjectChain(ScopeObjectChain \*pScopeObjectChain);

// fetch the Catch scope object which encloses the passed bytecode offset, returns NULL otherwise

Js::DebuggerScope \* GetDiagCatchScopeObjectAt(int byteCodeOffset);

ByteBlock \*GetProbeBackingBlock();

void SetProbeBackingBlock(ByteBlock\* probeBackingBlock);

bool HasLineBreak() const;

bool HasLineBreak(charcount\_t start, charcount\_t end) const;

bool HasGeneratedFromByteCodeCache() const { return this->byteCodeCache != nullptr; }

bool EndsAfter(size\_t offset) const;

void TrackLoad(int ichMin);

SmallSpanSequence\* GetStatementMapSpanSequence() const { return m\_sourceInfo.pSpanSequence; }

void RecordStatementMap(StatementMap\* statementMap);

void RecordStatementMap(SmallSpanSequenceIter &iter, StatementData \* data);

void RecordLoad(int ichMin, int bytecodeAfterLoad);

DebuggerScope\* RecordStartScopeObject(DiagExtraScopesType scopeType, int start, RegSlot scopeLocation, int\* index = nullptr);

void RecordEndScopeObject(DebuggerScope\* currentScope, int end);

DebuggerScope\* AddScopeObject(DiagExtraScopesType scopeType, int start, RegSlot scopeLocation);

bool TryGetDebuggerScopeAt(int index, DebuggerScope\*& debuggerScope);

StatementMapList \* GetStatementMaps() const;

static StatementMap \* GetNextNonSubexpressionStatementMap(StatementMapList \*statementMapList, int & startingAtIndex);

static StatementMap \* GetPrevNonSubexpressionStatementMap(StatementMapList \*statementMapList, int & startingAtIndex);

void RecordStatementAdjustment(uint offset, StatementAdjustmentType adjType);

void RecordCrossFrameEntryExitRecord(uint byteCodeOffset, bool isEnterBlock);

// Find out an offset falls within the range. returns TRUE if found.

BOOL GetBranchOffsetWithin(uint start, uint end, StatementAdjustmentRecord\* record);

bool GetLineCharOffset(int byteCodeOffset, ULONG\* line, LONG\* charOffset, bool canAllocateLineCache = true);

bool GetLineCharOffsetFromStartChar(int startCharOfStatement, ULONG\* \_line, LONG\* \_charOffset, bool canAllocateLineCache = true);

// Given bytecode position, returns the start position of the statement and length of the statement.

bool GetStatementIndexAndLengthAt(int byteCodeOffset, UINT32\* statementIndex, UINT32\* statementLength);

// skip any utf-8/utf-16 byte-order-mark. Returns the number of chars skipped.

static charcount\_t SkipByteOrderMark(\_\_in\_bcount\_z(4) LPCUTF8& documentStart)

{

charcount\_t retValue = 0;

Assert(documentStart != nullptr);

if (documentStart[0] == 0xEF &&

documentStart[1] == 0xBB &&

documentStart[2] == 0xBF)

{

// UTF-8 - EF BB BF

// 3 bytes skipped - reports one char skipped

documentStart += 3;

retValue = 1;

}

else if ((documentStart[0] == 0xFF && documentStart[1] == 0xFE) ||

(documentStart[0] == 0xFE && documentStart[1] == 0xFF))

{

// UTF-16 LE - FF FE

// UTF-16 BE - FE FF

// 2 bytes skipped - reports one char skipped

documentStart += 2;

retValue = 1;

}

return retValue;

}

StatementMap\* GetMatchingStatementMapFromByteCode(int byteCodeOffset, bool ignoreSubexpressions = false);

int GetEnclosingStatementIndexFromByteCode(int byteCodeOffset, bool ignoreSubexpressions = false);

StatementMap\* GetEnclosingStatementMapFromByteCode(int byteCodeOffset, bool ignoreSubexpressions = false);

StatementMap\* GetMatchingStatementMapFromSource(int byteCodeOffset, int\* pMapIndex = nullptr);

void RecordFrameDisplayRegister(RegSlot slot);

void RecordObjectRegister(RegSlot slot);

CrossFrameEntryExitRecordList\* GetCrossFrameEntryExitRecords();

#ifdef VTUNE\_PROFILING

uint GetStartOffset(uint statementIndex) const;

ULONG GetSourceLineNumber(uint statementIndex);

#endif

#pragma endregion

// Field accessors

bool GetHasBailoutInstrInJittedCode() const { return this->m\_hasBailoutInstrInJittedCode; }

void SetHasBailoutInstrInJittedCode(bool hasBailout) { this->m\_hasBailoutInstrInJittedCode = hasBailout; }

bool GetCanReleaseLoopHeaders() const { return (this->m\_depth == 0); }

void SetPendingLoopHeaderRelease(bool pendingLoopHeaderRelease) { this->m\_pendingLoopHeaderRelease = pendingLoopHeaderRelease; }

bool GetIsFromNativeCodeModule() const { return m\_isFromNativeCodeModule; }

void SetIsFromNativeCodeModule(bool isFromNativeCodeModule) { m\_isFromNativeCodeModule = isFromNativeCodeModule; }

uint GetLoopNumber(LoopHeader const \* loopHeader) const;

bool GetHasAllocatedLoopHeaders() { return this->loopHeaderArray != nullptr; }

#if ENABLE\_NATIVE\_CODEGEN

Js::JavascriptMethod GetLoopBodyEntryPoint(Js::LoopHeader \* loopHeader, int entryPointIndex);

void SetLoopBodyEntryPoint(Js::LoopHeader \* loopHeader, EntryPointInfo\* entryPointInfo, Js::JavascriptMethod entryPoint);

#endif

void RestoreOldDefaultEntryPoint(FunctionEntryPointInfo\* oldEntryPoint, JavascriptMethod oldOriginalEntryPoint, FunctionEntryPointInfo\* newEntryPoint);

FunctionEntryPointInfo\* CreateNewDefaultEntryPoint();

void AddEntryPointToEntryPointList(FunctionEntryPointInfo\* entryPoint);

// Kind of entry point for original entry point

BOOL IsInterpreterThunk() const;

BOOL IsDynamicInterpreterThunk() const;

BOOL IsNativeOriginalEntryPoint() const;

bool IsSimpleJitOriginalEntryPoint() const;

#if DYNAMIC\_INTERPRETER\_THUNK

static BYTE GetOffsetOfDynamicInterpreterThunk() { return offsetof(FunctionBody, m\_dynamicInterpreterThunk); }

void\* GetDynamicInterpreterEntryPoint() const

{

return m\_dynamicInterpreterThunk;

}

bool HasInterpreterThunkGenerated() const

{

return m\_dynamicInterpreterThunk != nullptr;

}

DWORD GetDynamicInterpreterThunkSize() const;

#endif

bool GetHasHotLoop() const { return hasHotLoop; };

void SetHasHotLoop();

bool GetHasNestedLoop() const { return hasNestedLoop; };

void SetHasNestedLoop(bool nest) { hasNestedLoop = nest; };

bool IsInlineApplyDisabled();

void InitDisableInlineApply();

void SetDisableInlineApply(bool set);

bool IsInlineSpreadDisabled() const { return disableInlineSpread; }

void InitDisableInlineSpread() { disableInlineSpread = this->functionId != Js::Constants::NoFunctionId && PHASE\_OFF(Js::InlinePhase, this); }

void SetDisableInlineSpread(bool set) { disableInlineSpread = set; }

bool CheckCalleeContextForInlining(FunctionProxy\* calleeFunctionProxy);

#if DBG

bool HasValidSourceInfo();

#endif

#if DYNAMIC\_INTERPRETER\_THUNK

JavascriptMethod EnsureDynamicInterpreterThunk(FunctionEntryPointInfo\* entryPointInfo);

#endif

void SetCheckCodeGenEntryPoint(FunctionEntryPointInfo\* entryPointInfo, JavascriptMethod entryPoint);

#if ENABLE\_NATIVE\_CODEGEN

typedef void (\*SetNativeEntryPointFuncType)(FunctionEntryPointInfo\* entryPointInfo, Js::FunctionBody \* functionBody, Js::JavascriptMethod entryPoint);

static void DefaultSetNativeEntryPoint(FunctionEntryPointInfo\* entryPointInfo, FunctionBody \* functionBody, JavascriptMethod entryPoint);

static void ProfileSetNativeEntryPoint(FunctionEntryPointInfo\* entryPointInfo, FunctionBody \* functionBody, JavascriptMethod entryPoint);

bool GetNativeEntryPointUsed() const { return m\_nativeEntryPointUsed; }

void SetNativeEntryPointUsed(bool nativeEntryPointUsed) { this->m\_nativeEntryPointUsed = nativeEntryPointUsed; }

#endif

bool GetIsFuncRegistered() { return m\_isFuncRegistered; }

void SetIsFuncRegistered(bool isRegistered) { m\_isFuncRegistered = isRegistered; }

bool GetHasLoops() const { return loopCount != 0; }

uint IncrLoopCount() { return this->loopCount++; }

uint GetLoopCount() const { return this->loopCount; }

bool AllocProfiledDivOrRem(ProfileId\* profileId) { if (this->profiledDivOrRemCount != Constants::NoProfileId) { \*profileId = this->profiledDivOrRemCount++; return true; } return false; }

ProfileId GetProfiledDivOrRemCount() { return this->profiledDivOrRemCount; }

bool AllocProfiledSwitch(ProfileId\* profileId) { if (this->profiledSwitchCount != Constants::NoProfileId) { \*profileId = this->profiledSwitchCount++; return true; } return false; }

ProfileId GetProfiledSwitchCount() { return this->profiledSwitchCount; }

bool AllocProfiledCallSiteId(ProfileId\* profileId) { if (this->profiledCallSiteCount != Constants::NoProfileId) { \*profileId = this->profiledCallSiteCount++; return true; } return false; }

ProfileId GetProfiledCallSiteCount() const { return this->profiledCallSiteCount; }

void SetProfiledCallSiteCount(ProfileId callSiteId) { this->profiledCallSiteCount = callSiteId; }

bool AllocProfiledArrayCallSiteId(ProfileId\* profileId) { if (this->profiledArrayCallSiteCount != Constants::NoProfileId) { \*profileId = this->profiledArrayCallSiteCount++; return true; } return false; }

ProfileId GetProfiledArrayCallSiteCount() const { return this->profiledArrayCallSiteCount; }

bool AllocProfiledReturnTypeId(ProfileId\* profileId) { if (this->profiledReturnTypeCount != Constants::NoProfileId) { \*profileId = this->profiledReturnTypeCount++; return true; } return false; }

ProfileId GetProfiledReturnTypeCount() const { return this->profiledReturnTypeCount; }

bool AllocProfiledSlotId(ProfileId\* profileId) { if (this->profiledSlotCount != Constants::NoProfileId) { \*profileId = this->profiledSlotCount++; return true; } return false; }

ProfileId GetProfiledSlotCount() const { return this->profiledSlotCount; }

ProfileId AllocProfiledLdElemId(ProfileId\* profileId) { if (this->profiledLdElemCount != Constants::NoProfileId) { \*profileId = this->profiledLdElemCount++; return true; } return false; }

ProfileId GetProfiledLdElemCount() const { return this->profiledLdElemCount; }

bool AllocProfiledStElemId(ProfileId\* profileId) { if (this->profiledStElemCount != Constants::NoProfileId) { \*profileId = this->profiledStElemCount++; return true; } return false; }

ProfileId GetProfiledStElemCount() const { return this->profiledStElemCount; }

uint GetProfiledFldCount() const { return this->GetInlineCacheCount(); }

ArgSlot GetProfiledInParamsCount() const { return this->GetInParamsCount() > 1? this->GetInParamsCount() - 1 : 0; }

bool IsPartialDeserializedFunction() { return this->m\_isPartialDeserializedFunction; }

#ifdef PERF\_COUNTERS

bool IsDeserializedFunction() { return this->m\_isDeserializedFunction; }

#endif

#ifdef IR\_VIEWER

bool IsIRDumpEnabled() const { return this->m\_isIRDumpEnabled; }

void SetIRDumpEnabled(bool enabled) { this->m\_isIRDumpEnabled = enabled; }

Js::DynamicObject \* GetIRDumpBaseObject();

#endif /\* IR\_VIEWER \*/

#if ENABLE\_NATIVE\_CODEGEN

void SetPolymorphicCallSiteInfoHead(PolymorphicCallSiteInfo \*polyCallSiteInfo) { this->polymorphicCallSiteInfoHead = polyCallSiteInfo;}

PolymorphicCallSiteInfo \* GetPolymorphicCallSiteInfoHead() { return this->polymorphicCallSiteInfoHead;}

#endif

PolymorphicInlineCache \* GetPolymorphicInlineCachesHead() { return this->polymorphicInlineCachesHead; }

void SetPolymorphicInlineCachesHead(PolymorphicInlineCache \* cache) { this->polymorphicInlineCachesHead = cache; }

bool PolyInliningUsingFixedMethodsAllowedByConfigFlags(FunctionBody\* topFunctionBody)

{

return !PHASE\_OFF(Js::InlinePhase, this) && !PHASE\_OFF(Js::InlinePhase, topFunctionBody) &&

!PHASE\_OFF(Js::PolymorphicInlinePhase, this) && !PHASE\_OFF(Js::PolymorphicInlinePhase, topFunctionBody) &&

!PHASE\_OFF(Js::FixedMethodsPhase, this) && !PHASE\_OFF(Js::FixedMethodsPhase, topFunctionBody) &&

!PHASE\_OFF(Js::PolymorphicInlineFixedMethodsPhase, this) && !PHASE\_OFF(Js::PolymorphicInlineFixedMethodsPhase, topFunctionBody);

}

Js::PropertyId \* GetPropertyIdsForScopeSlotArray() const

{

return this->propertyIdsForScopeSlotArray;

}

void SetPropertyIdsForScopeSlotArray(Js::PropertyId \* propertyIdsForScopeSlotArray, uint scopeSlotCount)

{

this->scopeSlotArraySize = scopeSlotCount;

this->propertyIdsForScopeSlotArray = propertyIdsForScopeSlotArray;

}

Js::PropertyIdOnRegSlotsContainer \* GetPropertyIdOnRegSlotsContainer() const

{

return this->propertyIdOnRegSlotsContainer;

}

private:

void ResetProfileIds();

void SetFlags(bool does, FunctionBodyFlags newFlags)

{

if (does)

{

flags = (FunctionBodyFlags)(flags | newFlags);

}

else

{

flags = (FunctionBodyFlags)(flags & ~newFlags);

}

}

public:

bool GetHasThis() const { return (flags & Flags\_HasThis) != 0; }

void SetHasThis(bool has) { SetFlags(has, Flags\_HasThis); }

bool GetHasTry() const { return (flags & Flags\_HasTry) != 0; }

void SetHasTry(bool has) { SetFlags(has, Flags\_HasTry); }

bool GetHasFinally() const { return m\_hasFinally; }

void SetHasFinally(bool has){ m\_hasFinally = has; }

bool GetFuncEscapes() const { return funcEscapes; }

void SetFuncEscapes(bool does) { funcEscapes = does; }

bool GetHasOrParentHasArguments() const { return (flags & Flags\_HasOrParentHasArguments) != 0; }

void SetHasOrParentHasArguments(bool has) { SetFlags(has, Flags\_HasOrParentHasArguments); }

bool DoStackNestedFunc() const { return (flags & Flags\_StackNestedFunc) != 0; }

void SetStackNestedFunc(bool does) { SetFlags(does, Flags\_StackNestedFunc); }

#if DBG

bool CanDoStackNestedFunc() const { return m\_canDoStackNestedFunc; }

void SetCanDoStackNestedFunc() { m\_canDoStackNestedFunc = true; }

#endif

FunctionBody \* GetStackNestedFuncParent();

FunctionBody \* GetAndClearStackNestedFuncParent();

void ClearStackNestedFuncParent();

void SetStackNestedFuncParent(FunctionBody \* parentFunctionBody);

#if defined(\_M\_IX86) || defined(\_M\_X64)

bool DoStackClosure() const

{

return DoStackNestedFunc() && GetNestedCount() != 0 && !PHASE\_OFF(StackClosurePhase, this) && scopeSlotArraySize != 0 && GetEnvDepth() != (uint16)-1;

}

#else

bool DoStackClosure() const

{

return false;

}

#endif

bool DoStackFrameDisplay() const { return DoStackClosure(); }

bool DoStackScopeSlots() const { return DoStackClosure(); }

bool IsNonUserCode() const { return (flags & Flags\_NonUserCode) != 0; }

void SetIsNonUserCode(bool set);

bool GetHasNoExplicitReturnValue() { return (flags & Flags\_HasNoExplicitReturnValue) != 0; }

void SetHasNoExplicitReturnValue(bool has) { SetFlags(has, Flags\_HasNoExplicitReturnValue); }

bool GetHasOnlyThisStmts() const { return (flags & Flags\_HasOnlyThisStatements) != 0; }

void SetHasOnlyThisStmts(bool has) { SetFlags(has, Flags\_HasOnlyThisStatements); }

bool GetIsFirstFunctionObject() const { return m\_firstFunctionObject; }

void SetIsNotFirstFunctionObject() { m\_firstFunctionObject = false; }

bool GetInlineCachesOnFunctionObject() { return m\_inlineCachesOnFunctionObject; }

void SetInlineCachesOnFunctionObject(bool has) { m\_inlineCachesOnFunctionObject = has; }

bool GetHasRestParameter() const { return (flags & Flags\_HasRestParameter) != 0; }

void SetHasRestParameter() { SetFlags(true, Flags\_HasRestParameter); }

uint GetNumberOfRecursiveCallSites();

bool CanInlineRecursively(uint depth, bool tryAggressive = true);

public:

bool CanInlineAgain() const

{

// Block excessive recursive inlining of the same function

return inlineDepth < static\_cast<byte>(max(1, min(0xff, CONFIG\_FLAG(MaxFuncInlineDepth))));

}

void OnBeginInlineInto()

{

++inlineDepth;

}

void OnEndInlineInto()

{

--inlineDepth;

}

uint8 IncrementBailOnMisingProfileCount() { return ++bailOnMisingProfileCount; }

void ResetBailOnMisingProfileCount() { bailOnMisingProfileCount = 0; }

uint8 IncrementBailOnMisingProfileRejitCount() { return ++bailOnMisingProfileRejitCount; }

uint32 GetFrameHeight(EntryPointInfo\* entryPointInfo) const;

void SetFrameHeight(EntryPointInfo\* entryPointInfo, uint32 frameHeight);

RegSlot GetLocalsCount();

RegSlot GetConstantCount() const { return m\_constCount; }

void SetConstantCount(RegSlot cNewConstants);

RegSlot GetVarCount();

void SetVarCount(RegSlot cNewVars);

RegSlot MapRegSlot(RegSlot reg)

{

if (this->RegIsConst(reg))

{

reg = CONSTREG\_TO\_REGSLOT(reg);

Assert(reg < this->GetConstantCount());

}

else

{

reg += this->GetConstantCount();

}

return reg;

}

bool RegIsConst(RegSlot reg) { return reg > REGSLOT\_TO\_CONSTREG(this->GetConstantCount()); }

uint32 GetNonTempLocalVarCount();

uint32 GetFirstNonTempLocalIndex();

uint32 GetEndNonTempLocalIndex();

bool IsNonTempLocalVar(uint32 varIndex);

bool GetSlotOffset(RegSlot slotId, int32 \* slotOffset, bool allowTemp = false);

RegSlot GetOutParamsDepth();

void SetOutParamDepth(RegSlot cOutParamsDepth);

RegSlot GetYieldRegister();

RegSlot GetFirstTmpReg();

void SetFirstTmpReg(RegSlot firstTmpReg);

RegSlot GetTempCount();

Js::ModuleID GetModuleID() const;

void CreateConstantTable();

void RecordNullObject(RegSlot location);

void RecordUndefinedObject(RegSlot location);

void RecordTrueObject(RegSlot location);

void RecordFalseObject(RegSlot location);

void RecordIntConstant(RegSlot location, unsigned int val);

void RecordStrConstant(RegSlot location, LPCOLESTR psz, ulong cch);

void RecordFloatConstant(RegSlot location, double d);

void RecordNullDisplayConstant(RegSlot location);

void RecordStrictNullDisplayConstant(RegSlot location);

void InitConstantSlots(Var \*dstSlots);

Var GetConstantVar(RegSlot location);

void\* GetConstTable() const{return m\_constTable;}

void CloneConstantTable(FunctionBody \*newFunc);

void MarkScript(ByteBlock \* pblkByteCode, ByteBlock \* pblkAuxiliaryData, ByteBlock\* auxContextBlock,

uint byteCodeCount, uint byteCodeInLoopCount, uint byteCodeWithoutLDACount);

void BeginExecution();

void EndExecution();

SourceInfo \* GetSourceInfo() { return &this->m\_sourceInfo; }

bool InstallProbe(int offset);

bool UninstallProbe(int offset);

bool ProbeAtOffset(int offsest, OpCode\* pOriginalOpcode);

FunctionBody \* Clone(ScriptContext \*scriptContext, uint sourceIndex = Js::Constants::InvalidSourceIndex);

static bool ShouldShareInlineCaches() { return CONFIG\_FLAG(ShareInlineCaches); }

uint GetInlineCacheCount() const { return inlineCacheCount; }

uint GetRootObjectLoadInlineCacheStart() const { return rootObjectLoadInlineCacheStart; }

uint GetRootObjectLoadMethodInlineCacheStart() const { return rootObjectLoadMethodInlineCacheStart; }

uint GetRootObjectStoreInlineCacheStart() const { return rootObjectStoreInlineCacheStart; }

uint GetIsInstInlineCacheCount() const { return isInstInlineCacheCount; }

uint GetReferencedPropertyIdCount() const { return referencedPropertyIdCount; }

void AllocateInlineCache();

InlineCache \* GetInlineCache(uint index);

bool CanFunctionObjectHaveInlineCaches();

void\*\* GetInlineCaches();

#if DBG

byte\* GetInlineCacheTypes();

#endif

InlineCache \* GetRootObjectInlineCache(uint index);

IsInstInlineCache \* GetIsInstInlineCache(uint index);

PolymorphicInlineCache \* GetPolymorphicInlineCache(uint index);

PolymorphicInlineCache \* CreateNewPolymorphicInlineCache(uint index, PropertyId propertyId, InlineCache \* inlineCache);

PolymorphicInlineCache \* CreateBiggerPolymorphicInlineCache(uint index, PropertyId propertyId);

private:

void ResetInlineCaches();

PolymorphicInlineCache \* CreatePolymorphicInlineCache(uint index, uint16 size);

uint32 m\_asmJsTotalLoopCount;

public:

void CreateCacheIdToPropertyIdMap();

void CreateCacheIdToPropertyIdMap(uint rootObjectLoadInlineCacheStart, uint rootObjectLoadMethodInlineCacheStart, uint rootObjectStoreInlineCacheStart,

uint totalFieldAccessInlineCacheCount, uint isInstInlineCacheCount);

void SetPropertyIdForCacheId(uint cacheId, PropertyId propertyId);

PropertyId GetPropertyIdFromCacheId(uint cacheId)

{

Assert(this->cacheIdToPropertyIdMap);

Assert(cacheId < this->GetInlineCacheCount());

return this->cacheIdToPropertyIdMap[cacheId];

}

#if DBG

void VerifyCacheIdToPropertyIdMap();

#endif

void CreateReferencedPropertyIdMap(uint referencedPropertyIdCount);

void CreateReferencedPropertyIdMap();

PropertyId GetReferencedPropertyIdWithMapIndex(uint mapIndex);

void SetReferencedPropertyIdWithMapIndex(uint mapIndex, PropertyId propertyId);

PropertyId GetReferencedPropertyId(uint index);

#if DBG

void VerifyReferencedPropertyIdMap();

#endif

#ifdef ENABLE\_DEBUG\_CONFIG\_OPTIONS

void DumpFullFunctionName();

void DumpFunctionId(bool pad);

uint GetTraceFunctionNumber() const;

#endif

public:

uint NewObjectLiteral();

void AllocateObjectLiteralTypeArray();

DynamicType \*\* GetObjectLiteralTypeRef(uint index);

uint NewLiteralRegex();

uint GetLiteralRegexCount() const;

void AllocateLiteralRegexArray();

UnifiedRegex::RegexPattern \*GetLiteralRegex(const uint index);

#ifndef TEMP\_DISABLE\_ASMJS

AsmJsFunctionInfo\* GetAsmJsFunctionInfo()const {return asmJsFunctionInfo; }

AsmJsFunctionInfo\* AllocateAsmJsFunctionInfo();

AsmJsModuleInfo\* GetAsmJsModuleInfo()const{ return asmJsModuleInfo; }

void ResetAsmJsInfo()

{

asmJsFunctionInfo = nullptr;

asmJsModuleInfo = nullptr;

}

bool IsAsmJSModule()const{ if (asmJsModuleInfo) return true; return false; }

AsmJsModuleInfo\* AllocateAsmJsModuleInfo();

#endif

void SetLiteralRegex(const uint index, UnifiedRegex::RegexPattern \*const pattern);

private:

void ResetLiteralRegexes();

void ResetObjectLiteralTypes();

public:

void ResetByteCodeGenState();

void ResetByteCodeGenVisitState();

void FindClosestStatements(long characterOffset, StatementLocation \*firstStatementLocation, StatementLocation \*secondStatementLocation);

#if ENABLE\_NATIVE\_CODEGEN

const FunctionCodeGenRuntimeData \*GetInlineeCodeGenRuntimeData(const ProfileId profiledCallSiteId) const;

const FunctionCodeGenRuntimeData \*GetInlineeCodeGenRuntimeDataForTargetInlinee(const ProfileId profiledCallSiteId, FunctionBody \*inlineeFuncBody) const;

FunctionCodeGenRuntimeData \*EnsureInlineeCodeGenRuntimeData(

Recycler \*const recycler,

\_\_in\_range(0, profiledCallSiteCount - 1) const ProfileId profiledCallSiteId,

FunctionBody \*const inlinee);

const FunctionCodeGenRuntimeData \*GetLdFldInlineeCodeGenRuntimeData(const InlineCacheIndex inlineCacheIndex) const;

FunctionCodeGenRuntimeData \*EnsureLdFldInlineeCodeGenRuntimeData(

Recycler \*const recycler,

\_\_in\_range(0, this->inlineCacheCount - 1) const InlineCacheIndex inlineCacheIndex,

FunctionBody \*const inlinee);

void LoadDynamicProfileInfo();

bool HasExecutionDynamicProfileInfo() const { return hasExecutionDynamicProfileInfo; }

bool HasDynamicProfileInfo() const { return dynamicProfileInfo != nullptr; }

bool NeedEnsureDynamicProfileInfo() const;

DynamicProfileInfo \* GetDynamicProfileInfo() const { Assert(HasExecutionDynamicProfileInfo()); return dynamicProfileInfo; }

DynamicProfileInfo \* GetAnyDynamicProfileInfo() const { Assert(HasDynamicProfileInfo()); return dynamicProfileInfo; }

DynamicProfileInfo \* EnsureDynamicProfileInfo();

DynamicProfileInfo \* AllocateDynamicProfile();

BYTE GetSavedInlinerVersion() const;

uint32 GetSavedPolymorphicCacheState() const;

ImplicitCallFlags GetSavedImplicitCallsFlags() const;

bool HasNonBuiltInCallee();

bool HasFunctionBailOutRecord() const { return functionBailOutRecord != nullptr; }

FunctionBailOutRecord \* GetFunctionBailOutRecord() const { Assert(HasFunctionBailOutRecord()); return functionBailOutRecord; }

void SetFunctionBailOutRecord(FunctionBailOutRecord \* record) { Assert(!HasFunctionBailOutRecord()); functionBailOutRecord = record; }

void RecordNativeThrowMap(SmallSpanSequenceIter& iter, uint32 offset, uint32 statementIndex, EntryPointInfo\* entryPoint, uint loopNum);

void RecordNativeBaseAddress(BYTE\* baseAddress, ptrdiff\_t codeSizeS, NativeCodeData \* data, NativeCodeData \* transferData, CodeGenNumberChunk \* numberChunks,

EntryPointInfo\* info, uint loopNum);

void SetNativeThrowSpanSequence(SmallSpanSequence \*seq, uint loopNum, LoopEntryPointInfo\* entryPoint);

BOOL GetMatchingStatementMapFromNativeAddress(DWORD\_PTR codeAddress, StatementData &data, uint loopNum, FunctionBody \*inlinee = nullptr);

BOOL GetMatchingStatementMapFromNativeOffset(DWORD\_PTR codeAddress, uint32 offset, StatementData &data, uint loopNum, FunctionBody \*inlinee = nullptr);

FunctionEntryPointInfo \* GetEntryPointFromNativeAddress(DWORD\_PTR codeAddress);

LoopEntryPointInfo \* GetLoopEntryPointInfoFromNativeAddress(DWORD\_PTR codeAddress, uint loopNum) const;

#endif

void InsertSymbolToRegSlotList(JsUtil::CharacterBuffer<WCHAR> const& propName, RegSlot reg, RegSlot totalRegsCount);

void InsertSymbolToRegSlotList(RegSlot reg, PropertyId propertyId, RegSlot totalRegsCount);

void SetPropertyIdsOfFormals(PropertyIdArray \* formalArgs);

bool DontRethunkAfterBailout() const { return dontRethunkAfterBailout; }

void SetDontRethunkAfterBailout() { dontRethunkAfterBailout = true; }

void ClearDontRethunkAfterBailout() { dontRethunkAfterBailout = false; }

void SaveState(ParseNodePtr pnode);

void RestoreState(ParseNodePtr pnode);

// Used for the debug purpose, this info will be stored (in the non-debug mode), when a function has all locals marked as non-local-referenced.

// So when we got to no-refresh debug mode, and try to re-use the same function body we can then enforce all locals to be non-local-referenced.

bool HasAllNonLocalReferenced() const { return m\_hasAllNonLocalReferenced; }

void SetAllNonLocalReferenced(bool set) { m\_hasAllNonLocalReferenced = set; }

bool HasSetIsObject() const { return m\_hasSetIsObject; }

void SetHasSetIsObject(bool set) { m\_hasSetIsObject = set; }

bool HasFuncExprNameReference() const { return m\_hasFunExprNameReference; }

void SetFuncExprNameReference(bool value) { m\_hasFunExprNameReference = value; }

bool GetChildCallsEval() const { return m\_ChildCallsEval; }

void SetChildCallsEval(bool value) { m\_ChildCallsEval = value; }

bool GetCallsEval() const { return m\_CallsEval; }

void SetCallsEval(bool set) { m\_CallsEval = set; }

bool HasReferenceableBuiltInArguments() const { return m\_hasReferenceableBuiltInArguments; }

void SetHasReferenceableBuiltInArguments(bool value) { m\_hasReferenceableBuiltInArguments = value; }

// Used for the debug purpose. This is to avoid setting all locals to non-local-referenced, multiple time for each child function.

bool HasDoneAllNonLocalReferenced() const { return m\_hasDoneAllNonLocalReferenced; }

void SetHasDoneAllNonLocalReferenced(bool set) { m\_hasDoneAllNonLocalReferenced = set; }

// Once the function compiled is sent m\_hasFunctionCompiledSent will be set to 'true'. The below check will be used only to determine during ProfileModeDeferredParse function.

bool HasFunctionCompiledSent() const { return m\_hasFunctionCompiledSent; }

void SetHasFunctionCompiledSent(bool set) { m\_hasFunctionCompiledSent = set; }

#if DBG\_DUMP

void DumpStatementMaps();

void Dump();

void PrintStatementSourceLine(uint statementIndex);

void PrintStatementSourceLineFromStartOffset(uint cchStartOffset);

void DumpScopes();

#endif

uint GetStatementStartOffset(const uint statementIndex);

#ifdef IR\_VIEWER

void GetSourceLineFromStartOffset(const uint startOffset, LPCUTF8 \*sourceBegin, LPCUTF8 \*sourceEnd,

ULONG \* line, LONG \* col);

void GetStatementSourceInfo(const uint statementIndex, LPCUTF8 \*sourceBegin, LPCUTF8 \*sourceEnd,

ULONG \* line, LONG \* col);

#endif

HRESULT RegisterFunction(BOOL fChangeMode, BOOL fOnlyCurrent = FALSE);

HRESULT ReportScriptCompiled();

HRESULT ReportFunctionCompiled();

void SetEntryToProfileMode();

void CheckAndRegisterFuncToDiag(ScriptContext \*scriptContext);

void SetEntryToDeferParseForDebugger();

void ResetEntryPoint();

void CleanupToReparse();

void AddDeferParseAttribute();

void RemoveDeferParseAttribute();

#if DBG

void MustBeInDebugMode();

#endif

static bool IsDummyGlobalRetStatement(const regex::Interval \*sourceSpan)

{

Assert(sourceSpan != nullptr);

return sourceSpan->begin == 0 && sourceSpan->end == 0;

}

static void GetShortNameFromUrl(\_\_in LPCWSTR pchUrl, \_Out\_writes\_z\_(cchBuffer) LPWSTR pchShortName, \_\_in size\_t cchBuffer);

template<class Fn>

void MapLoopHeaders(Fn fn) const

{

if(this->loopHeaderArray)

{

for(uint i = 0; i < loopCount; i++)

{

fn(i , &this->loopHeaderArray[i]);

}

}

}

template <class Fn>

void MapEntryPoints(Fn fn) const

{

if (this->entryPoints)

{

this->entryPoints->Map([&fn] (int index, RecyclerWeakReference<FunctionEntryPointInfo>\* entryPoint) {

FunctionEntryPointInfo\* strongRef = entryPoint->Get();

if (strongRef)

{

fn(index, strongRef);

}

});

}

}

bool DoJITLoopBody() const

{

return IsJitLoopBodyPhaseEnabled() && this->loopHeaderArray != nullptr;

}

bool ForceJITLoopBody() const

{

return IsJitLoopBodyPhaseForced() && !this->GetHasTry();

}

bool IsGeneratorAndJitIsDisabled()

{

return this->IsGenerator() && !(CONFIG\_ISENABLED(Js::JitES6GeneratorsFlag) && !this->GetHasTry());

}

FunctionBodyFlags \* GetAddressOfFlags() { return &this->flags; }

Js::RegSlot GetRestParamRegSlot();

public:

void RecordConstant(RegSlot location, Var var);

private:

inline void CheckEmpty();

inline void CheckNotExecuting();

SmallSpanSequence \*GetThrowSpanSequence(DWORD\_PTR codeAddress, uint loopNum);

BOOL GetMatchingStatementMap(StatementData &data, int statementIndex, FunctionBody \*inlinee);

#if ENABLE\_NATIVE\_CODEGEN

int GetStatementIndexFromNativeOffset(SmallSpanSequence \*pThrowSpanSequence, uint32 nativeOffset);

int GetStatementIndexFromNativeAddress(SmallSpanSequence \*pThrowSpanSequence, DWORD\_PTR codeAddress, DWORD\_PTR nativeBaseAddress);

#endif

void EnsureAuxStatementData();

StatementAdjustmentRecordList\* GetStatementAdjustmentRecords();

#ifdef DYNAMIC\_PROFILE\_MUTATOR

friend class DynamicProfileMutator;

friend class DynamicProfileMutatorImpl;

#endif

friend class RemoteFunctionBody;

};

typedef SynchronizableList<FunctionBody\*, JsUtil::List<FunctionBody\*, ArenaAllocator, false, Js::FreeListedRemovePolicy> > FunctionBodyList;

struct ScopeSlots

{

public:

static uint const MaxEncodedSlotCount = USHORT\_MAX;

// The slot index is at the same location as the vtable, so that we can distinguish between scope slot and frame display

static uint const EncodedSlotCountSlotIndex = 0;

static uint const ScopeMetadataSlotIndex = 1; // Either a FunctionBody\* or DebuggerScope\*

static uint const FirstSlotIndex = 2;

public:

ScopeSlots(Var\* slotArray) : slotArray(slotArray)

{

}

bool IsFunctionScopeSlotArray()

{

return FunctionBody::Is(slotArray[ScopeMetadataSlotIndex]);

}

FunctionBody\* GetFunctionBody()

{

Assert(IsFunctionScopeSlotArray());

return (FunctionBody\*)(slotArray[ScopeMetadataSlotIndex]);

}

DebuggerScope\* GetDebuggerScope()

{

Assert(!IsFunctionScopeSlotArray());

return (DebuggerScope\*)(slotArray[ScopeMetadataSlotIndex]);

}

Var GetScopeMetadataRaw() const

{

return slotArray[ScopeMetadataSlotIndex];

}

void SetScopeMetadata(Var scopeMetadataObj)

{

slotArray[ScopeMetadataSlotIndex] = scopeMetadataObj;

}

uint GetCount() const

{

return ::Math::PointerCastToIntegralTruncate<uint>(slotArray[EncodedSlotCountSlotIndex]);

}

void SetCount(uint count)

{

slotArray[EncodedSlotCountSlotIndex] = (Var)min<uint>(count, ScopeSlots::MaxEncodedSlotCount);

}

Var Get(uint i) const

{

Assert(i < GetCount());

return slotArray[i + FirstSlotIndex];

}

void Set(uint i, Var value)

{

Assert(i < GetCount());

slotArray[i + FirstSlotIndex] = value;

}

template<class Fn>

void Map(Fn fn)

{

uint count = GetCount();

for(uint i = 0; i < count; i++)

{

fn(GetSlot[i]);

}

}

// The first pointer sized value in the object for scope slots is the count, while it is a vtable

// for Activation object or with scope (a recyclable object)

// VTable values are always > 64K because they are a pointer, hence anything less than that implies

// a slot array.

// CONSIDER: Use TaggedInt instead of range of slot count to distinguish slot array with others.

static bool Is(void\* object)

{

size\_t slotCount = \*((size\_t\*)object);

if(slotCount <= MaxEncodedSlotCount)

{

return true;

}

return false;

}

private:

Var\* slotArray;

};

enum ScopeType

{

ScopeType\_ActivationObject,

ScopeType\_SlotArray,

ScopeType\_WithScope

};

// A FrameDisplay encodes a FunctionObject's scope chain. It is an array of scopes, where each scope can be either an inline slot array

// or a RecyclableObject. A FrameDisplay for a given FunctionObject will consist of the FrameDisplay from it's enclosing scope, with any additional

// scopes prepended. Due to with statements etc. a function may introduce multiple scopes to the FrameDisplay.

struct FrameDisplay

{

FrameDisplay(uint16 len, bool strictMode = false) :

tag(true),

length(len),

strictMode(strictMode)

#if \_M\_X64

, unused(0)

#endif

{

}

void SetTag(bool tag) { this->tag = tag; }

void SetItem(uint index, void\* item);

void \*GetItem(uint index);

uint16 GetLength() const { return length; }

void SetLength(uint16 len) { this->length = len; }

bool GetStrictMode() const { return strictMode; }

void SetStrictMode(bool flag) { this->strictMode = flag; }

void\*\* GetDataAddress() { return (void\*\*)&this->scopes; }

static uint32 GetOffsetOfStrictMode() { return offsetof(FrameDisplay, strictMode); }

static uint32 GetOffsetOfLength() { return offsetof(FrameDisplay, length); }

static uint32 GetOffsetOfScopes() { return offsetof(FrameDisplay, scopes); }

static ScopeType GetScopeType(void\* scope);

private:

bool tag; // Tag it so that the NativeCodeGenerator::IsValidVar would not think this is var

bool strictMode;

uint16 length;

#if defined(\_M\_X64\_OR\_ARM64)

uint32 unused;

#endif

void\* scopes[];

};

#pragma region Function Body helper classes

#pragma region Debugging related source classes

// Contains only the beginning part of the statement. This will mainly used in SmallSpanSequence which will further be compressed

// and stored in the buffer

struct StatementData

{

StatementData()

: sourceBegin(0),

bytecodeBegin(0)

{

}

int sourceBegin;

int bytecodeBegin;

};

struct StatementLocation

{

Js::FunctionBody\* function;

regex::Interval statement;

regex::Interval bytecodeSpan;

};

// Small span in the Statement buffer of the SmallSpanSequence

struct SmallSpan

{

ushort sourceBegin;

ushort bytecodeBegin;

SmallSpan(uint32 val)

{

sourceBegin = (ushort)(val >> 16);

bytecodeBegin = (ushort)(val & 0x0000FFFF);

}

operator unsigned int()

{

return (uint32)sourceBegin << 16 | bytecodeBegin;

}

};

// Iterator which contains the state at particular index. These values will used when fetching next item from

// SmallSpanSequence

class SmallSpanSequenceIter

{

friend class SmallSpanSequence;

public:

SmallSpanSequenceIter()

: accumulatedIndex(-1),

accumulatedSourceBegin(0),

accumulatedBytecodeBegin(0),

indexOfActualOffset(0)

{

}

// Below are used for fast access when the last access happened nearby.

// so the actual index would be accumulatedIndex / 2 + (remainder for which byte).

int accumulatedIndex;

int accumulatedSourceBegin;

int accumulatedBytecodeBegin;

int indexOfActualOffset;

};

// This class compacts the range of the statement to BYTEs instead of ints.

// Instead of having start and end as int32s we will have them stored in bytes, and they will be

// treated as start offset and end offset.

// For simplicity, this class should be heap allocated, since it can be allocated from either the background

// or main thread.

class SmallSpanSequence

{

friend class SmallSpanSequenceIter;

friend class ByteCodeBufferBuilder;

friend class ByteCodeBufferReader;

private:

// Each item in the list contains two set of begins (one for bytecode and for sourcespan).

// The allowed valued for source and bytecode span is in between SHORT\_MAX - 1 to SHORT\_MIN (inclusive).

// otherwise its a miss

JsUtil::GrowingUint32HeapArray \* pStatementBuffer;

// Contains list of values which are missed in StatementBuffer.

JsUtil::GrowingUint32HeapArray \* pActualOffsetList;

// The first value of the sequence

int baseValue;

BOOL GetRangeAt(int index, SmallSpanSequenceIter &iter, int \* pCountOfMissed, StatementData & data);

ushort GetDiff(int current, int prev);

public:

SmallSpanSequence();

~SmallSpanSequence()

{

Cleanup();

}

void Cleanup()

{

if (pStatementBuffer != nullptr)

{

HeapDelete(pStatementBuffer);

}

if (pActualOffsetList != nullptr)

{

HeapDelete(pActualOffsetList);

}

}

// Trys to match passed bytecode in the statement, and returns the statement which includes that.

BOOL GetMatchingStatementFromBytecode(int bytecode, SmallSpanSequenceIter &iter, StatementData & data);

// Record the statement data in the statement buffer in the compressed manner.

BOOL RecordARange(SmallSpanSequenceIter &iter, StatementData \* data);

// Reset the accumulator's state and value.

void Reset(SmallSpanSequenceIter &iter);

uint32 Count() const { return pStatementBuffer ? pStatementBuffer->Count() : 0; }

BOOL Item(int index, SmallSpanSequenceIter &iter, StatementData &data);

// Below function will not change any state, so it will not alter accumulated index and value

BOOL Seek(int index, StatementData & data);

SmallSpanSequence \* Clone();

};

#pragma endregion

// This container represent the property ids for the locals which are placed at direct slot

// and list of formals args if user has not used the arguments object in the script for the current function

struct PropertyIdOnRegSlotsContainer

{

PropertyId \* propertyIdsForRegSlots;

uint length;

PropertyIdArray \* propertyIdsForFormalArgs;

PropertyIdOnRegSlotsContainer();

static PropertyIdOnRegSlotsContainer \* New(Recycler \* recycler);

void CreateRegSlotsArray(Recycler \* recycler, uint \_length);

void SetFormalArgs(PropertyIdArray \* formalArgs);

// Helper methods

void Insert(RegSlot reg, PropertyId propId);

void FetchItemAt(uint index, FunctionBody \*pFuncBody, \_\_out PropertyId \*pPropId, \_\_out RegSlot \*pRegSlot);

// Whether reg belongs to non-temp locals

bool IsRegSlotFormal(RegSlot reg);

};

// Flags for the DebuggerScopeProperty object.

typedef int DebuggerScopePropertyFlags;

const int DebuggerScopePropertyFlags\_None = 0x000000000;

const int DebuggerScopePropertyFlags\_Const = 0x000000001;

const int DebuggerScopePropertyFlags\_CatchObject = 0x000000002;

const int DebuggerScopePropertyFlags\_WithObject = 0x000000004;

const int DebuggerScopePropertyFlags\_ForInOrOfCollection = 0x000000008;

// Used to store local property info for with/catch objects, lets, or consts

// that are needed for the debugger.

class DebuggerScopeProperty

{

public:

Js::PropertyId propId; // The property ID of the scope variable.

RegSlot location; // Contains the location of the scope variable (regslot, slotarray, direct).

int byteCodeInitializationOffset; // The byte code offset used when comparing let/const variables for dead zone exclusion debugger side.

DebuggerScopePropertyFlags flags; // Flags for the property.

bool IsConst() const { return (flags & DebuggerScopePropertyFlags\_Const) != 0; }

bool IsCatchObject() const { return (flags & DebuggerScopePropertyFlags\_CatchObject) != 0; }

bool IsWithObject() const { return (flags & DebuggerScopePropertyFlags\_WithObject) != 0; }

bool IsForInOrForOfCollectionScope() const { return (flags & DebuggerScopePropertyFlags\_ForInOrOfCollection) != 0; }

public:

// Determines if the current property is in a dead zone. Note that the property makes

// no assumptions about what scope it's in, that is determined by DebuggerScope.

// byteCodeOffset - The current offset in bytecode that the debugger is at.

bool IsInDeadZone(int byteCodeOffset) const

{

if (IsForInOrForOfCollectionScope())

{

// These are let/const loop variables of a for-in or for-of loop

// in the scope for the collection expression. They are always

// in TDZ in this scope, never initialized by the bytecode.

return true;

}

if (this->byteCodeInitializationOffset == Constants::InvalidByteCodeOffset && !(IsCatchObject() || IsWithObject()))

{

AssertMsg(false, "Debug let/const property never had its initialization point updated. This indicates that a Ld or St operation in ByteCodeGenerator was missed that needs to have DebuggerScope::UpdatePropertyInitializationOffset() added to it.");

return false;

}

return byteCodeOffset < this->byteCodeInitializationOffset;

}

};

// Used to track with, catch, and block scopes for the debugger to determine context.

class DebuggerScope

{

public:

typedef JsUtil::List<DebuggerScopeProperty> DebuggerScopePropertyList;

DebuggerScope(Recycler\* recycler, DiagExtraScopesType scopeType, RegSlot scopeLocation, int rangeBegin)

: scopeType(scopeType),

scopeProperties(nullptr),

parentScope(nullptr),

siblingScope(nullptr),

scopeLocation(scopeLocation),

recycler(recycler)

{

this->range.begin = rangeBegin;

this->range.end = -1;

}

DebuggerScope \* GetSiblingScope(RegSlot location, FunctionBody \*functionBody);

void AddProperty(RegSlot location, Js::PropertyId propertyId, DebuggerScopePropertyFlags flags);

bool GetPropertyIndex(Js::PropertyId propertyId, int& i);

bool IsOffsetInScope(int offset) const;

bool Contains(Js::PropertyId propertyId, RegSlot location) const;

bool IsBlockScope() const;

bool IsBlockObjectScope() const

{

return this->scopeType == Js::DiagBlockScopeInObject;

}

bool IsCatchScope() const;

bool IsWithScope() const;

bool IsSlotScope() const;

bool HasProperties() const;

bool IsAncestorOf(const DebuggerScope\* potentialChildScope);

bool AreAllPropertiesInDeadZone(int byteCodeOffset) const;

RegSlot GetLocation() const { Assert(IsOwnScope()); return scopeLocation; }

bool IsOwnScope() const { return scopeLocation != Js::Constants::NoRegister; }

bool TryGetProperty(Js::PropertyId propertyId, RegSlot location, DebuggerScopeProperty\* outScopeProperty) const;

bool TryGetValidProperty(Js::PropertyId propertyId, RegSlot location, int offset, DebuggerScopeProperty\* outScopeProperty, bool\* isInDeadZone) const;

bool UpdatePropertyInitializationOffset(RegSlot location, Js::PropertyId propertyId, int byteCodeOffset, bool isFunctionDeclaration = false);

void UpdateDueToByteCodeRegeneration(DiagExtraScopesType scopeType, int start, RegSlot scopeLocation);

void UpdatePropertiesInForInOrOfCollectionScope();

void SetParentScope(DebuggerScope\* parentScope) { this->parentScope = parentScope; }

DebuggerScope\* GetParentScope() const { return parentScope; }

DebuggerScope\* FindCommonAncestor(DebuggerScope\* debuggerScope);

int GetEnd() const { return range.end; }

int GetStart() const { return range.begin; }

void SetScopeLocation(RegSlot scopeLocation) { this->scopeLocation = scopeLocation; }

void SetBegin(int begin);

void SetEnd(int end);

#if DBG

void Dump();

PCWSTR GetDebuggerScopeTypeString(DiagExtraScopesType scopeType);

#endif

public:

// The list of scope properties in this scope object.

// For with scope: Has 1 property that represents the scoped object.

// For catch scope: Has 1 property that represents the exception object.

// For block scope: Has 0-n properties that represent let/const variables in that scope.

DebuggerScopePropertyList\* scopeProperties;

DiagExtraScopesType scopeType; // The type of scope being represented (With, Catch, or Block scope).

DebuggerScope\* siblingScope; // Valid only when current scope is slot/activationobject and symbols are on direct regslot

static const int InvalidScopeIndex = -1;

private:

int GetScopeDepth() const;

bool UpdatePropertyInitializationOffsetInternal(RegSlot location, Js::PropertyId propertyId, int byteCodeOffset, bool isFunctionDeclaration = false);

void EnsurePropertyListIsAllocated();

private:

DebuggerScope\* parentScope;

regex::Interval range; // The start and end byte code writer offsets used when comparing where the debugger is currently stopped at (breakpoint location).

RegSlot scopeLocation;

Recycler\* recycler;

};

class ScopeObjectChain

{

public:

typedef JsUtil::List<DebuggerScope\*> ScopeObjectChainList;

ScopeObjectChain(Recycler\* recycler)

: pScopeChain(nullptr)

{

pScopeChain = RecyclerNew(recycler, ScopeObjectChainList, recycler);

}

// This function will return DebuggerScopeProperty when the property is found and correctly in the range.

// If the property is found, but the scope is not in the range, it will return false, but the out param (isPropertyInDebuggerScope) will set to true,

// and isConst will be updated.

// If the property is not found at all, it will return false, and isPropertyInDebuggerScope will be false.

bool TryGetDebuggerScopePropertyInfo(PropertyId propertyId, RegSlot location, int offset, bool\* isPropertyInDebuggerScope, bool \*isConst, bool\* isInDeadZone);

// List of all Scope Objects in a function. Scopes are added to this list as when they are created in bytecode gen part.

ScopeObjectChainList\* pScopeChain;

};

#pragma endregion

} // 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 "RuntimeBasePch.h"

namespace Js

{

FunctionInfo::FunctionInfo(JavascriptMethod entryPoint, Attributes attributes, LocalFunctionId functionId, FunctionBody\* functionBodyImpl)

: originalEntryPoint(entryPoint), attributes(attributes), functionBodyImpl(functionBodyImpl), functionId(functionId)

{

#if !DYNAMIC\_INTERPRETER\_THUNK

Assert(entryPoint != nullptr);

#endif

}

void FunctionInfo::VerifyOriginalEntryPoint() const

{

Assert(!this->HasBody() || this->IsDeferredParseFunction() || this->IsDeferredDeserializeFunction() || this->GetFunctionProxy()->HasValidEntryPoint());

}

JavascriptMethod

FunctionInfo::GetOriginalEntryPoint() const

{

VerifyOriginalEntryPoint();

return GetOriginalEntryPoint\_Unchecked();

}

JavascriptMethod FunctionInfo::GetOriginalEntryPoint\_Unchecked() const

{

return originalEntryPoint;

}

void FunctionInfo::SetOriginalEntryPoint(const JavascriptMethod originalEntryPoint)

{

Assert(originalEntryPoint);

this->originalEntryPoint = originalEntryPoint;

}

FunctionBody \*

FunctionInfo::GetFunctionBody() const

{

Assert(functionBodyImpl == nullptr || functionBodyImpl->IsFunctionBody());

return (FunctionBody \*)functionBodyImpl;

}

FunctionInfo::Attributes FunctionInfo::GetAttributes(Js::RecyclableObject \* function)

{

return function->GetTypeId() == Js::TypeIds\_Function ?

Js::JavascriptFunction::FromVar(function)->GetFunctionInfo()->GetAttributes() : Js::FunctionInfo::None;

}

}

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

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

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

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

namespace Js

{

class FunctionProxy;

class FunctionBody;

class ParseableFunctionInfo;

class DeferDeserializeFunctionInfo;

class FunctionInfo: public FinalizableObject

{

friend class RemoteFunctionBody;

protected:

DEFINE\_VTABLE\_CTOR\_NOBASE(FunctionInfo);

public:

enum Attributes : uint32

{

None = 0x00000,

ErrorOnNew = 0x00001,

SkipDefaultNewObject = 0x00002,

DoNotProfile = 0x00004,

HasNoSideEffect = 0x00008, // calling function doesn�t cause an implicit flags to be set,

// the callee will detect and set implicit flags on its individual operations

NeedCrossSiteSecurityCheck = 0x00010,

DeferredDeserialize = 0x00020, // The function represents something that needs to be deserialized on use

DeferredParse = 0x00040, // The function represents something that needs to be parsed on use

CanBeHoisted = 0x00080, // The function return value won't be changed in a loop so the evaluation can be hoisted.

SuperReference = 0x00100,

ClassMethod = 0x00200, // The function is a class method

ClassConstructor = 0x00400, // The function is a class constructor

DefaultConstructor = 0x00800, // The function is a default class constructor

Lambda = 0x01000,

CapturesThis = 0x02000, // Only lambdas will set this; denotes whether the lambda referred to this, used by debugger

Generator = 0x04000,

BuiltInInlinableAsLdFldInlinee = 0x08000,

Async = 0x10000

};

FunctionInfo(JavascriptMethod entryPoint, Attributes attributes = None, LocalFunctionId functionId = Js::Constants::NoFunctionId, FunctionBody\* functionBodyImpl = NULL);

static DWORD GetFunctionBodyImplOffset() { return offsetof(FunctionInfo, functionBodyImpl); }

void VerifyOriginalEntryPoint() const;

JavascriptMethod GetOriginalEntryPoint() const;

JavascriptMethod GetOriginalEntryPoint\_Unchecked() const;

void SetOriginalEntryPoint(const JavascriptMethod originalEntryPoint);

bool IsAsync() const { return ((this->attributes & Async) != 0); }

bool IsDeferred() const { return ((this->attributes & (DeferredDeserialize | DeferredParse)) != 0); }

bool IsLambda() const { return ((this->attributes & Lambda) != 0); }

bool IsConstructor() const { return ((this->attributes & ErrorOnNew) == 0); }

bool IsGenerator() const { return ((this->attributes & Generator) != 0); }

bool IsDefaultConstructor() const { return ((this->attributes & DefaultConstructor) != 0); }

bool IsClassConstructor() const { return ((this->attributes & ClassConstructor) != 0); }

bool IsClassMethod() const { return ((this->attributes & ClassMethod) != 0); }

bool HasSuperReference() const { return ((this->attributes & SuperReference) != 0); }

BOOL HasBody() const { return functionBodyImpl != NULL; }

BOOL HasParseableInfo() const { return this->HasBody() && !this->IsDeferredDeserializeFunction(); }

FunctionProxy \* GetFunctionProxy() const

{

return functionBodyImpl;

}

ParseableFunctionInfo\* GetParseableFunctionInfo() const

{

Assert(functionBodyImpl == NULL || !IsDeferredDeserializeFunction());

return (ParseableFunctionInfo\*) functionBodyImpl;

}

ParseableFunctionInfo\*\* GetParseableFunctionInfoRef() const

{

Assert(functionBodyImpl == NULL || !IsDeferredDeserializeFunction());

return (ParseableFunctionInfo\*\*)&functionBodyImpl;

}

DeferDeserializeFunctionInfo\* GetDeferDeserializeFunctionInfo() const

{

Assert(functionBodyImpl == NULL || IsDeferredDeserializeFunction());

return (DeferDeserializeFunctionInfo\*)functionBodyImpl;

}

FunctionBody \* GetFunctionBody() const;

Attributes GetAttributes() const { return attributes; }

static Attributes GetAttributes(Js::RecyclableObject \* function);

Js::LocalFunctionId GetLocalFunctionId() const { return functionId; }

virtual void Finalize(bool isShutdown)

{

}

virtual void Dispose(bool isShutdown)

{

}

virtual void Mark(Recycler \*recycler) override { AssertMsg(false, "Mark called on object that isn't TrackableObject"); }

BOOL IsDeferredDeserializeFunction() const { return ((this->attributes & DeferredDeserialize) == DeferredDeserialize); }

BOOL IsDeferredParseFunction() const { return ((this->attributes & DeferredParse) == DeferredParse); }

protected:

JavascriptMethod originalEntryPoint;

// WriteBarrier-TODO: Fix this? This is used only by proxies to keep the deserialized version around

// However, proxies are not allocated as write barrier memory currently so its fine to not set the write barrier for this field

FunctionProxy \* functionBodyImpl; // Implementation of the function- null if the function doesn't have a body

LocalFunctionId functionId; // Per host source context (source file) function Id

Attributes attributes;

};

// Helper FunctionInfo for builtins that we don't want to profile (script profiler).

class NoProfileFunctionInfo : public FunctionInfo

{

public:

NoProfileFunctionInfo(JavascriptMethod entryPoint)

: FunctionInfo(entryPoint, Attributes::DoNotProfile)

{}

};

};

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

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

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

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

#include "RuntimeBasePch.h"

namespace Js

{

double HiResTimer::GetSystemTime()

{

SYSTEMTIME stTime;

::GetSystemTime(&stTime);

return DateUtilities::TimeFromSt(&stTime);

}

// determine if the system time is being adjusted every tick to gradually

// bring it inline with a time server.

double HiResTimer::GetAdjustFactor()

{

DWORD dwTimeAdjustment = 0;

DWORD dwTimeIncrement = 0;

BOOL fAdjustmentDisabled = FALSE;

BOOL fSuccess = GetSystemTimeAdjustment(&dwTimeAdjustment, &dwTimeIncrement, &fAdjustmentDisabled);

if (!fSuccess || fAdjustmentDisabled)

{

return 1;

}

return ((double)dwTimeAdjustment) / ((double)dwTimeIncrement);

}

double HiResTimer::Now()

{

if(!fHiResAvailable)

{

return GetSystemTime();

}

if(!fInit)

{

if (!QueryPerformanceFrequency((LARGE\_INTEGER \*) &freq))

{

fHiResAvailable = false;

return GetSystemTime();

}

fInit = true;

}

#if DBG

uint64 f;

Assert(QueryPerformanceFrequency((LARGE\_INTEGER \*)&f) && f == freq);

#endif

// try better resolution time using perf counters

uint64 count;

if( !QueryPerformanceCounter((LARGE\_INTEGER \*) &count))

{

fHiResAvailable = false;

return GetSystemTime();

}

double time = GetSystemTime();

// there is a base time and count set.

if (!fReset

&& (count >= baseMsCount)) // Make sure we don't regress

{

double elapsed = ((double)(count - baseMsCount)) \* 1000 / freq;

// if the system time is being adjusted every tick, adjust the

// precise time delta accordingly.

if (dAdjustFactor != 1)

{

elapsed = elapsed \* dAdjustFactor;

}

double preciseTime = dBaseTime + elapsed;

if (fabs(preciseTime - time) < 25 // the time computed via perf counter is off by 25ms

&& preciseTime >= dLastTime) // the time computed via perf counter is running backwards

{

dLastTime = preciseTime;

return dLastTime;

}

}

//reset

dBaseTime = time;

dAdjustFactor = GetAdjustFactor();

baseMsCount = count;

double dSinceLast = time - dLastTime;

if (dSinceLast < -3000) // if new time is significantly behind (3s), use it:

{ // the clock may have been set backwards.

dLastTime = time;

}

else

{

dLastTime = max(dLastTime, time); // otherwise, make sure we don't regress the time.

}

fReset = false;

return dLastTime;

}

} // 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.

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

#pragma once

namespace Js

{

class HiResTimer

{

private:

double dBaseTime;

double dLastTime;

double dAdjustFactor;

uint64 baseMsCount;

uint64 freq;

bool fReset;

bool fInit;

bool fHiResAvailable;

double GetAdjustFactor();

public:

HiResTimer(): fInit(false), dBaseTime(0), baseMsCount(0), fHiResAvailable(true), dLastTime(0), dAdjustFactor(1), fReset(true) {}

double Now();

void Reset() { fReset = true; }

static double GetSystemTime();

};

} // 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.

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

//

// NOTE: This file is intended to be "#include" multiple times. The call site should define various

// macros to be executed for each entry. Macros that are not provided will be given a default,

// empty implementation.

//

//

// NOTE: After modifying this file you might observe unit test failures, especially jd/Debugger unit test failures.

// This can be caused by the build-generated bytecode files for the Intl.js and Promise.js becoming invalidated.

// After building the Chakra solution, run 'core/lib/Runtime/Library/InJavascript/GenByteCode.cmd' ( note that both x86 and x64 will be generated).

//

#if !defined(ENTRY\_INTERNAL\_SYMBOL)

#define ENTRY\_INTERNAL\_SYMBOL

#endif

#if !defined(ENTRY\_SYMBOL)

#define ENTRY\_SYMBOL(name, description)

#endif

#if !defined(ENTRY)

#define ENTRY(name)

#endif

#if !defined(ENTRY2)

#define ENTRY2(name, str)

#endif

ENTRY\_INTERNAL\_SYMBOL(\_lexicalThisSlotSymbol)

ENTRY\_INTERNAL\_SYMBOL(\_superReferenceSymbol)

ENTRY\_INTERNAL\_SYMBOL(\_superCtorReferenceSymbol)

ENTRY\_INTERNAL\_SYMBOL(\_lexicalNewTargetSymbol)

ENTRY\_SYMBOL(\_symbolHasInstance, L"Symbol.hasInstance")

ENTRY\_SYMBOL(\_symbolIsConcatSpreadable, L"Symbol.isConcatSpreadable")

ENTRY\_SYMBOL(\_symbolIterator, L"Symbol.iterator")

ENTRY\_SYMBOL(\_symbolSpecies, L"[Symbol.species]")

ENTRY\_SYMBOL(\_symbolToPrimitive, L"Symbol.toPrimitive")

ENTRY\_SYMBOL(\_symbolToStringTag, L"Symbol.toStringTag")

ENTRY\_SYMBOL(\_symbolUnscopables, L"Symbol.unscopables")

// math functions must remain contiguous for quick modification check

ENTRY(abs)

ENTRY(acos)

ENTRY(asin)

ENTRY(atan)

ENTRY(atan2)

ENTRY(ceil)

ENTRY(cos)

ENTRY(exp)

ENTRY(floor)

ENTRY(log)

ENTRY(LOG10E)

ENTRY(LOG2E)

ENTRY(min)

ENTRY(PI)

ENTRY(pow)

ENTRY(random)

ENTRY(round)

ENTRY(sin)

ENTRY(sqrt)

ENTRY(tan)

ENTRY(log10)

ENTRY(log2)

ENTRY(log1p)

ENTRY(expm1)

ENTRY(cosh)

ENTRY(sinh)

ENTRY(tanh)

ENTRY(acosh)

ENTRY(asinh)

ENTRY(atanh)

ENTRY(hypot)

ENTRY(trunc)

ENTRY(sign)

ENTRY(cbrt)

ENTRY(imul)

ENTRY(clz32)

ENTRY(fround)

// End math functions

// SIMD

ENTRY(check)

ENTRY(signMask)

ENTRY(Float32x4)

ENTRY(Float64x2)

ENTRY(Int32x4)

ENTRY(Int8x16)

ENTRY(zero)

ENTRY(splat)

ENTRY2(bool\_, L"bool") // "bool" cannot be an identifier in C++ so using "bool\_" instead

ENTRY(fromFloat32x4)

ENTRY(fromFloat32x4Bits)

ENTRY(fromInt32x4)

ENTRY(fromInt32x4Bits)

ENTRY(fromFloat64x2)

ENTRY(fromFloat64x2Bits)

// keep these contiguous

ENTRY(withFlagX)

ENTRY(withFlagY)

ENTRY(withFlagZ)

ENTRY(withFlagW)

ENTRY(flagX)

ENTRY(flagY)

ENTRY(flagZ)

ENTRY(flagW)

ENTRY(extractLane)

ENTRY(replaceLane)

//

ENTRY(load)

ENTRY(store)

ENTRY(load1)

ENTRY(store1)

ENTRY(load2)

ENTRY(store2)

ENTRY(load3)

ENTRY(store3)

ENTRY(mul)

ENTRY(div)

ENTRY(and)

ENTRY(or)

ENTRY(xor)

ENTRY(neg)

ENTRY(not)

ENTRY(swizzle)

ENTRY(shuffle)

ENTRY(clamp)

ENTRY(select)

ENTRY(reciprocal)

ENTRY(reciprocalSqrt)

ENTRY(scale)

ENTRY(lessThan)

ENTRY(lessThanOrEqual)

ENTRY(equal)

ENTRY(notEqual)

ENTRY(greaterThanOrEqual)

ENTRY(greaterThan)

ENTRY(shiftLeft)

ENTRY(shiftRightLogical)

ENTRY(shiftRightArithmetic)

ENTRY(shiftLeftByScalar)

ENTRY(shiftRightLogicalByScalar)

ENTRY(shiftRightArithmeticByScalar)

ENTRY(add)

ENTRY(all)

ENTRY(anchor)

ENTRY(apply)

ENTRY(Array)

ENTRY(assign)

ENTRY(atEnd)

ENTRY(big)

ENTRY(bind)

ENTRY(blink)

ENTRY(bold)

ENTRY(Boolean)

ENTRY(call)

ENTRY(CanvasPixelArray)

ENTRY(cast)

ENTRY2(catch\_, L"catch") // "catch" cannot be an identifier in C++ so using "catch\_" instead

ENTRY(charAt)

ENTRY(charCodeAt)

ENTRY(codePointAt)

ENTRY(clear)

ENTRY(copyWithin)

ENTRY(CollectGarbage)

ENTRY(concat)

ENTRY(construct)

ENTRY(constructor)

ENTRY(ConversionError)

ENTRY(create)

ENTRY(Date)

ENTRY(decodeURI)

ENTRY(decodeURIComponent)

ENTRY2(delete\_, L"delete") // "delete" cannot be an identifier in C++ so using "delete\_" instead

ENTRY(deleteProperty)

ENTRY(description)

ENTRY(done)

ENTRY(E)

ENTRY(encodeURI)

ENTRY(encodeURIComponent)

ENTRY(endsWith)

ENTRY(entries)

ENTRY(enumerate)

ENTRY(Enumerator)

ENTRY(EPSILON)

ENTRY(escape)

ENTRY(Error)

ENTRY(eval)

ENTRY(EvalError)

ENTRY(every)

ENTRY(exec)

ENTRY(flags)

ENTRY(fill)

ENTRY(filter)

ENTRY(find)

ENTRY(findIndex)

ENTRY(fixed)

ENTRY(fontcolor)

ENTRY(fontsize)

ENTRY2(for\_, L"for") // "for" cannot be an identifier in C++ so using "for\_" instead

ENTRY(forEach)

ENTRY(freeze)

ENTRY(from)

ENTRY(fromCharCode)

ENTRY(fromCodePoint)

ENTRY(Function)

ENTRY(getDate)

ENTRY(getDay)

ENTRY(getFullYear)

ENTRY(getHours)

ENTRY(getMilliseconds)

ENTRY(getMinutes)

ENTRY(getMonth)

ENTRY(GetObject)

ENTRY(getOwnPropertyDescriptor)

ENTRY(getOwnPropertyNames)

ENTRY(getOwnPropertySymbols)

ENTRY(getSeconds)

ENTRY(getTime)

ENTRY(getTimezoneOffset)

ENTRY(getUTCDate)

ENTRY(getUTCDay)

ENTRY(getUTCFullYear)

ENTRY(getUTCHours)

ENTRY(getUTCMilliseconds)

ENTRY(getUTCMinutes)

ENTRY(getUTCMonth)

ENTRY(getUTCSeconds)

ENTRY(getVarDate)

ENTRY(getYear)

ENTRY(has)

ENTRY(hasInstance)

ENTRY(hasOwnProperty)

ENTRY(includes)

ENTRY(indexOf)

ENTRY(Infinity)

ENTRY(Intl)

ENTRY(is)

ENTRY(isArray)

ENTRY(isConcatSpreadable)

ENTRY(isExtensible)

ENTRY(isFinite)

ENTRY(isFrozen)

ENTRY(isInteger)

ENTRY(isNaN)

ENTRY(isPrototypeOf)

ENTRY(isSafeInteger)

ENTRY(isSealed)

ENTRY(isView)

ENTRY(italics)

ENTRY(item)

ENTRY(iterator)

ENTRY(join)

ENTRY(JSON)

ENTRY(keyFor)

ENTRY(keys)

ENTRY(lastIndexOf)

ENTRY(length)

ENTRY(link)

ENTRY(LN10)

ENTRY(LN2)

ENTRY(localeCompare)

ENTRY(map)

ENTRY(Map)

ENTRY(Math)

// SIMD\_JS

ENTRY(SIMD)

ENTRY(max)

ENTRY(MAX\_SAFE\_INTEGER)

ENTRY(MAX\_VALUE)

ENTRY(match)

ENTRY(message)

ENTRY(MIN\_SAFE\_INTEGER)

ENTRY(MIN\_VALUE)

ENTRY(moveFirst)

ENTRY(moveNext)

ENTRY(NaN)

ENTRY(name)

ENTRY(next)

ENTRY(normalize)

ENTRY(now)

ENTRY(Number)

ENTRY(number)

ENTRY(NEGATIVE\_INFINITY)

ENTRY(ownKeys)

ENTRY(Object)

ENTRY(of)

ENTRY(parse)

ENTRY(parseFloat)

ENTRY(parseInt)

ENTRY(pop)

ENTRY(POSITIVE\_INFINITY)

ENTRY(preventExtensions)

ENTRY(propertyIsEnumerable)

ENTRY(prototype)

ENTRY(proxy)

ENTRY(Proxy)

ENTRY(push)

ENTRY(race)

ENTRY(RangeError)

ENTRY(raw)

ENTRY(reduce)

ENTRY(reduceRight)

ENTRY(ReferenceError)

ENTRY(Reflect)

ENTRY(RegExpError)

ENTRY(reject)

ENTRY(replace)

ENTRY(resolve)

ENTRY2(return\_, L"return") // "return" cannot be an identifier in C++ so using "return\_" instead

ENTRY(reverse)

ENTRY(revocable)

ENTRY(revoke)

ENTRY(RegExp)

ENTRY(repeat)

ENTRY(seal)

ENTRY(search)

ENTRY(Set)

ENTRY(setDate)

ENTRY(setFullYear)

ENTRY(setHours)

ENTRY(setMilliseconds)

ENTRY(setMinutes)

ENTRY(setMonth)

ENTRY(setSeconds)

ENTRY(setTime)

ENTRY(setUTCDate)

ENTRY(setUTCFullYear)

ENTRY(setUTCHours)

ENTRY(setUTCMilliseconds)

ENTRY(setUTCMinutes)

ENTRY(setUTCMonth)

ENTRY(setUTCSeconds)

ENTRY(setYear)

ENTRY(shift)

ENTRY(size)

ENTRY(slice)

ENTRY(some)

ENTRY(sort)

ENTRY(source)

ENTRY(species)

ENTRY(splice)

ENTRY(split)

ENTRY(SQRT1\_2)

ENTRY(SQRT2)

ENTRY(strike)

ENTRY(String)

ENTRY(sub)

ENTRY(substring)

ENTRY(substr)

ENTRY(sup)

ENTRY(stack)

ENTRY(stackTraceLimit)

ENTRY(startsWith)

ENTRY(stringify)

ENTRY(Symbol)

ENTRY(SyntaxError)

ENTRY(test)

ENTRY(then)

ENTRY2(throw\_, L"throw") // "throw" cannot be an identifier in C++ so using "throw\_" instead

ENTRY(toDateString)

ENTRY(toExponential)

ENTRY(toFixed)

ENTRY(toISOString)

ENTRY(toJSON)

ENTRY(toLocaleDateString)

ENTRY(toLocaleLowerCase)

ENTRY(toLocaleString)

ENTRY(toLocaleTimeString)

ENTRY(toLocaleUpperCase)

ENTRY(toLowerCase)

ENTRY(toPrecision)

ENTRY(toPrimitive)

ENTRY(toString)

ENTRY(toStringTag)

ENTRY(toTimeString)

ENTRY(toUpperCase)

ENTRY(toUTCString)

ENTRY(transfer)

ENTRY(trim)

ENTRY(trimLeft)

ENTRY(trimRight)

ENTRY(TypeError)

ENTRY(undefined)

ENTRY(unescape)

ENTRY(unscopables)

ENTRY(unshift)

ENTRY(URIError)

ENTRY(UTC)

ENTRY(valueOf)

ENTRY(values)

ENTRY(WeakMap)

ENTRY(WeakSet)

ENTRY(WinRTError)

ENTRY(write)

ENTRY(writeln)

ENTRY(setNonUserCodeExceptions)

ENTRY(debuggerEnabled)

ENTRY(enumerable)

ENTRY(configurable)

ENTRY(writable)

ENTRY(value)

ENTRY(get)

ENTRY(getPrototypeOf)

ENTRY(setPrototypeOf)

ENTRY(set)

ENTRY(defineProperty)

ENTRY(defineProperties)

ENTRY(toGMTString)

ENTRY(compile)

ENTRY(global)

ENTRY(lastIndex)

ENTRY(multiline)

ENTRY(ignoreCase)

ENTRY(unicode)

ENTRY(sticky)

ENTRY(byteOffset)

ENTRY(byteLength)

ENTRY(buffer)

ENTRY(BYTES\_PER\_ELEMENT)

ENTRY(ArrayBuffer)

ENTRY(Int8Array)

ENTRY(Int16Array)

ENTRY(Int32Array)

ENTRY(Uint8Array)

ENTRY(Uint8ClampedArray)

ENTRY(Uint16Array)

ENTRY(Uint32Array)

ENTRY(Float32Array)

ENTRY(Float64Array)

ENTRY(CharArray)

ENTRY(Int64Array)

ENTRY(Uint64Array)

ENTRY(BooleanArray)

ENTRY(DataView)

ENTRY(setInt8)

ENTRY(setUint8)

ENTRY(setInt16)

ENTRY(setUint16)

ENTRY(setInt32)

ENTRY(setUint32)

ENTRY(setFloat32)

ENTRY(setFloat64)

ENTRY(getInt8)

ENTRY(getUint8)

ENTRY(getInt16)

ENTRY(getUint16)

ENTRY(getInt32)

ENTRY(getUint32)

ENTRY(getFloat32)

ENTRY(getFloat64)

ENTRY(subarray)

ENTRY(Debug)

ENTRY(caller)

ENTRY(callee)

ENTRY(arguments)

ENTRY($\_)

ENTRY($1)

ENTRY($2)

ENTRY($3)

ENTRY($4)

ENTRY($5)

ENTRY($6)

ENTRY($7)

ENTRY($8)

ENTRY($9)

ENTRY(index)

ENTRY(input)

ENTRY(lastMatch)

ENTRY(lastParen)

ENTRY(leftContext)

ENTRY(rightContext)

ENTRY(options)

ENTRY2(\_\_instance, L"$\_\_instance")

ENTRY2(Small, L"small")

ENTRY2($Ampersand, L"$&")

ENTRY2($Plus, L"$+")

ENTRY2($BackTick, L"$`")

ENTRY2($Tick, L"$\'")

ENTRY(\_\_defineGetter\_\_)

ENTRY(\_\_defineSetter\_\_)

ENTRY(\_\_lookupGetter\_\_)

ENTRY(\_\_lookupSetter\_\_)

ENTRY(\_\_proto\_\_)

//Intl

ENTRY(EngineInterface)

ENTRY(raiseAssert)

ENTRY(\_\_minimumFractionDigits)

ENTRY(\_\_maximumFractionDigits)

ENTRY(\_\_minimumIntegerDigits)

ENTRY(\_\_minimumSignificantDigits)

ENTRY(\_\_maximumSignificantDigits)

ENTRY(minimumSignificantDigits)

ENTRY(maximumSignificantDigits)

ENTRY(\_\_isDecimalPointAlwaysDisplayed)

ENTRY(\_\_useGrouping)

ENTRY(\_\_numberingSystem)

ENTRY(\_\_formatterToUse)

ENTRY(\_\_currency)

ENTRY(\_\_currencyDisplay)

ENTRY(\_\_currencyDisplayToUse)

ENTRY(\_\_locale)

ENTRY(\_\_templateString)

ENTRY(Collator)

ENTRY(NumberFormat)

ENTRY(DateTimeFormat)

ENTRY(format)

ENTRY(resolvedOptions)

ENTRY(compare)

ENTRY(locale)

ENTRY(sensitivity)

ENTRY(numeric)

ENTRY(ignorePunctuation)

ENTRY(\_\_windowsClock)

ENTRY(\_\_windowsCalendar)

ENTRY(\_\_patternStrings)

ENTRY(isWellFormedLanguageTag)

ENTRY(normalizeLanguageTag)

ENTRY(compareString)

ENTRY(resolveLocaleLookup)

ENTRY(resolveLocaleBestFit)

ENTRY(getDefaultLocale)

ENTRY(getExtensions)

ENTRY(formatNumber)

ENTRY(cacheNumberFormat)

ENTRY(createDateTimeFormat)

ENTRY(currencyDigits)

ENTRY(formatDateTime)

ENTRY(registerBuiltInFunction)

ENTRY(getHiddenObject)

ENTRY(setHiddenObject)

ENTRY(builtInSetPrototype)

ENTRY(builtInGetArrayLength)

ENTRY(builtInRegexMatch)

ENTRY(builtInCallInstanceFunction)

ENTRY(subTags)

ENTRY(builtInMathAbs)

ENTRY(builtInMathFloor)

ENTRY(builtInMathMax)

ENTRY(builtInMathPow)

ENTRY(builtInJavascriptObjectEntryDefineProperty)

ENTRY(builtInJavascriptObjectEntryGetPrototypeOf)

ENTRY(builtInJavascriptObjectEntryIsExtensible)

ENTRY(builtInJavascriptObjectEntryGetOwnPropertyNames)

ENTRY(builtInJavascriptObjectEntryHasOwnProperty)

ENTRY(builtInJavascriptObjectCreate)

ENTRY(builtInJavascriptObjectPreventExtensions)

ENTRY(builtInJavascriptObjectGetOwnPropertyDescriptor)

ENTRY(builtInJavascriptArrayEntryForEach)

ENTRY(builtInJavascriptArrayEntryIndexOf)

ENTRY(builtInJavascriptArrayEntryPush)

ENTRY(builtInJavascriptArrayEntryJoin)

ENTRY(builtInJavascriptFunctionEntryBind)

ENTRY(builtInJavascriptDateEntryGetDate)

ENTRY(builtInJavascriptDateEntryNow)

ENTRY(builtInJavascriptRegExpEntryTest)

ENTRY(builtInJavascriptStringEntryMatch)

ENTRY(builtInJavascriptStringEntryReplace)

ENTRY(builtInJavascriptStringEntryToLowerCase)

ENTRY(builtInJavascriptStringEntryToUpperCase)

ENTRY(builtInGlobalObjectEntryIsFinite)

ENTRY(builtInGlobalObjectEntryIsNaN)

ENTRY(builtInGlobalObjectEval)

ENTRY(raiseNeedObject)

ENTRY(raiseNeedObjectOrString)

ENTRY(raiseThis\_NullOrUndefined)

ENTRY(raiseNotAConstructor)

ENTRY(raiseInvalidDate)

ENTRY(ToLogicalBoolean)

ENTRY(ToString)

ENTRY(ToNumber)

ENTRY(raiseOptionValueOutOfRange\_3)

ENTRY(raiseOptionValueOutOfRange)

ENTRY(ToUint32)

ENTRY(raiseLocaleNotWellFormed)

ENTRY(isValid)

ENTRY(localeMatcher)

ENTRY(validateAndCanonicalizeTimeZone)

ENTRY(getDefaultTimeZone)

ENTRY(raiseObjectIsAlreadyInitialized)

ENTRY(raiseObjectIsNonExtensible)

ENTRY(raiseNeedObjectOfType)

ENTRY(raiseInvalidCurrencyCode)

ENTRY(raiseMissingCurrencyCode)

ENTRY(\_\_initializedIntlObject)

ENTRY(\_\_initializedNumberFormat)

ENTRY(\_\_initializedDateTimeFormat)

ENTRY(localeWithoutSubtags)

ENTRY(\_\_matcher)

ENTRY(\_\_localeForCompare)

ENTRY(\_\_usage)

ENTRY(\_\_sensitivity)

ENTRY(\_\_ignorePunctuation)

ENTRY(\_\_caseFirst)

ENTRY(\_\_numeric)

ENTRY(\_\_collation)

ENTRY(\_\_initializedCollator)

ENTRY(\_\_boundCompare)

ENTRY(\_\_boundFormat)

ENTRY(supportedLocalesOf)

ENTRY(\_\_style)

ENTRY(weekday)

ENTRY(year)

ENTRY(month)

ENTRY(day)

ENTRY(minute)

ENTRY(hour)

ENTRY(second)

ENTRY(era)

ENTRY(timeZone)

ENTRY(timeZoneName)

ENTRY(hour12)

ENTRY(\_\_weekday)

ENTRY(\_\_year)

ENTRY(\_\_month)

ENTRY(\_\_day)

ENTRY(\_\_minute)

ENTRY(\_\_hour)

ENTRY(\_\_second)

ENTRY(\_\_era)

ENTRY(\_\_timeZone)

ENTRY(\_\_timeZoneName)

ENTRY(\_\_hour12)

ENTRY(\_\_formatMatcher)

ENTRY(\_\_calendar)

ENTRY(calendar)

ENTRY(\_\_relevantExtensionKeys)

ENTRY(numberingSystem)

ENTRY(MS\_ASYNC\_OP\_STATUS\_SUCCESS)

ENTRY(MS\_ASYNC\_OP\_STATUS\_CANCELED)

ENTRY(MS\_ASYNC\_OP\_STATUS\_ERROR)

ENTRY(MS\_ASYNC\_CALLBACK\_STATUS\_ASSIGN\_DELEGATE)

ENTRY(MS\_ASYNC\_CALLBACK\_STATUS\_JOIN)

ENTRY(MS\_ASYNC\_CALLBACK\_STATUS\_CHOOSEANY)

ENTRY(MS\_ASYNC\_CALLBACK\_STATUS\_CANCEL)

ENTRY(MS\_ASYNC\_CALLBACK\_STATUS\_ERROR)

ENTRY(msTraceAsyncOperationStarting)

ENTRY(msTraceAsyncCallbackStarting)

ENTRY(msTraceAsyncCallbackCompleted)

ENTRY(msUpdateAsyncCallbackRelation)

ENTRY(msTraceAsyncOperationCompleted)

ENTRY(getErrorMessage)

ENTRY(logDebugMessage)

ENTRY(tagPublicLibraryCode)

ENTRY(enqueueTask)

ENTRY(Common)

ENTRY(Promise)

ENTRY(MissingProperty)

// These are ES5/6/7+ builtins that are tracked for telemetry purposes, but currently not implemented by Chakra or are otherwise special.

// IMPORTANT NOTE: See notes at top of this file regarding GenByteCode and test failures after changing this list.

ENTRY(observe)

ENTRY(unobserve)

ENTRY(setUInt16)

ENTRY(at)

ENTRY(contains)

ENTRY(rightPad)

ENTRY(rPad)

ENTRY(padRight)

ENTRY(leftPad)

ENTRY(lPad)

ENTRY(padLeft)

ENTRY(getOwnPropertyDescriptors)

ENTRY(\_\_constructor) // represents an invocation of the constructor function rather than a use of the constructor property (i.e. `new Foo()` rather than `(new Foo()).constructor`).

// SymbolFunctionNameId: for RuntimeFunction nameId

ENTRY2(\_RuntimeFunctionNameId\_hasInstance, L"[Symbol.hasInstance]")

ENTRY2(\_RuntimeFunctionNameId\_toStringTag, L"[Symbol.toStringTag]")

ENTRY2(\_RuntimeFunctionNameId\_toPrimitive, L"[Symbol.toPrimitive]")

ENTRY2(\_RuntimeFunctionNameId\_iterator, L"[Symbol.iterator]")

ENTRY2(\_RuntimeFunctionNameId\_species, L"[Symbol.species]")

//Previously we would add these right after standard built-ins were generated; now adding them here.

ENTRY(document)

ENTRY2(\_event, L"event")

ENTRY(history)

ENTRY(location)

ENTRY(navigator)

ENTRY(screen)

// Note: Do not add fields for conditionally-compiled PropertyIds into this file.

// The bytecode for internal javascript libraries is built on chk but re-used in fre builds.

// Having a mismatch in the number of PropertyIds will cause a failure loading bytecode.

// See BLUE #467459

#undef ENTRY\_INTERNAL\_SYMBOL

#undef ENTRY\_SYMBOL

#undef ENTRY

#undef ENTRY2

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

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

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

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

#include "RuntimeBasePch.h"

namespace Js

{

EnterScriptObject::EnterScriptObject(ScriptContext\* scriptContext, ScriptEntryExitRecord\* entryExitRecord,

void \* returnAddress, bool doCleanup, bool isCallRoot, bool hasCaller)

{

Assert(scriptContext);

#ifdef PROFILE\_EXEC

scriptContext->ProfileBegin(Js::RunPhase);

#endif

if (scriptContext->GetThreadContext() &&

scriptContext->GetThreadContext()->IsNoScriptScope())

{

FromDOM\_NoScriptScope\_fatal\_error();

}

// Keep a copy locally so the optimizer can just copy prop it to the dtor

this->scriptContext = scriptContext;

this->entryExitRecord = entryExitRecord;

this->doCleanup = doCleanup;

this->isCallRoot = isCallRoot;

this->hr = NOERROR;

this->hasForcedEnter = scriptContext->GetDebugContext() != nullptr ? scriptContext->GetDebugContext()->GetProbeContainer()->isForcedToEnterScriptStart : false;

// Initialize the entry exit record

entryExitRecord->returnAddrOfScriptEntryFunction = returnAddress;

entryExitRecord->hasCaller = hasCaller;

entryExitRecord->scriptContext = scriptContext;

#ifdef EXCEPTION\_CHECK

entryExitRecord->handledExceptionType = ExceptionCheck::ClearHandledExceptionType();

#endif

#if DBG\_DUMP

entryExitRecord->isCallRoot = isCallRoot;

#endif

if (!scriptContext->IsClosed())

{

library = scriptContext->GetLibrary();

}

try

{

AUTO\_NESTED\_HANDLED\_EXCEPTION\_TYPE(ExceptionType\_OutOfMemory);

scriptContext->GetThreadContext()->PushHostScriptContext(scriptContext->GetHostScriptContext());

}

catch (Js::OutOfMemoryException)

{

this->hr = E\_OUTOFMEMORY;

}

BEGIN\_NO\_EXCEPTION

{

// We can not have any exception in the constructor, otherwise the destructor will

// not run and we might be in an inconsistent state

// Put any code that may raise an exception in OnScriptStart

scriptContext->GetThreadContext()->EnterScriptStart(entryExitRecord, doCleanup);

}

END\_NO\_EXCEPTION

}

void EnterScriptObject::VerifyEnterScript()

{

if (FAILED(hr))

{

Assert(hr == E\_OUTOFMEMORY);

throw Js::OutOfMemoryException();

}

}

EnterScriptObject::~EnterScriptObject()

{

scriptContext->OnScriptEnd(isCallRoot, hasForcedEnter);

if (SUCCEEDED(hr))

{

scriptContext->GetThreadContext()->PopHostScriptContext();

}

scriptContext->GetThreadContext()->EnterScriptEnd(entryExitRecord, doCleanup);

#ifdef PROFILE\_EXEC

scriptContext->ProfileEnd(Js::RunPhase);

#endif

}

};

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

// 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

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

// EnterScriptStart/EnterScriptEnd

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

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

#define SAVE\_FS0() \_\_entryExitRecord.scriptEntryFS0 = (void\*)\_\_readfsdword(0)

#else

#define SAVE\_FS0()

#endif

#define BEGIN\_ENTER\_SCRIPT\_EX(scriptContext, doCleanup, isCallRoot, hasCaller, isScript) \

{ \

Js::ScriptContext\* \_\_localScriptContext = scriptContext; \

Js::ScriptEntryExitRecord \_\_entryExitRecord = {0}; \

SAVE\_FS0(); \

Js::EnterScriptObject \_\_enterScriptObject = Js::EnterScriptObject(\_\_localScriptContext, &\_\_entryExitRecord, \

\_ReturnAddress(), doCleanup, isCallRoot, hasCaller); \

\_\_localScriptContext->OnScriptStart(isCallRoot, isScript); \

\_\_enterScriptObject.VerifyEnterScript();

#define BEGIN\_ENTER\_SCRIPT(scriptContext, doCleanup, isCallRoot, hasCaller) \

BEGIN\_ENTER\_SCRIPT\_EX(scriptContext, doCleanup, isCallRoot, hasCaller, /\*isScript\*/true) \

#define END\_ENTER\_SCRIPT \

}

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

// EnterScriptStart/EnterScriptEnd with javascript exception handling

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

#define BEGIN\_JS\_RUNTIME\_CALL\_EX(scriptContext, doCleanup) \

BEGIN\_ENTER\_SCRIPT(scriptContext, doCleanup, /\*isCallRoot\*/ false, /\*hasCaller\*/false) \

{

#define BEGIN\_JS\_RUNTIME\_CALLROOT\_EX(scriptContext, hasCaller) \

BEGIN\_ENTER\_SCRIPT(scriptContext, /\*doCleanup\*/ true, /\*isCallRoot\*/ true, hasCaller) \

{

#define BEGIN\_JS\_RUNTIME\_CALL(scriptContext) \

BEGIN\_JS\_RUNTIME\_CALL\_EX(scriptContext, /\*doCleanup\*/ true)

// Use \_NOT\_SCRIPT to indicate we are not really starting script, avoid certain risky/lengthy work.

#define BEGIN\_JS\_RUNTIME\_CALL\_NOT\_SCRIPT(scriptContext) \

BEGIN\_ENTER\_SCRIPT\_EX(scriptContext, /\*doCleanup\*/false, /\*isCallRoot\*/false, /\*hasCaller\*/false, /\*isScript\*/false) \

{

#define END\_JS\_RUNTIME\_CALL(scriptContext) \

} \

END\_ENTER\_SCRIPT

#define BEGIN\_JS\_RUNTIME\_CALL\_EX\_AND\_TRANSLATE\_EXCEPTION\_AND\_ERROROBJECT\_TO\_HRESULT(scriptContext, doCleanup) \

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

BEGIN\_ENTER\_SCRIPT(scriptContext, doCleanup, /\*isCallRoot\*/ false, /\*hasCaller\*/false) \

{ \

IGNORE\_STACKWALK\_EXCEPTION(scriptContext); \

#define BEGIN\_JS\_RUNTIME\_CALL\_EX\_AND\_TRANSLATE\_AND\_GET\_EXCEPTION\_AND\_ERROROBJECT\_TO\_HRESULT(scriptContext, doCleanup) \

BEGIN\_JS\_RUNTIME\_CALL\_EX\_AND\_TRANSLATE\_EXCEPTION\_AND\_ERROROBJECT\_TO\_HRESULT(scriptContext, doCleanup) \

#define BEGIN\_JS\_RUNTIME\_CALL\_EX\_AND\_TRANSLATE\_EXCEPTION\_AND\_ERROROBJECT\_TO\_HRESULT\_NESTED(scriptContext, doCleanup) \

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

BEGIN\_ENTER\_SCRIPT(scriptContext, doCleanup, /\*isCallRoot\*/ false, /\*hasCaller\*/false) \

{ \

#define END\_JS\_RUNTIME\_CALL\_AND\_TRANSLATE\_EXCEPTION\_AND\_ERROROBJECT\_TO\_HRESULT(hr) \

} \

END\_ENTER\_SCRIPT \

END\_TRANSLATE\_KNOWN\_EXCEPTION\_TO\_HRESULT(hr) \

END\_TRANSLATE\_ERROROBJECT\_TO\_HRESULT(hr) \

CATCH\_UNHANDLED\_EXCEPTION(hr)

#define END\_JS\_RUNTIME\_CALL\_AND\_TRANSLATE\_AND\_GET\_EXCEPTION\_AND\_ERROROBJECT\_TO\_HRESULT(hr, scriptContext, exceptionObject) \

} \

END\_ENTER\_SCRIPT \

END\_TRANSLATE\_KNOWN\_EXCEPTION\_TO\_HRESULT(hr) \

END\_GET\_ERROROBJECT(hr, scriptContext, exceptionObject) \

CATCH\_UNHANDLED\_EXCEPTION(hr)

#define BEGIN\_JS\_RUNTIME\_CALL\_EX\_AND\_TRANSLATE\_OOM\_TO\_HRESULT(scriptContext, doCleanup, hasCaller) \

BEGIN\_TRANSLATE\_OOM\_TO\_HRESULT \

BEGIN\_ENTER\_SCRIPT(scriptContext, doCleanup, /\*isCallRoot\*/ false, hasCaller) \

{

#define END\_JS\_RUNTIME\_CALL\_AND\_TRANSLATE\_OOM\_TO\_HRESULT(hr) \

} \

END\_ENTER\_SCRIPT \

END\_TRANSLATE\_OOM\_TO\_HRESULT(hr)

#define END\_TRANSLATE\_SO\_OOM\_JSEXCEPTION(hr) \

} \

catch (Js::JavascriptExceptionObject \*) \

{ \

} \

catch (Js::OutOfMemoryException) \

{ \

} \

catch (Js::StackOverflowException) \

{ \

} \

catch (...) \

{ \

AssertMsg(false, "this exception didn't get handled"); \

hr = E\_FAIL; \

} \

}

#ifdef CHECK\_STACKWALK\_EXCEPTION

#define IGNORE\_STACKWALK\_EXCEPTION(scriptContext) \

scriptContext->SetIgnoreStackWalkException();

#else

#define IGNORE\_STACKWALK\_EXCEPTION(scriptContext)

#endif

// For debugging scenarios where execution will go to debugging manager and come back to engine again, enforce the current EER to have

// 'hasCaller' property set, which will enable the stack walking across frames.

#define ENFORCE\_ENTRYEXITRECORD\_HASCALLER(scriptContext) \

scriptContext->EnforceEERHasCaller();

#define BEGIN\_JS\_RUNTIME\_CALL\_EX\_AND\_TRANSLATE\_EXCEPTION\_AND\_ERROROBJECT\_TO\_HRESULT2(scriptContext, doCleanup, hasCaller) \

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

BEGIN\_ENTER\_SCRIPT(scriptContext, doCleanup, /\*isCallRoot\*/ false, /\*hasCaller\*/hasCaller) \

{ \

// Same as above but allows custom handling of exception object.

#define END\_JS\_RUNTIME\_CALL\_AND\_TRANSLATE\_EXCEPTION\_AND\_ERROROBJECT\_TO\_HRESULT2(hr, exceptionObject) \

} \

END\_ENTER\_SCRIPT \

END\_TRANSLATE\_KNOWN\_EXCEPTION\_TO\_HRESULT(hr) \

catch(Js::JavascriptExceptionObject \* exceptionObject)

namespace Js

{

class EnterScriptObject

{

private:

ScriptEntryExitRecord\* entryExitRecord;

bool doCleanup;

bool isCallRoot;

bool hasForcedEnter; // due to debugging.

ScriptContext\* scriptContext;

HRESULT hr; // we need to throw outside of constructor

JavascriptLibrary\* library; // stack pin the library.

public:

EnterScriptObject(ScriptContext\* scriptContext, ScriptEntryExitRecord\* entryExitRecord,

void \* returnAddress, bool doCleanup, bool isCallRoot, bool hasCaller);

void VerifyEnterScript();

~EnterScriptObject();

};

template<bool stackProbe, bool leaveForHost, bool isFPUControlRestoreNeeded>

class LeaveScriptObject

{

private:

ScriptContext \*const scriptContext;

void \*const frameAddress;

bool leftScript;

SmartFPUControlT<isFPUControlRestoreNeeded> savedFPUControl;

DECLARE\_EXCEPTION\_CHECK\_DATA;

public:

LeaveScriptObject(ScriptContext \*const scriptContext, void \*const frameAddress)

: scriptContext(scriptContext),

frameAddress(frameAddress),

savedFPUControl()

{

leftScript = scriptContext->LeaveScriptStart<stackProbe, leaveForHost>(frameAddress);

// We should be in script when we leave

Assert(leftScript);

SAVE\_EXCEPTION\_CHECK;

}

~LeaveScriptObject()

{

// We should be in script when we leave

Assert(leftScript);

RESTORE\_EXCEPTION\_CHECK;

if(leftScript)

{

scriptContext->LeaveScriptEnd<leaveForHost>(frameAddress);

}

}

};

}

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

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

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

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

#include "RuntimeBasePch.h"

const PerfHintItem s\_perfHintContainer[] =

{

#define PERFHINT\_REASON(name, isNotOptimized, level, desc, consequences, suggestion) {desc, consequences, suggestion, level, isNotOptimized},

#include "PerfHintDescriptions.h"

#undef PERFHINT\_REASON

};

void WritePerfHint(PerfHints hint, Js::FunctionBody \* functionBody, uint byteCodeOffset /\*= Js::Constants::NoByteCodeOffset\*/)

{

Assert(functionBody);

Assert(((uint)hint) < \_countof(s\_perfHintContainer));

PerfHintItem item = s\_perfHintContainer[(uint)hint];

int level = CONFIG\_FLAG(PerfHintLevel);

Assert(level <= (int)PerfHintLevels::VERBOSE);

if ((int)item.level <= level)

{

ULONG lineNumber = functionBody->GetLineNumber();

LONG columnNumber = functionBody->GetColumnNumber();

if (byteCodeOffset != Js::Constants::NoByteCodeOffset)

{

functionBody->GetLineCharOffset(byteCodeOffset, &lineNumber, &columnNumber, false /\*canAllocateLineCache\*/);

// returned values are 0-based. Adjusting.

lineNumber++;

columnNumber++;

}

// We will print the short name.

TCHAR shortName[255];

Js::FunctionBody::GetShortNameFromUrl(functionBody->GetSourceName(), shortName, 255);

OUTPUT\_TRACE(Js::PerfHintPhase, L"%s : %s {\n Function : %s [%s @ %u, %u]\n Consequences : %s\n Suggestion : %s\n}\n",

item.isNotOptimized ? L"Not optimized" : L"Optimized",

item.description,

functionBody->GetExternalDisplayName(),

shortName,

lineNumber,

columnNumber,

item.consequences,

item.suggestion);

Output::Flush();

}

}

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

// 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

enum class PerfHintLevels

{

L1 = 1, // Critical

L2, // Noisy but should give you all

VERBOSE = L2,

};

enum PerfHints : uint

{

#define PERFHINT\_REASON(name, isNotOptimized, level, desc, consequences, suggestion) name,

#include "PerfHintDescriptions.h"

#undef PERFHINT\_REASON

};

struct PerfHintItem

{

LPCWSTR description;

LPCWSTR consequences;

LPCWSTR suggestion;

PerfHintLevels level;

bool isNotOptimized;

};

extern const PerfHintItem s\_perfHintContainer[];

void WritePerfHint(PerfHints hint, Js::FunctionBody \* functionBody, uint byteCodeOffset = Js::Constants::NoByteCodeOffset);

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

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

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

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

// (hintCode, isNotOptimized, hintlevel, description, consequences, suggestion)

PERFHINT\_REASON(HasTryBlock, true, PerfHintLevels::L1, L"Function has try block", L"Un-optimized JIT code generated for this function", L"Move perf sensitive block inside of try to different function")

PERFHINT\_REASON(HasTryBlock\_Verbose, true, PerfHintLevels::VERBOSE, L"Function has try block", L"Un-optimized JIT code generated for this function", L"Move perf sensitive block inside of try to different function")

PERFHINT\_REASON(CallsEval, true, PerfHintLevels::L1, L"Function calls eval statement", L"Extra scopes, affect inlining, high overhead in the JIT code", L"Check usage of eval statement")

PERFHINT\_REASON(CallsEval\_Verbose, true, PerfHintLevels::VERBOSE, L"Function calls eval statement", L"Extra scopes, affect inlining, high overhead in the JIT code", L"Check usage of eval statement")

PERFHINT\_REASON(ChildCallsEval, true, PerfHintLevels::VERBOSE, L"Function's child calls eval statement", L"Extra scopes, affect inlining, high overhead in the JIT code", L"Check usage of eval statement")

PERFHINT\_REASON(HasWithBlock, true, PerfHintLevels::L1, L"Function has with statement", L"Slower lookups, high overhead in the JIT code", L"Avoid using with statement")

PERFHINT\_REASON(HeapArgumentsDueToFormals, true, PerfHintLevels::L1, L"Arguments object not optimized due to formals", L"Slower lookups, affects inlining, high overhead in the JIT code", L"Check the usage of formals in the function")

PERFHINT\_REASON(HeapArgumentsModification, true, PerfHintLevels::L1, L"Modification to arguments", L"Slower lookups, high overhead in the JIT code", L"Avoid modification to the arguments")

PERFHINT\_REASON(HeapArgumentsCreated, true, PerfHintLevels::L1, L"Arguments object not optimized", L"Slower lookups, high overhead in the JIT code", L"Check the usage of arguments in the function")

PERFHINT\_REASON(PolymorphicInilineCap, true, PerfHintLevels::L1, L"Function has reached polymorphic-inline cap", L"This function will not inline more than 4 functions for this call-site.", L"Check the polymorphic usage of this function")

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

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

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

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

#include "RuntimeBasePch.h"

namespace Js

{

DEFINE\_RECYCLER\_TRACKER\_PERF\_COUNTER(PropertyRecord);

DEFINE\_RECYCLER\_TRACKER\_WEAKREF\_PERF\_COUNTER(PropertyRecord);

// Constructor for runtime-constructed PropertyRecords

PropertyRecord::PropertyRecord(DWORD byteCount, bool isNumeric, uint hash, bool isSymbol)

: pid(Js::Constants::NoProperty), hash(hash), isNumeric(isNumeric), byteCount(byteCount), isBound(false), isSymbol(isSymbol)

{

}

// Constructor for built-in PropertyRecords

PropertyRecord::PropertyRecord(PropertyId pid, uint hash, bool isNumeric, DWORD byteCount, bool isSymbol)

: pid(pid), hash(hash), isNumeric(isNumeric), byteCount(byteCount), isBound(true), isSymbol(isSymbol)

{

}

void PropertyRecord::Finalize(bool isShutdown)

{

if (!isShutdown)

{

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

Assert(tc);

Assert(tc->IsActivePropertyId(this->GetPropertyId()));

tc->InvalidatePropertyRecord(this);

}

}

#ifdef DEBUG

// This is only used to assert that integer property names are not passed into

// the GetSetter, GetProperty, SetProperty etc methods that take JavascriptString

// instead of PropertyId. It is expected that integer property names will go

// through a fast path before reaching those APIs.

bool PropertyRecord::IsPropertyNameNumeric(const wchar\_t\* str, int length)

{

uint32 unused;

return IsPropertyNameNumeric(str, length, &unused);

}

#endif

bool PropertyRecord::IsPropertyNameNumeric(const wchar\_t\* str, int length, uint32\* intVal)

{

return (Js::JavascriptOperators::TryConvertToUInt32(str, length, intVal) &&

(\*intVal != Js::JavascriptArray::InvalidIndex));

}

uint32 PropertyRecord::GetNumericValue() const

{

Assert(IsNumeric());

return \*(uint32 \*)(this->GetBuffer() + this->GetLength() + 1);

}

// Initialize all Internal property records

#define INTERNALPROPERTY(name) \

const BuiltInPropertyRecord<1> InternalPropertyRecords::name = { PropertyRecord((PropertyId)InternalPropertyIds::name, (uint)InternalPropertyIds::name, false, 0, false), L"" };

#include "..\InternalPropertyList.h"

const PropertyRecord\* InternalPropertyRecords::GetInternalPropertyName(PropertyId propertyId)

{

Assert(IsInternalPropertyId(propertyId));

switch (propertyId)

{

#define INTERNALPROPERTY(name) \

case InternalPropertyIds::name: \

return InternalPropertyRecords::name;

#include "..\InternalPropertyList.h"

}

Throw::FatalInternalError();

}

PropertyAttributes PropertyRecord::DefaultAttributesForPropertyId(PropertyId propertyId, bool \_\_proto\_\_AsDeleted)

{

switch (propertyId)

{

case PropertyIds::\_\_proto\_\_:

if (\_\_proto\_\_AsDeleted)

{

//

// If the property name is \_\_proto\_\_, it could be either [[prototype]] or ignored, or become a local

// property depending on later environment and property value. To maintain enumeration order when it

// becomes a local property, add the entry as deleted.

//

return PropertyDeletedDefaults;

}

return PropertyDynamicTypeDefaults;

default:

return PropertyDynamicTypeDefaults;

}

}

// Initialize all BuiltIn property records

const BuiltInPropertyRecord<1> BuiltInPropertyRecords::EMPTY = { PropertyRecord(PropertyIds::\_none, 0, false, 0, false), L"" };

#define ENTRY\_INTERNAL\_SYMBOL(n) const BuiltInPropertyRecord<ARRAYSIZE(L"<" L#n L">")> BuiltInPropertyRecords::n = { PropertyRecord(PropertyIds::n, (uint)PropertyIds::n, false, (ARRAYSIZE(L"<" L#n L">") - 1) \* sizeof(wchar\_t), true), L"<" L#n L">" };

#define ENTRY\_SYMBOL(n, d) const BuiltInPropertyRecord<ARRAYSIZE(d)> BuiltInPropertyRecords::n = { PropertyRecord(PropertyIds::n, 0, false, (ARRAYSIZE(d) - 1) \* sizeof(wchar\_t), true), d };

#define ENTRY2(n, s) const BuiltInPropertyRecord<ARRAYSIZE(s)> BuiltInPropertyRecords::n = { PropertyRecord(PropertyIds::n, 0, false, (ARRAYSIZE(s) - 1) \* sizeof(wchar\_t), false), s };

#define ENTRY(n) ENTRY2(n, L#n)

#include "Base\JnDirectFields.h"

};

namespace JsUtil

{

bool NoCaseComparer<Js::CaseInvariantPropertyListWithHashCode\*>::Equals(\_In\_ Js::CaseInvariantPropertyListWithHashCode\* list1, JsUtil::CharacterBuffer<WCHAR> const& str)

{

Assert(list1 != nullptr);

const RecyclerWeakReference<Js::PropertyRecord const>\* propRecordWeakRef = list1->CompactEnd<true>();

// If the lists are empty post-compaction, thats fine, we'll just remove them later

if (propRecordWeakRef != nullptr)

{

const Js::PropertyRecord\* prop = propRecordWeakRef->Get();

// Since compaction returned this pointer, their strong refs should not be null

Assert(prop);

JsUtil::CharacterBuffer<WCHAR> string(prop->GetBuffer(), prop->GetLength());

return NoCaseComparer<JsUtil::CharacterBuffer<WCHAR> >::Equals(string, str);

}

// If either of the property strings contains no entries, the two lists are not equivalent

return false;

}

bool NoCaseComparer<Js::CaseInvariantPropertyListWithHashCode\*>::Equals(\_In\_ Js::CaseInvariantPropertyListWithHashCode\* list1, \_In\_ Js::CaseInvariantPropertyListWithHashCode\* list2)

{

Assert(list1 != nullptr && list2 != nullptr);

// If the two lists are the same, they're equal

if (list1 == list2)

{

return true;

}

// If they don't have the same case invariant hash code, they're not equal

if (list1->caseInvariantHashCode != list2->caseInvariantHashCode)

{

return false;

}

// Find a string from list 2

// If it's the same when compared with a string from list 1 in a case insensitive way, they're equal

const RecyclerWeakReference<Js::PropertyRecord const>\* propRecordWeakRef = list2->CompactEnd<true>();

if (propRecordWeakRef != nullptr)

{

const Js::PropertyRecord\* prop = propRecordWeakRef->Get();

// Since compaction returned this pointer, their strong refs should not be null

Assert(prop);

JsUtil::CharacterBuffer<WCHAR> string(prop->GetBuffer(), prop->GetLength());

return NoCaseComparer<Js::CaseInvariantPropertyListWithHashCode\*>::Equals(list1, string);

}

return false;

}

uint NoCaseComparer<Js::CaseInvariantPropertyListWithHashCode\*>::GetHashCode(\_In\_ Js::CaseInvariantPropertyListWithHashCode\* list)

{

Assert(list != nullptr);

if (list->caseInvariantHashCode == 0)

{

const RecyclerWeakReference<Js::PropertyRecord const>\* propRecordWeakRef = list->CompactEnd<true>();

if (propRecordWeakRef != nullptr)

{

const Js::PropertyRecord\* prop = propRecordWeakRef->Get();

Assert(prop);

JsUtil::CharacterBuffer<WCHAR> string(prop->GetBuffer(), prop->GetLength());

list->caseInvariantHashCode = NoCaseComparer<JsUtil::CharacterBuffer<WCHAR> >::GetHashCode(string);

}

}

return list->caseInvariantHashCode;

}

}

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

// 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 PROPERTY\_RECORD\_TRACE

#define PropertyRecordTrace(...) \

if (Js::Configuration::Global.flags.Trace.IsEnabled(Js::PropertyRecordPhase)) \

{ \

Output::Print(\_\_VA\_ARGS\_\_); \

}

#else

#define PropertyRecordTrace(...)

#endif

class ThreadContext;

namespace Js

{

class PropertyRecord : FinalizableObject

{

friend class ThreadContext;

template <int LEN>

friend struct BuiltInPropertyRecord;

friend class InternalPropertyRecords;

friend class BuiltInPropertyRecords;

friend class DOMBuiltInPropertyRecords;

private:

PropertyId pid;

//Made this mutable so that we can set it for Built-In js property records when we are adding it.

//If we try to set it when initializing; we get extra code added for each built in; and thus increasing the size of chakracore

mutable uint hash;

bool isNumeric;

bool isBound;

bool isSymbol;

// Have the length before the buffer so that the buffer would have a BSTR format

DWORD byteCount;

PropertyRecord(DWORD bytelength, bool isNumeric, uint hash, bool isSymbol);

PropertyRecord(PropertyId pid, uint hash, bool isNumeric, DWORD byteCount, bool isSymbol);

PropertyRecord() { Assert(false); } // never used, needed by compiler for BuiltInPropertyRecord

static bool IsPropertyNameNumeric(const wchar\_t\* str, int length, uint32\* intVal);

public:

#ifdef DEBUG

static bool IsPropertyNameNumeric(const wchar\_t\* str, int length);

#endif

static PropertyRecord \* New(Recycler \* recycler, JsUtil::CharacterBuffer<WCHAR> const& propertyName);

static PropertyAttributes DefaultAttributesForPropertyId(PropertyId propertyId, bool \_\_proto\_\_AsDeleted);

PropertyId GetPropertyId() const { return pid; }

uint GetHashCode() const { return hash; }

charcount\_t GetLength() const

{

return byteCount / sizeof(wchar\_t);

}

const wchar\_t\* GetBuffer() const

{

return (const wchar\_t \*)(this + 1);

}

bool IsNumeric() const { return isNumeric; }

uint32 GetNumericValue() const;

bool IsBound() const { return isBound; }

bool IsSymbol() const { return isSymbol; }

void SetHash(uint hash) const

{

this->hash = hash;

}

bool Equals(JsUtil::CharacterBuffer<WCHAR> const & str) const

{

return (this->GetLength() == str.GetLength() && !Js::IsInternalPropertyId(this->GetPropertyId()) &&

JsUtil::CharacterBuffer<WCHAR>::StaticEquals(this->GetBuffer(), str.GetBuffer(), this->GetLength()));

}

bool Equals(PropertyRecord const & propertyRecord) const

{

return (this->GetLength() == propertyRecord.GetLength() &&

Js::IsInternalPropertyId(this->GetPropertyId()) == Js::IsInternalPropertyId(propertyRecord.GetPropertyId()) &&

JsUtil::CharacterBuffer<WCHAR>::StaticEquals(this->GetBuffer(), propertyRecord.GetBuffer(), this->GetLength()));

}

public:

// Finalizable support

virtual void Finalize(bool isShutdown);

virtual void Dispose(bool isShutdown)

{

}

virtual void Mark(Recycler \*recycler) override { AssertMsg(false, "Mark called on object that isn't TrackableObject"); }

};

// This struct maps to the layout of runtime allocated PropertyRecord. Used for creating built-in PropertyRecords statically.

template <int LEN>

struct BuiltInPropertyRecord

{

PropertyRecord propertyRecord;

wchar\_t buffer[LEN];

operator const PropertyRecord\*() const

{

return &propertyRecord;

}

bool Equals(JsUtil::CharacterBuffer<WCHAR> const & str) const

{

return (LEN - 1 == str.GetLength() &&

JsUtil::CharacterBuffer<WCHAR>::StaticEquals(buffer, str.GetBuffer(), LEN - 1));

}

};

// Internal PropertyRecords mapping to InternalPropertyIds. Property names of internal PropertyRecords are not used

// and set to empty string.

class InternalPropertyRecords

{

public:

#define INTERNALPROPERTY(n) const static BuiltInPropertyRecord<1> n;

#include "..\InternalPropertyList.h"

static const PropertyRecord\* GetInternalPropertyName(PropertyId propertyId);

};

// Built-in PropertyRecords. Created statically with known PropertyIds.

class BuiltInPropertyRecords

{

public:

const static BuiltInPropertyRecord<1> EMPTY;

#define ENTRY\_INTERNAL\_SYMBOL(n) const static BuiltInPropertyRecord<ARRAYSIZE(L"<" L#n L">")> n;

#define ENTRY\_SYMBOL(n, d) const static BuiltInPropertyRecord<ARRAYSIZE(d)> n;

#define ENTRY(n) const static BuiltInPropertyRecord<ARRAYSIZE(L#n)> n;

#define ENTRY2(n, s) const static BuiltInPropertyRecord<ARRAYSIZE(s)> n;

#include "Base\JnDirectFields.h"

};

template <typename TChar>

class HashedCharacterBuffer : public JsUtil::CharacterBuffer<TChar>

{

private:

hash\_t hashCode;

public:

HashedCharacterBuffer(TChar const \* string, charcount\_t len) :

JsUtil::CharacterBuffer<TChar>(string, len)

{

this->hashCode = JsUtil::CharacterBuffer<WCHAR>::StaticGetHashCode(string, len);

}

hash\_t GetHashCode() const { return this->hashCode; }

};

struct PropertyRecordPointerComparer

{

\_\_inline static bool Equals(PropertyRecord const \* str1, PropertyRecord const \* str2)

{

return (str1->GetLength() == str2->GetLength() &&

JsUtil::CharacterBuffer<WCHAR>::StaticEquals(str1->GetBuffer(), str2->GetBuffer(), str1->GetLength()));

}

\_\_inline static bool Equals(PropertyRecord const \* str1, JsUtil::CharacterBuffer<WCHAR> const \* str2)

{

return (str1->GetLength() == str2->GetLength() && !Js::IsInternalPropertyId(str1->GetPropertyId()) &&

JsUtil::CharacterBuffer<WCHAR>::StaticEquals(str1->GetBuffer(), str2->GetBuffer(), str1->GetLength()));

}

\_\_inline static hash\_t GetHashCode(PropertyRecord const \* str)

{

return str->GetHashCode();

}

\_\_inline static hash\_t GetHashCode(JsUtil::CharacterBuffer<WCHAR> const \* str)

{

return JsUtil::CharacterBuffer<WCHAR>::StaticGetHashCode(str->GetBuffer(), str->GetLength());

}

};

template<typename T>

struct PropertyRecordStringHashComparer

{

\_\_inline static bool Equals(T str1, T str2)

{

static\_assert(false, "Unexpected type T; note T == PropertyId not allowed!");

}

\_\_inline static hash\_t GetHashCode(T str)

{

// T == PropertyId is not allowed because there is no way to get the string hash

// from just a PropertyId value, the PropertyRecord is required for that.

static\_assert(false, "Unexpected type T; note T == PropertyId not allowed!");

}

};

template<>

struct PropertyRecordStringHashComparer<PropertyRecord const \*>

{

\_\_inline static bool Equals(PropertyRecord const \* str1, PropertyRecord const \* str2)

{

return str1 == str2;

}

\_\_inline static bool Equals(PropertyRecord const \* str1, JsUtil::CharacterBuffer<WCHAR> const & str2)

{

return (!str1->IsSymbol() &&

str1->GetLength() == str2.GetLength() &&

!Js::IsInternalPropertyId(str1->GetPropertyId()) &&

JsUtil::CharacterBuffer<WCHAR>::StaticEquals(str1->GetBuffer(), str2.GetBuffer(), str1->GetLength()));

}

\_\_inline static bool Equals(PropertyRecord const \* str1, HashedCharacterBuffer<wchar\_t> const & str2)

{

return (!str1->IsSymbol() &&

str1->GetHashCode() == str2.GetHashCode() &&

str1->GetLength() == str2.GetLength() &&

!Js::IsInternalPropertyId(str1->GetPropertyId()) &&

JsUtil::CharacterBuffer<wchar\_t>::StaticEquals(str1->GetBuffer(), str2.GetBuffer(), str1->GetLength()));

}

\_\_inline static bool Equals(PropertyRecord const \* str1, JavascriptString \* str2);

\_\_inline static hash\_t GetHashCode(const PropertyRecord\* str)

{

return str->GetHashCode();

}

};

template<>

struct PropertyRecordStringHashComparer<JsUtil::CharacterBuffer<WCHAR>>

{

\_\_inline static bool Equals(JsUtil::CharacterBuffer<WCHAR> const & str1, JsUtil::CharacterBuffer<WCHAR> const & str2)

{

return (str1.GetLength() == str2.GetLength() &&

JsUtil::CharacterBuffer<WCHAR>::StaticEquals(str1.GetBuffer(), str2.GetBuffer(), str1.GetLength()));

}

\_\_inline static hash\_t GetHashCode(JsUtil::CharacterBuffer<WCHAR> const & str)

{

return JsUtil::CharacterBuffer<WCHAR>::StaticGetHashCode(str.GetBuffer(), str.GetLength());

}

};

template<>

struct PropertyRecordStringHashComparer<HashedCharacterBuffer<wchar\_t>>

{

\_\_inline static hash\_t GetHashCode(HashedCharacterBuffer<wchar\_t> const & str)

{

return str.GetHashCode();

}

};

class CaseInvariantPropertyListWithHashCode: public JsUtil::List<const RecyclerWeakReference<Js::PropertyRecord const>\*>

{

public:

CaseInvariantPropertyListWithHashCode(Recycler\* recycler, int increment):

JsUtil::List<const RecyclerWeakReference<Js::PropertyRecord const>\*>(recycler, increment),

caseInvariantHashCode(0)

{

}

uint caseInvariantHashCode;

};

}

// Hash and lookup by PropertyId

template <>

struct DefaultComparer<const Js::PropertyRecord\*>

{

\_\_inline static hash\_t GetHashCode(const Js::PropertyRecord\* str)

{

return DefaultComparer<Js::PropertyId>::GetHashCode(str->GetPropertyId());

}

\_\_inline static bool Equals(const Js::PropertyRecord\* str, Js::PropertyId propertyId)

{

return str->GetPropertyId() == propertyId;

}

\_\_inline static bool Equals(const Js::PropertyRecord\* str1, const Js::PropertyRecord\* str2)

{

return str1 == str2;

}

};

namespace JsUtil

{

template<>

struct NoCaseComparer<Js::CaseInvariantPropertyListWithHashCode\*>

{

static bool Equals(\_In\_ Js::CaseInvariantPropertyListWithHashCode\* list1, \_In\_ Js::CaseInvariantPropertyListWithHashCode\* list2);

static bool Equals(\_In\_ Js::CaseInvariantPropertyListWithHashCode\* list, JsUtil::CharacterBuffer<WCHAR> const& str);

static hash\_t GetHashCode(\_In\_ Js::CaseInvariantPropertyListWithHashCode\* list);

};

}

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

// 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

{

typedef JsUtil::MruDictionary<UnifiedRegex::RegexKey, UnifiedRegex::RegexPattern\*> RegexPatternMruMap;

};

namespace JsUtil

{

template <>

class ValueEntry<Js::RegexPatternMruMap::MruDictionaryData>: public BaseValueEntry<Js::RegexPatternMruMap::MruDictionaryData>

{

public:

void Clear()

{

this->value = 0;

}

};

};

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

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

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

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

#include "RuntimeBasePch.h"

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

// 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

#include "Parser.h"

#include "Runtime.h"

#include "Debug\DebuggingFlags.h"

#include "Debug\DiagProbe.h"

#include "Debug\DebugManager.h"

#include "Debug\ProbeContainer.h"

#include "Debug\DebugContext.h"

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

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

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

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

#include "RuntimeBasePch.h"

// Parser Includes

#include "RegexCommon.h"

#include "DebugWriter.h"

#include "RegexStats.h"

#include "ByteCode\ByteCodeAPI.h"

#include "Library\ProfileString.h"

#include "Debug\DiagHelperMethodWrapper.h"

#include "BackEndAPI.h"

#if PROFILE\_DICTIONARY

#include "DictionaryStats.h"

#endif

#include "Base\ScriptContextProfiler.h"

#include "Base\EtwTrace.h"

#include "Language\InterpreterStackFrame.h"

#include "Language\SourceDynamicProfileManager.h"

#include "Language\JavascriptStackWalker.h"

#include "Language\AsmJsTypes.h"

#include "Language\AsmJsModule.h"

#ifdef ASMJS\_PLAT

#include "Language\AsmJsEncoder.h"

#include "Language\AsmJsCodeGenerator.h"

#endif

#ifdef ENABLE\_BASIC\_TELEMETRY

#include "ScriptContextTelemetry.h"

#endif

namespace Js

{

ScriptContext \* ScriptContext::New(ThreadContext \* threadContext)

{

AutoPtr<ScriptContext> scriptContext(HeapNew(ScriptContext, threadContext));

scriptContext->InitializeAllocations();

return scriptContext.Detach();

}

void ScriptContext::Delete(ScriptContext\* scriptContext)

{

HeapDelete(scriptContext);

}

ScriptContext::ScriptContext(ThreadContext\* threadContext) :

ScriptContextBase(),

interpreterArena(nullptr),

dynamicFunctionReference(nullptr),

moduleSrcInfoCount(0),

// Regex globals

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

regexStatsDatabase(0),

regexDebugWriter(0),

#endif

trigramAlphabet(nullptr),

regexStacks(nullptr),

arrayMatchInit(false),

config(threadContext->GetConfig(), threadContext->IsOptimizedForManyInstances()),

#if ENABLE\_BACKGROUND\_PARSING

backgroundParser(nullptr),

#endif

#if ENABLE\_NATIVE\_CODEGEN

nativeCodeGen(nullptr),

#endif

threadContext(threadContext),

scriptStartEventHandler(nullptr),

scriptEndEventHandler(nullptr),

#ifdef FAULT\_INJECTION

disposeScriptByFaultInjectionEventHandler(nullptr),

#endif

integerStringMap(this->GeneralAllocator()),

guestArena(nullptr),

raiseMessageToDebuggerFunctionType(nullptr),

transitionToDebugModeIfFirstSourceFn(nullptr),

lastTimeZoneUpdateTickCount(0),

sourceSize(0),

deferredBody(false),

isScriptContextActuallyClosed(false),

isInvalidatedForHostObjects(false),

fastDOMenabled(false),

directHostTypeId(TypeIds\_GlobalObject),

isPerformingNonreentrantWork(false),

isDiagnosticsScriptContext(false),

m\_enumerateNonUserFunctionsOnly(false),

recycler(threadContext->EnsureRecycler()),

CurrentThunk(DefaultEntryThunk),

CurrentCrossSiteThunk(CrossSite::DefaultThunk),

DeferredParsingThunk(DefaultDeferredParsingThunk),

DeferredDeserializationThunk(DefaultDeferredDeserializeThunk),

m\_pBuiltinFunctionIdMap(nullptr),

diagnosticArena(nullptr),

hostScriptContext(nullptr),

scriptEngineHaltCallback(nullptr),

#if DYNAMIC\_INTERPRETER\_THUNK

interpreterThunkEmitter(nullptr),

#endif

#ifdef ASMJS\_PLAT

asmJsInterpreterThunkEmitter(nullptr),

asmJsCodeGenerator(nullptr),

#endif

generalAllocator(L"SC-General", threadContext->GetPageAllocator(), Throw::OutOfMemory),

#ifdef ENABLE\_BASIC\_TELEMETRY

telemetryAllocator(L"SC-Telemetry", threadContext->GetPageAllocator(), Throw::OutOfMemory),

#endif

dynamicProfileInfoAllocator(L"SC-DynProfileInfo", threadContext->GetPageAllocator(), Throw::OutOfMemory),

#ifdef SEPARATE\_ARENA

sourceCodeAllocator(L"SC-Code", threadContext->GetPageAllocator(), Throw::OutOfMemory),

regexAllocator(L"SC-Regex", threadContext->GetPageAllocator(), Throw::OutOfMemory),

#endif

#ifdef NEED\_MISC\_ALLOCATOR

miscAllocator(L"GC-Misc", threadContext->GetPageAllocator(), Throw::OutOfMemory),

#endif

inlineCacheAllocator(L"SC-InlineCache", threadContext->GetPageAllocator(), Throw::OutOfMemory),

isInstInlineCacheAllocator(L"SC-IsInstInlineCache", threadContext->GetPageAllocator(), Throw::OutOfMemory),

hasRegisteredInlineCache(false),

hasRegisteredIsInstInlineCache(false),

entryInScriptContextWithInlineCachesRegistry(nullptr),

entryInScriptContextWithIsInstInlineCachesRegistry(nullptr),

registeredPrototypeChainEnsuredToHaveOnlyWritableDataPropertiesScriptContext(nullptr),

cache(nullptr),

bindRefChunkCurrent(nullptr),

bindRefChunkEnd(nullptr),

firstInterpreterFrameReturnAddress(nullptr),

builtInLibraryFunctions(nullptr),

isWeakReferenceDictionaryListCleared(false)

#if ENABLE\_PROFILE\_INFO

, referencesSharedDynamicSourceContextInfo(false)

#endif

#if DBG

, isInitialized(false)

, isCloningGlobal(false)

, bindRef(MiscAllocator())

#endif

#ifdef REJIT\_STATS

, rejitStatsMap(nullptr)

#endif

#ifdef ENABLE\_BASIC\_TELEMETRY

, telemetry(nullptr)

#endif

#ifdef INLINE\_CACHE\_STATS

, cacheDataMap(nullptr)

#endif

#ifdef FIELD\_ACCESS\_STATS

, fieldAccessStatsByFunctionNumber(nullptr)

#endif

, webWorkerId(Js::Constants::NonWebWorkerContextId)

, url(L"")

, startupComplete(false)

, isEnumeratingRecyclerObjects(false)

#ifdef EDIT\_AND\_CONTINUE

, activeScriptEditQuery(nullptr)

#endif

, heapEnum(nullptr)

#ifdef RECYCLER\_PERF\_COUNTERS

, bindReferenceCount(0)

#endif

, nextPendingClose(nullptr)

, m\_fTraceDomCall(FALSE)

#ifdef ENABLE\_DOM\_FAST\_PATH

, domFastPathIRHelperMap(nullptr)

#endif

, intConstPropsOnGlobalObject(nullptr)

, intConstPropsOnGlobalUserObject(nullptr)

#ifdef PROFILE\_STRINGS

, stringProfiler(nullptr)

#endif

#ifdef PROFILE\_BAILOUT\_RECORD\_MEMORY

, codeSize(0)

, bailOutRecordBytes(0)

, bailOutOffsetBytes(0)

, debugContext(nullptr)

#endif

{

// This may allocate memory and cause exception, but it is ok, as we all we have done so far

// are field init and those dtor will be called if exception occurs

threadContext->EnsureDebugManager();

// Don't use throwing memory allocation in ctor, as exception in ctor doesn't cause the dtor to be called

// potentially causing memory leaks

BEGIN\_NO\_EXCEPTION;

#ifdef RUNTIME\_DATA\_COLLECTION

createTime = time(nullptr);

#endif

#ifdef BGJIT\_STATS

interpretedCount = maxFuncInterpret = funcJITCount = bytecodeJITCount = interpretedCallsHighPri = jitCodeUsed = funcJitCodeUsed = loopJITCount = speculativeJitCount = 0;

#endif

#ifdef PROFILE\_TYPES

convertNullToSimpleCount = 0;

convertNullToSimpleDictionaryCount = 0;

convertNullToDictionaryCount = 0;

convertDeferredToDictionaryCount = 0;

convertDeferredToSimpleDictionaryCount = 0;

convertSimpleToDictionaryCount = 0;

convertSimpleToSimpleDictionaryCount = 0;

convertPathToDictionaryCount1 = 0;

convertPathToDictionaryCount2 = 0;

convertPathToDictionaryCount3 = 0;

convertPathToDictionaryCount4 = 0;

convertPathToSimpleDictionaryCount = 0;

convertSimplePathToPathCount = 0;

convertSimpleDictionaryToDictionaryCount = 0;

convertSimpleSharedDictionaryToNonSharedCount = 0;

convertSimpleSharedToNonSharedCount = 0;

simplePathTypeHandlerCount = 0;

pathTypeHandlerCount = 0;

promoteCount = 0;

cacheCount = 0;

branchCount = 0;

maxPathLength = 0;

memset(typeCount, 0, sizeof(typeCount));

memset(instanceCount, 0, sizeof(instanceCount));

#endif

#ifdef PROFILE\_OBJECT\_LITERALS

objectLiteralInstanceCount = 0;

objectLiteralPathCount = 0;

memset(objectLiteralCount, 0, sizeof(objectLiteralCount));

objectLiteralSimpleDictionaryCount = 0;

objectLiteralMaxLength = 0;

objectLiteralPromoteCount = 0;

objectLiteralCacheCount = 0;

objectLiteralBranchCount = 0;

#endif

#if DBG\_DUMP

byteCodeDataSize = 0;

byteCodeAuxiliaryDataSize = 0;

byteCodeAuxiliaryContextDataSize = 0;

memset(byteCodeHistogram, 0, sizeof(byteCodeHistogram));

#endif

memset(propertyStrings, 0, sizeof(PropertyStringMap\*)\* 80);

#if DBG || defined(RUNTIME\_DATA\_COLLECTION)

this->allocId = threadContext->GetUnreleasedScriptContextCount();

#endif

#if DBG

this->hadProfiled = false;

#endif

#if DBG\_DUMP

forinCache = 0;

forinNoCache = 0;

#endif

callCount = 0;

threadContext->GetHiResTimer()->Reset();

#ifdef PROFILE\_EXEC

profiler = nullptr;

isProfilerCreated = false;

disableProfiler = false;

ensureParentInfo = false;

#endif

#ifdef PROFILE\_MEM

profileMemoryDump = true;

#endif

m\_pProfileCallback = nullptr;

m\_pProfileCallback2 = nullptr;

m\_inProfileCallback = FALSE;

CleanupDocumentContext = nullptr;

// Do this after all operations that may cause potential exceptions

threadContext->RegisterScriptContext(this);

numberAllocator.Initialize(this->GetRecycler());

#if DEBUG

m\_iProfileSession = -1;

#endif

#ifdef LEAK\_REPORT

this->urlRecord = nullptr;

this->isRootTrackerScriptContext = false;

#endif

PERF\_COUNTER\_INC(Basic, ScriptContext);

PERF\_COUNTER\_INC(Basic, ScriptContextActive);

END\_NO\_EXCEPTION;

}

void ScriptContext::InitializeAllocations()

{

this->charClassifier = Anew(GeneralAllocator(), CharClassifier, this);

this->valueOfInlineCache = AllocatorNewZ(InlineCacheAllocator, GetInlineCacheAllocator(), InlineCache);

this->toStringInlineCache = AllocatorNewZ(InlineCacheAllocator, GetInlineCacheAllocator(), InlineCache);

#ifdef REJIT\_STATS

if (PHASE\_STATS1(Js::ReJITPhase))

{

rejitReasonCounts = AnewArrayZ(GeneralAllocator(), uint, NumRejitReasons);

bailoutReasonCounts = Anew(GeneralAllocator(), BailoutStatsMap, GeneralAllocator());

}

#endif

#ifdef ENABLE\_BASIC\_TELEMETRY

this->telemetry = Anew(this->TelemetryAllocator(), ScriptContextTelemetry, \*this);

#endif

#ifdef PROFILE\_STRINGS

if (Js::Configuration::Global.flags.ProfileStrings)

{

stringProfiler = Anew(MiscAllocator(), StringProfiler, threadContext->GetPageAllocator());

}

#endif

intConstPropsOnGlobalObject = Anew(GeneralAllocator(), PropIdSetForConstProp, GeneralAllocator());

intConstPropsOnGlobalUserObject = Anew(GeneralAllocator(), PropIdSetForConstProp, GeneralAllocator());

this->debugContext = HeapNew(DebugContext, this);

}

void ScriptContext::EnsureClearDebugDocument()

{

if (this->sourceList)

{

this->sourceList->Map([=](uint i, RecyclerWeakReference<Js::Utf8SourceInfo>\* sourceInfoWeakRef) {

Js::Utf8SourceInfo\* sourceInfo = sourceInfoWeakRef->Get();

if (sourceInfo)

{

sourceInfo->ClearDebugDocument();

}

});

}

}

void ScriptContext::ShutdownClearSourceLists()

{

if (this->sourceList)

{

// In the unclean shutdown case, we might not have destroyed the script context when

// this is called- in which case, skip doing this work and simply release the source list

// so that it doesn't show up as a leak. Since we're doing unclean shutdown, it's ok to

// skip cleanup here for expediency.

if (this->isClosed)

{

this->MapFunction([this](Js::FunctionBody\* functionBody) {

Assert(functionBody->GetScriptContext() == this);

functionBody->CleanupSourceInfo(true);

});

}

EnsureClearDebugDocument();

// Don't need the source list any more so ok to release

this->sourceList.Unroot(this->GetRecycler());

}

if (this->calleeUtf8SourceInfoList)

{

this->calleeUtf8SourceInfoList.Unroot(this->GetRecycler());

}

}

ScriptContext::~ScriptContext()

{

// Take etw rundown lock on this thread context. We are going to change/destroy this scriptContext.

AutoCriticalSection autocs(GetThreadContext()->GetEtwRundownCriticalSection());

// TODO: Can we move this on Close()?

ClearHostScriptContext();

threadContext->UnregisterScriptContext(this);

// Only call RemoveFromPendingClose if we are in a pending close state.

if (isClosed && !isScriptContextActuallyClosed)

{

threadContext->RemoveFromPendingClose(this);

}

this->isClosed = true;

bool closed = Close(true);

// JIT may access number allocator. Need to close the script context first,

// which will close the native code generator and abort any current job on this generator.

numberAllocator.Uninitialize();

ShutdownClearSourceLists();

if (regexStacks)

{

Adelete(RegexAllocator(), regexStacks);

regexStacks = nullptr;

}

if (javascriptLibrary != nullptr)

{

javascriptLibrary->scriptContext = nullptr;

javascriptLibrary = nullptr;

if (closed)

{

// if we just closed, we haven't unpin the object yet.

// We need to null out the script context in the global object first

// before we unpin the global object so that script context dtor doesn't get called twice

#if ENABLE\_NATIVE\_CODEGEN

Assert(this->IsClosedNativeCodeGenerator());

#endif

this->recycler->RootRelease(globalObject);

}

}

#if ENABLE\_BACKGROUND\_PARSING

if (this->backgroundParser != nullptr)

{

BackgroundParser::Delete(this->backgroundParser);

this->backgroundParser = nullptr;

}

#endif

#if ENABLE\_NATIVE\_CODEGEN

if (this->nativeCodeGen != nullptr)

{

DeleteNativeCodeGenerator(this->nativeCodeGen);

nativeCodeGen = NULL;

}

#endif

#if DYNAMIC\_INTERPRETER\_THUNK

if (this->interpreterThunkEmitter != nullptr)

{

HeapDelete(interpreterThunkEmitter);

this->interpreterThunkEmitter = NULL;

}

#endif

#ifdef ASMJS\_PLAT

if (this->asmJsInterpreterThunkEmitter != nullptr)

{

HeapDelete(asmJsInterpreterThunkEmitter);

this->asmJsInterpreterThunkEmitter = nullptr;

}

if (this->asmJsCodeGenerator != nullptr)

{

HeapDelete(asmJsCodeGenerator);

this->asmJsCodeGenerator = NULL;

}

#endif

if (this->hasRegisteredInlineCache)

{

// TODO (PersistentInlineCaches): It really isn't necessary to clear inline caches in all script contexts.

// Since this script context is being destroyed, the inline cache arena will also go away and release its

// memory back to the page allocator. Thus, we cannot leave this script context's inline caches on the

// thread context's invalidation lists. However, it should suffice to remove this script context's caches

// without touching other script contexts' caches. We could call some form of RemoveInlineCachesFromInvalidationLists()

// on the inline cache allocator, which would walk all inline caches and zap values pointed to by strongRef.

// clear out all inline caches to remove our proto inline caches from the thread context

threadContext->ClearInlineCaches();

Assert(!this->hasRegisteredInlineCache);

Assert(this->entryInScriptContextWithInlineCachesRegistry == nullptr);

}

else if (this->entryInScriptContextWithInlineCachesRegistry != nullptr)

{

// UnregisterInlineCacheScriptContext may throw, set up the correct state first

ScriptContext \*\* entry = this->entryInScriptContextWithInlineCachesRegistry;

this->entryInScriptContextWithInlineCachesRegistry = nullptr;

threadContext->UnregisterInlineCacheScriptContext(entry);

}

if (this->hasRegisteredIsInstInlineCache)

{

// clear out all inline caches to remove our proto inline caches from the thread context

threadContext->ClearIsInstInlineCaches();

Assert(!this->hasRegisteredIsInstInlineCache);

Assert(this->entryInScriptContextWithIsInstInlineCachesRegistry == nullptr);

}

else if (this->entryInScriptContextWithInlineCachesRegistry != nullptr)

{

// UnregisterInlineCacheScriptContext may throw, set up the correct state first

ScriptContext \*\* entry = this->entryInScriptContextWithInlineCachesRegistry;

this->entryInScriptContextWithInlineCachesRegistry = nullptr;

threadContext->UnregisterIsInstInlineCacheScriptContext(entry);

}

// In case there is something added to the list between close and dtor, just reset the list again

this->weakReferenceDictionaryList.Reset();

PERF\_COUNTER\_DEC(Basic, ScriptContext);

}

void ScriptContext::SetUrl(BSTR bstrUrl)

{

// Assumption: this method is never called multiple times

Assert(this->url != nullptr && wcslen(this->url) == 0);

charcount\_t length = SysStringLen(bstrUrl) + 1; // Add 1 for the NULL.

wchar\_t\* urlCopy = AnewArray(this->GeneralAllocator(), wchar\_t, length);

js\_memcpy\_s(urlCopy, (length - 1) \* sizeof(wchar\_t), bstrUrl, (length - 1) \* sizeof(wchar\_t));

urlCopy[length - 1] = L'\0';

this->url = urlCopy;

#ifdef LEAK\_REPORT

if (Js::Configuration::Global.flags.IsEnabled(Js::LeakReportFlag))

{

this->urlRecord = LeakReport::LogUrl(urlCopy, this->globalObject);

}

#endif

}

uint ScriptContext::GetNextSourceContextId()

{

Assert(this->cache);

Assert(this->cache->sourceContextInfoMap ||

this->cache->dynamicSourceContextInfoMap);

uint nextSourceContextId = 0;

if (this->cache->sourceContextInfoMap)

{

nextSourceContextId = this->cache->sourceContextInfoMap->Count();

}

if (this->cache->dynamicSourceContextInfoMap)

{

nextSourceContextId += this->cache->dynamicSourceContextInfoMap->Count();

}

return nextSourceContextId + 1;

}

// Do most of the Close() work except the final release which could delete the scriptContext.

void ScriptContext::InternalClose()

{

this->PrintStats();

isScriptContextActuallyClosed = true;

PERF\_COUNTER\_DEC(Basic, ScriptContextActive);

#if DBG\_DUMP

if (Js::Configuration::Global.flags.TraceWin8Allocations)

{

Output::Print(L"MemoryTrace: ScriptContext Close\n");

Output::Flush();

}

#endif

#ifdef ENABLE\_JS\_ETW

EventWriteJSCRIPT\_HOST\_SCRIPT\_CONTEXT\_CLOSE(this);

#endif

#if ENABLE\_PROFILE\_INFO

HRESULT hr = S\_OK;

BEGIN\_TRANSLATE\_OOM\_TO\_HRESULT\_NESTED

{

DynamicProfileInfo::Save(this);

}

END\_TRANSLATE\_OOM\_TO\_HRESULT(hr);

#if DBG\_DUMP || defined(DYNAMIC\_PROFILE\_STORAGE) || defined(RUNTIME\_DATA\_COLLECTION)

this->ClearDynamicProfileList();

#endif

#endif

#if ENABLE\_NATIVE\_CODEGEN

if (nativeCodeGen != nullptr)

{

Assert(!isInitialized || this->globalObject != nullptr);

CloseNativeCodeGenerator(this->nativeCodeGen);

}

#endif

if (this->fakeGlobalFuncForUndefer)

{

this->fakeGlobalFuncForUndefer->Cleanup(true);

this->fakeGlobalFuncForUndefer.Unroot(this->GetRecycler());

}

if (this->sourceList)

{

bool hasFunctions = false;

this->sourceList->MapUntil([&hasFunctions](int, RecyclerWeakReference<Utf8SourceInfo>\* sourceInfoWeakRef) -> bool

{

Utf8SourceInfo\* sourceInfo = sourceInfoWeakRef->Get();

if (sourceInfo)

{

hasFunctions = sourceInfo->HasFunctions();

}

return hasFunctions;

});

if (hasFunctions)

{

// We still need to walk through all the function bodies and call cleanup

// because otherwise ETW events might not get fired if a GC doesn't happen

// and the thread context isn't shut down cleanly (process detach case)

this->MapFunction([this](Js::FunctionBody\* functionBody) {

Assert(functionBody->GetScriptContext() == this);

functionBody->Cleanup(/\* isScriptContextClosing \*/ true);

});

}

}

JS\_ETW(EtwTrace::LogSourceUnloadEvents(this));

this->GetThreadContext()->SubSourceSize(this->GetSourceSize());

#if DYNAMIC\_INTERPRETER\_THUNK

if (this->interpreterThunkEmitter != nullptr)

{

this->interpreterThunkEmitter->Close();

}

#endif

#ifdef ASMJS\_PLAT

if (this->asmJsInterpreterThunkEmitter != nullptr)

{

this->asmJsInterpreterThunkEmitter->Close();

}

#endif

// Stop profiling if present

DeRegisterProfileProbe(S\_OK, nullptr);

if (this->diagnosticArena != nullptr)

{

HeapDelete(this->diagnosticArena);

this->diagnosticArena = nullptr;

}

if (this->debugContext != nullptr)

{

this->debugContext->Close();

HeapDelete(this->debugContext);

this->debugContext = nullptr;

}

// Need to print this out before the native code gen is deleted

// which will delete the codegenProfiler

#ifdef PROFILE\_EXEC

if (Js::Configuration::Global.flags.IsEnabled(Js::ProfileFlag))

{

if (isProfilerCreated)

{

this->ProfilePrint();

}

if (profiler != nullptr)

{

profiler->Release();

profiler = nullptr;

}

}

#endif

#if ENABLE\_PROFILE\_INFO

// Release this only after native code gen is shut down, as there may be

// profile info allocated from the SourceDynamicProfileManager arena.

// The first condition might not be true if the dynamic functions have already been freed by the time

// ScriptContext closes

if (referencesSharedDynamicSourceContextInfo)

{

// For the host provided dynamic code, we may not have added any dynamic context to the dynamicSourceContextInfoMap

Assert(this->GetDynamicSourceContextInfoMap() != nullptr);

this->GetThreadContext()->ReleaseSourceDynamicProfileManagers(this->GetUrl());

}

#endif

RECYCLER\_PERF\_COUNTER\_SUB(BindReference, bindReferenceCount);

if (this->interpreterArena)

{

ReleaseInterpreterArena();

interpreterArena = nullptr;

}

if (this->guestArena)

{

ReleaseGuestArena();

guestArena = nullptr;

cache = nullptr;

bindRefChunkCurrent = nullptr;

bindRefChunkEnd = nullptr;

}

builtInLibraryFunctions = nullptr;

pActiveScriptDirect = nullptr;

isWeakReferenceDictionaryListCleared = true;

this->weakReferenceDictionaryList.Clear(this->GeneralAllocator());

// This can be null if the script context initialization threw

// and InternalClose gets called in the destructor code path

if (javascriptLibrary != nullptr)

{

javascriptLibrary->Uninitialize();

}

if (registeredPrototypeChainEnsuredToHaveOnlyWritableDataPropertiesScriptContext != nullptr)

{

// UnregisterPrototypeChainEnsuredToHaveOnlyWritableDataPropertiesScriptContext may throw, set up the correct state first

ScriptContext \*\* registeredScriptContext = registeredPrototypeChainEnsuredToHaveOnlyWritableDataPropertiesScriptContext;

ClearPrototypeChainEnsuredToHaveOnlyWritableDataPropertiesCaches();

Assert(registeredPrototypeChainEnsuredToHaveOnlyWritableDataPropertiesScriptContext == nullptr);

threadContext->UnregisterPrototypeChainEnsuredToHaveOnlyWritableDataPropertiesScriptContext(registeredScriptContext);

}

threadContext->ReleaseDebugManager();

}

bool ScriptContext::Close(bool inDestructor)

{

if (isScriptContextActuallyClosed)

return false;

// Limit the lock scope. We require the same lock in ~ScriptContext(), which may be called next.

{

// Take etw rundown lock on this thread context. We are going to change this scriptContext.

AutoCriticalSection autocs(GetThreadContext()->GetEtwRundownCriticalSection());

InternalClose();

}

if (!inDestructor && globalObject != nullptr)

{

//A side effect of releasing globalObject that this script context could be deleted, so the release call here

//must be the last thing in close.

#if ENABLE\_NATIVE\_CODEGEN

Assert(this->IsClosedNativeCodeGenerator());

#endif

GetRecycler()->RootRelease(globalObject);

}

// A script context closing is a signal to the thread context that it

// needs to do an idle GC independent of what the heuristics are

this->threadContext->SetForceOneIdleCollection();

return true;

}

PropertyString\* ScriptContext::GetPropertyString2(wchar\_t ch1, wchar\_t ch2)

{

if (ch1 < '0' || ch1 > 'z' || ch2 < '0' || ch2 > 'z')

{

return NULL;

}

const uint i = PropertyStringMap::PStrMapIndex(ch1);

if (propertyStrings[i] == NULL)

{

return NULL;

}

const uint j = PropertyStringMap::PStrMapIndex(ch2);

return propertyStrings[i]->strLen2[j];

}

void ScriptContext::FindPropertyRecord(JavascriptString \*pstName, PropertyRecord const \*\* propertyRecord)

{

threadContext->FindPropertyRecord(pstName, propertyRecord);

}

void ScriptContext::FindPropertyRecord(\_\_in LPCWSTR propertyName, \_\_in int propertyNameLength, PropertyRecord const \*\* propertyRecord)

{

threadContext->FindPropertyRecord(propertyName, propertyNameLength, propertyRecord);

}

JsUtil::List<const RecyclerWeakReference<Js::PropertyRecord const>\*>\* ScriptContext::FindPropertyIdNoCase(\_\_in LPCWSTR propertyName, \_\_in int propertyNameLength)

{

return threadContext->FindPropertyIdNoCase(this, propertyName, propertyNameLength);

}

PropertyId ScriptContext::GetOrAddPropertyIdTracked(JsUtil::CharacterBuffer<WCHAR> const& propName)

{

Js::PropertyRecord const \* propertyRecord;

threadContext->GetOrAddPropertyId(propName, &propertyRecord);

this->TrackPid(propertyRecord);

return propertyRecord->GetPropertyId();

}

void ScriptContext::GetOrAddPropertyRecord(JsUtil::CharacterBuffer<WCHAR> const& propertyName, PropertyRecord const \*\* propertyRecord)

{

threadContext->GetOrAddPropertyId(propertyName, propertyRecord);

}

PropertyId ScriptContext::GetOrAddPropertyIdTracked(\_\_in\_ecount(propertyNameLength) LPCWSTR propertyName, \_\_in int propertyNameLength)

{

Js::PropertyRecord const \* propertyRecord;

threadContext->GetOrAddPropertyId(propertyName, propertyNameLength, &propertyRecord);

if (propertyNameLength == 2)

{

CachePropertyString2(propertyRecord);

}

this->TrackPid(propertyRecord);

return propertyRecord->GetPropertyId();

}

void ScriptContext::GetOrAddPropertyRecord(\_\_in\_ecount(propertyNameLength) LPCWSTR propertyName, \_\_in int propertyNameLength, PropertyRecord const \*\* propertyRecord)

{

threadContext->GetOrAddPropertyId(propertyName, propertyNameLength, propertyRecord);

if (propertyNameLength == 2)

{

CachePropertyString2(\*propertyRecord);

}

}

BOOL ScriptContext::IsNumericPropertyId(PropertyId propertyId, uint32\* value)

{

BOOL isNumericPropertyId = threadContext->IsNumericPropertyId(propertyId, value);

#if DEBUG

PropertyRecord const \* name = this->GetPropertyName(propertyId);

if (name != nullptr)

{

// Symbol properties are not numeric - description should not be used.

if (name->IsSymbol())

{

return false;

}

ulong index;

BOOL isIndex = JavascriptArray::GetIndex(name->GetBuffer(), &index);

if (isNumericPropertyId != isIndex)

{

// WOOB 1137798: JavascriptArray::GetIndex does not handle embedded NULLs. So if we have a property

// name "1234\0", JavascriptArray::GetIndex would incorrectly accepts it as an array index property

// name.

Assert((size\_t)(name->GetLength()) != wcslen(name->GetBuffer()));

}

else if (isNumericPropertyId)

{

Assert((ulong)\*value == index);

}

}

#endif

return isNumericPropertyId;

}

void ScriptContext::RegisterWeakReferenceDictionary(JsUtil::IWeakReferenceDictionary\* weakReferenceDictionary)

{

this->weakReferenceDictionaryList.Prepend(this->GeneralAllocator(), weakReferenceDictionary);

}

RecyclableObject \*ScriptContext::GetMissingPropertyResult(Js::RecyclableObject \*instance, Js::PropertyId id)

{

return GetLibrary()->GetUndefined();

}

RecyclableObject \*ScriptContext::GetMissingItemResult(Js::RecyclableObject \*instance, uint32 index)

{

return GetLibrary()->GetUndefined();

}

RecyclableObject \*ScriptContext::GetMissingParameterValue(Js::JavascriptFunction \*function, uint32 paramIndex)

{

return GetLibrary()->GetUndefined();

}

RecyclableObject \*ScriptContext::GetNullPropertyResult(Js::RecyclableObject \*instance, Js::PropertyId id)

{

return GetLibrary()->GetNull();

}

RecyclableObject \*ScriptContext::GetNullItemResult(Js::RecyclableObject \*instance, uint32 index)

{

return GetLibrary()->GetUndefined();

}

SRCINFO \*ScriptContext::AddHostSrcInfo(SRCINFO const \*pSrcInfo)

{

Assert(pSrcInfo != nullptr);

return RecyclerNewZ(this->GetRecycler(), SRCINFO, \*pSrcInfo);

}

#ifdef PROFILE\_TYPES

void ScriptContext::ProfileTypes()

{

Output::Print(L"===============================================================================\n");

Output::Print(L"Types Profile\n");

Output::Print(L"-------------------------------------------------------------------------------\n");

Output::Print(L"Dynamic Type Conversions:\n");

Output::Print(L" Null to Simple %8d\n", convertNullToSimpleCount);

Output::Print(L" Deferred to SimpleMap %8d\n", convertDeferredToSimpleDictionaryCount);

Output::Print(L" Simple to Map %8d\n", convertSimpleToDictionaryCount);

Output::Print(L" Simple to SimpleMap %8d\n", convertSimpleToSimpleDictionaryCount);

Output::Print(L" Path to SimpleMap (set) %8d\n", convertPathToDictionaryCount1);

Output::Print(L" Path to SimpleMap (delete) %8d\n", convertPathToDictionaryCount2);

Output::Print(L" Path to SimpleMap (attribute) %8d\n", convertPathToDictionaryCount3);

Output::Print(L" Path to SimpleMap %8d\n", convertPathToSimpleDictionaryCount);

Output::Print(L" SimplePath to Path %8d\n", convertSimplePathToPathCount);

Output::Print(L" Shared SimpleMap to non-shared %8d\n", convertSimpleSharedDictionaryToNonSharedCount);

Output::Print(L" Deferred to Map %8d\n", convertDeferredToDictionaryCount);

Output::Print(L" Path to Map (accessor) %8d\n", convertPathToDictionaryCount4);

Output::Print(L" SimpleMap to Map %8d\n", convertSimpleDictionaryToDictionaryCount);

Output::Print(L" Path Cache Hits %8d\n", cacheCount);

Output::Print(L" Path Branches %8d\n", branchCount);

Output::Print(L" Path Promotions %8d\n", promoteCount);

Output::Print(L" Path Length (max) %8d\n", maxPathLength);

Output::Print(L" SimplePathTypeHandlers %8d\n", simplePathTypeHandlerCount);

Output::Print(L" PathTypeHandlers %8d\n", pathTypeHandlerCount);

Output::Print(L"\n");

Output::Print(L"Type Statistics: %8s %8s\n", L"Types", L"Instances");

Output::Print(L" Undefined %8d %8d\n", typeCount[TypeIds\_Undefined], instanceCount[TypeIds\_Undefined]);

Output::Print(L" Null %8d %8d\n", typeCount[TypeIds\_Null], instanceCount[TypeIds\_Null]);

Output::Print(L" Boolean %8d %8d\n", typeCount[TypeIds\_Boolean], instanceCount[TypeIds\_Boolean]);

Output::Print(L" Integer %8d %8d\n", typeCount[TypeIds\_Integer], instanceCount[TypeIds\_Integer]);

Output::Print(L" Number %8d %8d\n", typeCount[TypeIds\_Number], instanceCount[TypeIds\_Number]);

Output::Print(L" String %8d %8d\n", typeCount[TypeIds\_String], instanceCount[TypeIds\_String]);

Output::Print(L" Object %8d %8d\n", typeCount[TypeIds\_Object], instanceCount[TypeIds\_Object]);

Output::Print(L" Function %8d %8d\n", typeCount[TypeIds\_Function], instanceCount[TypeIds\_Function]);

Output::Print(L" Array %8d %8d\n", typeCount[TypeIds\_Array], instanceCount[TypeIds\_Array]);

Output::Print(L" Date %8d %8d\n", typeCount[TypeIds\_Date], instanceCount[TypeIds\_Date] + instanceCount[TypeIds\_WinRTDate]);

Output::Print(L" Symbol %8d %8d\n", typeCount[TypeIds\_Symbol], instanceCount[TypeIds\_Symbol]);

Output::Print(L" RegEx %8d %8d\n", typeCount[TypeIds\_RegEx], instanceCount[TypeIds\_RegEx]);

Output::Print(L" Error %8d %8d\n", typeCount[TypeIds\_Error], instanceCount[TypeIds\_Error]);

Output::Print(L" Proxy %8d %8d\n", typeCount[TypeIds\_Proxy], instanceCount[TypeIds\_Proxy]);

Output::Print(L" BooleanObject %8d %8d\n", typeCount[TypeIds\_BooleanObject], instanceCount[TypeIds\_BooleanObject]);

Output::Print(L" NumberObject %8d %8d\n", typeCount[TypeIds\_NumberObject], instanceCount[TypeIds\_NumberObject]);

Output::Print(L" StringObject %8d %8d\n", typeCount[TypeIds\_StringObject], instanceCount[TypeIds\_StringObject]);

Output::Print(L" SymbolObject %8d %8d\n", typeCount[TypeIds\_SymbolObject], instanceCount[TypeIds\_SymbolObject]);

Output::Print(L" GlobalObject %8d %8d\n", typeCount[TypeIds\_GlobalObject], instanceCount[TypeIds\_GlobalObject]);

Output::Print(L" Enumerator %8d %8d\n", typeCount[TypeIds\_Enumerator], instanceCount[TypeIds\_Enumerator]);

Output::Print(L" Int8Array %8d %8d\n", typeCount[TypeIds\_Int8Array], instanceCount[TypeIds\_Int8Array]);

Output::Print(L" Uint8Array %8d %8d\n", typeCount[TypeIds\_Uint8Array], instanceCount[TypeIds\_Uint8Array]);

Output::Print(L" Uint8ClampedArray %8d %8d\n", typeCount[TypeIds\_Uint8ClampedArray], instanceCount[TypeIds\_Uint8ClampedArray]);

Output::Print(L" Int16Array %8d %8d\n", typeCount[TypeIds\_Int16Array], instanceCount[TypeIds\_Int16Array]);

Output::Print(L" Int16Array %8d %8d\n", typeCount[TypeIds\_Uint16Array], instanceCount[TypeIds\_Uint16Array]);

Output::Print(L" Int32Array %8d %8d\n", typeCount[TypeIds\_Int32Array], instanceCount[TypeIds\_Int32Array]);

Output::Print(L" Uint32Array %8d %8d\n", typeCount[TypeIds\_Uint32Array], instanceCount[TypeIds\_Uint32Array]);

Output::Print(L" Float32Array %8d %8d\n", typeCount[TypeIds\_Float32Array], instanceCount[TypeIds\_Float32Array]);

Output::Print(L" Float64Array %8d %8d\n", typeCount[TypeIds\_Float64Array], instanceCount[TypeIds\_Float64Array]);

Output::Print(L" DataView %8d %8d\n", typeCount[TypeIds\_DataView], instanceCount[TypeIds\_DataView]);

Output::Print(L" ModuleRoot %8d %8d\n", typeCount[TypeIds\_ModuleRoot], instanceCount[TypeIds\_ModuleRoot]);

Output::Print(L" HostObject %8d %8d\n", typeCount[TypeIds\_HostObject], instanceCount[TypeIds\_HostObject]);

Output::Print(L" VariantDate %8d %8d\n", typeCount[TypeIds\_VariantDate], instanceCount[TypeIds\_VariantDate]);

Output::Print(L" HostDispatch %8d %8d\n", typeCount[TypeIds\_HostDispatch], instanceCount[TypeIds\_HostDispatch]);

Output::Print(L" Arguments %8d %8d\n", typeCount[TypeIds\_Arguments], instanceCount[TypeIds\_Arguments]);

Output::Print(L" ActivationObject %8d %8d\n", typeCount[TypeIds\_ActivationObject], instanceCount[TypeIds\_ActivationObject]);

Output::Print(L" Map %8d %8d\n", typeCount[TypeIds\_Map], instanceCount[TypeIds\_Map]);

Output::Print(L" Set %8d %8d\n", typeCount[TypeIds\_Set], instanceCount[TypeIds\_Set]);

Output::Print(L" WeakMap %8d %8d\n", typeCount[TypeIds\_WeakMap], instanceCount[TypeIds\_WeakMap]);

Output::Print(L" WeakSet %8d %8d\n", typeCount[TypeIds\_WeakSet], instanceCount[TypeIds\_WeakSet]);

Output::Print(L" ArrayIterator %8d %8d\n", typeCount[TypeIds\_ArrayIterator], instanceCount[TypeIds\_ArrayIterator]);

Output::Print(L" MapIterator %8d %8d\n", typeCount[TypeIds\_MapIterator], instanceCount[TypeIds\_MapIterator]);

Output::Print(L" SetIterator %8d %8d\n", typeCount[TypeIds\_SetIterator], instanceCount[TypeIds\_SetIterator]);

Output::Print(L" StringIterator %8d %8d\n", typeCount[TypeIds\_StringIterator], instanceCount[TypeIds\_StringIterator]);

Output::Print(L" Generator %8d %8d\n", typeCount[TypeIds\_Generator], instanceCount[TypeIds\_Generator]);

#if !DBG

Output::Print(L" \*\* Instance statistics only available on debug builds...\n");

#endif

Output::Flush();

}

#endif

#ifdef PROFILE\_OBJECT\_LITERALS

void ScriptContext::ProfileObjectLiteral()

{

Output::Print(L"===============================================================================\n");

Output::Print(L" Object Lit Instances created.. %d\n", objectLiteralInstanceCount);

Output::Print(L" Object Lit Path Types......... %d\n", objectLiteralPathCount);

Output::Print(L" Object Lit Simple Map......... %d\n", objectLiteralSimpleDictionaryCount);

Output::Print(L" Object Lit Max # of properties %d\n", objectLiteralMaxLength);

Output::Print(L" Object Lit Promote count...... %d\n", objectLiteralPromoteCount);

Output::Print(L" Object Lit Cache Hits......... %d\n", objectLiteralCacheCount);

Output::Print(L" Object Lit Branch count....... %d\n", objectLiteralBranchCount);

for (int i = 0; i < TypePath::MaxPathTypeHandlerLength; i++)

{

if (objectLiteralCount[i] != 0)

{

Output::Print(L" Object Lit properties [ %2d] .. %d\n", i, objectLiteralCount[i]);

}

}

Output::Flush();

}

#endif

//

// Regex helpers

//

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

UnifiedRegex::RegexStatsDatabase\* ScriptContext::GetRegexStatsDatabase()

{

if (regexStatsDatabase == 0)

{

ArenaAllocator\* allocator = MiscAllocator();

regexStatsDatabase = Anew(allocator, UnifiedRegex::RegexStatsDatabase, allocator);

}

return regexStatsDatabase;

}

UnifiedRegex::DebugWriter\* ScriptContext::GetRegexDebugWriter()

{

if (regexDebugWriter == 0)

{

ArenaAllocator\* allocator = MiscAllocator();

regexDebugWriter = Anew(allocator, UnifiedRegex::DebugWriter);

}

return regexDebugWriter;

}

#endif

bool ScriptContext::DoUndeferGlobalFunctions() const

{

return CONFIG\_FLAG(DeferTopLevelTillFirstCall) && !AutoSystemInfo::Data.IsLowMemoryProcess();

}

RegexPatternMruMap\* ScriptContext::GetDynamicRegexMap() const

{

Assert(!isScriptContextActuallyClosed);

Assert(guestArena);

Assert(cache);

Assert(cache->dynamicRegexMap);

return cache->dynamicRegexMap;

}

void ScriptContext::SetTrigramAlphabet(UnifiedRegex::TrigramAlphabet \* trigramAlphabet)

{

this->trigramAlphabet = trigramAlphabet;

}

UnifiedRegex::RegexStacks \*ScriptContext::RegexStacks()

{

UnifiedRegex::RegexStacks \* stacks = regexStacks;

if (stacks)

{

return stacks;

}

return AllocRegexStacks();

}

UnifiedRegex::RegexStacks \* ScriptContext::AllocRegexStacks()

{

Assert(this->regexStacks == nullptr);

UnifiedRegex::RegexStacks \* stacks = Anew(RegexAllocator(), UnifiedRegex::RegexStacks, threadContext->GetPageAllocator());

this->regexStacks = stacks;

return stacks;

}

UnifiedRegex::RegexStacks \*ScriptContext::SaveRegexStacks()

{

Assert(regexStacks);

const auto saved = regexStacks;

regexStacks = nullptr;

return saved;

}

void ScriptContext::RestoreRegexStacks(UnifiedRegex::RegexStacks \*const stacks)

{

Assert(stacks);

Assert(stacks != regexStacks);

if (regexStacks)

{

Adelete(RegexAllocator(), regexStacks);

}

regexStacks = stacks;

}

Js::TempArenaAllocatorObject\* ScriptContext::GetTemporaryAllocator(LPCWSTR name)

{

return this->threadContext->GetTemporaryAllocator(name);

}

void ScriptContext::ReleaseTemporaryAllocator(Js::TempArenaAllocatorObject\* tempAllocator)

{

AssertMsg(tempAllocator != nullptr, "tempAllocator should not be null");

this->threadContext->ReleaseTemporaryAllocator(tempAllocator);

}

Js::TempGuestArenaAllocatorObject\* ScriptContext::GetTemporaryGuestAllocator(LPCWSTR name)

{

return this->threadContext->GetTemporaryGuestAllocator(name);

}

void ScriptContext::ReleaseTemporaryGuestAllocator(Js::TempGuestArenaAllocatorObject\* tempGuestAllocator)

{

AssertMsg(tempGuestAllocator != nullptr, "tempAllocator should not be null");

this->threadContext->ReleaseTemporaryGuestAllocator(tempGuestAllocator);

}

void ScriptContext::InitializePreGlobal()

{

this->guestArena = this->GetRecycler()->CreateGuestArena(L"Guest", Throw::OutOfMemory);

#if ENABLE\_PROFILE\_INFO

#if DBG\_DUMP || defined(DYNAMIC\_PROFILE\_STORAGE) || defined(RUNTIME\_DATA\_COLLECTION)

if (DynamicProfileInfo::NeedProfileInfoList())

{

this->profileInfoList.Root(RecyclerNew(this->GetRecycler(), SListBase<DynamicProfileInfo \*>), recycler);

}

#endif

#endif

{

AutoCriticalSection critSec(this->threadContext->GetEtwRundownCriticalSection());

this->cache = AnewStructZ(guestArena, Cache);

}

this->cache->rootPath = TypePath::New(recycler);

this->cache->dynamicRegexMap =

RegexPatternMruMap::New(

recycler,

REGEX\_CONFIG\_FLAG(DynamicRegexMruListSize) <= 0 ? 16 : REGEX\_CONFIG\_FLAG(DynamicRegexMruListSize));

SourceContextInfo\* sourceContextInfo = RecyclerNewStructZ(this->GetRecycler(), SourceContextInfo);

sourceContextInfo->dwHostSourceContext = Js::Constants::NoHostSourceContext;

sourceContextInfo->isHostDynamicDocument = false;

sourceContextInfo->sourceContextId = Js::Constants::NoSourceContext;

this->cache->noContextSourceContextInfo = sourceContextInfo;

SRCINFO\* srcInfo = RecyclerNewStructZ(this->GetRecycler(), SRCINFO);

srcInfo->sourceContextInfo = this->cache->noContextSourceContextInfo;

srcInfo->moduleID = kmodGlobal;

this->cache->noContextGlobalSourceInfo = srcInfo;

#if ENABLE\_BACKGROUND\_PARSING

if (PHASE\_ON1(Js::ParallelParsePhase))

{

this->backgroundParser = BackgroundParser::New(this);

}

#endif

#if ENABLE\_NATIVE\_CODEGEN

// Create the native code gen before the profiler

this->nativeCodeGen = NewNativeCodeGenerator(this);

#endif

#ifdef PROFILE\_EXEC

this->CreateProfiler();

#endif

#ifdef FIELD\_ACCESS\_STATS

this->fieldAccessStatsByFunctionNumber = RecyclerNew(this->recycler, FieldAccessStatsByFunctionNumberMap, recycler);

BindReference(this->fieldAccessStatsByFunctionNumber);

#endif

this->operationStack = Anew(GeneralAllocator(), JsUtil::Stack<Var>, GeneralAllocator());

this->GetDebugContext()->Initialize();

Tick::InitType();

}

void ScriptContext::Initialize()

{

SmartFPUControl defaultControl;

InitializePreGlobal();

InitializeGlobalObject();

InitializePostGlobal();

}

void ScriptContext::InitializePostGlobal()

{

this->GetDebugContext()->GetProbeContainer()->Initialize(this);

AssertMsg(this->CurrentThunk == DefaultEntryThunk, "Creating non default thunk while initializing");

AssertMsg(this->DeferredParsingThunk == DefaultDeferredParsingThunk, "Creating non default thunk while initializing");

AssertMsg(this->DeferredDeserializationThunk == DefaultDeferredDeserializeThunk, "Creating non default thunk while initializing");

if (!sourceList)

{

AutoCriticalSection critSec(threadContext->GetEtwRundownCriticalSection());

sourceList.Root(RecyclerNew(this->GetRecycler(), SourceList, this->GetRecycler()), this->GetRecycler());

}

#if DYNAMIC\_INTERPRETER\_THUNK

interpreterThunkEmitter = HeapNew(InterpreterThunkEmitter, this->GetThreadContext()->GetAllocationPolicyManager(),

SourceCodeAllocator(), Js::InterpreterStackFrame::InterpreterThunk);

#endif

#ifdef ASMJS\_PLAT

asmJsInterpreterThunkEmitter = HeapNew(InterpreterThunkEmitter, this->GetThreadContext()->GetAllocationPolicyManager(),

SourceCodeAllocator(), Js::InterpreterStackFrame::InterpreterAsmThunk);

#endif

JS\_ETW(EtwTrace::LogScriptContextLoadEvent(this));

JS\_ETW(EventWriteJSCRIPT\_HOST\_SCRIPT\_CONTEXT\_START(this));

#ifdef PROFILE\_EXEC

if (profiler != nullptr)

{

this->threadContext->GetRecycler()->SetProfiler(profiler->GetProfiler(), profiler->GetBackgroundRecyclerProfiler());

}

#endif

#if DBG

this->javascriptLibrary->DumpLibraryByteCode();

isInitialized = TRUE;

#endif

}

#ifdef ASMJS\_PLAT

AsmJsCodeGenerator\* ScriptContext::InitAsmJsCodeGenerator()

{

if( !asmJsCodeGenerator )

{

asmJsCodeGenerator = HeapNew( AsmJsCodeGenerator, this );

}

return asmJsCodeGenerator;

}

#endif

void ScriptContext::MarkForClose()

{

SaveStartupProfileAndRelease(true);

this->isClosed = true;

#ifdef LEAK\_REPORT

if (this->isRootTrackerScriptContext)

{

this->GetThreadContext()->ClearRootTrackerScriptContext(this);

}

#endif

if (!threadContext->IsInScript())

{

Close(FALSE);

}

else

{

threadContext->AddToPendingScriptContextCloseList(this);

}

}

void ScriptContext::InitializeGlobalObject()

{

GlobalObject \* localGlobalObject = GlobalObject::New(this);

GetRecycler()->RootAddRef(localGlobalObject);

// Assigned the global Object after we have successfully AddRef (in case of OOM)

globalObject = localGlobalObject;

globalObject->Initialize(this);

}

ArenaAllocator\* ScriptContext::AllocatorForDiagnostics()

{

if (this->diagnosticArena == nullptr)

{

this->diagnosticArena = HeapNew(ArenaAllocator, L"Diagnostic", this->GetThreadContext()->GetDebugManager()->GetDiagnosticPageAllocator(), Throw::OutOfMemory);

}

Assert(this->diagnosticArena != nullptr);

return this->diagnosticArena;

}

void ScriptContext::PushObject(Var object)

{

operationStack->Push(object);

}

Var ScriptContext::PopObject()

{

return operationStack->Pop();

}

BOOL ScriptContext::CheckObject(Var object)

{

return operationStack->Contains(object);

}

void ScriptContext::SetHostScriptContext(HostScriptContext \* hostScriptContext)

{

Assert(this->hostScriptContext == nullptr);

this->hostScriptContext = hostScriptContext;

#ifdef PROFILE\_EXEC

this->ensureParentInfo = true;

#endif

}

//

// Enables chakradiag to get the HaltCallBack pointer that is implemented by

// the ScriptEngine.

//

void ScriptContext::SetScriptEngineHaltCallback(HaltCallback\* scriptEngine)

{

Assert(this->scriptEngineHaltCallback == NULL);

Assert(scriptEngine != NULL);

this->scriptEngineHaltCallback = scriptEngine;

}

void ScriptContext::ClearHostScriptContext()

{

if (this->hostScriptContext != nullptr)

{

this->hostScriptContext->Delete();

#ifdef PROFILE\_EXEC

this->ensureParentInfo = false;

#endif

}

}

IActiveScriptProfilerHeapEnum\* ScriptContext::GetHeapEnum()

{

Assert(this->GetThreadContext());

return this->GetThreadContext()->GetHeapEnum();

}

void ScriptContext::SetHeapEnum(IActiveScriptProfilerHeapEnum\* newHeapEnum)

{

Assert(this->GetThreadContext());

this->GetThreadContext()->SetHeapEnum(newHeapEnum);

}

void ScriptContext::ClearHeapEnum()

{

Assert(this->GetThreadContext());

this->GetThreadContext()->ClearHeapEnum();

}

BOOL ScriptContext::VerifyAlive(BOOL isJSFunction, ScriptContext\* requestScriptContext)

{

if (isClosed)

{

if (!requestScriptContext)

{

requestScriptContext = this;

}

#if ENABLE\_PROFILE\_INFO

if (!GetThreadContext()->RecordImplicitException())

{

return FALSE;

}

#endif

if (isJSFunction)

{

Js::JavascriptError::MapAndThrowError(requestScriptContext, JSERR\_CantExecute);

}

else

{

Js::JavascriptError::MapAndThrowError(requestScriptContext, E\_ACCESSDENIED);

}

}

return TRUE;

}

void ScriptContext::VerifyAliveWithHostContext(BOOL isJSFunction, HostScriptContext\* requestHostScriptContext)

{

if (requestHostScriptContext)

{

VerifyAlive(isJSFunction, requestHostScriptContext->GetScriptContext());

}

else

{

Assert(!GetThreadContext()->GetIsThreadBound() || !GetHostScriptContext()->HasCaller());

VerifyAlive(isJSFunction, NULL);

}

}

PropertyRecord const \* ScriptContext::GetPropertyName(PropertyId propertyId)

{

return threadContext->GetPropertyName(propertyId);

}

PropertyRecord const \* ScriptContext::GetPropertyNameLocked(PropertyId propertyId)

{

return threadContext->GetPropertyNameLocked(propertyId);

}

void ScriptContext::InitPropertyStringMap(int i)

{

propertyStrings[i] = AnewStruct(GeneralAllocator(), PropertyStringMap);

memset(propertyStrings[i]->strLen2, 0, sizeof(PropertyString\*)\* 80);

}

void ScriptContext::TrackPid(const PropertyRecord\* propertyRecord)

{

if (IsBuiltInPropertyId(propertyRecord->GetPropertyId()) || propertyRecord->IsBound())

{

return;

}

if (-1 != this->GetLibrary()->EnsureReferencedPropertyRecordList()->AddNew(propertyRecord))

{

RECYCLER\_PERF\_COUNTER\_INC(PropertyRecordBindReference);

}

}

void ScriptContext::TrackPid(PropertyId propertyId)

{

if (IsBuiltInPropertyId(propertyId))

{

return;

}

const PropertyRecord\* propertyRecord = this->GetPropertyName(propertyId);

Assert(propertyRecord != nullptr);

this->TrackPid(propertyRecord);

}

bool ScriptContext::IsTrackedPropertyId(Js::PropertyId propertyId)

{

if (IsBuiltInPropertyId(propertyId))

{

return true;

}

const PropertyRecord\* propertyRecord = this->GetPropertyName(propertyId);

Assert(propertyRecord != nullptr);

if (propertyRecord->IsBound())

{

return true;

}

JavascriptLibrary::ReferencedPropertyRecordHashSet \* referencedPropertyRecords

= this->GetLibrary()->GetReferencedPropertyRecordList();

return referencedPropertyRecords && referencedPropertyRecords->Contains(propertyRecord);

}

PropertyString\* ScriptContext::AddPropertyString2(const Js::PropertyRecord\* propString)

{

const wchar\_t\* buf = propString->GetBuffer();

const uint i = PropertyStringMap::PStrMapIndex(buf[0]);

if (propertyStrings[i] == NULL)

{

InitPropertyStringMap(i);

}

const uint j = PropertyStringMap::PStrMapIndex(buf[1]);

if (propertyStrings[i]->strLen2[j] == NULL && !isClosed)

{

propertyStrings[i]->strLen2[j] = GetLibrary()->CreatePropertyString(propString, this->GeneralAllocator());

this->TrackPid(propString);

}

return propertyStrings[i]->strLen2[j];

}

PropertyString\* ScriptContext::CachePropertyString2(const PropertyRecord\* propString)

{

Assert(propString->GetLength() == 2);

const wchar\_t\* propertyName = propString->GetBuffer();

if ((propertyName[0] <= 'z') && (propertyName[1] <= 'z') && (propertyName[0] >= '0') && (propertyName[1] >= '0') && ((propertyName[0] > '9') || (propertyName[1] > '9')))

{

return AddPropertyString2(propString);

}

return NULL;

}

PropertyString\* ScriptContext::GetPropertyString(PropertyId propertyId)

{

PropertyStringCacheMap\* propertyStringMap = this->GetLibrary()->EnsurePropertyStringMap();

PropertyString \*string;

RecyclerWeakReference<PropertyString>\* stringReference;

if (propertyStringMap->TryGetValue(propertyId, &stringReference))

{

string = stringReference->Get();

if (string != nullptr)

{

return string;

}

}

const Js::PropertyRecord\* propertyName = this->GetPropertyName(propertyId);

string = this->GetLibrary()->CreatePropertyString(propertyName);

propertyStringMap->Item(propertyId, recycler->CreateWeakReferenceHandle(string));

return string;

}

void ScriptContext::InvalidatePropertyStringCache(PropertyId propertyId, Type\* type)

{

PropertyStringCacheMap\* propertyStringMap = this->javascriptLibrary->GetPropertyStringMap();

if (propertyStringMap != nullptr)

{

PropertyString \*string = nullptr;

RecyclerWeakReference<PropertyString>\* stringReference;

if (propertyStringMap->TryGetValue(propertyId, &stringReference))

{

string = stringReference->Get();

}

if (string)

{

PropertyCache const\* cache = string->GetPropertyCache();

if (cache->type == type)

{

string->ClearPropertyCache();

}

}

}

}

void ScriptContext::CleanupWeakReferenceDictionaries()

{

if (!isWeakReferenceDictionaryListCleared)

{

SListBase<JsUtil::IWeakReferenceDictionary\*>::Iterator iter(&this->weakReferenceDictionaryList);

while (iter.Next())

{

JsUtil::IWeakReferenceDictionary\* weakReferenceDictionary = iter.Data();

weakReferenceDictionary->Cleanup();

}

}

}

JavascriptString\* ScriptContext::GetIntegerString(Var aValue)

{

return this->GetIntegerString(TaggedInt::ToInt32(aValue));

}

JavascriptString\* ScriptContext::GetIntegerString(uint value)

{

if (value <= INT\_MAX)

{

return this->GetIntegerString((int)value);

}

return TaggedInt::ToString(value, this);

}

JavascriptString\* ScriptContext::GetIntegerString(int value)

{

// Optimize for 0-9

if (0 <= value && value <= 9)

{

return GetLibrary()->GetCharStringCache().GetStringForCharA('0' + static\_cast<char>(value));

}

JavascriptString \*string;

if (!this->integerStringMap.TryGetValue(value, &string))

{

// Add the string to hash table cache

// Don't add if table is getting too full. We'll be holding on to

// too many strings, and table lookup will become too slow.

if (this->integerStringMap.Count() > 1024)

{

// Use recycler memory

string = TaggedInt::ToString(value, this);

}

else

{

wchar\_t stringBuffer[20];

TaggedInt::ToBuffer(value, stringBuffer, \_countof(stringBuffer));

string = JavascriptString::NewCopySzFromArena(stringBuffer, this, this->GeneralAllocator());

this->integerStringMap.AddNew(value, string);

}

}

return string;

}

void ScriptContext::CheckEvalRestriction()

{

HRESULT hr = S\_OK;

Var domError = nullptr;

HostScriptContext\* hostScriptContext = this->GetHostScriptContext();

BEGIN\_LEAVE\_SCRIPT(this)

{

if (!FAILED(hr = hostScriptContext->CheckEvalRestriction()))

{

return;

}

hr = hostScriptContext->HostExceptionFromHRESULT(hr, &domError);

}

END\_LEAVE\_SCRIPT(this);

if (FAILED(hr))

{

Js::JavascriptError::MapAndThrowError(this, hr);

}

if (domError != nullptr)

{

JavascriptExceptionOperators::Throw(domError, this);

}

AssertMsg(false, "We should have thrown by now.");

Js::JavascriptError::MapAndThrowError(this, E\_FAIL);

}

JavascriptFunction\* ScriptContext::LoadScript(const wchar\_t\* script, SRCINFO const \* pSrcInfo, CompileScriptException \* pse, bool isExpression, bool disableDeferredParse, bool isByteCodeBufferForLibrary, Utf8SourceInfo\*\* ppSourceInfo, const wchar\_t \*rootDisplayName, bool disableAsmJs)

{

if (pSrcInfo == nullptr)

{

pSrcInfo = this->cache->noContextGlobalSourceInfo;

}

Assert(!this->threadContext->IsScriptActive());

Assert(pse != nullptr);

try

{

AUTO\_NESTED\_HANDLED\_EXCEPTION\_TYPE((ExceptionType)(ExceptionType\_OutOfMemory | ExceptionType\_StackOverflow));

Js::AutoDynamicCodeReference dynamicFunctionReference(this);

// Convert to UTF8 and then load that

size\_t length = wcslen(script);

if (!IsValidCharCount(length))

{

Js::Throw::OutOfMemory();

}

// Allocate memory for the UTF8 output buffer.

// We need at most 3 bytes for each Unicode code point.

// The + 1 is to include the terminating NUL.

// Nit: Technically, we know that the NUL only needs 1 byte instead of

// 3, but that's difficult to express in a SAL annotation for "EncodeInto".

size\_t cbUtf8Buffer = AllocSizeMath::Mul(AllocSizeMath::Add(length , 1), 3);

LPUTF8 utf8Script = RecyclerNewArrayLeafTrace(this->GetRecycler(), utf8char\_t, cbUtf8Buffer);

size\_t cbNeeded = utf8::EncodeIntoAndNullTerminate(utf8Script, script, static\_cast<charcount\_t>(length));

#if DBG\_DUMP

if (Js::Configuration::Global.flags.TraceMemory.IsEnabled(Js::ParsePhase) && Configuration::Global.flags.Verbose)

{

Output::Print(L"Loading script.\n"

L" Unicode (in bytes) %u\n"

L" UTF-8 size (in bytes) %u\n"

L" Expected savings %d\n", length \* sizeof(wchar\_t), cbNeeded, length \* sizeof(wchar\_t)-cbNeeded);

}

#endif

// Free unused bytes

Assert(cbNeeded + 1 <= cbUtf8Buffer);

\*ppSourceInfo = Utf8SourceInfo::New(this, utf8Script, (int)length, cbNeeded, pSrcInfo);

//

// Parse and execute the JavaScript file.

//

HRESULT hr;

Parser parser(this);

SourceContextInfo \* sourceContextInfo = pSrcInfo->sourceContextInfo;

// Invoke the parser, passing in the global function name, which we will then run to execute

// the script.

// This is global function called from jc or scriptengine::parse, in both case we can return the value to the caller.

ULONG grfscr = fscrGlobalCode | (isExpression ? fscrReturnExpression : 0);

if (!disableDeferredParse && (length > Parser::GetDeferralThreshold(sourceContextInfo->IsSourceProfileLoaded())))

{

grfscr |= fscrDeferFncParse;

}

if (disableAsmJs)

{

grfscr |= fscrNoAsmJs;

}

if (PHASE\_FORCE1(Js::EvalCompilePhase))

{

// pretend it is eval

grfscr |= (fscrEval | fscrEvalCode);

}

if (isByteCodeBufferForLibrary)

{

grfscr |= (fscrNoAsmJs | fscrNoPreJit);

}

ParseNodePtr parseTree;

hr = parser.ParseCesu8Source(&parseTree, utf8Script, cbNeeded, grfscr, pse, &sourceContextInfo->nextLocalFunctionId,

sourceContextInfo);

(\*ppSourceInfo)->SetParseFlags(grfscr);

if (FAILED(hr) || parseTree == nullptr)

{

return nullptr;

}

Assert(length < MAXLONG);

uint sourceIndex = this->SaveSourceNoCopy(\*ppSourceInfo, static\_cast<charcount\_t>(length), /\*isCesu8\*/ true);

JavascriptFunction \* pFunction = GenerateRootFunction(parseTree, sourceIndex, &parser, grfscr, pse, rootDisplayName);

if (pse->ei.scode == JSERR\_AsmJsCompileError)

{

Assert(!disableAsmJs);

pse->Clear();

return LoadScript(script, pSrcInfo, pse, isExpression, disableDeferredParse, isByteCodeBufferForLibrary, ppSourceInfo, rootDisplayName, true);

}

if (pFunction != nullptr && this->IsProfiling())

{

RegisterScript(pFunction->GetFunctionProxy());

}

return pFunction;

}

catch (Js::OutOfMemoryException)

{

pse->ProcessError(nullptr, E\_OUTOFMEMORY, nullptr);

return nullptr;

}

catch (Js::StackOverflowException)

{

pse->ProcessError(nullptr, VBSERR\_OutOfStack, nullptr);

return nullptr;

}

}

JavascriptFunction\* ScriptContext::LoadScript(LPCUTF8 script, size\_t cb, SRCINFO const \* pSrcInfo, CompileScriptException \* pse, bool isExpression, bool disableDeferredParse, bool isByteCodeBufferForLibrary, Utf8SourceInfo\*\* ppSourceInfo, const wchar\_t \*rootDisplayName, bool disableAsmJs)

{

if (pSrcInfo == nullptr)

{

pSrcInfo = this->cache->noContextGlobalSourceInfo;

}

Assert(!this->threadContext->IsScriptActive());

Assert(pse != nullptr);

try

{

AUTO\_HANDLED\_EXCEPTION\_TYPE((ExceptionType)(ExceptionType\_OutOfMemory | ExceptionType\_StackOverflow));

Js::AutoDynamicCodeReference dynamicFunctionReference(this);

//

// Parse and execute the JavaScript file.

//

HRESULT hr;

Parser parser(this);

SourceContextInfo \* sourceContextInfo = pSrcInfo->sourceContextInfo;

// Invoke the parser, passing in the global function name, which we will then run to execute

// the script.

ULONG grfscr = fscrGlobalCode | (isExpression ? fscrReturnExpression : 0);

if (!disableDeferredParse && (cb > Parser::GetDeferralThreshold(sourceContextInfo->IsSourceProfileLoaded())))

{

grfscr |= fscrDeferFncParse;

}

if (disableAsmJs)

{

grfscr |= fscrNoAsmJs;

}

if (PHASE\_FORCE1(Js::EvalCompilePhase))

{

// pretend it is eval

grfscr |= (fscrEval | fscrEvalCode);

}

if (isByteCodeBufferForLibrary)

{

grfscr |= (fscrNoAsmJs | fscrNoPreJit);

}

#if DBG\_DUMP

if (Js::Configuration::Global.flags.TraceMemory.IsEnabled(Js::ParsePhase) && Configuration::Global.flags.Verbose)

{

size\_t length = utf8::ByteIndexIntoCharacterIndex(script, cb, utf8::doAllowThreeByteSurrogates);

Output::Print(L"Direct UTF-8 parsing.\n"

L" Would have expanded into: %u (in bytes)\n"

L" UTF-8 size (in bytes): %u (in bytes)\n"

L" Expected savings: %d (in bytes)\n", length \* sizeof(wchar\_t), cb, length \* sizeof(wchar\_t)-cb);

}

#endif

ParseNodePtr parseTree;

hr = parser.ParseUtf8Source(&parseTree, script, cb, grfscr, pse, &sourceContextInfo->nextLocalFunctionId,

sourceContextInfo);

if (FAILED(hr) || parseTree == nullptr)

{

return nullptr;

}

// We do not own the memory passed into DefaultLoadScriptUtf8. We need to save it so we copy the memory.

\*ppSourceInfo = Utf8SourceInfo::New(this, script, parser.GetSourceIchLim(), cb, pSrcInfo);

(\*ppSourceInfo)->SetParseFlags(grfscr);

uint sourceIndex = this->SaveSourceNoCopy(\*ppSourceInfo, parser.GetSourceIchLim(), /\* isCesu8\*/ false);

JavascriptFunction \* pFunction = GenerateRootFunction(parseTree, sourceIndex, &parser, grfscr, pse, rootDisplayName);

if (pse->ei.scode == JSERR\_AsmJsCompileError)

{

Assert(!disableAsmJs);

pse->Clear();

return LoadScript(script, cb, pSrcInfo, pse, isExpression, disableDeferredParse, isByteCodeBufferForLibrary, ppSourceInfo, rootDisplayName, true);

}

if (pFunction != nullptr && this->IsProfiling())

{

RegisterScript(pFunction->GetFunctionProxy());

}

return pFunction;

}

catch (Js::OutOfMemoryException)

{

pse->ProcessError(nullptr, E\_OUTOFMEMORY, nullptr);

return nullptr;

}

catch (Js::StackOverflowException)

{

pse->ProcessError(nullptr, VBSERR\_OutOfStack, nullptr);

return nullptr;

}

}

JavascriptFunction\* ScriptContext::GenerateRootFunction(ParseNodePtr parseTree, uint sourceIndex, Parser\* parser, ulong grfscr, CompileScriptException \* pse, const wchar\_t \*rootDisplayName)

{

HRESULT hr;

// Get the source code to keep it alive during the bytecode generation process

LPCUTF8 source = this->GetSource(sourceIndex)->GetSource(L"ScriptContext::GenerateRootFunction");

Assert(source != nullptr); // Source should not have been reclaimed by now

// Generate bytecode and native code

ParseableFunctionInfo\* body = NULL;

hr = GenerateByteCode(parseTree, grfscr, this, &body, sourceIndex, false, parser, pse);

this->GetSource(sourceIndex)->SetByteCodeGenerationFlags(grfscr);

if (FAILED(hr))

{

return nullptr;

}

body->SetDisplayName(rootDisplayName);

body->SetIsTopLevel(true);

JavascriptFunction\* rootFunction = javascriptLibrary->CreateScriptFunction(body);

return rootFunction;

}

BOOL ScriptContext::ReserveStaticTypeIds(\_\_in int first, \_\_in int last)

{

return threadContext->ReserveStaticTypeIds(first, last);

}

TypeId ScriptContext::ReserveTypeIds(int count)

{

return threadContext->ReserveTypeIds(count);

}

TypeId ScriptContext::CreateTypeId()

{

return threadContext->CreateTypeId();

}

void ScriptContext::OnScriptStart(bool isRoot, bool isScript)

{

const bool isForcedEnter = this->GetDebugContext() != nullptr ? this->GetDebugContext()->GetProbeContainer()->isForcedToEnterScriptStart : false;

if (this->scriptStartEventHandler != nullptr && ((isRoot && threadContext->GetCallRootLevel() == 1) || isForcedEnter))

{

if (this->GetDebugContext() != nullptr)

{

this->GetDebugContext()->GetProbeContainer()->isForcedToEnterScriptStart = false;

}

this->scriptStartEventHandler(this);

}

#if ENABLE\_NATIVE\_CODEGEN

//Blue 5491: Only start codegen if isScript. Avoid it if we are not really starting script and called from risky region such as catch handler.

if (isScript)

{

NativeCodeGenEnterScriptStart(this->GetNativeCodeGenerator());

}

#endif

}

void ScriptContext::OnScriptEnd(bool isRoot, bool isForcedEnd)

{

if ((isRoot && threadContext->GetCallRootLevel() == 1) || isForcedEnd)

{

if (this->scriptEndEventHandler != nullptr)

{

this->scriptEndEventHandler(this);

}

}

}

#ifdef FAULT\_INJECTION

void ScriptContext::DisposeScriptContextByFaultInjection() {

if (this->disposeScriptByFaultInjectionEventHandler != nullptr)

{

this->disposeScriptByFaultInjectionEventHandler(this);

}

}

#endif

template <bool stackProbe, bool leaveForHost>

bool ScriptContext::LeaveScriptStart(void \* frameAddress)

{

ThreadContext \* threadContext = this->threadContext;

if (!threadContext->IsScriptActive())

{

// we should have enter always.

AssertMsg(FALSE, "Leaving ScriptStart while script is not active.");

return false;

}

// Make sure the host function will have at least 32k of stack available.

if (stackProbe)

{

threadContext->ProbeStack(Js::Constants::MinStackCallout, this);

}

else

{

AssertMsg(ExceptionCheck::HasStackProbe(), "missing stack probe");

}

threadContext->LeaveScriptStart<leaveForHost>(frameAddress);

return true;

}

template <bool leaveForHost>

void ScriptContext::LeaveScriptEnd(void \* frameAddress)

{

this->threadContext->LeaveScriptEnd<leaveForHost>(frameAddress);

}

// explicit instantiations

template bool ScriptContext::LeaveScriptStart<true, true>(void \* frameAddress);

template bool ScriptContext::LeaveScriptStart<true, false>(void \* frameAddress);

template bool ScriptContext::LeaveScriptStart<false, true>(void \* frameAddress);

template void ScriptContext::LeaveScriptEnd<true>(void \* frameAddress);

template void ScriptContext::LeaveScriptEnd<false>(void \* frameAddress);

bool ScriptContext::EnsureInterpreterArena(ArenaAllocator \*\*ppAlloc)

{

bool fNew = false;

if (this->interpreterArena == nullptr)

{

this->interpreterArena = this->GetRecycler()->CreateGuestArena(L"Interpreter", Throw::OutOfMemory);

fNew = true;

}

\*ppAlloc = this->interpreterArena;

return fNew;

}

void ScriptContext::ReleaseInterpreterArena()

{

AssertMsg(this->interpreterArena, "No interpreter arena to release");

if (this->interpreterArena)

{

this->GetRecycler()->DeleteGuestArena(this->interpreterArena);

this->interpreterArena = nullptr;

}

}

void ScriptContext::ReleaseGuestArena()

{

AssertMsg(this->guestArena, "No guest arena to release");

if (this->guestArena)

{

this->GetRecycler()->DeleteGuestArena(this->guestArena);

this->guestArena = nullptr;

}

}

void ScriptContext::SetScriptStartEventHandler(ScriptContext::EventHandler eventHandler)

{

AssertMsg(this->scriptStartEventHandler == nullptr, "Do not support multi-cast yet");

this->scriptStartEventHandler = eventHandler;

}

void ScriptContext::SetScriptEndEventHandler(ScriptContext::EventHandler eventHandler)

{

AssertMsg(this->scriptEndEventHandler == nullptr, "Do not support multi-cast yet");

this->scriptEndEventHandler = eventHandler;

}

#ifdef FAULT\_INJECTION

void ScriptContext::SetDisposeDisposeByFaultInjectionEventHandler(ScriptContext::EventHandler eventHandler)

{

AssertMsg(this->disposeScriptByFaultInjectionEventHandler == nullptr, "Do not support multi-cast yet");

this->disposeScriptByFaultInjectionEventHandler = eventHandler;

}

#endif

bool ScriptContext::SaveSourceCopy(Utf8SourceInfo\* const sourceInfo, int cchLength, bool isCesu8, uint \* index)

{

HRESULT hr = S\_OK;

BEGIN\_TRANSLATE\_OOM\_TO\_HRESULT

{

\*index = this->SaveSourceCopy(sourceInfo, cchLength, isCesu8);

}

END\_TRANSLATE\_OOM\_TO\_HRESULT(hr);

return hr == S\_OK;

}

uint ScriptContext::SaveSourceCopy(Utf8SourceInfo\* sourceInfo, int cchLength, bool isCesu8)

{

Utf8SourceInfo\* newSource = Utf8SourceInfo::Clone(this, sourceInfo);

return SaveSourceNoCopy(newSource, cchLength, isCesu8);

}

Utf8SourceInfo\* ScriptContext::CloneSourceCrossContext(Utf8SourceInfo\* crossContextSourceInfo, SRCINFO const\* srcInfo)

{

return Utf8SourceInfo::CloneNoCopy(this, crossContextSourceInfo, srcInfo);

}

uint ScriptContext::SaveSourceNoCopy(Utf8SourceInfo\* sourceInfo, int cchLength, bool isCesu8)

{

Assert(sourceInfo->GetScriptContext() == this);

if (this->IsInDebugMode() && sourceInfo->debugModeSource == nullptr && !sourceInfo->debugModeSourceIsEmpty)

{

sourceInfo->SetInDebugMode(true);

}

RecyclerWeakReference<Utf8SourceInfo>\* sourceWeakRef = this->GetRecycler()->CreateWeakReferenceHandle<Utf8SourceInfo>(sourceInfo);

sourceInfo->SetIsCesu8(isCesu8);

return sourceList->SetAtFirstFreeSpot(sourceWeakRef);

}

void ScriptContext::CloneSources(ScriptContext\* sourceContext)

{

sourceContext->sourceList->Map([=](int index, RecyclerWeakReference<Utf8SourceInfo>\* sourceInfo)

{

Utf8SourceInfo\* info = sourceInfo->Get();

if (info)

{

CloneSource(info);

}

});

}

uint ScriptContext::CloneSource(Utf8SourceInfo\* info)

{

return this->SaveSourceCopy(info, info->GetCchLength(), info->GetIsCesu8());

}

Utf8SourceInfo\* ScriptContext::GetSource(uint index)

{

Assert(this->sourceList->IsItemValid(index)); // This assert should be a subset of info != null- if info was null, in the last collect, we'd have invalidated the item

Utf8SourceInfo\* info = this->sourceList->Item(index)->Get();

Assert(info != nullptr); // Should still be alive if this method is being called

return info;

}

bool ScriptContext::IsItemValidInSourceList(int index)

{

return (index < this->sourceList->Count()) && this->sourceList->IsItemValid(index);

}

void ScriptContext::RecordException(JavascriptExceptionObject \* exceptionObject, bool propagateToDebugger)

{

Assert(this->threadContext->GetRecordedException() == nullptr || GetThreadContext()->HasUnhandledException());

this->threadContext->SetRecordedException(exceptionObject, propagateToDebugger);

#if DBG

exceptionObject->FillStackBackTrace();

#endif

}

void ScriptContext::RethrowRecordedException(JavascriptExceptionObject::HostWrapperCreateFuncType hostWrapperCreateFunc)

{

bool considerPassingToDebugger = false;

JavascriptExceptionObject \* exceptionObject = this->GetAndClearRecordedException(&considerPassingToDebugger);

if (hostWrapperCreateFunc)

{

exceptionObject->SetHostWrapperCreateFunc(exceptionObject->GetScriptContext() != this ? hostWrapperCreateFunc : nullptr);

}

JavascriptExceptionOperators::RethrowExceptionObject(exceptionObject, this, considerPassingToDebugger);

}

Js::JavascriptExceptionObject \* ScriptContext::GetAndClearRecordedException(bool \*considerPassingToDebugger)

{

JavascriptExceptionObject \* exceptionObject = this->threadContext->GetRecordedException();

Assert(exceptionObject != nullptr);

if (considerPassingToDebugger)

{

\*considerPassingToDebugger = this->threadContext->GetPropagateException();

}

exceptionObject = exceptionObject->CloneIfStaticExceptionObject(this);

this->threadContext->SetRecordedException(nullptr);

return exceptionObject;

}

bool ScriptContext::IsInEvalMap(FastEvalMapString const& key, BOOL isIndirect, ScriptFunction \*\*ppFuncScript)

{

EvalCacheDictionary \*dict = isIndirect ? this->cache->indirectEvalCacheDictionary : this->cache->evalCacheDictionary;

if (dict == nullptr)

{

return false;

}

#ifdef PROFILE\_EVALMAP

if (Configuration::Global.flags.ProfileEvalMap)

{

charcount\_t len = key.str.GetLength();

if (dict->TryGetValue(key, ppFuncScript))

{

Output::Print(L"EvalMap cache hit:\t source size = %d\n", len);

}

else

{

Output::Print(L"EvalMap cache miss:\t source size = %d\n", len);

}

}

#endif

// If eval map cleanup is false, to preserve existing behavior, add it to the eval map MRU list

bool success = dict->TryGetValue(key, ppFuncScript);

if (success)

{

dict->NotifyAdd(key);

#ifdef VERBOSE\_EVAL\_MAP

#if DBG

dict->DumpKeepAlives();

#endif

#endif

}

return success;

}

void ScriptContext::BeginDynamicFunctionReferences()

{

if (this->dynamicFunctionReference == nullptr)

{

this->dynamicFunctionReference = RecyclerNew(this->recycler, FunctionReferenceList, this->recycler);

this->BindReference(this->dynamicFunctionReference);

this->dynamicFunctionReferenceDepth = 0;

}

this->dynamicFunctionReferenceDepth++;

}

void ScriptContext::EndDynamicFunctionReferences()

{

Assert(this->dynamicFunctionReference != nullptr);

this->dynamicFunctionReferenceDepth--;

if (this->dynamicFunctionReferenceDepth == 0)

{

this->dynamicFunctionReference->Clear();

}

}

void ScriptContext::RegisterDynamicFunctionReference(FunctionProxy\* func)

{

Assert(this->dynamicFunctionReferenceDepth > 0);

this->dynamicFunctionReference->Push(func);

}

void ScriptContext::AddToEvalMap(FastEvalMapString const& key, BOOL isIndirect, ScriptFunction \*pFuncScript)

{

EvalCacheDictionary \*dict = isIndirect ? this->cache->indirectEvalCacheDictionary : this->cache->evalCacheDictionary;

if (dict == nullptr)

{

EvalCacheTopLevelDictionary\* evalTopDictionary = RecyclerNew(this->recycler, EvalCacheTopLevelDictionary, this->recycler);

dict = RecyclerNew(this->recycler, EvalCacheDictionary, evalTopDictionary, recycler);

if (isIndirect)

{

this->cache->indirectEvalCacheDictionary = dict;

}

else

{

this->cache->evalCacheDictionary = dict;

}

}

dict->Add(key, pFuncScript);

}

bool ScriptContext::IsInNewFunctionMap(EvalMapString const& key, ParseableFunctionInfo \*\*ppFuncBody)

{

if (this->cache->newFunctionCache == nullptr)

{

return false;

}

// If eval map cleanup is false, to preserve existing behavior, add it to the eval map MRU list

bool success = this->cache->newFunctionCache->TryGetValue(key, ppFuncBody);

if (success)

{

this->cache->newFunctionCache->NotifyAdd(key);

#ifdef VERBOSE\_EVAL\_MAP

#if DBG

this->cache->newFunctionCache->DumpKeepAlives();

#endif

#endif

}

return success;

}

void ScriptContext::AddToNewFunctionMap(EvalMapString const& key, ParseableFunctionInfo \*pFuncBody)

{

if (this->cache->newFunctionCache == nullptr)

{

this->cache->newFunctionCache = RecyclerNew(this->recycler, NewFunctionCache, this->recycler);

}

this->cache->newFunctionCache->Add(key, pFuncBody);

}

void ScriptContext::EnsureSourceContextInfoMap()

{

if (this->cache->sourceContextInfoMap == nullptr)

{

this->cache->sourceContextInfoMap = RecyclerNew(this->GetRecycler(), SourceContextInfoMap, this->GetRecycler());

}

}

void ScriptContext::EnsureDynamicSourceContextInfoMap()

{

if (this->cache->dynamicSourceContextInfoMap == nullptr)

{

this->cache->dynamicSourceContextInfoMap = RecyclerNew(this->GetRecycler(), DynamicSourceContextInfoMap, this->GetRecycler());

}

}

SourceContextInfo\* ScriptContext::GetSourceContextInfo(uint hash)

{

SourceContextInfo \* sourceContextInfo;

if (this->cache->dynamicSourceContextInfoMap && this->cache->dynamicSourceContextInfoMap->TryGetValue(hash, &sourceContextInfo))

{

return sourceContextInfo;

}

return nullptr;

}

SourceContextInfo\* ScriptContext::CreateSourceContextInfo(uint hash, DWORD\_PTR hostSourceContext)

{

EnsureDynamicSourceContextInfoMap();

if (this->GetSourceContextInfo(hash) != nullptr)

{

return const\_cast<SourceContextInfo\*>(this->cache->noContextSourceContextInfo);

}

if (this->cache->dynamicSourceContextInfoMap->Count() > INMEMORY\_CACHE\_MAX\_PROFILE\_MANAGER)

{

OUTPUT\_TRACE(Js::DynamicProfilePhase, L"Max of dynamic script profile info reached.\n");

return const\_cast<SourceContextInfo\*>(this->cache->noContextSourceContextInfo);

}

// This is capped so we can continue allocating in the arena

SourceContextInfo \* sourceContextInfo = RecyclerNewStructZ(this->GetRecycler(), SourceContextInfo);

sourceContextInfo->sourceContextId = this->GetNextSourceContextId();

sourceContextInfo->dwHostSourceContext = hostSourceContext;

sourceContextInfo->isHostDynamicDocument = true;

sourceContextInfo->hash = hash;

#if ENABLE\_PROFILE\_INFO

sourceContextInfo->sourceDynamicProfileManager = this->threadContext->GetSourceDynamicProfileManager(this->GetUrl(), hash, &referencesSharedDynamicSourceContextInfo);

#endif

// For the host provided dynamic code (if hostSourceContext is not NoHostSourceContext), do not add to dynamicSourceContextInfoMap

if (hostSourceContext == Js::Constants::NoHostSourceContext)

{

this->cache->dynamicSourceContextInfoMap->Add(hash, sourceContextInfo);

}

return sourceContextInfo;

}

//

// Makes a copy of the URL to be stored in the map.

//

SourceContextInfo \* ScriptContext::CreateSourceContextInfo(DWORD\_PTR sourceContext, wchar\_t const \* url, size\_t len,

IActiveScriptDataCache\* profileDataCache, wchar\_t const \* sourceMapUrl /\*= NULL\*/, size\_t sourceMapUrlLen /\*= 0\*/)

{

// Take etw rundown lock on this thread context. We are going to init/add to sourceContextInfoMap.

AutoCriticalSection autocs(GetThreadContext()->GetEtwRundownCriticalSection());

EnsureSourceContextInfoMap();

Assert(this->GetSourceContextInfo(sourceContext, profileDataCache) == nullptr);

SourceContextInfo \* sourceContextInfo = RecyclerNewStructZ(this->GetRecycler(), SourceContextInfo);

sourceContextInfo->sourceContextId = this->GetNextSourceContextId();

sourceContextInfo->dwHostSourceContext = sourceContext;

sourceContextInfo->isHostDynamicDocument = false;

#if ENABLE\_PROFILE\_INFO

sourceContextInfo->sourceDynamicProfileManager = nullptr;

#endif

if (url != nullptr)

{

sourceContextInfo->url = CopyString(url, len, this->SourceCodeAllocator());

JS\_ETW(EtwTrace::LogSourceModuleLoadEvent(this, sourceContext, url));

}

if (sourceMapUrl != nullptr && sourceMapUrlLen != 0)

{

sourceContextInfo->sourceMapUrl = CopyString(sourceMapUrl, sourceMapUrlLen, this->SourceCodeAllocator());

}

#if ENABLE\_PROFILE\_INFO

if (!this->startupComplete)

{

sourceContextInfo->sourceDynamicProfileManager = SourceDynamicProfileManager::LoadFromDynamicProfileStorage(sourceContextInfo, this, profileDataCache);

Assert(sourceContextInfo->sourceDynamicProfileManager != NULL);

}

this->cache->sourceContextInfoMap->Add(sourceContext, sourceContextInfo);

#endif

return sourceContextInfo;

}

// static

const wchar\_t\* ScriptContext::CopyString(const wchar\_t\* str, size\_t charCount, ArenaAllocator\* alloc)

{

size\_t length = charCount + 1; // Add 1 for the NULL.

wchar\_t\* copy = AnewArray(alloc, wchar\_t, length);

js\_wmemcpy\_s(copy, length, str, charCount);

copy[length - 1] = L'\0';

return copy;

}

SourceContextInfo \* ScriptContext::GetSourceContextInfo(DWORD\_PTR sourceContext, IActiveScriptDataCache\* profileDataCache)

{

if (sourceContext == Js::Constants::NoHostSourceContext)

{

return const\_cast<SourceContextInfo\*>(this->cache->noContextSourceContextInfo);

}

// We only init sourceContextInfoMap, don't need to lock.

EnsureSourceContextInfoMap();

SourceContextInfo \* sourceContextInfo;

if (this->cache->sourceContextInfoMap->TryGetValue(sourceContext, &sourceContextInfo))

{

#if ENABLE\_PROFILE\_INFO

if (profileDataCache &&

sourceContextInfo->sourceDynamicProfileManager != nullptr &&

!sourceContextInfo->sourceDynamicProfileManager->IsProfileLoadedFromWinInet() &&

!this->startupComplete)

{

bool profileLoaded = sourceContextInfo->sourceDynamicProfileManager->LoadFromProfileCache(profileDataCache, sourceContextInfo->url);

if (profileLoaded)

{

JS\_ETW(EventWriteJSCRIPT\_PROFILE\_LOAD(sourceContextInfo->dwHostSourceContext, this));

}

}

#endif

return sourceContextInfo;

}

return nullptr;

}

SRCINFO const \*

ScriptContext::GetModuleSrcInfo(Js::ModuleID moduleID)

{

if (moduleSrcInfoCount <= moduleID)

{

uint newCount = moduleID + 4; // Preallocate 4 more slots, moduleID don't usually grow much

SRCINFO const \*\* newModuleSrcInfo = RecyclerNewArrayZ(this->GetRecycler(), SRCINFO const\*, newCount);

memcpy(newModuleSrcInfo, cache->moduleSrcInfo, sizeof(SRCINFO const \*)\* moduleSrcInfoCount);

cache->moduleSrcInfo = newModuleSrcInfo;

moduleSrcInfoCount = newCount;

cache->moduleSrcInfo[0] = this->cache->noContextGlobalSourceInfo;

}

SRCINFO const \* si = cache->moduleSrcInfo[moduleID];

if (si == nullptr)

{

SRCINFO \* newSrcInfo = RecyclerNewStructZ(this->GetRecycler(), SRCINFO);

newSrcInfo->sourceContextInfo = this->cache->noContextSourceContextInfo;

newSrcInfo->moduleID = moduleID;

cache->moduleSrcInfo[moduleID] = newSrcInfo;

si = newSrcInfo;

}

return si;

}

void ScriptContext::UpdateTimeZoneInfo()

{

GetTimeZoneInformation(&timeZoneInfo);

\_tzset();

}

#ifdef PROFILE\_EXEC

void

ScriptContext::DisableProfiler()

{

disableProfiler = true;

}

Profiler \*

ScriptContext::CreateProfiler()

{

Assert(profiler == nullptr);

if (Js::Configuration::Global.flags.IsEnabled(Js::ProfileFlag))

{

this->profiler = NoCheckHeapNew(ScriptContextProfiler);

this->profiler->Initialize(GetThreadContext()->GetPageAllocator(), threadContext->GetRecycler());

#if ENABLE\_NATIVE\_CODEGEN

CreateProfilerNativeCodeGen(this->nativeCodeGen, this->profiler);

#endif

this->isProfilerCreated = true;

Profiler \* oldProfiler = this->threadContext->GetRecycler()->GetProfiler();

this->threadContext->GetRecycler()->SetProfiler(this->profiler->GetProfiler(), this->profiler->GetBackgroundRecyclerProfiler());

return oldProfiler;

}

return nullptr;

}

void

ScriptContext::SetRecyclerProfiler()

{

Assert(Js::Configuration::Global.flags.IsEnabled(Js::ProfileFlag));

AssertMsg(this->profiler != nullptr, "Profiler tag is supplied but the profiler pointer is NULL");

if (this->ensureParentInfo)

{

this->hostScriptContext->EnsureParentInfo();

this->ensureParentInfo = false;

}

this->GetRecycler()->SetProfiler(this->profiler->GetProfiler(), this->profiler->GetBackgroundRecyclerProfiler());

}

void

ScriptContext::SetProfilerFromScriptContext(ScriptContext \* scriptContext)

{

// this function needs to be called before any code gen happens so

// that access to codegenProfiler won't have concurrency issues

if (Js::Configuration::Global.flags.IsEnabled(Js::ProfileFlag))

{

Assert(this->profiler != nullptr);

Assert(this->isProfilerCreated);

Assert(scriptContext->profiler != nullptr);

Assert(scriptContext->isProfilerCreated);

scriptContext->profiler->ProfileMerge(this->profiler);

this->profiler->Release();

this->profiler = scriptContext->profiler;

this->profiler->AddRef();

this->isProfilerCreated = false;

#if ENABLE\_NATIVE\_CODEGEN

SetProfilerFromNativeCodeGen(this->nativeCodeGen, scriptContext->GetNativeCodeGenerator());

#endif

this->threadContext->GetRecycler()->SetProfiler(this->profiler->GetProfiler(), this->profiler->GetBackgroundRecyclerProfiler());

}

}

void

ScriptContext::ProfileBegin(Js::Phase phase)

{

AssertMsg((this->profiler != nullptr) == Js::Configuration::Global.flags.IsEnabled(Js::ProfileFlag),

"Profiler tag is supplied but the profiler pointer is NULL");

if (this->profiler)

{

if (this->ensureParentInfo)

{

this->hostScriptContext->EnsureParentInfo();

this->ensureParentInfo = false;

}

this->profiler->ProfileBegin(phase);

}

}

void

ScriptContext::ProfileEnd(Js::Phase phase)

{

AssertMsg((this->profiler != nullptr) == Js::Configuration::Global.flags.IsEnabled(Js::ProfileFlag),

"Profiler tag is supplied but the profiler pointer is NULL");

if (this->profiler)

{

this->profiler->ProfileEnd(phase);

}

}

void

ScriptContext::ProfileSuspend(Js::Phase phase, Js::Profiler::SuspendRecord \* suspendRecord)

{

AssertMsg((this->profiler != nullptr) == Js::Configuration::Global.flags.IsEnabled(Js::ProfileFlag),

"Profiler tag is supplied but the profiler pointer is NULL");

if (this->profiler)

{

this->profiler->ProfileSuspend(phase, suspendRecord);

}

}

void

ScriptContext::ProfileResume(Js::Profiler::SuspendRecord \* suspendRecord)

{

AssertMsg((this->profiler != nullptr) == Js::Configuration::Global.flags.IsEnabled(Js::ProfileFlag),

"Profiler tag is supplied but the profiler pointer is NULL");

if (this->profiler)

{

this->profiler->ProfileResume(suspendRecord);

}

}

void

ScriptContext::ProfilePrint()

{

if (disableProfiler)

{

return;

}

Assert(profiler != nullptr);

recycler->EnsureNotCollecting();

profiler->ProfilePrint(Js::Configuration::Global.flags.Profile.GetFirstPhase());

#if ENABLE\_NATIVE\_CODEGEN

ProfilePrintNativeCodeGen(this->nativeCodeGen);

#endif

}

#endif

inline void ScriptContext::CoreSetProfileEventMask(DWORD dwEventMask)

{

AssertMsg(m\_pProfileCallback != NULL, "Assigning the event mask when there is no callback");

m\_dwEventMask = dwEventMask;

m\_fTraceFunctionCall = (dwEventMask & PROFILER\_EVENT\_MASK\_TRACE\_SCRIPT\_FUNCTION\_CALL);

m\_fTraceNativeFunctionCall = (dwEventMask & PROFILER\_EVENT\_MASK\_TRACE\_NATIVE\_FUNCTION\_CALL);

m\_fTraceDomCall = (dwEventMask & PROFILER\_EVENT\_MASK\_TRACE\_DOM\_FUNCTION\_CALL);

}

HRESULT ScriptContext::RegisterProfileProbe(IActiveScriptProfilerCallback \*pProfileCallback, DWORD dwEventMask, DWORD dwContext, RegisterExternalLibraryType RegisterExternalLibrary, JavascriptMethod dispatchInvoke)

{

if (m\_pProfileCallback != NULL)

{

return ACTIVPROF\_E\_PROFILER\_PRESENT;

}

OUTPUT\_TRACE(Js::ScriptProfilerPhase, L"ScriptContext::RegisterProfileProbe\n");

OUTPUT\_TRACE(Js::ScriptProfilerPhase, L"Info\nThunks Address :\n");

OUTPUT\_TRACE(Js::ScriptProfilerPhase, L"DefaultEntryThunk : 0x%08X, CrossSite::DefaultThunk : 0x%08X, DefaultDeferredParsingThunk : 0x%08X\n", DefaultEntryThunk, CrossSite::DefaultThunk, DefaultDeferredParsingThunk);

OUTPUT\_TRACE(Js::ScriptProfilerPhase, L"ProfileEntryThunk : 0x%08X, CrossSite::ProfileThunk : 0x%08X, ProfileDeferredParsingThunk : 0x%08X, ProfileDeferredDeserializeThunk : 0x%08X,\n", ProfileEntryThunk, CrossSite::ProfileThunk, ProfileDeferredParsingThunk, ProfileDeferredDeserializeThunk);

OUTPUT\_TRACE(Js::ScriptProfilerPhase, L"ScriptType :\n");

OUTPUT\_TRACE(Js::ScriptProfilerPhase, L"PROFILER\_SCRIPT\_TYPE\_USER : 0, PROFILER\_SCRIPT\_TYPE\_DYNAMIC : 1, PROFILER\_SCRIPT\_TYPE\_NATIVE : 2, PROFILER\_SCRIPT\_TYPE\_DOM : 3\n");

HRESULT hr = pProfileCallback->Initialize(dwContext);

if (SUCCEEDED(hr))

{

m\_pProfileCallback = pProfileCallback;

pProfileCallback->AddRef();

CoreSetProfileEventMask(dwEventMask);

if (m\_fTraceDomCall)

{

if (FAILED(pProfileCallback->QueryInterface(&m\_pProfileCallback2)))

{

m\_fTraceDomCall = FALSE;

}

}

if (webWorkerId != Js::Constants::NonWebWorkerContextId)

{

IActiveScriptProfilerCallback3 \* pProfilerCallback3;

if (SUCCEEDED(pProfileCallback->QueryInterface(&pProfilerCallback3)))

{

pProfilerCallback3->SetWebWorkerId(webWorkerId);

pProfilerCallback3->Release();

// Omitting the HRESULT since it is up to the callback to make use of the webWorker information.

}

}

#if DEBUG

StartNewProfileSession();

#endif

#if ENABLE\_NATIVE\_CODEGEN

NativeCodeGenerator \*pNativeCodeGen = this->GetNativeCodeGenerator();

AutoOptionalCriticalSection autoAcquireCodeGenQueue(GetNativeCodeGenCriticalSection(pNativeCodeGen));

#endif

this->SetProfileMode(TRUE);

#if ENABLE\_NATIVE\_CODEGEN

SetProfileModeNativeCodeGen(pNativeCodeGen, TRUE);

#endif

// Register builtin functions

if (m\_fTraceNativeFunctionCall)

{

hr = this->RegisterBuiltinFunctions(RegisterExternalLibrary);

if (FAILED(hr))

{

return hr;

}

}

this->RegisterAllScripts();

// Set the dispatch profiler:

this->SetDispatchProfile(TRUE, dispatchInvoke);

// Update the function objects currently present in there.

this->SetFunctionInRecyclerToProfileMode();

}

return hr;

}

HRESULT ScriptContext::SetProfileEventMask(DWORD dwEventMask)

{

if (m\_pProfileCallback == NULL)

{

return ACTIVPROF\_E\_PROFILER\_ABSENT;

}

return ACTIVPROF\_E\_UNABLE\_TO\_APPLY\_ACTION;

}

HRESULT ScriptContext::DeRegisterProfileProbe(HRESULT hrReason, JavascriptMethod dispatchInvoke)

{

if (m\_pProfileCallback == NULL)

{

return ACTIVPROF\_E\_PROFILER\_ABSENT;

}

OUTPUT\_TRACE(Js::ScriptProfilerPhase, L"ScriptContext::DeRegisterProfileProbe\n");

#if ENABLE\_NATIVE\_CODEGEN

// Acquire the code gen working queue - we are going to change the thunks

NativeCodeGenerator \*pNativeCodeGen = this->GetNativeCodeGenerator();

Assert(pNativeCodeGen);

{

AutoOptionalCriticalSection lock(GetNativeCodeGenCriticalSection(pNativeCodeGen));

this->SetProfileMode(FALSE);

SetProfileModeNativeCodeGen(pNativeCodeGen, FALSE);

// DisableJIT-TODO: Does need to happen even with JIT disabled?

// Unset the dispatch profiler:

if (dispatchInvoke != nullptr)

{

this->SetDispatchProfile(FALSE, dispatchInvoke);

}

}

#endif

m\_inProfileCallback = TRUE;

HRESULT hr = m\_pProfileCallback->Shutdown(hrReason);

m\_inProfileCallback = FALSE;

m\_pProfileCallback->Release();

m\_pProfileCallback = NULL;

if (m\_pProfileCallback2 != NULL)

{

m\_pProfileCallback2->Release();

m\_pProfileCallback2 = NULL;

}

#if DEBUG

StopProfileSession();

#endif

return hr;

}

void ScriptContext::SetProfileMode(BOOL fSet)

{

if (fSet)

{

AssertMsg(m\_pProfileCallback != NULL, "In profile mode when there is no call back");

this->CurrentThunk = ProfileEntryThunk;

this->CurrentCrossSiteThunk = CrossSite::ProfileThunk;

this->DeferredParsingThunk = ProfileDeferredParsingThunk;

this->DeferredDeserializationThunk = ProfileDeferredDeserializeThunk;

this->globalObject->EvalHelper = &Js::GlobalObject::ProfileModeEvalHelper;

#if DBG

this->hadProfiled = true;

#endif

}

else

{

this->CurrentThunk = DefaultEntryThunk;

this->CurrentCrossSiteThunk = CrossSite::DefaultThunk;

this->DeferredParsingThunk = DefaultDeferredParsingThunk;

this->globalObject->EvalHelper = &Js::GlobalObject::DefaultEvalHelper;

// In Debug mode/Fast F12 library is still needed for built-in wrappers.

if (!(this->IsInDebugMode() && this->IsExceptionWrapperForBuiltInsEnabled()))

{

this->javascriptLibrary->SetProfileMode(FALSE);

}

}

}

HRESULT ScriptContext::RegisterScript(Js::FunctionProxy \* proxy, BOOL fRegisterScript /\*default TRUE\*/)

{

if (m\_pProfileCallback == nullptr)

{

return ACTIVPROF\_E\_PROFILER\_ABSENT;

}

OUTPUT\_TRACE(Js::ScriptProfilerPhase, L"ScriptContext::RegisterScript, fRegisterScript : %s, IsFunctionDefer : %s\n", IsTrueOrFalse(fRegisterScript), IsTrueOrFalse(proxy->IsDeferred()));

AssertMsg(proxy != nullptr, "Function body cannot be null when calling reporting");

AssertMsg(proxy->GetScriptContext() == this, "wrong script context while reporting the function?");

if (fRegisterScript)

{

// Register the script to the callback.

// REVIEW: do we really need to undefer everything?

HRESULT hr = proxy->EnsureDeserialized()->Parse()->ReportScriptCompiled();

if (FAILED(hr))

{

return hr;

}

}

return !proxy->IsDeferred() ? proxy->GetFunctionBody()->RegisterFunction(false) : S\_OK;

}

HRESULT ScriptContext::RegisterAllScripts()

{

AssertMsg(m\_pProfileCallback != nullptr, "Called register scripts when we don't have profile callback");

OUTPUT\_TRACE(Js::ScriptProfilerPhase, L"ScriptContext::RegisterAllScripts started\n");

// Future Work: Once Utf8SourceInfo can generate the debug document text without requiring a function body,

// this code can be considerably simplified to doing the following:

// - scriptContext->MapScript() : Fire script compiled for each script

// - scriptContext->MapFunction(): Fire function compiled for each function

this->MapScript([](Utf8SourceInfo\* sourceInfo)

{

FunctionBody\* functionBody = sourceInfo->GetAnyParsedFunction();

if (functionBody)

{

functionBody->ReportScriptCompiled();

}

});

// FunctionCompiled events for all functions.

this->MapFunction([](Js::FunctionBody\* pFuncBody)

{

if (!pFuncBody->GetIsTopLevel() && pFuncBody->GetIsGlobalFunc())

{

// This must be the dummy function, generated due to the deferred parsing.

return;

}

pFuncBody->RegisterFunction(TRUE, TRUE); // Ignore potential failure (worst case is not profiling).

});

OUTPUT\_TRACE(Js::ScriptProfilerPhase, L"ScriptContext::RegisterAllScripts ended\n");

return S\_OK;

}

// Shuts down and recreates the native code generator. This is used when

// attaching and detaching the debugger in order to clear the list of work

// items that are pending in the JIT job queue.

// Alloc first and then free so that the native code generator is at a different address

#if ENABLE\_NATIVE\_CODEGEN

HRESULT ScriptContext::RecreateNativeCodeGenerator()

{

NativeCodeGenerator\* oldCodeGen = this->nativeCodeGen;

HRESULT hr = S\_OK;

BEGIN\_TRANSLATE\_OOM\_TO\_HRESULT\_NESTED

this->nativeCodeGen = NewNativeCodeGenerator(this);

SetProfileModeNativeCodeGen(this->GetNativeCodeGenerator(), this->IsProfiling());

END\_TRANSLATE\_OOM\_TO\_HRESULT(hr);

// Delete the native code generator and recreate so that all jobs get cleared properly

// and re-jitted.

CloseNativeCodeGenerator(oldCodeGen);

DeleteNativeCodeGenerator(oldCodeGen);

return hr;

}

#endif

HRESULT ScriptContext::OnDebuggerAttached()

{

OUTPUT\_TRACE(Js::DebuggerPhase, L"ScriptContext::OnDebuggerAttached: start 0x%p\n", this);

Js::StepController\* stepController = &this->GetThreadContext()->GetDebugManager()->stepController;

if (stepController->IsActive())

{

AssertMsg(stepController->GetActivatedContext() == nullptr, "StepController should not be active when we attach.");

stepController->Deactivate(); // Defense in depth

}

bool shouldPerformSourceRundown = false;

if (this->IsInNonDebugMode())

{

// Today we do source rundown as a part of attach to support VS attaching without

// first calling PerformSourceRundown. PerformSourceRundown will be called once

// by debugger host prior to attaching.

this->GetDebugContext()->SetInSourceRundownMode();

// Need to perform rundown only once.

shouldPerformSourceRundown = true;

}

// Rundown on all existing functions and change their thunks so that they will go to debug mode once they are called.

HRESULT hr = OnDebuggerAttachedDetached(/\*attach\*/ true);

if (SUCCEEDED(hr))

{

// Disable QC while functions are re-parsed as this can be time consuming

AutoDisableInterrupt autoDisableInterrupt(this->threadContext->GetInterruptPoller(), true);

if ((hr = this->GetDebugContext()->RundownSourcesAndReparse(shouldPerformSourceRundown, /\*shouldReparseFunctions\*/ true)) == S\_OK)

{

HRESULT hr2 = this->GetLibrary()->EnsureReadyIfHybridDebugging(); // Prepare library if hybrid debugging attach

Assert(hr2 != E\_FAIL); // Omitting HRESULT

}

if (!this->IsClosed())

{

HRESULT hrEntryPointUpdate = S\_OK;

BEGIN\_TRANSLATE\_OOM\_TO\_HRESULT\_NESTED

#ifdef ASMJS\_PLAT

TempArenaAllocatorObject\* tmpAlloc = GetTemporaryAllocator(L"DebuggerTransition");

debugTransitionAlloc = tmpAlloc->GetAllocator();

asmJsEnvironmentMap = Anew(debugTransitionAlloc, AsmFunctionMap, debugTransitionAlloc);

#endif

// Still do the pass on the function's entrypoint to reflect its state with the functionbody's entrypoint.

this->UpdateRecyclerFunctionEntryPointsForDebugger();

#ifdef ASMJS\_PLAT

auto asmEnvIter = asmJsEnvironmentMap->GetIterator();

while (asmEnvIter.IsValid())

{

// we are attaching, change frame setup for asm.js frame to javascript frame

SList<AsmJsScriptFunction \*> \* funcList = asmEnvIter.CurrentValue();

Assert(!funcList->Empty());

void\* newEnv = AsmJsModuleInfo::ConvertFrameForJavascript(asmEnvIter.CurrentKey(), funcList->Head());

funcList->Iterate([&](AsmJsScriptFunction \* func)

{

func->GetEnvironment()->SetItem(0, newEnv);

});

asmEnvIter.MoveNext();

}

// walk through and clean up the asm.js fields as a discrete step, because module might be multiply linked

auto asmCleanupIter = asmJsEnvironmentMap->GetIterator();

while (asmCleanupIter.IsValid())

{

SList<AsmJsScriptFunction \*> \* funcList = asmCleanupIter.CurrentValue();

Assert(!funcList->Empty());

funcList->Iterate([](AsmJsScriptFunction \* func)

{

func->SetModuleMemory(nullptr);

func->GetFunctionBody()->ResetAsmJsInfo();

});

asmCleanupIter.MoveNext();

}

ReleaseTemporaryAllocator(tmpAlloc);

#endif

END\_TRANSLATE\_OOM\_TO\_HRESULT(hrEntryPointUpdate);

if (hrEntryPointUpdate != S\_OK)

{

// should only be here for OOM

Assert(hrEntryPointUpdate == E\_OUTOFMEMORY);

return hrEntryPointUpdate;

}

}

}

else

{

// Let's find out on what conditions it fails

RAISE\_FATL\_INTERNAL\_ERROR\_IFFAILED(hr);

}

OUTPUT\_TRACE(Js::DebuggerPhase, L"ScriptContext::OnDebuggerAttached: done 0x%p, hr = 0x%X\n", this, hr);

return hr;

}

// Reverts the script context state back to the state before debugging began.

HRESULT ScriptContext::OnDebuggerDetached()

{

OUTPUT\_TRACE(Js::DebuggerPhase, L"ScriptContext::OnDebuggerDetached: start 0x%p\n", this);

Js::StepController\* stepController = &this->GetThreadContext()->GetDebugManager()->stepController;

if (stepController->IsActive())

{

// Normally step controller is deactivated on start of dispatch (step, async break, exception, etc),

// and in the beginning of interpreter loop we check for step complete (can cause check whether current bytecode belong to stmt).

// But since it holds to functionBody/statementMaps, we have to deactivate it as func bodies are going away/reparsed.

stepController->Deactivate();

}

// Go through all existing functions and change their thunks back to using non-debug mode versions when called

// and notify the script context that the debugger has detached to allow it to revert the runtime to the proper

// state (JIT enabled).

HRESULT hr = OnDebuggerAttachedDetached(/\*attach\*/ false);

if (SUCCEEDED(hr))

{

// Move the debugger into source rundown mode.

this->GetDebugContext()->SetInSourceRundownMode();

// Disable QC while functions are re-parsed as this can be time consuming

AutoDisableInterrupt autoDisableInterrupt(this->threadContext->GetInterruptPoller(), true);

// Force a reparse so that indirect function caches are updated.

hr = this->GetDebugContext()->RundownSourcesAndReparse(/\*shouldPerformSourceRundown\*/ false, /\*shouldReparseFunctions\*/ true);

// Let's find out on what conditions it fails

RAISE\_FATL\_INTERNAL\_ERROR\_IFFAILED(hr);

// Still do the pass on the function's entrypoint to reflect its state with the functionbody's entrypoint.

this->UpdateRecyclerFunctionEntryPointsForDebugger();

}

else

{

// Let's find out on what conditions it fails

RAISE\_FATL\_INTERNAL\_ERROR\_IFFAILED(hr);

}

OUTPUT\_TRACE(Js::DebuggerPhase, L"ScriptContext::OnDebuggerDetached: done 0x%p, hr = 0x%X\n", this, hr);

return hr;

}

HRESULT ScriptContext::OnDebuggerAttachedDetached(bool attach)

{

// notify threadContext that debugger is attaching so do not do expire

struct AutoRestore

{

AutoRestore(ThreadContext\* threadContext)

:threadContext(threadContext)

{

this->threadContext->GetDebugManager()->SetDebuggerAttaching(true);

}

~AutoRestore()

{

this->threadContext->GetDebugManager()->SetDebuggerAttaching(false);

}

private:

ThreadContext\* threadContext;

} autoRestore(this->GetThreadContext());

if (!Js::Configuration::Global.EnableJitInDebugMode())

{

if (attach)

{

// Now force nonative, so the job will not be put in jit queue.

ForceNoNative();

}

else

{

// Take the runtime out of interpreted mode so the JIT

// queue can be exercised.

this->ForceNative();

}

}

// Invalidate all the caches.

this->threadContext->InvalidateAllProtoInlineCaches();

this->threadContext->InvalidateAllStoreFieldInlineCaches();

this->threadContext->InvalidateAllIsInstInlineCaches();

if (!attach)

{

this->UnRegisterDebugThunk();

// Remove all breakpoint probes

this->GetDebugContext()->GetProbeContainer()->RemoveAllProbes();

}

HRESULT hr = S\_OK;

if (!CONFIG\_FLAG(ForceDiagnosticsMode))

{

#if ENABLE\_NATIVE\_CODEGEN

// Recreate the native code generator so that all pending

// JIT work items will be cleared.

hr = RecreateNativeCodeGenerator();

if (FAILED(hr))

{

return hr;

}

#endif

if (attach)

{

// We need to transition to debug mode after the NativeCodeGenerator is cleared/closed. Since the NativeCodeGenerator will be working on a different thread - it may

// be checking on the DebuggerState (from ScriptContext) while emitting code.

this->GetDebugContext()->SetInDebugMode();

#if ENABLE\_NATIVE\_CODEGEN

UpdateNativeCodeGeneratorForDebugMode(this->nativeCodeGen);

#endif

}

}

else if (attach)

{

this->GetDebugContext()->SetInDebugMode();

}

BEGIN\_TRANSLATE\_OOM\_TO\_HRESULT\_NESTED

{

// Remap all the function entry point thunks.

this->sourceList->Map([=](uint i, RecyclerWeakReference<Js::Utf8SourceInfo>\* sourceInfoWeakRef) {

Js::Utf8SourceInfo\* sourceInfo = sourceInfoWeakRef->Get();

if (sourceInfo) {

sourceInfo->SetInDebugMode(attach);

if (!sourceInfo->GetIsLibraryCode())

{

sourceInfo->MapFunction([](Js::FunctionBody\* functionBody) {

functionBody->SetEntryToDeferParseForDebugger();

});

}

else

{

sourceInfo->MapFunction([](Js::FunctionBody\* functionBody) {

functionBody->ResetEntryPoint();

});

}

}

});

}

END\_TRANSLATE\_OOM\_TO\_HRESULT(hr);

if (FAILED(hr))

{

return hr;

}

if (attach)

{

this->RegisterDebugThunk();

}

#if ENABLE\_PROFILE\_INFO

#if DBG\_DUMP || defined(DYNAMIC\_PROFILE\_STORAGE) || defined(RUNTIME\_DATA\_COLLECTION)

// Reset the dynamic profile list

if (this->profileInfoList)

{

this->profileInfoList->Reset();

}

#endif

#endif

return hr;

}

// We use ProfileThunk under debugger.

void ScriptContext::RegisterDebugThunk(bool calledDuringAttach /\*= true\*/)

{

if (this->IsExceptionWrapperForBuiltInsEnabled())

{

this->CurrentThunk = ProfileEntryThunk;

this->CurrentCrossSiteThunk = CrossSite::ProfileThunk;

#if ENABLE\_NATIVE\_CODEGEN

SetProfileModeNativeCodeGen(this->GetNativeCodeGenerator(), TRUE);

#endif

// Set library to profile mode so that for built-ins all new instances of functions

// are created with entry point set to the ProfileThunk.

this->javascriptLibrary->SetProfileMode(true);

this->javascriptLibrary->SetDispatchProfile(true, DispatchProfileInoke);

if (!calledDuringAttach)

{

m\_fTraceDomCall = TRUE; // This flag is always needed in DebugMode to wrap external functions with DebugProfileThunk

// Update the function objects currently present in there.

this->SetFunctionInRecyclerToProfileMode(true/\*enumerateNonUserFunctionsOnly\*/);

}

}

}

void ScriptContext::UnRegisterDebugThunk()

{

if (!this->IsProfiling() && this->IsExceptionWrapperForBuiltInsEnabled())

{

this->CurrentThunk = DefaultEntryThunk;

this->CurrentCrossSiteThunk = CrossSite::DefaultThunk;

#if ENABLE\_NATIVE\_CODEGEN

SetProfileModeNativeCodeGen(this->GetNativeCodeGenerator(), FALSE);

#endif

if (!this->IsProfiling())

{

this->javascriptLibrary->SetProfileMode(false);

this->javascriptLibrary->SetDispatchProfile(false, DispatchDefaultInvoke);

}

}

}

HRESULT ScriptContext::RegisterBuiltinFunctions(RegisterExternalLibraryType RegisterExternalLibrary)

{

Assert(m\_pProfileCallback != NULL);

OUTPUT\_TRACE(Js::ScriptProfilerPhase, L"ScriptContext::RegisterBuiltinFunctions\n");

HRESULT hr = S\_OK;

// Consider creating ProfileArena allocator instead of General allocator

if (m\_pBuiltinFunctionIdMap == NULL)

{

// Anew throws if it OOMs, so the caller into this function needs to handle that exception

m\_pBuiltinFunctionIdMap = Anew(GeneralAllocator(), BuiltinFunctionIdDictionary,

GeneralAllocator(), 17);

}

this->javascriptLibrary->SetProfileMode(TRUE);

if (FAILED(hr = OnScriptCompiled(BuiltInFunctionsScriptId, PROFILER\_SCRIPT\_TYPE\_NATIVE, NULL)))

{

return hr;

}

if (FAILED(hr = this->javascriptLibrary->ProfilerRegisterBuiltIns()))

{

return hr;

}

// External Library

if (RegisterExternalLibrary != NULL)

{

(\*RegisterExternalLibrary)(this);

}

return hr;

}

void ScriptContext::SetFunctionInRecyclerToProfileMode(bool enumerateNonUserFunctionsOnly/\* = false\*/)

{

OUTPUT\_TRACE(Js::ScriptProfilerPhase, L"ScriptContext::SetFunctionInRecyclerToProfileMode started (m\_fTraceDomCall : %s)\n", IsTrueOrFalse(m\_fTraceDomCall));

// Mark this script context isEnumeratingRecyclerObjects

AutoEnumeratingRecyclerObjects enumeratingRecyclerObjects(this);

m\_enumerateNonUserFunctionsOnly = enumerateNonUserFunctionsOnly;

this->recycler->EnumerateObjects(JavascriptLibrary::EnumFunctionClass, &ScriptContext::RecyclerEnumClassEnumeratorCallback);

OUTPUT\_TRACE(Js::ScriptProfilerPhase, L"ScriptContext::SetFunctionInRecyclerToProfileMode ended\n");

}

void ScriptContext::UpdateRecyclerFunctionEntryPointsForDebugger()

{

// Mark this script context isEnumeratingRecyclerObjects

AutoEnumeratingRecyclerObjects enumeratingRecyclerObjects(this);

this->recycler->EnumerateObjects(JavascriptLibrary::EnumFunctionClass, &ScriptContext::RecyclerFunctionCallbackForDebugger);

}

#ifdef ASMJS\_PLAT

void ScriptContext::TransitionEnvironmentForDebugger(ScriptFunction \* scriptFunction)

{

if (scriptFunction->GetScriptContext()->IsInDebugMode() &&

scriptFunction->GetFunctionBody()->GetAsmJsFunctionInfo() != nullptr &&

scriptFunction->GetFunctionBody()->GetAsmJsFunctionInfo()->GetModuleFunctionBody() != nullptr)

{

void\* env = scriptFunction->GetEnvironment()->GetItem(0);

SList<AsmJsScriptFunction\*> \* funcList = nullptr;

if (asmJsEnvironmentMap->TryGetValue(env, &funcList))

{

funcList->Push((AsmJsScriptFunction\*)scriptFunction);

}

else

{

SList<AsmJsScriptFunction\*> \* newList = Anew(debugTransitionAlloc, SList<AsmJsScriptFunction\*>, debugTransitionAlloc);

asmJsEnvironmentMap->AddNew(env, newList);

newList->Push((AsmJsScriptFunction\*)scriptFunction);

}

}

}

#endif

/\*static\*/

void ScriptContext::RecyclerFunctionCallbackForDebugger(void \*address, size\_t size)

{

JavascriptFunction \*pFunction = (JavascriptFunction \*)address;

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

if (scriptContext == nullptr || scriptContext->IsClosed())

{

// Can't enumerate from closed scriptcontext

return;

}

if (!scriptContext->IsEnumeratingRecyclerObjects())

{

return; // function not from enumerating script context

}

// Wrapped function are not allocated with the EnumClass bit

Assert(pFunction->GetFunctionInfo() != &JavascriptExternalFunction::EntryInfo::WrappedFunctionThunk);

JavascriptMethod entryPoint = pFunction->GetEntryPoint();

FunctionInfo \* info = pFunction->GetFunctionInfo();

FunctionProxy \* proxy = info->GetFunctionProxy();

if (proxy != info)

{

// Not a script function or, the thunk can deal with moving to the function body

Assert(proxy == nullptr || entryPoint == DefaultDeferredParsingThunk || entryPoint == ProfileDeferredParsingThunk

|| entryPoint == DefaultDeferredDeserializeThunk || entryPoint == ProfileDeferredDeserializeThunk ||

entryPoint == CrossSite::DefaultThunk || entryPoint == CrossSite::ProfileThunk);

// Replace entry points for built-ins/external/winrt functions so that we can wrap them with try-catch for "continue after exception".

if (!pFunction->IsScriptFunction() && IsExceptionWrapperForBuiltInsEnabled(scriptContext))

{

if (scriptContext->IsInDebugMode())

{

// We are attaching.

// For built-ins, WinRT and DOM functions which are already in recycler, change entry points to route to debug/profile thunk.

ScriptContext::SetEntryPointToProfileThunk(pFunction);

}

else

{

// We are detaching.

// For built-ins, WinRT and DOM functions which are already in recycler, restore entry points to original.

if (!scriptContext->IsProfiling())

{

ScriptContext::RestoreEntryPointFromProfileThunk(pFunction);

}

// If we are profiling, don't change anything.

}

}

return;

}

if (!proxy->IsFunctionBody())

{

// REVIEW: why we still have function that is still deferred?

return;

}

Assert(pFunction->IsScriptFunction());

// Excluding the internal library code, which is not debuggable already

if (!proxy->GetUtf8SourceInfo()->GetIsLibraryCode())

{

// Reset the constructor cache to default, so that it will not pick up the cached type, created before debugging.

// Look bug: 301517

pFunction->ResetConstructorCacheToDefault();

}

if (ScriptFunctionWithInlineCache::Is(pFunction))

{

ScriptFunctionWithInlineCache::FromVar(pFunction)->ClearInlineCacheOnFunctionObject();

}

// We should have force parsed the function, and have a function body

FunctionBody \* pBody = proxy->GetFunctionBody();

#ifdef ENABLE\_DEBUG\_CONFIG\_OPTIONS

if (scriptContext->IsInDebugMode())

{

if (!(proxy->GetUtf8SourceInfo()->GetIsLibraryCode() || pBody->IsByteCodeDebugMode()))

{

// Identifying if any function escaped for not being in debug mode. (This can be removed as a part of TFS : 935011)

Throw::FatalInternalError();

}

}

#endif

ScriptFunction \* scriptFunction = ScriptFunction::FromVar(pFunction);

#ifdef ASMJS\_PLAT

scriptContext->TransitionEnvironmentForDebugger(scriptFunction);

#endif

JavascriptMethod newEntryPoint;

if (CrossSite::IsThunk(entryPoint))

{

// Can't change from cross-site to non-cross-site, but still need to update the e.p.info on ScriptFunctionType.

newEntryPoint = entryPoint;

}

else

{

newEntryPoint = pBody->GetDirectEntryPoint(pBody->GetDefaultFunctionEntryPointInfo());

}

scriptFunction->ChangeEntryPoint(pBody->GetDefaultFunctionEntryPointInfo(), newEntryPoint);

}

void ScriptContext::RecyclerEnumClassEnumeratorCallback(void \*address, size\_t size)

{

// TODO: we are assuming its function because for now we are enumerating only on functions

// In future if the RecyclerNewEnumClass is used of Recyclable objects or Dynamic object, we would need a check if it is function

JavascriptFunction \*pFunction = (JavascriptFunction \*)address;

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

if (scriptContext == nullptr || scriptContext->IsClosed())

{

// Can't enumerate from closed scriptcontext

return;

}

if (!scriptContext->IsEnumeratingRecyclerObjects())

{

return; // function not from enumerating script context

}

if (!scriptContext->IsTraceDomCall() && (pFunction->IsExternalFunction() || pFunction->IsWinRTFunction()))

{

return;

}

if (scriptContext->IsEnumerateNonUserFunctionsOnly() && pFunction->IsScriptFunction())

{

return;

}

// Wrapped function are not allocated with the EnumClass bit

Assert(pFunction->GetFunctionInfo() != &JavascriptExternalFunction::EntryInfo::WrappedFunctionThunk);

JavascriptMethod entryPoint = pFunction->GetEntryPoint();

FunctionProxy \*proxy = pFunction->GetFunctionProxy();

if (proxy != NULL)

{

#if ENABLE\_DEBUG\_CONFIG\_OPTIONS

wchar\_t debugStringBuffer[MAX\_FUNCTION\_BODY\_DEBUG\_STRING\_SIZE];

#endif

OUTPUT\_TRACE(Js::ScriptProfilerPhase, L"ScriptContext::RecyclerEnumClassEnumeratorCallback\n");

OUTPUT\_TRACE(Js::ScriptProfilerPhase, L"\tFunctionProxy : 0x%08X, FunctionNumber : %s, DeferredParseAttributes : %d, EntryPoint : 0x%08X",

(DWORD\_PTR)proxy, proxy->GetDebugNumberSet(debugStringBuffer), proxy->GetAttributes(), (DWORD\_PTR)entryPoint);

#if ENABLE\_NATIVE\_CODEGEN

OUTPUT\_TRACE(Js::ScriptProfilerPhase, L" (IsIntermediateCodeGenThunk : %s, isNative : %s)\n",

IsTrueOrFalse(IsIntermediateCodeGenThunk(entryPoint)), IsTrueOrFalse(scriptContext->IsNativeAddress(entryPoint)));

#endif

OUTPUT\_TRACE(Js::ScriptProfilerPhase, L"\n");

FunctionInfo \* info = pFunction->GetFunctionInfo();

if (proxy != info)

{

// The thunk can deal with moving to the function body

Assert(entryPoint == DefaultDeferredParsingThunk || entryPoint == ProfileDeferredParsingThunk

|| entryPoint == DefaultDeferredDeserializeThunk || entryPoint == ProfileDeferredDeserializeThunk

|| entryPoint == CrossSite::DefaultThunk || entryPoint == CrossSite::ProfileThunk);

Assert(!proxy->IsDeferred());

Assert(proxy->GetFunctionBody()->GetProfileSession() == proxy->GetScriptContext()->GetProfileSession());

return;

}

#if ENABLE\_NATIVE\_CODEGEN

if (!IsIntermediateCodeGenThunk(entryPoint) && entryPoint != DynamicProfileInfo::EnsureDynamicProfileInfoThunk)

#endif

{

OUTPUT\_TRACE(Js::ScriptProfilerPhase, L"\t\tJs::ScriptContext::GetProfileModeThunk : 0x%08X\n", (DWORD\_PTR)Js::ScriptContext::GetProfileModeThunk(entryPoint));

ScriptFunction \* scriptFunction = ScriptFunction::FromVar(pFunction);

scriptFunction->ChangeEntryPoint(proxy->GetDefaultEntryPointInfo(), Js::ScriptContext::GetProfileModeThunk(entryPoint));

#if ENABLE\_NATIVE\_CODEGEN

OUTPUT\_TRACE(Js::ScriptProfilerPhase, L"\tUpdated entrypoint : 0x%08X (isNative : %s)\n", (DWORD\_PTR)pFunction->GetEntryPoint(), IsTrueOrFalse(scriptContext->IsNativeAddress(entryPoint)));

#endif

}

}

else

{

ScriptContext::SetEntryPointToProfileThunk(pFunction);

}

}

// static

void ScriptContext::SetEntryPointToProfileThunk(JavascriptFunction\* function)

{

JavascriptMethod entryPoint = function->GetEntryPoint();

if (entryPoint == Js::CrossSite::DefaultThunk)

{

function->SetEntryPoint(Js::CrossSite::ProfileThunk);

}

else if (entryPoint != Js::CrossSite::ProfileThunk && entryPoint != ProfileEntryThunk)

{

function->SetEntryPoint(ProfileEntryThunk);

}

}

// static

void ScriptContext::RestoreEntryPointFromProfileThunk(JavascriptFunction\* function)

{

JavascriptMethod entryPoint = function->GetEntryPoint();

if (entryPoint == Js::CrossSite::ProfileThunk)

{

function->SetEntryPoint(Js::CrossSite::DefaultThunk);

}

else if (entryPoint == ProfileEntryThunk)

{

function->SetEntryPoint(function->GetFunctionInfo()->GetOriginalEntryPoint());

}

}

JavascriptMethod ScriptContext::GetProfileModeThunk(JavascriptMethod entryPoint)

{

#if ENABLE\_NATIVE\_CODEGEN

Assert(!IsIntermediateCodeGenThunk(entryPoint));

#endif

if (entryPoint == DefaultDeferredParsingThunk || entryPoint == ProfileDeferredParsingThunk)

{

return ProfileDeferredParsingThunk;

}

if (entryPoint == DefaultDeferredDeserializeThunk || entryPoint == ProfileDeferredDeserializeThunk)

{

return ProfileDeferredDeserializeThunk;

}

if (CrossSite::IsThunk(entryPoint))

{

return CrossSite::ProfileThunk;

}

return ProfileEntryThunk;

}

#if \_M\_IX86

\_\_declspec(naked)

Var ScriptContext::ProfileModeDeferredParsingThunk(RecyclableObject\* function, CallInfo callInfo, ...)

{

// Register functions

\_\_asm

{

push ebp

mov ebp, esp

lea eax, [esp + 8]

push eax

call ScriptContext::ProfileModeDeferredParse

#ifdef \_CONTROL\_FLOW\_GUARD

// verify that the call target is valid

mov ecx, eax

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

mov eax, ecx

#endif

pop ebp

// Although we don't restore ESP here on WinCE, this is fine because script profiler is not shipped for WinCE.

jmp eax

}

}

#elif defined(\_M\_X64) || defined(\_M\_ARM32\_OR\_ARM64)

// Do nothing: the implementation of ScriptContext::ProfileModeDeferredParsingThunk is declared (appropriately decorated) in

// Language\amd64\amd64\_Thunks.asm and Language\arm\arm\_Thunks.asm and Language\arm64\arm64\_Thunks.asm respectively.

#else

Var ScriptContext::ProfileModeDeferredParsingThunk(RecyclableObject\* function, CallInfo callInfo, ...)

{

Js::Throw::NotImplemented();

return nullptr;

}

#endif

Js::JavascriptMethod ScriptContext::ProfileModeDeferredParse(ScriptFunction \*\* functionRef)

{

#if ENABLE\_DEBUG\_CONFIG\_OPTIONS

wchar\_t debugStringBuffer[MAX\_FUNCTION\_BODY\_DEBUG\_STRING\_SIZE];

#endif

OUTPUT\_TRACE(Js::ScriptProfilerPhase, L"ScriptContext::ProfileModeDeferredParse FunctionNumber : %s, startEntrypoint : 0x%08X\n", (\*functionRef)->GetFunctionProxy()->GetDebugNumberSet(debugStringBuffer), (\*functionRef)->GetEntryPoint());

BOOL fParsed = FALSE;

JavascriptMethod entryPoint = Js::JavascriptFunction::DeferredParseCore(functionRef, fParsed);

OUTPUT\_TRACE(Js::ScriptProfilerPhase, L"\t\tIsParsed : %s, updatedEntrypoint : 0x%08X\n", IsTrueOrFalse(fParsed), entryPoint);

//To get the scriptContext we only need the functionProxy

FunctionProxy \*pRootBody = (\*functionRef)->GetFunctionProxy();

ScriptContext \*pScriptContext = pRootBody->GetScriptContext();

if (pScriptContext->IsProfiling() && !pRootBody->GetFunctionBody()->HasFunctionCompiledSent())

{

pScriptContext->RegisterScript(pRootBody, FALSE /\*fRegisterScript\*/);

}

// We can come to this function even though we have stopped profiling.

Assert(!pScriptContext->IsProfiling() || (\*functionRef)->GetFunctionBody()->GetProfileSession() == pScriptContext->GetProfileSession());

return entryPoint;

}

#if \_M\_IX86

\_\_declspec(naked)

Var ScriptContext::ProfileModeDeferredDeserializeThunk(RecyclableObject\* function, CallInfo callInfo, ...)

{

// Register functions

\_\_asm

{

push ebp

mov ebp, esp

push[esp + 8]

call ScriptContext::ProfileModeDeferredDeserialize

#ifdef \_CONTROL\_FLOW\_GUARD

// verify that the call target is valid

mov ecx, eax

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

mov eax, ecx

#endif

pop ebp

// Although we don't restore ESP here on WinCE, this is fine because script profiler is not shipped for WinCE.

jmp eax

}

}

#elif defined(\_M\_X64) || defined(\_M\_ARM32\_OR\_ARM64)

// Do nothing: the implementation of ScriptContext::ProfileModeDeferredDeserializeThunk is declared (appropriately decorated) in

// Language\amd64\amd64\_Thunks.asm and Language\arm\arm\_Thunks.asm respectively.

#endif

Js::JavascriptMethod ScriptContext::ProfileModeDeferredDeserialize(ScriptFunction \*function)

{

#if ENABLE\_DEBUG\_CONFIG\_OPTIONS

wchar\_t debugStringBuffer[MAX\_FUNCTION\_BODY\_DEBUG\_STRING\_SIZE];

#endif

OUTPUT\_TRACE(Js::ScriptProfilerPhase, L"ScriptContext::ProfileModeDeferredDeserialize FunctionNumber : %s\n", function->GetFunctionProxy()->GetDebugNumberSet(debugStringBuffer));

JavascriptMethod entryPoint = Js::JavascriptFunction::DeferredDeserialize(function);

//To get the scriptContext; we only need the FunctionProxy

FunctionProxy \*pRootBody = function->GetFunctionProxy();

ScriptContext \*pScriptContext = pRootBody->GetScriptContext();

if (pScriptContext->IsProfiling() && !pRootBody->GetFunctionBody()->HasFunctionCompiledSent())

{

pScriptContext->RegisterScript(pRootBody, FALSE /\*fRegisterScript\*/);

}

// We can come to this function even though we have stopped profiling.

Assert(!pScriptContext->IsProfiling() || function->GetFunctionBody()->GetProfileSession() == pScriptContext->GetProfileSession());

return entryPoint;

}

BOOL ScriptContext::GetProfileInfo(

JavascriptFunction\* function,

PROFILER\_TOKEN &scriptId,

PROFILER\_TOKEN &functionId)

{

BOOL fCanProfile = (m\_pProfileCallback != nullptr && m\_fTraceFunctionCall);

if (!fCanProfile)

{

return FALSE;

}

Js::FunctionInfo\* functionInfo = function->GetFunctionInfo();

if (functionInfo->GetAttributes() & FunctionInfo::DoNotProfile)

{

return FALSE;

}

Js::FunctionBody \* functionBody = functionInfo->GetFunctionBody();

if (functionBody == nullptr)

{

functionId = GetFunctionNumber(functionInfo->GetOriginalEntryPoint());

if (functionId == -1)

{

// Dom Call

return m\_fTraceDomCall && (m\_pProfileCallback2 != nullptr);

}

else

{

// Builtin function

scriptId = BuiltInFunctionsScriptId;

return m\_fTraceNativeFunctionCall;

}

}

else if (!functionBody->GetUtf8SourceInfo()->GetIsLibraryCode() || functionBody->IsPublicLibraryCode()) // user script or public library code

{

scriptId = (PROFILER\_TOKEN)functionBody->GetUtf8SourceInfo()->GetSourceInfoId();

functionId = functionBody->GetFunctionNumber();

return TRUE;

}

return FALSE;

}

bool ScriptContext::IsForceNoNative()

{

bool forceNoNative = false;

if (!this->IsInNonDebugMode())

{

forceNoNative = this->IsInterpreted();

}

else if (!Js::Configuration::Global.EnableJitInDebugMode())

{

forceNoNative = true;

this->ForceNoNative();

}

return forceNoNative;

}

void ScriptContext::InitializeDebugging()

{

if (!this->IsInDebugMode()) // If we already in debug mode, we would have done below changes already.

{

this->GetDebugContext()->SetInDebugMode();

if (this->IsInDebugMode())

{

// Note: for this we need final IsInDebugMode and NativeCodeGen initialized,

// and inside EnsureScriptContext, which seems appropriate as well,

// it's too early as debugger manager is not registered, thus IsDebuggerEnvironmentAvailable is false.

this->RegisterDebugThunk(false/\*calledDuringAttach\*/);

// TODO: for launch scenario for external and WinRT functions it might be too late to register debug thunk here,

// as we need the thunk registered before FunctionInfo's for built-ins, that may throw, are created.

// Need to verify. If that's the case, one way would be to enumerate and fix all external/winRT thunks here.

}

}

}

// Combined profile/debug wrapper thunk.

// - used when we profile to send profile events

// - used when we debug, only used for built-in functions

// - used when we profile and debug

Var ScriptContext::DebugProfileProbeThunk(RecyclableObject\* callable, CallInfo callInfo, ...)

{

RUNTIME\_ARGUMENTS(args, callInfo);

JavascriptFunction\* function = JavascriptFunction::FromVar(callable);

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

PROFILER\_TOKEN scriptId = -1;

PROFILER\_TOKEN functionId = -1;

bool functionEnterEventSent = false;

const bool isProfilingUserCode = scriptContext->GetThreadContext()->IsProfilingUserCode();

const bool isUserCode = !function->IsLibraryCode();

wchar\_t \*pwszExtractedFunctionName = NULL;

const wchar\_t \*pwszFunctionName = NULL;

HRESULT hrOfEnterEvent = S\_OK;

// We can come here when profiling is not on

// e.g. User starts profiling, we update all thinks and then stop profiling - we don't update thunk

// So we still get this call

const bool fProfile = (isUserCode || isProfilingUserCode) // Only report user code or entry library code

&& scriptContext->GetProfileInfo(function, scriptId, functionId);

if (fProfile)

{

Js::FunctionBody \*pBody = function->GetFunctionBody();

if (pBody != nullptr && !pBody->HasFunctionCompiledSent())

{

pBody->RegisterFunction(false/\*changeThunk\*/);

}

#if DEBUG

{ // scope

Assert(scriptContext->IsProfiling());

if (pBody && pBody->GetProfileSession() != pBody->GetScriptContext()->GetProfileSession())

{

wchar\_t debugStringBuffer[MAX\_FUNCTION\_BODY\_DEBUG\_STRING\_SIZE];

OUTPUT\_TRACE\_DEBUGONLY(Js::ScriptProfilerPhase, L"ScriptContext::ProfileProbeThunk, ProfileSession does not match (%d != %d), functionNumber : %s, functionName : %s\n",

pBody->GetProfileSession(), pBody->GetScriptContext()->GetProfileSession(), pBody->GetDebugNumberSet(debugStringBuffer), pBody->GetDisplayName());

}

AssertMsg(pBody == NULL || pBody->GetProfileSession() == pBody->GetScriptContext()->GetProfileSession(), "Function info wasn't reported for this profile session");

}

#endif

if (functionId == -1)

{

Var sourceString = function->GetSourceString();

// SourceString will be null for the Js::BoundFunction, don't throw Enter/Exit notification in that case.

if (sourceString != NULL)

{

if (TaggedInt::Is(sourceString))

{

PropertyId nameId = TaggedInt::ToInt32(sourceString);

pwszFunctionName = scriptContext->GetPropertyString(nameId)->GetSz();

}

else

{

// it is string because user had called in toString extract name from it

Assert(JavascriptString::Is(sourceString));

const wchar\_t \*pwszToString = ((JavascriptString \*)sourceString)->GetSz();

const wchar\_t \*pwszNameStart = wcsstr(pwszToString, L" ");

const wchar\_t \*pwszNameEnd = wcsstr(pwszToString, L"(");

if (pwszNameStart == nullptr || pwszNameEnd == nullptr || ((int)(pwszNameEnd - pwszNameStart) <= 0))

{

int len = ((JavascriptString \*)sourceString)->GetLength() + 1;

pwszExtractedFunctionName = new wchar\_t[len];

wcsncpy\_s(pwszExtractedFunctionName, len, pwszToString, \_TRUNCATE);

}

else

{

int len = (int)(pwszNameEnd - pwszNameStart);

AssertMsg(len > 0, "Allocating array with zero or negative length?");

pwszExtractedFunctionName = new wchar\_t[len];

wcsncpy\_s(pwszExtractedFunctionName, len, pwszNameStart + 1, \_TRUNCATE);

}

pwszFunctionName = pwszExtractedFunctionName;

}

functionEnterEventSent = true;

Assert(pwszFunctionName != NULL);

hrOfEnterEvent = scriptContext->OnDispatchFunctionEnter(pwszFunctionName);

}

}

else

{

hrOfEnterEvent = scriptContext->OnFunctionEnter(scriptId, functionId);

}

scriptContext->GetThreadContext()->SetIsProfilingUserCode(isUserCode); // Update IsProfilingUserCode state

}

Var aReturn = NULL;

JavascriptMethod origEntryPoint = function->GetFunctionInfo()->GetOriginalEntryPoint();

\_\_try

{

Assert(!function->IsScriptFunction() || function->GetFunctionProxy());

// No need to wrap script functions, also can't if the wrapper is already on the stack.

// Treat "library code" script functions, such as Intl, as built-ins:

// use the wrapper when calling them, and do not reset the wrapper when calling them.

bool isDebugWrapperEnabled = scriptContext->IsInDebugMode() && IsExceptionWrapperForBuiltInsEnabled(scriptContext);

bool useDebugWrapper =

isDebugWrapperEnabled &&

function->IsLibraryCode() &&

!AutoRegisterIgnoreExceptionWrapper::IsRegistered(scriptContext->GetThreadContext());

OUTPUT\_VERBOSE\_TRACE(Js::DebuggerPhase, L"DebugProfileProbeThunk: calling function: %s isWrapperRegistered=%d useDebugWrapper=%d\n",

function->GetFunctionInfo()->HasBody() ? function->GetFunctionBody()->GetDisplayName() : L"built-in/library", AutoRegisterIgnoreExceptionWrapper::IsRegistered(scriptContext->GetThreadContext()), useDebugWrapper);

if (scriptContext->IsInDebugMode())

{

scriptContext->GetDebugContext()->GetProbeContainer()->StartRecordingCall();

}

if (useDebugWrapper)

{

// For native use wrapper and bail out on to ignore exception.

// Extract try-catch out of hot path in normal profile mode (presence of try-catch in a function is bad for perf).

aReturn = ProfileModeThunk\_DebugModeWrapper(function, scriptContext, origEntryPoint, args);

}

else

{

if (isDebugWrapperEnabled && !function->IsLibraryCode())

{

// We want to ignore exception and continue into closest user/script function down on the stack.

// Thus, if needed, reset the wrapper for the time of this call,

// so that if there is library/helper call after script function, it will use try-catch.

// Can't use smart/destructor object here because of \_\_try\_\_finally.

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

bool isOrigWrapperPresent = threadContext->GetDebugManager()->GetDebuggingFlags()->IsBuiltInWrapperPresent();

if (isOrigWrapperPresent)

{

threadContext->GetDebugManager()->GetDebuggingFlags()->SetIsBuiltInWrapperPresent(false);

}

\_\_try

{

aReturn = JavascriptFunction::CallFunction<true>(function, origEntryPoint, args);

}

\_\_finally

{

threadContext->GetDebugManager()->GetDebuggingFlags()->SetIsBuiltInWrapperPresent(isOrigWrapperPresent);

}

}

else

{

// Can we update return address to a thunk that sends Exit event and then jmp to entry instead of Calling it.

// Saves stack space and it might be something we would be doing anyway for handling profile.Start/stop

// which can come anywhere on the stack.

aReturn = JavascriptFunction::CallFunction<true>(function, origEntryPoint, args);

}

}

}

\_\_finally

{

if (fProfile)

{

if (hrOfEnterEvent != ACTIVPROF\_E\_PROFILER\_ABSENT)

{

if (functionId == -1)

{

// Check whether we have sent the Enter event or not.

if (functionEnterEventSent)

{

scriptContext->OnDispatchFunctionExit(pwszFunctionName);

if (pwszExtractedFunctionName != NULL)

{

delete[]pwszExtractedFunctionName;

}

}

}

else

{

scriptContext->OnFunctionExit(scriptId, functionId);

}

}

scriptContext->GetThreadContext()->SetIsProfilingUserCode(isProfilingUserCode); // Restore IsProfilingUserCode state

}

if (scriptContext->IsInDebugMode())

{

scriptContext->GetDebugContext()->GetProbeContainer()->EndRecordingCall(aReturn, function);

}

}

return aReturn;

}

// Part of ProfileModeThunk which is called in debug mode (debug or debug & profile).

Var ScriptContext::ProfileModeThunk\_DebugModeWrapper(JavascriptFunction\* function, ScriptContext\* scriptContext, JavascriptMethod entryPoint, Arguments& args)

{

AutoRegisterIgnoreExceptionWrapper autoWrapper(scriptContext->GetThreadContext());

Var aReturn = HelperOrLibraryMethodWrapper<true>(scriptContext, [=] {

return JavascriptFunction::CallFunction<true>(function, entryPoint, args);

});

return aReturn;

}

HRESULT ScriptContext::OnScriptCompiled(PROFILER\_TOKEN scriptId, PROFILER\_SCRIPT\_TYPE type, IUnknown \*pIDebugDocumentContext)

{

// TODO : can we do a delay send of these events or can we send a event before doing all this stuff that could calculate overhead?

Assert(m\_pProfileCallback != NULL);

OUTPUT\_TRACE(Js::ScriptProfilerPhase, L"ScriptContext::OnScriptCompiled scriptId : %d, ScriptType : %d\n", scriptId, type);

HRESULT hr = S\_OK;

if ((type == PROFILER\_SCRIPT\_TYPE\_NATIVE && m\_fTraceNativeFunctionCall) ||

(type != PROFILER\_SCRIPT\_TYPE\_NATIVE && m\_fTraceFunctionCall))

{

m\_inProfileCallback = TRUE;

hr = m\_pProfileCallback->ScriptCompiled(scriptId, type, pIDebugDocumentContext);

m\_inProfileCallback = FALSE;

}

return hr;

}

HRESULT ScriptContext::OnFunctionCompiled(

PROFILER\_TOKEN functionId,

PROFILER\_TOKEN scriptId,

const WCHAR \*pwszFunctionName,

const WCHAR \*pwszFunctionNameHint,

IUnknown \*pIDebugDocumentContext)

{

Assert(m\_pProfileCallback != NULL);

#ifdef ENABLE\_DEBUG\_CONFIG\_OPTIONS

if (scriptId != BuiltInFunctionsScriptId || Js::Configuration::Global.flags.Verbose)

{

OUTPUT\_TRACE(Js::ScriptProfilerPhase, L"ScriptContext::OnFunctionCompiled scriptId : %d, functionId : %d, FunctionName : %s, FunctionNameHint : %s\n", scriptId, functionId, pwszFunctionName, pwszFunctionNameHint);

}

#endif

HRESULT hr = S\_OK;

if ((scriptId == BuiltInFunctionsScriptId && m\_fTraceNativeFunctionCall) ||

(scriptId != BuiltInFunctionsScriptId && m\_fTraceFunctionCall))

{

m\_inProfileCallback = TRUE;

hr = m\_pProfileCallback->FunctionCompiled(functionId, scriptId, pwszFunctionName, pwszFunctionNameHint, pIDebugDocumentContext);

m\_inProfileCallback = FALSE;

}

return hr;

}

HRESULT ScriptContext::OnFunctionEnter(PROFILER\_TOKEN scriptId, PROFILER\_TOKEN functionId)

{

if (m\_pProfileCallback == NULL)

{

return ACTIVPROF\_E\_PROFILER\_ABSENT;

}

OUTPUT\_VERBOSE\_TRACE(Js::ScriptProfilerPhase, L"ScriptContext::OnFunctionEnter scriptId : %d, functionId : %d\n", scriptId, functionId);

HRESULT hr = S\_OK;

if ((scriptId == BuiltInFunctionsScriptId && m\_fTraceNativeFunctionCall) ||

(scriptId != BuiltInFunctionsScriptId && m\_fTraceFunctionCall))

{

m\_inProfileCallback = TRUE;

hr = m\_pProfileCallback->OnFunctionEnter(scriptId, functionId);

m\_inProfileCallback = FALSE;

}

return hr;

}

HRESULT ScriptContext::OnFunctionExit(PROFILER\_TOKEN scriptId, PROFILER\_TOKEN functionId)

{

if (m\_pProfileCallback == NULL)

{

return ACTIVPROF\_E\_PROFILER\_ABSENT;

}

OUTPUT\_VERBOSE\_TRACE(Js::ScriptProfilerPhase, L"ScriptContext::OnFunctionExit scriptId : %d, functionId : %d\n", scriptId, functionId);

HRESULT hr = S\_OK;

if ((scriptId == BuiltInFunctionsScriptId && m\_fTraceNativeFunctionCall) ||

(scriptId != BuiltInFunctionsScriptId && m\_fTraceFunctionCall))

{

m\_inProfileCallback = TRUE;

hr = m\_pProfileCallback->OnFunctionExit(scriptId, functionId);

m\_inProfileCallback = FALSE;

}

return hr;

}

HRESULT ScriptContext::FunctionExitSenderThunk(PROFILER\_TOKEN functionId, PROFILER\_TOKEN scriptId, ScriptContext \*pScriptContext)

{

return pScriptContext->OnFunctionExit(scriptId, functionId);

}

HRESULT ScriptContext::FunctionExitByNameSenderThunk(const wchar\_t \*pwszFunctionName, ScriptContext \*pScriptContext)

{

return pScriptContext->OnDispatchFunctionExit(pwszFunctionName);

}

Js::PropertyId ScriptContext::GetFunctionNumber(JavascriptMethod entryPoint)

{

return (m\_pBuiltinFunctionIdMap == NULL) ? -1 : m\_pBuiltinFunctionIdMap->Lookup(entryPoint, -1);

}

HRESULT ScriptContext::RegisterLibraryFunction(const wchar\_t \*pwszObjectName, const wchar\_t \*pwszFunctionName, Js::PropertyId functionPropertyId, JavascriptMethod entryPoint)

{

#if DEBUG

const wchar\_t \*pwszObjectNameFromProperty = const\_cast<wchar\_t \*>(GetPropertyName(functionPropertyId)->GetBuffer());

if (GetPropertyName(functionPropertyId)->IsSymbol())

{

// The spec names functions whose property is a well known symbol as the description from the symbol

// wrapped in square brackets, so verify by skipping past first bracket

Assert(!wcsncmp(pwszFunctionName + 1, pwszObjectNameFromProperty, wcslen(pwszObjectNameFromProperty)));

Assert(wcslen(pwszFunctionName) == wcslen(pwszObjectNameFromProperty) + 2);

}

else

{

Assert(!wcscmp(pwszFunctionName, pwszObjectNameFromProperty));

}

Assert(m\_pBuiltinFunctionIdMap != NULL);

#endif

// Create the propertyId as object.functionName if it is not global function

// the global functions would be recognized by just functionName

// e.g. with functionName, toString, depending on objectName, it could be Object.toString, or Date.toString

wchar\_t szTempName[70];

if (pwszObjectName != NULL)

{

// Create name as "object.function"

swprintf\_s(szTempName, 70, L"%s.%s", pwszObjectName, pwszFunctionName);

functionPropertyId = GetOrAddPropertyIdTracked(szTempName, (uint)wcslen(szTempName));

}

Js::PropertyId cachedFunctionId;

bool keyFound = m\_pBuiltinFunctionIdMap->TryGetValue(entryPoint, &cachedFunctionId);

if (keyFound)

{

// Entry point is already in the map

if (cachedFunctionId != functionPropertyId)

{

// This is the scenario where we could be using same function for multiple builtin functions

// e.g. Error.toString, WinRTError.toString etc.

// We would ignore these extra entrypoints because while profiling, identifying which object's toString is too costly for its worth

return S\_OK;

}

// else is the scenario where map was created by earlier profiling session and we are yet to send function compiled for this session

}

else

{

#if DBG

m\_pBuiltinFunctionIdMap->MapUntil([&](JavascriptMethod, Js::PropertyId propertyId) -> bool

{

if (functionPropertyId == propertyId)

{

Assert(false);

return true;

}

return false;

});

#endif

// throws, this must always be in a function that handles OOM

m\_pBuiltinFunctionIdMap->Add(entryPoint, functionPropertyId);

}

// Use name with "Object." if its not a global function

if (pwszObjectName != NULL)

{

return OnFunctionCompiled(functionPropertyId, BuiltInFunctionsScriptId, szTempName, NULL, NULL);

}

else

{

return OnFunctionCompiled(functionPropertyId, BuiltInFunctionsScriptId, pwszFunctionName, NULL, NULL);

}

}

void ScriptContext::BindReference(void \* addr)

{

Assert(!this->isClosed);

Assert(this->guestArena);

Assert(recycler->IsValidObject(addr));

#if DBG

Assert(!bindRef.ContainsKey(addr)); // Make sure we don't bind the same pointer twice

bindRef.AddNew(addr);

#endif

if (bindRefChunkCurrent == bindRefChunkEnd)

{

bindRefChunkCurrent = AnewArrayZ(this->guestArena, void \*, ArenaAllocator::ObjectAlignment / sizeof(void \*));

bindRefChunkEnd = bindRefChunkCurrent + ArenaAllocator::ObjectAlignment / sizeof(void \*);

}

Assert((bindRefChunkCurrent + 1) <= bindRefChunkEnd);

\*bindRefChunkCurrent = addr;

bindRefChunkCurrent++;

#ifdef RECYCLER\_PERF\_COUNTERS

this->bindReferenceCount++;

RECYCLER\_PERF\_COUNTER\_INC(BindReference);

#endif

}

#ifdef PROFILE\_STRINGS

StringProfiler\* ScriptContext::GetStringProfiler()

{

return stringProfiler;

}

#endif

void ScriptContext::FreeLoopBody(void\* address)

{

#if ENABLE\_NATIVE\_CODEGEN

FreeNativeCodeGenAllocation(this, address);

#endif

}

void ScriptContext::FreeFunctionEntryPoint(Js::JavascriptMethod method)

{

#if ENABLE\_NATIVE\_CODEGEN

FreeNativeCodeGenAllocation(this, method);

#endif

}

void ScriptContext::RegisterAsScriptContextWithInlineCaches()

{

if (this->entryInScriptContextWithInlineCachesRegistry == nullptr)

{

DoRegisterAsScriptContextWithInlineCaches();

}

}

void ScriptContext::DoRegisterAsScriptContextWithInlineCaches()

{

Assert(this->entryInScriptContextWithInlineCachesRegistry == nullptr);

// this call may throw OOM

this->entryInScriptContextWithInlineCachesRegistry = threadContext->RegisterInlineCacheScriptContext(this);

}

void ScriptContext::RegisterAsScriptContextWithIsInstInlineCaches()

{

if (this->entryInScriptContextWithIsInstInlineCachesRegistry == nullptr)

{

DoRegisterAsScriptContextWithIsInstInlineCaches();

}

}

bool ScriptContext::IsRegisteredAsScriptContextWithIsInstInlineCaches()

{

return this->entryInScriptContextWithIsInstInlineCachesRegistry != nullptr;

}

void ScriptContext::DoRegisterAsScriptContextWithIsInstInlineCaches()

{

Assert(this->entryInScriptContextWithIsInstInlineCachesRegistry == nullptr);

// this call may throw OOM

this->entryInScriptContextWithIsInstInlineCachesRegistry = threadContext->RegisterIsInstInlineCacheScriptContext(this);

}

void ScriptContext::RegisterProtoInlineCache(InlineCache \*pCache, PropertyId propId)

{

hasRegisteredInlineCache = true;

threadContext->RegisterProtoInlineCache(pCache, propId);

}

void ScriptContext::InvalidateProtoCaches(const PropertyId propertyId)

{

threadContext->InvalidateProtoInlineCaches(propertyId);

// Because setter inline caches get registered in the store field chain, we must invalidate that

// chain whenever we invalidate the proto chain.

threadContext->InvalidateStoreFieldInlineCaches(propertyId);

#if ENABLE\_NATIVE\_CODEGEN

threadContext->InvalidatePropertyGuards(propertyId);

#endif

threadContext->InvalidateProtoTypePropertyCaches(propertyId);

}

void ScriptContext::InvalidateAllProtoCaches()

{

threadContext->InvalidateAllProtoInlineCaches();

// Because setter inline caches get registered in the store field chain, we must invalidate that

// chain whenever we invalidate the proto chain.

threadContext->InvalidateAllStoreFieldInlineCaches();

#if ENABLE\_NATIVE\_CODEGEN

threadContext->InvalidateAllPropertyGuards();

#endif

threadContext->InvalidateAllProtoTypePropertyCaches();

}

void ScriptContext::RegisterStoreFieldInlineCache(InlineCache \*pCache, PropertyId propId)

{

hasRegisteredInlineCache = true;

threadContext->RegisterStoreFieldInlineCache(pCache, propId);

}

void ScriptContext::InvalidateStoreFieldCaches(const PropertyId propertyId)

{

threadContext->InvalidateStoreFieldInlineCaches(propertyId);

#if ENABLE\_NATIVE\_CODEGEN

threadContext->InvalidatePropertyGuards(propertyId);

#endif

}

void ScriptContext::InvalidateAllStoreFieldCaches()

{

threadContext->InvalidateAllStoreFieldInlineCaches();

}

void ScriptContext::RegisterIsInstInlineCache(Js::IsInstInlineCache \* cache, Js::Var function)

{

Assert(JavascriptFunction::FromVar(function)->GetScriptContext() == this);

hasRegisteredIsInstInlineCache = true;

threadContext->RegisterIsInstInlineCache(cache, function);

}

#if DBG

bool ScriptContext::IsIsInstInlineCacheRegistered(Js::IsInstInlineCache \* cache, Js::Var function)

{

return threadContext->IsIsInstInlineCacheRegistered(cache, function);

}

#endif

void ScriptContext::CleanSourceListInternal(bool calledDuringMark)

{

bool fCleanupDocRequired = false;

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

{

if (this->sourceList->IsItemValid(i))

{

RecyclerWeakReference<Utf8SourceInfo>\* sourceInfoWeakRef = this->sourceList->Item(i);

Utf8SourceInfo\* strongRef = nullptr;

if (calledDuringMark)

{

strongRef = sourceInfoWeakRef->FastGet();

}

else

{

strongRef = sourceInfoWeakRef->Get();

}

if (strongRef == nullptr)

{

this->sourceList->RemoveAt(i);

fCleanupDocRequired = true;

}

}

}

// If the sourceList got changed, in we need to refresh the nondebug document list in the profiler mode.

if (fCleanupDocRequired && m\_pProfileCallback != NULL)

{

Assert(CleanupDocumentContext != NULL);

CleanupDocumentContext(this);

}

}

void ScriptContext::ClearScriptContextCaches()

{

// Prevent reentrancy for the following work, which is not required to be done on every call to this function including

// reentrant calls

if (this->isPerformingNonreentrantWork)

{

return;

}

class AutoCleanup

{

private:

ScriptContext \*const scriptContext;

public:

AutoCleanup(ScriptContext \*const scriptContext) : scriptContext(scriptContext)

{

scriptContext->isPerformingNonreentrantWork = true;

}

~AutoCleanup()

{

scriptContext->isPerformingNonreentrantWork = false;

}

} autoCleanup(this);

if (this->isScriptContextActuallyClosed)

{

return;

}

Assert(this->guestArena);

Assert(this->cache);

if (EnableEvalMapCleanup())

{

// The eval map is not re-entrant, so make sure it's not in the middle of adding an entry

// Also, don't clean the eval map if the debugger is attached

if (!this->IsInDebugMode())

{

if (this->cache->evalCacheDictionary != nullptr)

{

this->CleanDynamicFunctionCache<Js::EvalCacheTopLevelDictionary>(this->cache->evalCacheDictionary->GetDictionary());

}

if (this->cache->indirectEvalCacheDictionary != nullptr)

{

this->CleanDynamicFunctionCache<Js::EvalCacheTopLevelDictionary>(this->cache->indirectEvalCacheDictionary->GetDictionary());

}

if (this->cache->newFunctionCache != nullptr)

{

this->CleanDynamicFunctionCache<Js::NewFunctionCache>(this->cache->newFunctionCache);

}

if (this->hostScriptContext != nullptr)

{

this->hostScriptContext->CleanDynamicCodeCache();

}

}

}

if (REGEX\_CONFIG\_FLAG(DynamicRegexMruListSize) > 0)

{

GetDynamicRegexMap()->RemoveRecentlyUnusedItems();

}

CleanSourceListInternal(true);

}

void ScriptContext::ClearInlineCaches()

{

Assert(this->entryInScriptContextWithInlineCachesRegistry != nullptr);

// For persistent inline caches, we assume here that all thread context's invalidation lists

// will be reset, such that all invalidationListSlotPtr will get zeroed. We will not be zeroing

// this field here to preserve the free list, which uses the field to link caches together.

GetInlineCacheAllocator()->ZeroAll();

this->entryInScriptContextWithInlineCachesRegistry = nullptr; // caller will remove us from the thread context

this->hasRegisteredInlineCache = false;

}

void ScriptContext::ClearIsInstInlineCaches()

{

Assert(entryInScriptContextWithIsInstInlineCachesRegistry != nullptr);

GetIsInstInlineCacheAllocator()->ZeroAll();

this->entryInScriptContextWithIsInstInlineCachesRegistry = nullptr; // caller will remove us from the thread context.

this->hasRegisteredIsInstInlineCache = false;

}

#ifdef PERSISTENT\_INLINE\_CACHES

void ScriptContext::ClearInlineCachesWithDeadWeakRefs()

{

// Review: I should be able to assert this here just like in ClearInlineCaches.

Assert(this->entryInScriptContextWithInlineCachesRegistry != nullptr);

GetInlineCacheAllocator()->ClearCachesWithDeadWeakRefs(this->recycler);

Assert(GetInlineCacheAllocator()->HasNoDeadWeakRefs(this->recycler));

}

#endif

#if ENABLE\_NATIVE\_CODEGEN

void ScriptContext::RegisterConstructorCache(Js::PropertyId propertyId, Js::ConstructorCache\* cache)

{

this->threadContext->RegisterConstructorCache(propertyId, cache);

}

#endif

void ScriptContext::RegisterPrototypeChainEnsuredToHaveOnlyWritableDataPropertiesScriptContext()

{

Assert(!IsClosed());

if (registeredPrototypeChainEnsuredToHaveOnlyWritableDataPropertiesScriptContext == nullptr)

{

DoRegisterPrototypeChainEnsuredToHaveOnlyWritableDataPropertiesScriptContext();

}

}

void ScriptContext::DoRegisterPrototypeChainEnsuredToHaveOnlyWritableDataPropertiesScriptContext()

{

Assert(!IsClosed());

Assert(registeredPrototypeChainEnsuredToHaveOnlyWritableDataPropertiesScriptContext == nullptr);

// this call may throw OOM

registeredPrototypeChainEnsuredToHaveOnlyWritableDataPropertiesScriptContext = threadContext->RegisterPrototypeChainEnsuredToHaveOnlyWritableDataPropertiesScriptContext(this);

}

void ScriptContext::ClearPrototypeChainEnsuredToHaveOnlyWritableDataPropertiesCaches()

{

Assert(registeredPrototypeChainEnsuredToHaveOnlyWritableDataPropertiesScriptContext != nullptr);

javascriptLibrary->NoPrototypeChainsAreEnsuredToHaveOnlyWritableDataProperties();

// Caller will unregister the script context from the thread context

registeredPrototypeChainEnsuredToHaveOnlyWritableDataPropertiesScriptContext = nullptr;

}

JavascriptString \* ScriptContext::GetLastNumberToStringRadix10(double value)

{

if (value == lastNumberToStringRadix10)

{

return cache->lastNumberToStringRadix10String;

}

return nullptr;

}

void

ScriptContext::SetLastNumberToStringRadix10(double value, JavascriptString \* str)

{

lastNumberToStringRadix10 = value;

cache->lastNumberToStringRadix10String = str;

}

bool ScriptContext::GetLastUtcTimeFromStr(JavascriptString \* str, double& dbl)

{

Assert(str != nullptr);

if (str != cache->lastUtcTimeFromStrString)

{

if (cache->lastUtcTimeFromStrString == nullptr

|| !JavascriptString::Equals(str, cache->lastUtcTimeFromStrString))

{

return false;

}

}

dbl = lastUtcTimeFromStr;

return true;

}

void

ScriptContext::SetLastUtcTimeFromStr(JavascriptString \* str, double value)

{

lastUtcTimeFromStr = value;

cache->lastUtcTimeFromStrString = str;

}

#if ENABLE\_NATIVE\_CODEGEN

BOOL ScriptContext::IsNativeAddress(void \* codeAddr)

{

PreReservedVirtualAllocWrapper \*preReservedVirtualAllocWrapper = this->threadContext->GetPreReservedVirtualAllocator();

if (preReservedVirtualAllocWrapper->IsPreReservedRegionPresent())

{

if (preReservedVirtualAllocWrapper->IsInRange(codeAddr))

{

Assert(!this->IsDynamicInterpreterThunk(codeAddr));

return true;

}

else if (this->threadContext->IsAllJITCodeInPreReservedRegion())

{

return false;

}

}

// Try locally first and then all script context on the thread

//Slow path

return IsNativeFunctionAddr(this, codeAddr) || this->threadContext->IsNativeAddress(codeAddr);

}

#endif

bool ScriptContext::SetDispatchProfile(bool fSet, JavascriptMethod dispatchInvoke)

{

if (!fSet)

{

this->javascriptLibrary->SetDispatchProfile(false, dispatchInvoke);

return true;

}

else if (m\_fTraceDomCall)

{

this->javascriptLibrary->SetDispatchProfile(true, dispatchInvoke);

return true;

}

return false;

}

HRESULT ScriptContext::OnDispatchFunctionEnter(const WCHAR \*pwszFunctionName)

{

if (m\_pProfileCallback2 == NULL)

{

return ACTIVPROF\_E\_PROFILER\_ABSENT;

}

HRESULT hr = S\_OK;

if (m\_fTraceDomCall)

{

m\_inProfileCallback = TRUE;

hr = m\_pProfileCallback2->OnFunctionEnterByName(pwszFunctionName, PROFILER\_SCRIPT\_TYPE\_DOM);

m\_inProfileCallback = FALSE;

}

return hr;

}

HRESULT ScriptContext::OnDispatchFunctionExit(const WCHAR \*pwszFunctionName)

{

if (m\_pProfileCallback2 == NULL)

{

return ACTIVPROF\_E\_PROFILER\_ABSENT;

}

HRESULT hr = S\_OK;

if (m\_fTraceDomCall)

{

m\_inProfileCallback = TRUE;

hr = m\_pProfileCallback2->OnFunctionExitByName(pwszFunctionName, PROFILER\_SCRIPT\_TYPE\_DOM);

m\_inProfileCallback = FALSE;

}

return hr;

}

void ScriptContext::SetBuiltInLibraryFunction(JavascriptMethod entryPoint, JavascriptFunction\* function)

{

if (!isClosed)

{

if (builtInLibraryFunctions == NULL)

{

Assert(this->recycler);

builtInLibraryFunctions = RecyclerNew(this->recycler, BuiltInLibraryFunctionMap, this->recycler);

BindReference(builtInLibraryFunctions);

}

builtInLibraryFunctions->Item(entryPoint, function);

}

}

JavascriptFunction\* ScriptContext::GetBuiltInLibraryFunction(JavascriptMethod entryPoint)

{

JavascriptFunction \* function = NULL;

if (builtInLibraryFunctions)

{

builtInLibraryFunctions->TryGetValue(entryPoint, &function);

}

return function;

}

#ifdef ENABLE\_DOM\_FAST\_PATH

DOMFastPathIRHelperMap\* ScriptContext::EnsureDOMFastPathIRHelperMap()

{

if (domFastPathIRHelperMap == nullptr)

{

// Anew throws if it OOMs, so the caller into this function needs to handle that exception

domFastPathIRHelperMap = Anew(GeneralAllocator(), DOMFastPathIRHelperMap,

GeneralAllocator(), 17); // initial capacity set to 17; unlikely to grow much bigger.

}

return domFastPathIRHelperMap;

}

#endif

#if ENABLE\_PROFILE\_INFO

void ScriptContext::AddDynamicProfileInfo(FunctionBody \* functionBody, WriteBarrierPtr<DynamicProfileInfo>\* dynamicProfileInfo)

{

Assert(functionBody->GetScriptContext() == this);

Assert(functionBody->HasValidSourceInfo());

DynamicProfileInfo \* newDynamicProfileInfo = \*dynamicProfileInfo;

// If it is a dynamic script - we should create a profile info bound to the threadContext for its lifetime.

SourceContextInfo\* sourceContextInfo = functionBody->GetSourceContextInfo();

SourceDynamicProfileManager\* profileManager = sourceContextInfo->sourceDynamicProfileManager;

if (sourceContextInfo->IsDynamic())

{

if (profileManager != nullptr)

{

// There is an in-memory cache and dynamic profile info is coming from there

if (newDynamicProfileInfo == nullptr)

{

newDynamicProfileInfo = DynamicProfileInfo::New(this->GetRecycler(), functionBody, true /\* persistsAcrossScriptContexts \*/);

profileManager->UpdateDynamicProfileInfo(functionBody->GetLocalFunctionId(), newDynamicProfileInfo);

\*dynamicProfileInfo = newDynamicProfileInfo;

}

profileManager->MarkAsExecuted(functionBody->GetLocalFunctionId());

newDynamicProfileInfo->UpdateFunctionInfo(functionBody, this->GetRecycler());

}

else

{

if (newDynamicProfileInfo == nullptr)

{

newDynamicProfileInfo = functionBody->AllocateDynamicProfile();

}

\*dynamicProfileInfo = newDynamicProfileInfo;

}

}

else

{

if (newDynamicProfileInfo == nullptr)

{

newDynamicProfileInfo = functionBody->AllocateDynamicProfile();

\*dynamicProfileInfo = newDynamicProfileInfo;

}

Assert(functionBody->interpretedCount == 0);

#if DBG\_DUMP || defined(DYNAMIC\_PROFILE\_STORAGE) || defined(RUNTIME\_DATA\_COLLECTION)

if (profileInfoList)

{

profileInfoList->Prepend(this->GetRecycler(), newDynamicProfileInfo);

}

#endif

if (!startupComplete)

{

Assert(profileManager);

profileManager->MarkAsExecuted(functionBody->GetLocalFunctionId());

}

}

Assert(\*dynamicProfileInfo != nullptr);

}

#endif

CharClassifier const \* ScriptContext::GetCharClassifier(void) const

{

return this->charClassifier;

}

void ScriptContext::OnStartupComplete()

{

JS\_ETW(EventWriteJSCRIPT\_ON\_STARTUP\_COMPLETE(this));

SaveStartupProfileAndRelease();

}

void ScriptContext::SaveStartupProfileAndRelease(bool isSaveOnClose)

{

if (!startupComplete && this->cache->sourceContextInfoMap)

{

#if ENABLE\_PROFILE\_INFO

this->cache->sourceContextInfoMap->Map([&](DWORD\_PTR dwHostSourceContext, SourceContextInfo\* info)

{

Assert(info->sourceDynamicProfileManager);

uint bytesWritten = info->sourceDynamicProfileManager->SaveToProfileCacheAndRelease(info);

if (bytesWritten > 0)

{

JS\_ETW(EventWriteJSCRIPT\_PROFILE\_SAVE(info->dwHostSourceContext, this, bytesWritten, isSaveOnClose));

OUTPUT\_TRACE(Js::DynamicProfilePhase, L"Profile saving succeeded\n");

}

});

#endif

}

startupComplete = true;

}

void ScriptContextBase::ClearGlobalObject()

{

#if ENABLE\_NATIVE\_CODEGEN

ScriptContext\* scriptContext = static\_cast<ScriptContext\*>(this);

Assert(scriptContext->IsClosedNativeCodeGenerator());

#endif

globalObject = nullptr;

javascriptLibrary = nullptr;

}

void ScriptContext::SetFastDOMenabled()

{

fastDOMenabled = true; Assert(globalObject->GetDirectHostObject() != NULL);

}

#if DYNAMIC\_INTERPRETER\_THUNK

JavascriptMethod ScriptContext::GetNextDynamicAsmJsInterpreterThunk(PVOID\* ppDynamicInterpreterThunk)

{

#ifdef ASMJS\_PLAT

return (JavascriptMethod)this->asmJsInterpreterThunkEmitter->GetNextThunk(ppDynamicInterpreterThunk);

#else

\_\_debugbreak();

return nullptr;

#endif

}

JavascriptMethod ScriptContext::GetNextDynamicInterpreterThunk(PVOID\* ppDynamicInterpreterThunk)

{

return (JavascriptMethod)this->interpreterThunkEmitter->GetNextThunk(ppDynamicInterpreterThunk);

}

BOOL ScriptContext::IsDynamicInterpreterThunk(void\* address)

{

return this->interpreterThunkEmitter->IsInRange(address);

}

void ScriptContext::ReleaseDynamicInterpreterThunk(BYTE\* address, bool addtoFreeList)

{

this->interpreterThunkEmitter->Release(address, addtoFreeList);

}

void ScriptContext::ReleaseDynamicAsmJsInterpreterThunk(BYTE\* address, bool addtoFreeList)

{

#ifdef ASMJS\_PLAT

this->asmJsInterpreterThunkEmitter->Release(address, addtoFreeList);

#else

Assert(UNREACHED);

#endif

}

#endif

bool ScriptContext::IsExceptionWrapperForBuiltInsEnabled()

{

return ScriptContext::IsExceptionWrapperForBuiltInsEnabled(this);

}

// static

bool ScriptContext::IsExceptionWrapperForBuiltInsEnabled(ScriptContext\* scriptContext)

{

Assert(scriptContext);

return CONFIG\_FLAG(EnableContinueAfterExceptionWrappersForBuiltIns);

}

bool ScriptContext::IsExceptionWrapperForHelpersEnabled(ScriptContext\* scriptContext)

{

Assert(scriptContext);

return CONFIG\_FLAG(EnableContinueAfterExceptionWrappersForHelpers);

}

void ScriptContextBase::SetGlobalObject(GlobalObject \*globalObject)

{

#if DBG

ScriptContext\* scriptContext = static\_cast<ScriptContext\*>(this);

Assert(scriptContext->IsCloningGlobal() && !this->globalObject);

#endif

this->globalObject = globalObject;

}

void ConvertKey(const FastEvalMapString& src, EvalMapString& dest)

{

dest.str = src.str;

dest.strict = src.strict;

dest.moduleID = src.moduleID;

dest.hash = TAGHASH((hash\_t)dest.str);

}

void ScriptContext::PrintStats()

{

#if ENABLE\_PROFILE\_INFO

#if DBG\_DUMP

DynamicProfileInfo::DumpScriptContext(this);

#endif

#ifdef RUNTIME\_DATA\_COLLECTION

DynamicProfileInfo::DumpScriptContextToFile(this);

#endif

#endif

#ifdef PROFILE\_TYPES

if (Configuration::Global.flags.ProfileTypes)

{

ProfileTypes();

}

#endif

#ifdef PROFILE\_BAILOUT\_RECORD\_MEMORY

if (Configuration::Global.flags.ProfileBailOutRecordMemory)

{

Output::Print(L"CodeSize: %6d\nBailOutRecord Size: %6d\nLocalOffsets Size: %6d\n", codeSize, bailOutRecordBytes, bailOutOffsetBytes);

}

#endif

#ifdef PROFILE\_OBJECT\_LITERALS

if (Configuration::Global.flags.ProfileObjectLiteral)

{

ProfileObjectLiteral();

}

#endif

#ifdef PROFILE\_STRINGS

if (stringProfiler != nullptr)

{

stringProfiler->PrintAll();

Adelete(MiscAllocator(), stringProfiler);

stringProfiler = nullptr;

}

#endif

#ifdef PROFILE\_MEM

if (profileMemoryDump && MemoryProfiler::IsTraceEnabled())

{

MemoryProfiler::PrintAll();

#ifdef PROFILE\_RECYCLER\_ALLOC

if (Js::Configuration::Global.flags.TraceMemory.IsEnabled(Js::AllPhase)

|| Js::Configuration::Global.flags.TraceMemory.IsEnabled(Js::RunPhase))

{

GetRecycler()->PrintAllocStats();

}

#endif

}

#endif

#if DBG\_DUMP

if (PHASE\_STATS1(Js::ByteCodePhase))

{

Output::Print(L" Total Bytecode size: <%d, %d, %d> = %d\n",

byteCodeDataSize,

byteCodeAuxiliaryDataSize,

byteCodeAuxiliaryContextDataSize,

byteCodeDataSize + byteCodeAuxiliaryDataSize + byteCodeAuxiliaryContextDataSize);

}

if (Configuration::Global.flags.BytecodeHist)

{

Output::Print(L"ByteCode Histogram\n");

Output::Print(L"\n");

uint total = 0;

uint unique = 0;

for (int j = 0; j < (int)OpCode::ByteCodeLast; j++)

{

total += byteCodeHistogram[j];

if (byteCodeHistogram[j] > 0)

{

unique++;

}

}

Output::Print(L"%9u Total executed ops\n", total);

Output::Print(L"\n");

uint max = UINT\_MAX;

double pctcume = 0.0;

while (true)

{

uint upper = 0;

int index = -1;

for (int j = 0; j < (int)OpCode::ByteCodeLast; j++)

{

if (OpCodeUtil::IsValidOpcode((OpCode)j) && byteCodeHistogram[j] > upper && byteCodeHistogram[j] < max)

{

index = j;

upper = byteCodeHistogram[j];

}

}

if (index == -1)

{

break;

}

max = byteCodeHistogram[index];

for (OpCode j = (OpCode)0; j < OpCode::ByteCodeLast; j++)

{

if (OpCodeUtil::IsValidOpcode(j) && max == byteCodeHistogram[(int)j])

{

double pct = ((double)max) / total;

pctcume += pct;

Output::Print(L"%9u %5.1lf %5.1lf %04x %s\n", max, pct \* 100, pctcume \* 100, j, OpCodeUtil::GetOpCodeName(j));

}

}

}

Output::Print(L"\n");

Output::Print(L"Unique opcodes: %d\n", unique);

}

#endif

#if ENABLE\_NATIVE\_CODEGEN

#ifdef BGJIT\_STATS

// We do not care about small script contexts without much activity - unless t

if (PHASE\_STATS1(Js::BGJitPhase) && (this->interpretedCount > 50 || Js::Configuration::Global.flags.IsEnabled(Js::ForceFlag)))

{

uint loopJitCodeUsed = 0;

uint bucketSize1 = 20;

uint bucketSize2 = 100;

uint size1CutOffbucketId = 4;

uint totalBuckets[15] = { 0 };

uint nativeCodeBuckets[15] = { 0 };

uint usedNativeCodeBuckets[15] = { 0 };

uint rejits[15] = { 0 };

uint zeroInterpretedFunctions = 0;

uint oneInterpretedFunctions = 0;

uint nonZeroBytecodeFunctions = 0;

Output::Print(L"Script Context: 0x%p Url: %s\n", this, this->url);

FunctionBody\* anyFunctionBody = this->FindFunction([](FunctionBody\* body) { return body != nullptr; });

if (anyFunctionBody)

{

OUTPUT\_VERBOSE\_STATS(Js::BGJitPhase, L"Function list\n");

OUTPUT\_VERBOSE\_STATS(Js::BGJitPhase, L"===============================\n");

OUTPUT\_VERBOSE\_STATS(Js::BGJitPhase, L"%-24s, %-8s, %-10s, %-10s, %-10s, %-10s, %-10s\n", L"Function", L"InterpretedCount", L"ByteCodeInLoopSize", L"ByteCodeSize", L"IsJitted", L"IsUsed", L"NativeCodeSize");

this->MapFunction([&](FunctionBody\* body)

{

bool isNativeCode = false;

// Filtering interpreted count lowers a lot of noise

if (body->interpretedCount > 1 || Js::Configuration::Global.flags.IsEnabled(Js::ForceFlag))

{

body->MapEntryPoints([&](uint entryPointIndex, FunctionEntryPointInfo\* entryPoint)

{

wchar\_t debugStringBuffer[MAX\_FUNCTION\_BODY\_DEBUG\_STRING\_SIZE];

char rejit = entryPointIndex > 0 ? '\*' : ' ';

isNativeCode = entryPoint->IsNativeCode() | isNativeCode;

OUTPUT\_VERBOSE\_STATS(Js::BGJitPhase, L"%-20s %16s %c, %8d , %10d , %10d, %-10s, %-10s, %10d\n",

body->GetExternalDisplayName(),

body->GetDebugNumberSet(debugStringBuffer),

rejit,

body->interpretedCount,

body->GetByteCodeInLoopCount(),

body->GetByteCodeCount(),

entryPoint->IsNativeCode() ? L"Jitted" : L"Interpreted",

body->GetNativeEntryPointUsed() ? L"Used" : L"NotUsed",

entryPoint->IsNativeCode() ? entryPoint->GetCodeSize() : 0);

});

}

if (body->interpretedCount == 0)

{

zeroInterpretedFunctions++;

if (body->GetByteCodeCount() > 0)

{

nonZeroBytecodeFunctions++;

}

}

else if (body->interpretedCount == 1)

{

oneInterpretedFunctions++;

}

// Generate a histogram using interpreted counts.

uint bucket;

uint intrpCount = body->interpretedCount;

if (intrpCount < 100)

{

bucket = intrpCount / bucketSize1;

}

else if (intrpCount < 1000)

{

bucket = size1CutOffbucketId + intrpCount / bucketSize2;

}

else

{

bucket = \_countof(totalBuckets) - 1;

}

// Explicitly assume that the bucket count is less than the following counts (which are all equal)

// This is because min will return \_countof(totalBuckets) - 1 if the count exceeds \_countof(totalBuckets) - 1.

\_\_analysis\_assume(bucket < \_countof(totalBuckets));

\_\_analysis\_assume(bucket < \_countof(nativeCodeBuckets));

\_\_analysis\_assume(bucket < \_countof(usedNativeCodeBuckets));

\_\_analysis\_assume(bucket < \_countof(rejits));

totalBuckets[bucket]++;

if (isNativeCode)

{

nativeCodeBuckets[bucket]++;

if (body->GetNativeEntryPointUsed())

{

usedNativeCodeBuckets[bucket]++;

}

if (body->HasRejit())

{

rejits[bucket]++;

}

}

body->MapLoopHeaders([&](uint loopNumber, LoopHeader\* header)

{

wchar\_t loopBodyName[256];

body->GetLoopBodyName(loopNumber, loopBodyName, \_countof(loopBodyName));

header->MapEntryPoints([&](int index, LoopEntryPointInfo \* entryPoint)

{

if (entryPoint->IsNativeCode())

{

wchar\_t debugStringBuffer[MAX\_FUNCTION\_BODY\_DEBUG\_STRING\_SIZE];

char rejit = index > 0 ? '\*' : ' ';

OUTPUT\_VERBOSE\_STATS(Js::BGJitPhase, L"%-20s %16s %c, %8d , %10d , %10d, %-10s, %-10s, %10d\n",

loopBodyName,

body->GetDebugNumberSet(debugStringBuffer),

rejit,

header->interpretCount,

header->GetByteCodeCount(),

header->GetByteCodeCount(),

L"Jitted",

entryPoint->IsUsed() ? L"Used" : L"NotUsed",

entryPoint->GetCodeSize());

if (entryPoint->IsUsed())

{

loopJitCodeUsed++;

}

}

});

});

});

}

Output::Print(L"\*\* SpeculativelyJitted: %6d FunctionsJitted: %6d JittedUsed: %6d Usage:%f ByteCodesJitted: %6d JitCodeUsed: %6d Usage: %f \n",

speculativeJitCount, funcJITCount, funcJitCodeUsed, ((float)(funcJitCodeUsed) / funcJITCount) \* 100, bytecodeJITCount, jitCodeUsed, ((float)(jitCodeUsed) / bytecodeJITCount) \* 100);

Output::Print(L"\*\* LoopJITCount: %6d LoopJitCodeUsed: %6d Usage: %f\n",

loopJITCount, loopJitCodeUsed, ((float)loopJitCodeUsed / loopJITCount) \* 100);

Output::Print(L"\*\* TotalInterpretedCalls: %6d MaxFuncInterp: %6d InterpretedHighPri: %6d \n",

interpretedCount, maxFuncInterpret, interpretedCallsHighPri);

Output::Print(L"\*\* ZeroInterpretedFunctions: %6d OneInterpretedFunctions: %6d ZeroInterpretedWithNonZeroBytecode: %6d \n ", zeroInterpretedFunctions, oneInterpretedFunctions, nonZeroBytecodeFunctions);

Output::Print(L"\*\* %-24s : %-10s %-10s %-10s %-10s %-10s\n", L"InterpretedCounts", L"Total", L"NativeCode", L"Used", L"Usage", L"Rejits");

uint low = 0;

uint high = 0;

for (uint i = 0; i < \_countof(totalBuckets); i++)

{

low = high;

if (i <= size1CutOffbucketId)

{

high = low + bucketSize1;

}

else if (i < (\_countof(totalBuckets) - 1))

{

high = low + bucketSize2; }

else

{

high = 100000;

}

Output::Print(L"\*\* %10d - %10d : %10d %10d %10d %7.2f %10d\n", low, high, totalBuckets[i], nativeCodeBuckets[i], usedNativeCodeBuckets[i], ((float)usedNativeCodeBuckets[i] / nativeCodeBuckets[i]) \* 100, rejits[i]);

}

Output::Print(L"\n\n");

}

#endif

#ifdef REJIT\_STATS

if (PHASE\_STATS1(Js::ReJITPhase))

{

uint totalBailouts = 0;

uint totalRejits = 0;

WCHAR buf[256];

// Dump bailout data.

Output::Print(L"%-40s %6s\n", L"Bailout Reason,", L"Count");

bailoutReasonCounts->Map([&totalBailouts](uint kind, uint val) {

WCHAR buf[256];

totalBailouts += val;

if (val != 0)

{

swprintf\_s(buf, L"%S,", GetBailOutKindName((IR::BailOutKind)kind));

Output::Print(L"%-40s %6d\n", buf, val);

}

});

Output::Print(L"%-40s %6d\n", L"TOTAL,", totalBailouts);

Output::Print(L"\n\n");

// Dump rejit data.

Output::Print(L"%-40s %6s\n", L"Rejit Reason,", L"Count");

for (uint i = 0; i < NumRejitReasons; ++i)

{

totalRejits += rejitReasonCounts[i];

if (rejitReasonCounts[i] != 0)

{

swprintf\_s(buf, L"%S,", RejitReasonNames[i]);

Output::Print(L"%-40s %6d\n", buf, rejitReasonCounts[i]);

}

}

Output::Print(L"%-40s %6d\n", L"TOTAL,", totalRejits);

Output::Print(L"\n\n");

// If in verbose mode, dump data for each FunctionBody

if (Configuration::Global.flags.Verbose && rejitStatsMap != NULL)

{

// Aggregated data

Output::Print(L"%-30s %14s %14s\n", L"Function (#),", L"Bailout Count,", L"Rejit Count");

rejitStatsMap->Map([](Js::FunctionBody const \*body, RejitStats \*stats, RecyclerWeakReference<const Js::FunctionBody> const\*) {

wchar\_t debugStringBuffer[MAX\_FUNCTION\_BODY\_DEBUG\_STRING\_SIZE];

for (uint i = 0; i < NumRejitReasons; ++i)

stats->m\_totalRejits += stats->m\_rejitReasonCounts[i];

stats->m\_bailoutReasonCounts->Map([stats](uint kind, uint val) {

stats->m\_totalBailouts += val;

});

WCHAR buf[256];

swprintf\_s(buf, L"%s (%s),", body->GetExternalDisplayName(), (const\_cast<Js::FunctionBody\*>(body))->GetDebugNumberSet(debugStringBuffer)); //TODO Kount

Output::Print(L"%-30s %14d, %14d\n", buf, stats->m\_totalBailouts, stats->m\_totalRejits);

});

Output::Print(L"\n\n");

// Per FunctionBody data

rejitStatsMap->Map([](Js::FunctionBody const \*body, RejitStats \*stats, RecyclerWeakReference<const Js::FunctionBody> const \*) {

wchar\_t debugStringBuffer[MAX\_FUNCTION\_BODY\_DEBUG\_STRING\_SIZE];

WCHAR buf[256];

swprintf\_s(buf, L"%s (%s),", body->GetExternalDisplayName(), (const\_cast<Js::FunctionBody\*>(body))->GetDebugNumberSet(debugStringBuffer)); //TODO Kount

Output::Print(L"%-30s\n\n", buf);

// Dump bailout data

if (stats->m\_totalBailouts != 0)

{

Output::Print(L"%10sBailouts:\n", L"");

stats->m\_bailoutReasonCounts->Map([](uint kind, uint val) {

if (val != 0)

{

WCHAR buf[256];

swprintf\_s(buf, L"%S,", GetBailOutKindName((IR::BailOutKind)kind));

Output::Print(L"%10s%-40s %6d\n", L"", buf, val);

}

});

}

Output::Print(L"\n");

// Dump rejit data.

if (stats->m\_totalRejits != 0)

{

Output::Print(L"%10sRejits:\n", L"");

for (uint i = 0; i < NumRejitReasons; ++i)

{

if (stats->m\_rejitReasonCounts[i] != 0)

{

swprintf\_s(buf, L"%S,", RejitReasonNames[i]);

Output::Print(L"%10s%-40s %6d\n", L"", buf, stats->m\_rejitReasonCounts[i]);

}

}

Output::Print(L"\n\n");

}

});

}

}

#endif

#ifdef FIELD\_ACCESS\_STATS

if (PHASE\_STATS1(Js::ObjTypeSpecPhase))

{

FieldAccessStats globalStats;

if (this->fieldAccessStatsByFunctionNumber != nullptr)

{

this->fieldAccessStatsByFunctionNumber->Map([&globalStats](uint functionNumber, FieldAccessStatsEntry\* entry)

{

FieldAccessStats functionStats;

entry->stats.Map([&functionStats](FieldAccessStatsPtr entryPointStats)

{

functionStats.Add(entryPointStats);

});

if (PHASE\_VERBOSE\_STATS1(Js::ObjTypeSpecPhase))

{

FunctionBody\* functionBody = entry->functionBodyWeakRef->Get();

const wchar\_t\* functionName = functionBody != nullptr ? functionBody->GetDisplayName() : L"<unknown>";

Output::Print(L"FieldAccessStats: function %s (#%u): inline cache stats:\n", functionName, functionNumber);

Output::Print(L" overall: total %u, no profile info %u\n", functionStats.totalInlineCacheCount, functionStats.noInfoInlineCacheCount);

Output::Print(L" mono: total %u, empty %u, cloned %u\n",

functionStats.monoInlineCacheCount, functionStats.emptyMonoInlineCacheCount, functionStats.clonedMonoInlineCacheCount);

Output::Print(L" poly: total %u (high %u, low %u), null %u, empty %u, ignored %u, disabled %u, equivalent %u, non-equivalent %u, cloned %u\n",

functionStats.polyInlineCacheCount, functionStats.highUtilPolyInlineCacheCount, functionStats.lowUtilPolyInlineCacheCount,

functionStats.nullPolyInlineCacheCount, functionStats.emptyPolyInlineCacheCount, functionStats.ignoredPolyInlineCacheCount, functionStats.disabledPolyInlineCacheCount,

functionStats.equivPolyInlineCacheCount, functionStats.nonEquivPolyInlineCacheCount, functionStats.clonedPolyInlineCacheCount);

}

globalStats.Add(&functionStats);

});

}

Output::Print(L"FieldAccessStats: totals\n");

Output::Print(L" overall: total %u, no profile info %u\n", globalStats.totalInlineCacheCount, globalStats.noInfoInlineCacheCount);

Output::Print(L" mono: total %u, empty %u, cloned %u\n",

globalStats.monoInlineCacheCount, globalStats.emptyMonoInlineCacheCount, globalStats.clonedMonoInlineCacheCount);

Output::Print(L" poly: total %u (high %u, low %u), null %u, empty %u, ignored %u, disabled %u, equivalent %u, non-equivalent %u, cloned %u\n",

globalStats.polyInlineCacheCount, globalStats.highUtilPolyInlineCacheCount, globalStats.lowUtilPolyInlineCacheCount,

globalStats.nullPolyInlineCacheCount, globalStats.emptyPolyInlineCacheCount, globalStats.ignoredPolyInlineCacheCount, globalStats.disabledPolyInlineCacheCount,

globalStats.equivPolyInlineCacheCount, globalStats.nonEquivPolyInlineCacheCount, globalStats.clonedPolyInlineCacheCount);

}

#endif

#ifdef MISSING\_PROPERTY\_STATS

if (PHASE\_STATS1(Js::MissingPropertyCachePhase))

{

Output::Print(L"MissingPropertyStats: hits = %d, misses = %d, cache attempts = %d.\n",

this->missingPropertyHits, this->missingPropertyMisses, this->missingPropertyCacheAttempts);

}

#endif

#ifdef INLINE\_CACHE\_STATS

if (PHASE\_STATS1(Js::PolymorphicInlineCachePhase))

{

Output::Print(L"%s,%s,%s,%s,%s,%s,%s,%s,%s\n", L"Function", L"Property", L"Kind", L"Accesses", L"Misses", L"Miss Rate", L"Collisions", L"Collision Rate", L"Slot Count");

cacheDataMap->Map([this](Js::PolymorphicInlineCache const \*cache, CacheData \*data) {

wchar\_t debugStringBuffer[MAX\_FUNCTION\_BODY\_DEBUG\_STRING\_SIZE];

uint total = data->hits + data->misses;

wchar\_t const \*propName = this->threadContext->GetPropertyName(data->propertyId)->GetBuffer();

wchar funcName[1024];

swprintf\_s(funcName, L"%s (%s)", cache->functionBody->GetExternalDisplayName(), cache->functionBody->GetDebugNumberSet(debugStringBuffer));

Output::Print(L"%s,%s,%s,%d,%d,%f,%d,%f,%d\n",

funcName,

propName,

data->isGetCache ? L"get" : L"set",

total,

data->misses,

static\_cast<float>(data->misses) / total,

data->collisions,

static\_cast<float>(data->collisions) / total,

cache->GetSize()

);

});

}

#endif

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

if (regexStatsDatabase != 0)

regexStatsDatabase->Print(GetRegexDebugWriter());

#endif

OUTPUT\_STATS(Js::EmitterPhase, L"Script Context: 0x%p Url: %s\n", this, this->url);

OUTPUT\_STATS(Js::EmitterPhase, L" Total thread committed code size = %d\n", this->GetThreadContext()->GetCodeSize());

OUTPUT\_STATS(Js::ParsePhase, L"Script Context: 0x%p Url: %s\n", this, this->url);

OUTPUT\_STATS(Js::ParsePhase, L" Total ThreadContext source size %d\n", this->GetThreadContext()->GetSourceSize());

#endif

#ifdef ENABLE\_BASIC\_TELEMETRY

if (this->telemetry != nullptr)

{

// If an exception (e.g. out-of-memory) happens during InitializeAllocations then `this->telemetry` will be null and the Close method will still be called, hence this guard expression.

this->telemetry->OutputPrint();

}

#endif

Output::Flush();

}

void ScriptContext::SetNextPendingClose(ScriptContext \* nextPendingClose) {

Assert(this->nextPendingClose == nullptr && nextPendingClose != nullptr);

this->nextPendingClose = nextPendingClose;

}

#ifdef ENABLE\_MUTATION\_BREAKPOINT

bool ScriptContext::HasMutationBreakpoints()

{

if (this->GetDebugContext() != nullptr && this->GetDebugContext()->GetProbeContainer() != nullptr)

{

return this->GetDebugContext()->GetProbeContainer()->HasMutationBreakpoints();

}

return false;

}

void ScriptContext::InsertMutationBreakpoint(Js::MutationBreakpoint \*mutationBreakpoint)

{

this->GetDebugContext()->GetProbeContainer()->InsertMutationBreakpoint(mutationBreakpoint);

}

#endif

#ifdef REJIT\_STATS

void ScriptContext::LogDataForFunctionBody(Js::FunctionBody \*body, uint idx, bool isRejit)

{

if (rejitStatsMap == NULL)

{

rejitStatsMap = RecyclerNew(this->recycler, RejitStatsMap, this->recycler);

BindReference(rejitStatsMap);

}

RejitStats \*stats = NULL;

if (!rejitStatsMap->TryGetValue(body, &stats))

{

stats = Anew(GeneralAllocator(), RejitStats, this);

rejitStatsMap->Item(body, stats);

}

if (isRejit)

{

stats->m\_rejitReasonCounts[idx]++;

}

else

{

if (!stats->m\_bailoutReasonCounts->ContainsKey(idx))

{

stats->m\_bailoutReasonCounts->Item(idx, 1);

}

else

{

uint val = stats->m\_bailoutReasonCounts->Item(idx);

++val;

stats->m\_bailoutReasonCounts->Item(idx, val);

}

}

}

void ScriptContext::LogRejit(Js::FunctionBody \*body, uint reason)

{

Assert(reason < NumRejitReasons);

rejitReasonCounts[reason]++;

if (Js::Configuration::Global.flags.Verbose)

{

LogDataForFunctionBody(body, reason, true);

}

}

void ScriptContext::LogBailout(Js::FunctionBody \*body, uint kind)

{

if (!bailoutReasonCounts->ContainsKey(kind))

{

bailoutReasonCounts->Item(kind, 1);

}

else

{

uint val = bailoutReasonCounts->Item(kind);

++val;

bailoutReasonCounts->Item(kind, val);

}

if (Js::Configuration::Global.flags.Verbose)

{

LogDataForFunctionBody(body, kind, false);

}

}

#endif

#ifdef ENABLE\_BASIC\_TELEMETRY

ScriptContextTelemetry& ScriptContext::GetTelemetry()

{

return \*this->telemetry;

}

bool ScriptContext::HasTelemetry()

{

return this->telemetry != nullptr;

}

#endif

bool ScriptContext::IsInNonDebugMode() const

{

if (this->debugContext != nullptr)

{

return this->GetDebugContext()->IsInNonDebugMode();

}

return true;

}

bool ScriptContext::IsInSourceRundownMode() const

{

if (this->debugContext != nullptr)

{

return this->GetDebugContext()->IsInSourceRundownMode();

}

return false;

}

bool ScriptContext::IsInDebugMode() const

{

if (this->debugContext != nullptr)

{

return this->GetDebugContext()->IsInDebugMode();

}

return false;

}

bool ScriptContext::IsInDebugOrSourceRundownMode() const

{

if (this->debugContext != nullptr)

{

return this->GetDebugContext()->IsInDebugOrSourceRundownMode();

}

return false;

}

#ifdef INLINE\_CACHE\_STATS

void ScriptContext::LogCacheUsage(Js::PolymorphicInlineCache \*cache, bool isGetter, Js::PropertyId propertyId, bool hit, bool collision)

{

if (cacheDataMap == NULL)

{

cacheDataMap = RecyclerNew(this->recycler, CacheDataMap, this->recycler);

BindReference(cacheDataMap);

}

CacheData \*data = NULL;

if (!cacheDataMap->TryGetValue(cache, &data))

{

data = Anew(GeneralAllocator(), CacheData);

cacheDataMap->Item(cache, data);

data->isGetCache = isGetter;

data->propertyId = propertyId;

}

Assert(data->isGetCache == isGetter);

Assert(data->propertyId == propertyId);

if (hit)

{

data->hits++;

}

else

{

data->misses++;

}

if (collision)

{

data->collisions++;

}

}

#endif

#ifdef FIELD\_ACCESS\_STATS

void ScriptContext::RecordFieldAccessStats(FunctionBody\* functionBody, FieldAccessStatsPtr fieldAccessStats)

{

Assert(fieldAccessStats != nullptr);

if (!PHASE\_STATS1(Js::ObjTypeSpecPhase))

{

return;

}

FieldAccessStatsEntry\* entry;

if (!this->fieldAccessStatsByFunctionNumber->TryGetValue(functionBody->GetFunctionNumber(), &entry))

{

RecyclerWeakReference<FunctionBody>\* functionBodyWeakRef;

this->recycler->FindOrCreateWeakReferenceHandle(functionBody, &functionBodyWeakRef);

entry = RecyclerNew(this->recycler, FieldAccessStatsEntry, functionBodyWeakRef, this->recycler);

this->fieldAccessStatsByFunctionNumber->AddNew(functionBody->GetFunctionNumber(), entry);

}

entry->stats.Prepend(fieldAccessStats);

}

#endif

#ifdef MISSING\_PROPERTY\_STATS

void ScriptContext::RecordMissingPropertyMiss()

{

this->missingPropertyMisses++;

}

void ScriptContext::RecordMissingPropertyHit()

{

this->missingPropertyHits++;

}

void ScriptContext::RecordMissingPropertyCacheAttempt()

{

this->missingPropertyCacheAttempts++;

}

#endif

bool ScriptContext::IsIntConstPropertyOnGlobalObject(Js::PropertyId propId)

{

return intConstPropsOnGlobalObject->ContainsKey(propId);

}

void ScriptContext::TrackIntConstPropertyOnGlobalObject(Js::PropertyId propertyId)

{

intConstPropsOnGlobalObject->AddNew(propertyId);

}

bool ScriptContext::IsIntConstPropertyOnGlobalUserObject(Js::PropertyId propertyId)

{

return intConstPropsOnGlobalUserObject->ContainsKey(propertyId) != NULL;

}

void ScriptContext::TrackIntConstPropertyOnGlobalUserObject(Js::PropertyId propertyId)

{

intConstPropsOnGlobalUserObject->AddNew(propertyId);

}

void ScriptContext::AddCalleeSourceInfoToList(Utf8SourceInfo\* sourceInfo)

{

Assert(sourceInfo);

RecyclerWeakReference<Js::Utf8SourceInfo>\* sourceInfoWeakRef = nullptr;

this->GetRecycler()->FindOrCreateWeakReferenceHandle(sourceInfo, &sourceInfoWeakRef);

Assert(sourceInfoWeakRef);

if (!calleeUtf8SourceInfoList)

{

Recycler \*recycler = this->GetRecycler();

calleeUtf8SourceInfoList.Root(RecyclerNew(recycler, CalleeSourceList, recycler), recycler);

}

if (!calleeUtf8SourceInfoList->Contains(sourceInfoWeakRef))

{

calleeUtf8SourceInfoList->Add(sourceInfoWeakRef);

}

}

#ifdef ENABLE\_JS\_ETW

void ScriptContext::EmitStackTraceEvent(\_\_in UINT64 operationID, \_\_in USHORT maxFrameCount, bool emitV2AsyncStackEvent)

{

// If call root level is zero, there is no EntryExitRecord and the stack walk will fail.

if (GetThreadContext()->GetCallRootLevel() == 0)

{

return;

}

Assert(EventEnabledJSCRIPT\_STACKTRACE() || EventEnabledJSCRIPT\_ASYNCCAUSALITY\_STACKTRACE\_V2() || PHASE\_TRACE1(Js::StackFramesEventPhase));

BEGIN\_TEMP\_ALLOCATOR(tempAllocator, this, L"StackTraceEvent")

{

JsUtil::List<StackFrameInfo, ArenaAllocator> stackFrames(tempAllocator);

Js::JavascriptStackWalker walker(this);

unsigned short nameBufferLength = 0;

Js::StringBuilder<ArenaAllocator> nameBuffer(tempAllocator);

nameBuffer.Reset();

OUTPUT\_TRACE(Js::StackFramesEventPhase, L"\nPosting stack trace via ETW:\n");

ushort frameCount = walker.WalkUntil((ushort)maxFrameCount, [&](Js::JavascriptFunction\* function, ushort frameIndex) -> bool

{

ULONG lineNumber = 0;

LONG columnNumber = 0;

UINT32 methodIdOrNameId = 0;

UINT8 isFrameIndex = 0; // FALSE

const WCHAR\* name = nullptr;

if (function->IsScriptFunction() && !function->IsLibraryCode())

{

Js::FunctionBody \* functionBody = function->GetFunctionBody();

functionBody->GetLineCharOffset(walker.GetByteCodeOffset(), &lineNumber, &columnNumber);

methodIdOrNameId = EtwTrace::GetFunctionId(functionBody);

name = functionBody->GetExternalDisplayName();

}

else

{

if (function->IsScriptFunction())

{

name = function->GetFunctionBody()->GetExternalDisplayName();

}

else

{

name = walker.GetCurrentNativeLibraryEntryName();

}

ushort nameLen = ProcessNameAndGetLength(&nameBuffer, name);

methodIdOrNameId = nameBufferLength;

// Keep track of the current length of the buffer. The next nameIndex will be at this position (+1 for each '\\', '\"', and ';' character added above).

nameBufferLength += nameLen;

isFrameIndex = 1; // TRUE;

}

StackFrameInfo frame((DWORD\_PTR)function->GetScriptContext(),

(UINT32)lineNumber,

(UINT32)columnNumber,

methodIdOrNameId,

isFrameIndex);

OUTPUT\_TRACE(Js::StackFramesEventPhase, L"Frame : (%s : %u) (%s), LineNumber : %u, ColumnNumber : %u\n",

(isFrameIndex == 1) ? (L"NameBufferIndex") : (L"MethodID"),

methodIdOrNameId,

name,

lineNumber,

columnNumber);

stackFrames.Add(frame);

return false;

});

Assert(frameCount == (ushort)stackFrames.Count());

if (frameCount > 0) // No need to emit event if there are no script frames.

{

auto nameBufferString = nameBuffer.Detach();

if (nameBufferLength > 0)

{

// Account for the terminating null character.

nameBufferLength++;

}

if (emitV2AsyncStackEvent)

{

JS\_ETW(EventWriteJSCRIPT\_ASYNCCAUSALITY\_STACKTRACE\_V2(operationID, frameCount, nameBufferLength, sizeof(StackFrameInfo), &stackFrames.Item(0), nameBufferString));

}

else

{

JS\_ETW(EventWriteJSCRIPT\_STACKTRACE(operationID, frameCount, nameBufferLength, sizeof(StackFrameInfo), &stackFrames.Item(0), nameBufferString));

}

}

}

END\_TEMP\_ALLOCATOR(tempAllocator, this);

OUTPUT\_FLUSH();

}

#endif

// Info: Append sourceString to stringBuilder after escaping charToEscape with escapeChar.

// "SomeBadly\0Formed\0String" => "SomeBadly\\\0Formed\\\0String"

// Parameters: stringBuilder - The Js::StringBuilder to which we should append sourceString.

// sourceString - The string we want to escape and append to stringBuilder.

// sourceStringLen - Length of sourceString.

// escapeChar - Char to use for escaping.

// charToEscape - The char which we should escape with escapeChar.

// Returns: Count of chars written to stringBuilder.

charcount\_t ScriptContext::AppendWithEscapeCharacters(Js::StringBuilder<ArenaAllocator>\* stringBuilder, const WCHAR\* sourceString, charcount\_t sourceStringLen, WCHAR escapeChar, WCHAR charToEscape)

{

const WCHAR\* charToEscapePtr = wcschr(sourceString, charToEscape);

charcount\_t charsPadding = 0;

// Only escape characters if sourceString contains one.

if (charToEscapePtr)

{

charcount\_t charsWritten = 0;

charcount\_t charsToAppend = 0;

while (charToEscapePtr)

{

charsToAppend = static\_cast<charcount\_t>(charToEscapePtr - sourceString) - charsWritten;

stringBuilder->Append(sourceString + charsWritten, charsToAppend);

stringBuilder->Append(escapeChar);

stringBuilder->Append(charToEscape);

// Keep track of this extra escapeChar character so we can update the buffer length correctly below.

charsPadding++;

// charsWritten is a count of the chars from sourceString which have been written - not count of chars Appended to stringBuilder.

charsWritten += charsToAppend + 1;

// Find next charToEscape.

charToEscapePtr++;

charToEscapePtr = wcschr(charToEscapePtr, charToEscape);

}

// Append the final part of the string if there is any left after the final charToEscape.

if (charsWritten != sourceStringLen)

{

charsToAppend = sourceStringLen - charsWritten;

stringBuilder->Append(sourceString + charsWritten, charsToAppend);

}

}

else

{

stringBuilder->AppendSz(sourceString);

}

return sourceStringLen + charsPadding;

}

/\*static\*/

ushort ScriptContext::ProcessNameAndGetLength(Js::StringBuilder<ArenaAllocator>\* nameBuffer, const WCHAR\* name)

{

Assert(nameBuffer);

Assert(name);

ushort nameLen = (ushort)wcslen(name);

// Surround each function name with quotes and escape any quote characters in the function name itself with '\\'.

nameBuffer->Append('\"');

// Adjust nameLen based on any escape characters we added to escape the '\"' in name.

nameLen = (unsigned short)AppendWithEscapeCharacters(nameBuffer, name, nameLen, '\\', '\"');

nameBuffer->AppendCppLiteral(L"\";");

// Add 3 padding characters here - one for initial '\"' character, too.

nameLen += 3;

return nameLen;

}

} // End namespace Js

SRCINFO\* SRCINFO::Clone(Js::ScriptContext\* scriptContext) const

{

SRCINFO\* srcInfo;

if (this->sourceContextInfo->dwHostSourceContext == Js::Constants::NoHostSourceContext &&

this->dlnHost == 0 && this->ulColumnHost == 0 && this->ulCharOffset == 0 &&

this->ichMinHost == 0 && this->ichLimHost == 0 && this->grfsi == 0)

{

srcInfo = const\_cast<SRCINFO\*>(scriptContext->GetModuleSrcInfo(this->moduleID));

}

else

{

SourceContextInfo\* sourceContextInfo = this->sourceContextInfo->Clone(scriptContext);

srcInfo = SRCINFO::Copy(scriptContext->GetRecycler(), this);

srcInfo->sourceContextInfo = sourceContextInfo;

}

return srcInfo;

}

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

// 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

#include "activprof.h"

#if DBG || ENABLE\_REGEX\_CONFIG\_OPTIONS || defined(PROFILE\_STRINGS)

#define NEED\_MISC\_ALLOCATOR

#endif

#define BuiltInFunctionsScriptId 0

class NativeCodeGenerator;

class BackgroundParser;

struct IActiveScriptDirect;

#ifdef ENABLE\_BASIC\_TELEMETRY

class ScriptContextTelemetry;

#endif

namespace Js

{

class ScriptContext;

class ScriptEditQuery;

class MutationBreakpoint;

class StringProfiler;

class DebugContext;

struct HaltCallback;

struct DebuggerOptionsCallback;

}

// Created for every source buffer passed by host.

// This structure has all the details.

class SRCINFO

{

// We currently don't free SRCINFO object so we don't want to add extra variables here.

// In future, when we do make it freeable and will be able to allocate more than one per Module,

// we can move variables m\_isGlobalFunc and m\_isEval from FunctionBody.cpp here.

public:

SourceContextInfo \* sourceContextInfo;

ULONG dlnHost; // Line number passed to ParseScriptText

ULONG ulColumnHost; // Column number on the line where the parse script text started

ULONG lnMinHost; // Line offset of first host-supplied line

ULONG ichMinHost; // Range of host supplied characters

ULONG ichLimHost;

ULONG ulCharOffset; // Char offset of the source text relative to the document. (Populated using IActiveScriptContext)

Js::ModuleID moduleID;

ULONG grfsi;

static SRCINFO\* Copy(Recycler\* recycler, const SRCINFO\* srcInfo)

{

SRCINFO\* copySrcInfo = RecyclerNew(recycler, SRCINFO, \*srcInfo);

return copySrcInfo;

}

SRCINFO\* Clone(Js::ScriptContext\* scriptContext) const;

};

struct CustomExternalObjectOperations

{

size\_t offsetOfOperationsUsage;

DWORD operationFlagEquals;

DWORD operationFlagStrictEquals;

};

enum ExternalJitData

{

ExternalJitData\_CustomExternalObjectOperations

};

class HostScriptContext

{

public:

HostScriptContext(Js::ScriptContext\* inScriptContext) { this->scriptContext = inScriptContext; }

virtual void Delete() = 0;

virtual HRESULT GetPreviousHostScriptContext(\_\_deref\_out HostScriptContext\*\* ppUnkCaller) = 0;

virtual HRESULT PushHostScriptContext() = 0;

virtual void PopHostScriptContext() = 0;

virtual HRESULT SetCaller(IUnknown \*punkNew, IUnknown \*\*ppunkPrev) = 0;

virtual HRESULT GetDispatchExCaller(\_\_deref\_out void\*\* dispatchExCaller) = 0;

virtual void ReleaseDispatchExCaller(\_\_in void\* dispatchExCaler) = 0;

virtual Js::ModuleRoot \* GetModuleRoot(int moduleID) = 0;

virtual HRESULT CheckCrossDomainScriptContext(\_\_in Js::ScriptContext\* scriptContext) = 0;

virtual HRESULT GetHostContextUrl(\_\_in DWORD\_PTR hostSourceContext, \_\_out BSTR& pUrl) = 0;

virtual BOOL HasCaller() = 0;

virtual void CleanDynamicCodeCache() = 0;

virtual HRESULT VerifyDOMSecurity(Js::ScriptContext\* targetContext, Js::Var obj) = 0;

virtual HRESULT CheckEvalRestriction() = 0;

virtual HRESULT HostExceptionFromHRESULT(HRESULT hr, Js::Var\* outError) = 0;

virtual HRESULT GetExternalJitData(ExternalJitData id, void \*data) = 0;

virtual HRESULT SetDispatchInvoke(Js::JavascriptMethod dispatchInvoke) = 0;

virtual HRESULT ArrayBufferFromExternalObject(\_\_in Js::RecyclableObject \*obj,

\_\_out Js::ArrayBuffer \*\*ppArrayBuffer) = 0;

virtual Js::JavascriptError\* CreateWinRTError(IErrorInfo\* perrinfo, Js::RestrictedErrorStrings \* proerrstr) = 0;

virtual Js::JavascriptFunction\* InitializeHostPromiseContinuationFunction() = 0;

Js::ScriptContext\* GetScriptContext() { return scriptContext; }

virtual bool SetCrossSiteForFunctionType(Js::JavascriptFunction \* function) = 0;

#if DBG\_DUMP || defined(PROFILE\_EXEC) || defined(PROFILE\_MEM)

virtual void EnsureParentInfo(Js::ScriptContext\* scriptContext = NULL) = 0;

#endif

#if DBG

virtual bool IsHostCrossSiteThunk(Js::JavascriptMethod address) = 0;

#endif

private:

Js::ScriptContext\* scriptContext;

};

namespace Js

{

#pragma pack(push, 1)

struct StackFrameInfo

{

StackFrameInfo() { }

StackFrameInfo(DWORD\_PTR \_scriptContextID

, UINT32 \_sourceLocationLineNumber

, UINT32 \_sourceLocationColumnNumber

, UINT32 \_methodIDOrNameIndex

, UINT8 \_isFrameIndex)

: scriptContextID(\_scriptContextID)

, sourceLocationLineNumber(\_sourceLocationLineNumber)

, sourceLocationColumnNumber(\_sourceLocationColumnNumber)

, methodIDOrNameIndex(\_methodIDOrNameIndex)

, isFrameIndex(\_isFrameIndex)

{ }

DWORD\_PTR scriptContextID;

UINT32 sourceLocationLineNumber;

UINT32 sourceLocationColumnNumber;

UINT32 methodIDOrNameIndex;

UINT8 isFrameIndex;

};

#pragma pack(pop)

class ProjectionConfiguration

{

public:

ProjectionConfiguration() : targetVersion(0)

{

}

DWORD GetTargetVersion() const { return this->targetVersion; }

void SetTargetVersion(DWORD version) { this->targetVersion = version; }

bool IsTargetWindows8() const { return this->targetVersion == NTDDI\_WIN8; }

bool IsTargetWindowsBlueOrLater() const { return this->targetVersion >= NTDDI\_WINBLUE; }

private:

DWORD targetVersion;

};

class ScriptConfiguration

{

public:

ScriptConfiguration(const ThreadConfiguration \* const threadConfig, const bool isOptimizedForManyInstances) :

#ifdef ENABLE\_PROJECTION

HostType(Configuration::Global.flags.HostType),

WinRTConstructorAllowed(Configuration::Global.flags.WinRTConstructorAllowed),

#endif

NoNative(Configuration::Global.flags.NoNative),

isOptimizedForManyInstances(isOptimizedForManyInstances),

threadConfig(threadConfig)

{

}

// Version

bool SupportsES3() const { return true; }

bool SupportsES3Extensions() const {

#ifdef ENABLE\_PROJECTION

return HostType != HostTypeApplication;

#else

return true;

#endif

}

#define FORWARD\_THREAD\_CONFIG(flag) inline bool flag() const { return threadConfig->flag(); }

#define FLAG(threadFlag, globalFlag) FORWARD\_THREAD\_CONFIG(threadFlag)

#define FLAG\_RELEASE(threadFlag, globalFlag) FORWARD\_THREAD\_CONFIG(threadFlag)

#include "../Base/ThreadConfigFlagsList.h"

#undef FLAG\_RELEASE

#undef FLAG

#undef FORWARD\_THREAD\_CONFIG

bool SupportsCollectGarbage() const { return true; }

bool IsTypedArrayEnabled() const { return true; }

bool BindDeferredPidRefs() const { return IsLetAndConstEnabled(); }

void ForceNoNative() { this->NoNative = true; }

void ForceNative() { this->NoNative = false; }

bool IsNoNative() const { return this->NoNative; }

void SetCanOptimizeGlobalLookupFlag(BOOL f){ this->fCanOptimizeGlobalLookup = f;}

BOOL CanOptimizeGlobalLookup() const { return this->fCanOptimizeGlobalLookup;}

bool IsOptimizedForManyInstances() const { return isOptimizedForManyInstances; }

bool IsBlockScopeEnabled() const { return true; }

void CopyFrom(ScriptConfiguration& other)

{

this->NoNative = other.NoNative;

this->fCanOptimizeGlobalLookup = other.fCanOptimizeGlobalLookup;

#ifdef ENABLE\_PROJECTION

this->HostType = other.HostType;

this->WinRTConstructorAllowed = other.WinRTConstructorAllowed;

this->projectionConfiguration = other.projectionConfiguration;

#endif

}

#ifdef ENABLE\_PROJECTION

Number GetHostType() const // Returns one of enum HostType values (see ConfigFlagsTable.h).

{

AssertMsg(this->HostType >= HostTypeMin && this->HostType <= HostTypeMax, "HostType value is out of valid range.");

return this->HostType;

}

ProjectionConfiguration const \* GetProjectionConfig() const

{

return &projectionConfiguration;

}

void SetHostType(long hostType) { this->HostType = hostType; }

void SetWinRTConstructorAllowed(bool allowed) { this->WinRTConstructorAllowed = allowed; }

void SetProjectionTargetVersion(DWORD version)

{

projectionConfiguration.SetTargetVersion(version);

}

bool IsWinRTEnabled() const { return (GetHostType() == Js::HostTypeApplication) || (GetHostType() == Js::HostTypeWebview); }

bool IsWinRTConstructorAllowed() const { return (GetHostType() != Js::HostTypeWebview) || this->WinRTConstructorAllowed; }

#endif

private:

// Per script configurations

bool NoNative;

BOOL fCanOptimizeGlobalLookup;

const bool isOptimizedForManyInstances;

const ThreadConfiguration \* const threadConfig;

#ifdef ENABLE\_PROJECTION

Number HostType; // One of enum HostType values (see ConfigFlagsTable.h).

bool WinRTConstructorAllowed; // whether allow constructor in webview host type. Also note that this is not a security feature.

ProjectionConfiguration projectionConfiguration;

#endif

};

struct ScriptEntryExitRecord

{

BOOL hasCaller : 1;

BOOL hasReentered : 1;

#if DBG\_DUMP

BOOL isCallRoot : 1;

#endif

#if DBG || defined(PROFILE\_EXEC)

BOOL leaveForHost : 1;

#endif

#if DBG

BOOL leaveForAsyncHostOperation : 1;

#endif

#ifdef CHECK\_STACKWALK\_EXCEPTION

BOOL ignoreStackWalkException: 1;

#endif

Js::ImplicitCallFlags savedImplicitCallFlags;

void \* returnAddrOfScriptEntryFunction;

void \* frameIdOfScriptExitFunction; // the frameAddres in x86, the return address in amd64/arm\_soc

ScriptContext \* scriptContext;

struct ScriptEntryExitRecord \* next;

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

void \* scriptEntryFS0;

#endif

#ifdef EXCEPTION\_CHECK

ExceptionType handledExceptionType;

#endif

};

static const unsigned int EvalMRUSize = 15;

typedef JsUtil::BaseDictionary<DWORD\_PTR, SourceContextInfo \*, Recycler, PowerOf2SizePolicy> SourceContextInfoMap;

typedef JsUtil::BaseDictionary<uint, SourceContextInfo \*, Recycler, PowerOf2SizePolicy> DynamicSourceContextInfoMap;

typedef JsUtil::BaseDictionary<EvalMapString, ScriptFunction\*, RecyclerNonLeafAllocator, PrimeSizePolicy> SecondLevelEvalCache;

typedef TwoLevelHashRecord<FastEvalMapString, ScriptFunction\*, SecondLevelEvalCache, EvalMapString> EvalMapRecord;

typedef JsUtil::Cache<FastEvalMapString, EvalMapRecord\*, RecyclerNonLeafAllocator, PrimeSizePolicy, JsUtil::MRURetentionPolicy<FastEvalMapString, EvalMRUSize>, FastEvalMapStringComparer> EvalCacheTopLevelDictionary;

typedef SList<Js::FunctionProxy\*, Recycler> FunctionReferenceList;

typedef JsUtil::Cache<EvalMapString, ParseableFunctionInfo\*, RecyclerNonLeafAllocator, PrimeSizePolicy, JsUtil::MRURetentionPolicy<EvalMapString, EvalMRUSize>> NewFunctionCache;

typedef JsUtil::BaseDictionary<ParseableFunctionInfo\*, ParseableFunctionInfo\*, Recycler, PrimeSizePolicy, RecyclerPointerComparer> ParseableFunctionInfoMap;

// This is the dictionary used by script context to cache the eval.

typedef TwoLevelHashDictionary<FastEvalMapString, ScriptFunction\*, EvalMapRecord, EvalCacheTopLevelDictionary, EvalMapString> EvalCacheDictionary;

struct PropertyStringMap

{

PropertyString\* strLen2[80];

\_\_inline static uint PStrMapIndex(wchar\_t ch)

{

Assert(ch >= '0' && ch <= 'z');

return ch - '0';

}

};

#ifdef ENABLE\_DOM\_FAST\_PATH

typedef JsUtil::BaseDictionary<Js::FunctionInfo\*, IR::JnHelperMethod, ArenaAllocator, PowerOf2SizePolicy> DOMFastPathIRHelperMap;

#endif

// valid if object!= NULL

struct EnumeratedObjectCache {

static const int kMaxCachedPropStrings=16;

DynamicObject\* object;

DynamicType\* type;

PropertyString\* propertyStrings[kMaxCachedPropStrings];

int validPropStrings;

};

// Holder for all cached pointers. These are allocated on a guest arena

// ensuring they cause the related objects to be pinned.

struct Cache

{

JavascriptString \* lastNumberToStringRadix10String;

EnumeratedObjectCache enumObjCache;

JavascriptString \* lastUtcTimeFromStrString;

TypePath\* rootPath;

EvalCacheDictionary\* evalCacheDictionary;

EvalCacheDictionary\* indirectEvalCacheDictionary;

NewFunctionCache\* newFunctionCache;

RegexPatternMruMap \*dynamicRegexMap;

SourceContextInfoMap\* sourceContextInfoMap; // maps host provided context cookie to the URL of the script buffer passed.

DynamicSourceContextInfoMap\* dynamicSourceContextInfoMap;

SourceContextInfo\* noContextSourceContextInfo;

SRCINFO\* noContextGlobalSourceInfo;

SRCINFO const \*\* moduleSrcInfo;

};

class ScriptContext : public ScriptContextBase

{

friend class LowererMD;

friend class RemoteScriptContext;

public:

static DWORD GetThreadContextOffset() { return offsetof(ScriptContext, threadContext); }

static DWORD GetOptimizationOverridesOffset() { return offsetof(ScriptContext, optimizationOverrides); }

static DWORD GetRecyclerOffset() { return offsetof(ScriptContext, recycler); }

static DWORD GetNumberAllocatorOffset() { return offsetof(ScriptContext, numberAllocator); }

static DWORD GetAsmIntDbValOffset() { return offsetof(ScriptContext, retAsmIntDbVal); }

ScriptContext \*next;

ScriptContext \*prev;

double retAsmIntDbVal; // stores the double & float result for Asm interpreter

AsmJsSIMDValue retAsmSimdVal; // stores raw simd result for Asm interpreter

static DWORD GetAsmSimdValOffset() { return offsetof(ScriptContext, retAsmSimdVal); }

ScriptContextOptimizationOverrideInfo optimizationOverrides;

Js::JavascriptMethod CurrentThunk;

Js::JavascriptMethod CurrentCrossSiteThunk;

Js::JavascriptMethod DeferredParsingThunk;

Js::JavascriptMethod DeferredDeserializationThunk;

Js::JavascriptMethod DispatchDefaultInvoke;

Js::JavascriptMethod DispatchProfileInoke;

typedef HRESULT (\*GetDocumentContextFunction)(

ScriptContext \*pContext,

Js::FunctionBody \*pFunctionBody,

IDebugDocumentContext \*\*ppDebugDocumentContext);

GetDocumentContextFunction GetDocumentContext;

typedef HRESULT (\*CleanupDocumentContextFunction)(ScriptContext \*pContext);

CleanupDocumentContextFunction CleanupDocumentContext;

const ScriptContextBase\* GetScriptContextBase() const { return static\_cast<const ScriptContextBase\*>(this); }

bool DoUndeferGlobalFunctions() const;

bool IsUndeclBlockVar(Var var) const { return this->javascriptLibrary->IsUndeclBlockVar(var); }

void TrackPid(const PropertyRecord\* propertyRecord);

void TrackPid(PropertyId propertyId);

bool IsTrackedPropertyId(Js::PropertyId propertyId);

void InvalidateHostObjects()

{

AssertMsg(!isClosed, "Host Object invalidation should occur before the engine is fully closed. Figure our how isClosed got set beforehand.");

isInvalidatedForHostObjects = true;

}

bool IsInvalidatedForHostObjects()

{

return isInvalidatedForHostObjects;

}

#ifdef ENABLE\_JS\_ETW

void EmitStackTraceEvent(\_\_in UINT64 operationID, \_\_in USHORT maxFrameCount, bool emitV2AsyncStackEvent);

#endif

void SetIsDiagnosticsScriptContext(bool set) { this->isDiagnosticsScriptContext = set; }

bool IsDiagnosticsScriptContext() const { return this->isDiagnosticsScriptContext; }

bool IsInNonDebugMode() const;

bool IsInSourceRundownMode() const;

bool IsInDebugMode() const;

bool IsInDebugOrSourceRundownMode() const;

bool IsRunningScript() const { return this->threadContext->GetScriptEntryExit() != nullptr; }

typedef JsUtil::List<RecyclerWeakReference<Utf8SourceInfo>\*, Recycler, false, Js::WeakRefFreeListedRemovePolicy> CalleeSourceList;

RecyclerRootPtr<CalleeSourceList> calleeUtf8SourceInfoList;

void AddCalleeSourceInfoToList(Utf8SourceInfo\* sourceInfo);

bool HaveCalleeSources() { return calleeUtf8SourceInfoList && !calleeUtf8SourceInfoList->Empty(); }

template<class TMapFunction>

void MapCalleeSources(TMapFunction map)

{

if (this->HaveCalleeSources())

{

calleeUtf8SourceInfoList->Map([&](uint i, RecyclerWeakReference<Js::Utf8SourceInfo>\* sourceInfoWeakRef)

{

if (calleeUtf8SourceInfoList->IsItemValid(i))

{

Js::Utf8SourceInfo\* sourceInfo = sourceInfoWeakRef->Get();

map(sourceInfo);

}

});

}

if (calleeUtf8SourceInfoList)

{

calleeUtf8SourceInfoList.Unroot(this->GetRecycler());

}

}

#ifdef ASMJS\_PLAT

inline AsmJsCodeGenerator\* GetAsmJsCodeGenerator() const{return asmJsCodeGenerator;}

AsmJsCodeGenerator\* InitAsmJsCodeGenerator();

#endif

bool IsExceptionWrapperForBuiltInsEnabled();

bool IsEnumerateNonUserFunctionsOnly() const { return m\_enumerateNonUserFunctionsOnly; }

bool IsTraceDomCall() const { return !!m\_fTraceDomCall; }

static bool IsExceptionWrapperForBuiltInsEnabled(ScriptContext\* scriptContext);

static bool IsExceptionWrapperForHelpersEnabled(ScriptContext\* scriptContext);

InlineCache \* GetValueOfInlineCache() const { return valueOfInlineCache;}

InlineCache \* GetToStringInlineCache() const { return toStringInlineCache; }

FunctionBody \* GetFakeGlobalFuncForUndefer() const { return fakeGlobalFuncForUndefer; }

void SetFakeGlobalFuncForUndefer(FunctionBody \* func) { fakeGlobalFuncForUndefer.Root(func, GetRecycler()); }

private:

PropertyStringMap\* propertyStrings[80];

JavascriptFunction\* GenerateRootFunction(ParseNodePtr parseTree, uint sourceIndex, Parser\* parser, ulong grfscr, CompileScriptException \* pse, const wchar\_t \*rootDisplayName);

typedef void (\*EventHandler)(ScriptContext \*);

ScriptContext \*\* entryInScriptContextWithInlineCachesRegistry;

ScriptContext \*\* entryInScriptContextWithIsInstInlineCachesRegistry;

ScriptContext \*\* registeredPrototypeChainEnsuredToHaveOnlyWritableDataPropertiesScriptContext;

ArenaAllocator generalAllocator;

#ifdef ENABLE\_BASIC\_TELEMETRY

ArenaAllocator telemetryAllocator;

#endif

ArenaAllocator dynamicProfileInfoAllocator;

InlineCacheAllocator inlineCacheAllocator;

IsInstInlineCacheAllocator isInstInlineCacheAllocator;

ArenaAllocator\* interpreterArena;

ArenaAllocator\* guestArena;

ArenaAllocator\* diagnosticArena;

void \*\* bindRefChunkCurrent;

void \*\* bindRefChunkEnd;

bool startupComplete; // Indicates if the heuristic startup phase for this script context is complete

bool isInvalidatedForHostObjects; // Indicates that we've invalidate all objects in the host so stop calling them.

bool isEnumeratingRecyclerObjects; // Indicates this scriptContext is enumerating recycler objects. Used by recycler enumerating callbacks to filter out other unrelated scriptContexts.

bool m\_enumerateNonUserFunctionsOnly; // Indicates that recycler enumeration callback will consider only non-user functions (which are built-ins, external, winrt etc).

ThreadContext\* threadContext;

TypeId directHostTypeId;

InlineCache \* valueOfInlineCache;

InlineCache \* toStringInlineCache;

typedef JsUtil::BaseHashSet<Js::PropertyId, ArenaAllocator> PropIdSetForConstProp;

PropIdSetForConstProp \* intConstPropsOnGlobalObject;

PropIdSetForConstProp \* intConstPropsOnGlobalUserObject;

void \* firstInterpreterFrameReturnAddress;

#ifdef SEPARATE\_ARENA

ArenaAllocator sourceCodeAllocator;

ArenaAllocator regexAllocator;

#endif

#ifdef NEED\_MISC\_ALLOCATOR

ArenaAllocator miscAllocator;

#endif

#if DBG

JsUtil::BaseHashSet<void \*, ArenaAllocator> bindRef;

int m\_iProfileSession;

#endif

#ifdef PROFILE\_EXEC

ScriptContextProfiler \* profiler;

bool isProfilerCreated;

bool disableProfiler;

bool ensureParentInfo;

Profiler \* CreateProfiler();

#endif

#ifdef PROFILE\_MEM

bool profileMemoryDump;

#endif

#ifdef PROFILE\_STRINGS

StringProfiler\* stringProfiler;

#endif

RecyclerRootPtr<FunctionBody> fakeGlobalFuncForUndefer;

public:

#ifdef PROFILE\_TYPES

int convertNullToSimpleCount;

int convertNullToSimpleDictionaryCount;

int convertNullToDictionaryCount;

int convertDeferredToDictionaryCount;

int convertDeferredToSimpleDictionaryCount;

int convertSimpleToDictionaryCount;

int convertSimpleToSimpleDictionaryCount;

int convertPathToDictionaryCount1;

int convertPathToDictionaryCount2;

int convertPathToDictionaryCount3;

int convertPathToDictionaryCount4;

int convertPathToSimpleDictionaryCount;

int convertSimplePathToPathCount;

int convertSimpleDictionaryToDictionaryCount;

int convertSimpleSharedDictionaryToNonSharedCount;

int convertSimpleSharedToNonSharedCount;

int simplePathTypeHandlerCount;

int pathTypeHandlerCount;

int promoteCount;

int cacheCount;

int branchCount;

int maxPathLength;

int typeCount[TypeIds\_Limit];

int instanceCount[TypeIds\_Limit];

#endif

#ifdef PROFILE\_BAILOUT\_RECORD\_MEMORY

\_\_int64 bailOutRecordBytes;

\_\_int64 bailOutOffsetBytes;

\_\_int64 codeSize;

#endif

#ifdef PROFILE\_OBJECT\_LITERALS

int objectLiteralInstanceCount;

int objectLiteralPathCount;

int objectLiteralCount[TypePath::MaxPathTypeHandlerLength];

int objectLiteralSimpleDictionaryCount;

uint32 objectLiteralMaxLength;

int objectLiteralPromoteCount;

int objectLiteralCacheCount;

int objectLiteralBranchCount;

#endif

#if DBG\_DUMP

uint byteCodeDataSize;

uint byteCodeAuxiliaryDataSize;

uint byteCodeAuxiliaryContextDataSize;

uint byteCodeHistogram[OpCode::ByteCodeLast];

uint32 forinCache;

uint32 forinNoCache;

#endif

#ifdef BGJIT\_STATS

uint interpretedCount;

uint funcJITCount;

uint loopJITCount;

uint bytecodeJITCount;

uint interpretedCallsHighPri;

uint maxFuncInterpret;

uint jitCodeUsed;

uint funcJitCodeUsed;

uint speculativeJitCount;

#endif

#ifdef REJIT\_STATS

// Used to store bailout stats

typedef JsUtil::BaseDictionary<uint, uint, ArenaAllocator> BailoutStatsMap;

struct RejitStats

{

uint \*m\_rejitReasonCounts;

BailoutStatsMap\* m\_bailoutReasonCounts;

uint m\_totalRejits;

uint m\_totalBailouts;

RejitStats(ScriptContext \*scriptContext) : m\_totalRejits(0), m\_totalBailouts(0)

{

m\_rejitReasonCounts = AnewArrayZ(scriptContext->GeneralAllocator(), uint, NumRejitReasons);

m\_bailoutReasonCounts = Anew(scriptContext->GeneralAllocator(), BailoutStatsMap, scriptContext->GeneralAllocator());

}

};

void LogDataForFunctionBody(Js::FunctionBody \*body, uint idx, bool isRejit);

void LogRejit(Js::FunctionBody \*body, uint reason);

void LogBailout(Js::FunctionBody \*body, uint kind);

// Used to centrally collect stats for all function bodies.

typedef JsUtil::WeaklyReferencedKeyDictionary<const Js::FunctionBody, RejitStats\*> RejitStatsMap;

RejitStatsMap\* rejitStatsMap;

BailoutStatsMap \*bailoutReasonCounts;

uint \*rejitReasonCounts;

#endif

#ifdef ENABLE\_BASIC\_TELEMETRY

private:

ScriptContextTelemetry\* telemetry;

public:

ScriptContextTelemetry& GetTelemetry();

bool HasTelemetry();

#endif

#ifdef INLINE\_CACHE\_STATS

// Used to store inline cache stats

struct CacheData

{

uint hits;

uint misses;

uint collisions;

bool isGetCache;

Js::PropertyId propertyId;

CacheData() : hits(0), misses(0), collisions(0), isGetCache(false), propertyId(Js::PropertyIds::\_none) { }

};

// This is a strongly referenced dictionary, since we want to know hit rates for dead caches.

typedef JsUtil::BaseDictionary<const Js::PolymorphicInlineCache\*, CacheData\*, Recycler> CacheDataMap;

CacheDataMap \*cacheDataMap;

void LogCacheUsage(Js::PolymorphicInlineCache \*cache, bool isGet, Js::PropertyId propertyId, bool hit, bool collision);

#endif

#ifdef FIELD\_ACCESS\_STATS

typedef SList<FieldAccessStatsPtr, Recycler> FieldAccessStatsList;

struct FieldAccessStatsEntry

{

RecyclerWeakReference<FunctionBody>\* functionBodyWeakRef;

FieldAccessStatsList stats;

FieldAccessStatsEntry(RecyclerWeakReference<FunctionBody>\* functionBodyWeakRef, Recycler\* recycler)

: functionBodyWeakRef(functionBodyWeakRef), stats(recycler) {}

};

typedef JsUtil::BaseDictionary<uint, FieldAccessStatsEntry\*, Recycler> FieldAccessStatsByFunctionNumberMap;

FieldAccessStatsByFunctionNumberMap\* fieldAccessStatsByFunctionNumber;

void RecordFieldAccessStats(FunctionBody\* functionBody, FieldAccessStatsPtr fieldAccessStats);

#endif

#ifdef MISSING\_PROPERTY\_STATS

int missingPropertyMisses;

int missingPropertyHits;

int missingPropertyCacheAttempts;

void RecordMissingPropertyMiss();

void RecordMissingPropertyHit();

void RecordMissingPropertyCacheAttempt();

#endif

bool IsIntConstPropertyOnGlobalObject(Js::PropertyId propId);

void TrackIntConstPropertyOnGlobalObject(Js::PropertyId propId);

bool IsIntConstPropertyOnGlobalUserObject(Js::PropertyId propertyId);

void TrackIntConstPropertyOnGlobalUserObject(Js::PropertyId propertyId);

private:

//

// Regex globals

//

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

UnifiedRegex::DebugWriter\* regexDebugWriter;

UnifiedRegex::RegexStatsDatabase\* regexStatsDatabase;

#endif

UnifiedRegex::TrigramAlphabet\* trigramAlphabet;

UnifiedRegex::RegexStacks \*regexStacks;

FunctionReferenceList\* dynamicFunctionReference;

uint dynamicFunctionReferenceDepth;

JsUtil::Stack<Var>\* operationStack;

Recycler\* recycler;

RecyclerJavascriptNumberAllocator numberAllocator;

ScriptConfiguration config;

CharClassifier \*charClassifier;

// DisableJIT-TODO: Switch this to Dynamic thunk ifdef instead

#if ENABLE\_NATIVE\_CODEGEN

InterpreterThunkEmitter\* interpreterThunkEmitter;

BackgroundParser \*backgroundParser;

#ifdef ASMJS\_PLAT

InterpreterThunkEmitter\* asmJsInterpreterThunkEmitter;

AsmJsCodeGenerator\* asmJsCodeGenerator;

typedef JsUtil::BaseDictionary<void \*, SList<AsmJsScriptFunction \*>\*, ArenaAllocator> AsmFunctionMap;

AsmFunctionMap\* asmJsEnvironmentMap;

ArenaAllocator\* debugTransitionAlloc;

#endif

NativeCodeGenerator\* nativeCodeGen;

#endif

TIME\_ZONE\_INFORMATION timeZoneInfo;

uint lastTimeZoneUpdateTickCount;

DaylightTimeHelper daylightTimeHelper;

HostScriptContext \* hostScriptContext;

HaltCallback\* scriptEngineHaltCallback;

EventHandler scriptStartEventHandler;

EventHandler scriptEndEventHandler;

#ifdef FAULT\_INJECTION

EventHandler disposeScriptByFaultInjectionEventHandler;

#endif

JsUtil::BaseDictionary<uint, JavascriptString \*, ArenaAllocator> integerStringMap;

double lastNumberToStringRadix10;

double lastUtcTimeFromStr;

#if ENABLE\_PROFILE\_INFO

bool referencesSharedDynamicSourceContextInfo;

#endif

// We could delay the actual close after callRootLevel is 0.

// this is to indicate the actual close is called once only.

bool isScriptContextActuallyClosed;

#if DBG

bool isInitialized;

bool isCloningGlobal;

#endif

bool fastDOMenabled;

bool hasRegisteredInlineCache;

bool hasRegisteredIsInstInlineCache;

bool deferredBody;

bool isPerformingNonreentrantWork;

bool isDiagnosticsScriptContext; // mentions that current script context belongs to the diagnostics OM.

size\_t sourceSize;

void CleanSourceListInternal(bool calledDuringMark);

typedef JsUtil::List<RecyclerWeakReference<Utf8SourceInfo>\*, Recycler, false, Js::FreeListedRemovePolicy> SourceList;

RecyclerRootPtr<SourceList> sourceList;

IActiveScriptProfilerHeapEnum\* heapEnum;

// Profiler Probes

// In future these can be list of callbacks ?

// Profiler Callback information

IActiveScriptProfilerCallback \*m\_pProfileCallback;

BOOL m\_fTraceFunctionCall;

BOOL m\_fTraceNativeFunctionCall;

DWORD m\_dwEventMask;

IActiveScriptProfilerCallback2 \*m\_pProfileCallback2;

BOOL m\_fTraceDomCall;

BOOL m\_inProfileCallback;

#if ENABLE\_PROFILE\_INFO

#if DBG\_DUMP || defined(DYNAMIC\_PROFILE\_STORAGE) || defined(RUNTIME\_DATA\_COLLECTION)

RecyclerRootPtr<SListBase<DynamicProfileInfo \*>> profileInfoList;

#endif

#endif

#if DEBUG

static int scriptContextCount;

#endif

// List of weak reference dictionaries. We'll walk through them

// and clean them up post-collection

// IWeakReferenceDictionary objects are added to this list by calling

// RegisterWeakReferenceDictionary. If you use JsUtil::WeakReferenceDictionary,

// which also exposes the IWeakReferenceDictionary interface, it'll

// automatically register the dictionary on the script context

SListBase<JsUtil::IWeakReferenceDictionary\*> weakReferenceDictionaryList;

bool isWeakReferenceDictionaryListCleared;

typedef void(\*RaiseMessageToDebuggerFunctionType)(ScriptContext \*, DEBUG\_EVENT\_INFO\_TYPE, LPCWSTR, LPCWSTR);

RaiseMessageToDebuggerFunctionType raiseMessageToDebuggerFunctionType;

typedef void(\*TransitionToDebugModeIfFirstSourceFn)(ScriptContext \*, Utf8SourceInfo \*);

TransitionToDebugModeIfFirstSourceFn transitionToDebugModeIfFirstSourceFn;

#ifdef ENABLE\_DOM\_FAST\_PATH

// Theoretically we can put this in ThreadContext; don't want to keep the dictionary forever, and preserve the possibility of

// using JavascriptFunction as key.

DOMFastPathIRHelperMap\* domFastPathIRHelperMap;

#endif

ScriptContext(ThreadContext\* threadContext);

void InitializeAllocations();

void InitializePreGlobal();

void InitializePostGlobal();

// Source Info

void EnsureSourceContextInfoMap();

void EnsureDynamicSourceContextInfoMap();

uint moduleSrcInfoCount;

#ifdef RUNTIME\_DATA\_COLLECTION

time\_t createTime;

#endif

wchar\_t const \* url;

void PrintStats();

BOOL LeaveScriptStartCore(void \* frameAddress, bool leaveForHost);

void InternalClose();

DebugContext\* debugContext;

public:

static const int kArrayMatchCh=72;

static const int kMaxArrayMatchIndex=8192;

short arrayMatchItems[kArrayMatchCh];

bool arrayMatchInit;

#ifdef LEAK\_REPORT

LeakReport::UrlRecord \* urlRecord;

bool isRootTrackerScriptContext;

#endif

DaylightTimeHelper \*GetDaylightTimeHelper() { return &daylightTimeHelper; }

bool IsClosed() const { return isClosed; }

bool IsActuallyClosed() const { return isScriptContextActuallyClosed; }

#if ENABLE\_NATIVE\_CODEGEN

bool IsClosedNativeCodeGenerator() const

{

return !nativeCodeGen || ::IsClosedNativeCodeGenerator(nativeCodeGen);

}

#endif

void SetDirectHostTypeId(TypeId typeId) {directHostTypeId = typeId; }

TypeId GetDirectHostTypeId() const { return directHostTypeId; }

TypePath\* GetRootPath() { return cache->rootPath; }

#ifdef ENABLE\_DOM\_FAST\_PATH

DOMFastPathIRHelperMap\* EnsureDOMFastPathIRHelperMap();

#endif

wchar\_t const \* GetUrl() const { return url; }

void SetUrl(BSTR bstr);

#ifdef RUNTIME\_DATA\_COLLECTION

time\_t GetCreateTime() const { return createTime; }

uint GetAllocId() const { return allocId; }

#endif

void InitializeArrayMatch()

{

if (!arrayMatchInit)

{

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

{

arrayMatchItems[i]= -1;

}

arrayMatchInit=true;

}

}

#ifdef HEAP\_ENUMERATION\_VALIDATION

bool IsInitialized() { return this->isInitialized; }

#endif

DebugContext\* GetDebugContext() const { return this->debugContext; }

uint callCount;

// Guest arena allocated cache holding references that need to be pinned.

Cache\* cache;

// if the current context is for webworker

DWORD webWorkerId;

static ScriptContext \* New(ThreadContext \* threadContext);

static void Delete(ScriptContext\* scriptContext);

~ScriptContext();

#ifdef PROFILE\_TYPES

void ProfileTypes();

#endif

#ifdef PROFILE\_OBJECT\_LITERALS

void ProfileObjectLiteral();

#endif

//

// Regex helpers

//

#if ENABLE\_REGEX\_CONFIG\_OPTIONS

UnifiedRegex::RegexStatsDatabase\* GetRegexStatsDatabase();

UnifiedRegex::DebugWriter\* GetRegexDebugWriter();

#endif

RegexPatternMruMap\* GetDynamicRegexMap() const;

UnifiedRegex::TrigramAlphabet\* GetTrigramAlphabet() { return trigramAlphabet; }

void SetTrigramAlphabet(UnifiedRegex::TrigramAlphabet \* trigramAlphabet);

UnifiedRegex::RegexStacks \*RegexStacks();

UnifiedRegex::RegexStacks \*AllocRegexStacks();

UnifiedRegex::RegexStacks \*SaveRegexStacks();

void RestoreRegexStacks(UnifiedRegex::RegexStacks \*const contStack);

void InitializeGlobalObject();

JavascriptLibrary\* GetLibrary() const { return javascriptLibrary; }

const JavascriptLibraryBase\* GetLibraryBase() const { return javascriptLibrary->GetLibraryBase(); }

#if DBG

BOOL IsCloningGlobal() const { return isCloningGlobal;}

#endif

void PushObject(Var object);

Var PopObject();

BOOL CheckObject(Var object);

inline bool IsHeapEnumInProgress() { return GetRecycler()->IsHeapEnumInProgress(); }

bool IsInterpreted() { return config.IsNoNative(); }

void ForceNoNative() { config.ForceNoNative(); }

void ForceNative() { config.ForceNative(); }

ScriptConfiguration const \* GetConfig(void) const { return &config; }

CharClassifier const \* GetCharClassifier(void) const;

ThreadContext \* GetThreadContext() const { return threadContext; }

TIME\_ZONE\_INFORMATION \* GetTimeZoneInfo()

{

uint tickCount = GetTickCount();

if (tickCount - lastTimeZoneUpdateTickCount > 1000)

{

UpdateTimeZoneInfo();

lastTimeZoneUpdateTickCount = tickCount;

}

return &timeZoneInfo;

}

void UpdateTimeZoneInfo();

static const int MaxEvalSourceSize = 400;

bool IsInEvalMap(FastEvalMapString const& key, BOOL isIndirect, ScriptFunction \*\*ppFuncScript);

void AddToEvalMap(FastEvalMapString const& key, BOOL isIndirect, ScriptFunction \*pFuncScript);

template <typename TCacheType>

void CleanDynamicFunctionCache(TCacheType\* cacheType);

void CleanEvalMapCache(Js::EvalCacheTopLevelDictionary \* cacheType);

template <class TDelegate>

void MapFunction(TDelegate mapper);

template <class TDelegate>

FunctionBody\* FindFunction(TDelegate predicate);

\_\_inline bool EnableEvalMapCleanup() { return CONFIG\_FLAG(EnableEvalMapCleanup); };

void BeginDynamicFunctionReferences();

void EndDynamicFunctionReferences();

void RegisterDynamicFunctionReference(FunctionProxy\* func);

uint GetNextSourceContextId();

bool IsInNewFunctionMap(EvalMapString const& key, ParseableFunctionInfo \*\*ppFuncBody);

void AddToNewFunctionMap(EvalMapString const& key, ParseableFunctionInfo \*pFuncBody);

SourceContextInfo \* GetSourceContextInfo(DWORD\_PTR hostSourceContext, IActiveScriptDataCache\* profileDataCache);

SourceContextInfo \* GetSourceContextInfo(uint hash);

SourceContextInfo \* CreateSourceContextInfo(uint hash, DWORD\_PTR hostSourceContext);

SourceContextInfo \* CreateSourceContextInfo(DWORD\_PTR hostSourceContext, wchar\_t const \* url, size\_t len,

IActiveScriptDataCache\* profileDataCache, wchar\_t const \* sourceMapUrl = nullptr, size\_t sourceMapUrlLen = 0);

#if defined(LEAK\_REPORT) || defined(CHECK\_MEMORY\_LEAK)

void ClearSourceContextInfoMaps()

{

if (this->cache != nullptr)

{

this->cache->sourceContextInfoMap = nullptr;

this->cache->dynamicSourceContextInfoMap = nullptr;

#if ENABLE\_PROFILE\_INFO

this->referencesSharedDynamicSourceContextInfo = false;

#endif

}

}

#endif

#if ENABLE\_PROFILE\_INFO

#if DBG\_DUMP || defined(DYNAMIC\_PROFILE\_STORAGE) || defined(RUNTIME\_DATA\_COLLECTION)

void ClearDynamicProfileList()

{

if (profileInfoList)

{

profileInfoList->Reset();

profileInfoList.Unroot(this->recycler);

}

}

SListBase<DynamicProfileInfo \*> \* GetProfileInfoList() { return profileInfoList; }

#endif

#endif

SRCINFO const \* GetModuleSrcInfo(Js::ModuleID moduleID);

SourceContextInfoMap\* GetSourceContextInfoMap()

{

return (this->cache ? this->cache->sourceContextInfoMap : nullptr);

}

DynamicSourceContextInfoMap\* GetDynamicSourceContextInfoMap()

{

return (this->cache ? this->cache->dynamicSourceContextInfoMap : nullptr);

}

void SetFirstInterpreterFrameReturnAddress(void \* returnAddress) { firstInterpreterFrameReturnAddress = returnAddress;}

void \*GetFirstInterpreterFrameReturnAddress() { return firstInterpreterFrameReturnAddress;}

void CleanupWeakReferenceDictionaries();

void Initialize();

bool Close(bool inDestructor);

void MarkForClose();

#ifdef ENABLE\_PROJECTION

void SetHostType(long hostType) { config.SetHostType(hostType); }

void SetWinRTConstructorAllowed(bool allowed) { config.SetWinRTConstructorAllowed(allowed); }

void SetProjectionTargetVersion(DWORD version) { config.SetProjectionTargetVersion(version); }

#endif

void SetCanOptimizeGlobalLookupFlag(BOOL f){ config.SetCanOptimizeGlobalLookupFlag(f);}

BOOL CanOptimizeGlobalLookup(){ return config.CanOptimizeGlobalLookup();}

bool IsClosed() { return isClosed; }

bool IsFastDOMEnabled() { return fastDOMenabled; }

void SetFastDOMenabled();

BOOL VerifyAlive(BOOL isJSFunction = FALSE, ScriptContext\* requestScriptContext = nullptr);

void VerifyAliveWithHostContext(BOOL isJSFunction, HostScriptContext\* requestHostScriptContext);

void AddFunctionBodyToPropIdMap(FunctionBody\* body);

void BindReference(void\* addr);

void InitPropertyStringMap(int i);

PropertyString\* AddPropertyString2(const Js::PropertyRecord\* propertyRecord);

PropertyString\* CachePropertyString2(const Js::PropertyRecord\* propertyRecord);

PropertyString\* GetPropertyString2(wchar\_t ch1, wchar\_t ch2);

void FindPropertyRecord(\_\_in LPCWSTR pszPropertyName, \_\_in int propertyNameLength, PropertyRecord const\*\* propertyRecord);

JsUtil::List<const RecyclerWeakReference<Js::PropertyRecord const>\*>\* FindPropertyIdNoCase(\_\_in LPCWSTR pszPropertyName, \_\_in int propertyNameLength);

void FindPropertyRecord(JavascriptString\* pstName, PropertyRecord const\*\* propertyRecord);

PropertyRecord const \* GetPropertyName(PropertyId propertyId);

PropertyRecord const \* GetPropertyNameLocked(PropertyId propertyId);

void GetOrAddPropertyRecord(JsUtil::CharacterBuffer<WCHAR> const& propName, PropertyRecord const\*\* propertyRecord);

template <size\_t N> void GetOrAddPropertyRecord(const wchar\_t(&propertyName)[N], PropertyRecord const\*\* propertyRecord)

{

GetOrAddPropertyRecord(propertyName, N - 1, propertyRecord);

}

PropertyId GetOrAddPropertyIdTracked(JsUtil::CharacterBuffer<WCHAR> const& propName);

template <size\_t N> PropertyId GetOrAddPropertyIdTracked(const wchar\_t(&propertyName)[N])

{

return GetOrAddPropertyIdTracked(propertyName, N - 1);

}

PropertyId GetOrAddPropertyIdTracked(\_\_in\_ecount(propertyNameLength) LPCWSTR pszPropertyName, \_\_in int propertyNameLength);

void GetOrAddPropertyRecord(\_\_in\_ecount(propertyNameLength) LPCWSTR pszPropertyName, \_\_in int propertyNameLength, PropertyRecord const\*\* propertyRecord);

BOOL IsNumericPropertyId(PropertyId propertyId, uint32\* value);

void RegisterWeakReferenceDictionary(JsUtil::IWeakReferenceDictionary\* weakReferenceDictionary);

void ResetWeakReferenceDicitionaryList() { weakReferenceDictionaryList.Reset(); }

BOOL ReserveStaticTypeIds(\_\_in int first, \_\_in int last);

TypeId ReserveTypeIds(int count);

TypeId CreateTypeId();

WellKnownHostType GetWellKnownHostType(Js::TypeId typeId) { return threadContext->GetWellKnownHostType(typeId); }

void SetWellKnownHostTypeId(WellKnownHostType wellKnownType, Js::TypeId typeId) { threadContext->SetWellKnownHostTypeId(wellKnownType, typeId); }

JavascriptFunction\* LoadScript(const wchar\_t\* script, SRCINFO const \* pSrcInfo, CompileScriptException \* pse, bool isExpression, bool disableDeferredParse, bool isByteCodeBufferForLibrary, Utf8SourceInfo\*\* ppSourceInfo, const wchar\_t \*rootDisplayName, bool disableAsmJs = false);

JavascriptFunction\* LoadScript(LPCUTF8 script, size\_t cb, SRCINFO const \* pSrcInfo, CompileScriptException \* pse, bool isExpression, bool disableDeferredParse, bool isByteCodeBufferForLibrary, Utf8SourceInfo\*\* ppSourceInfo, const wchar\_t \*rootDisplayName, bool disableAsmJs = false);

ArenaAllocator\* GeneralAllocator() { return &generalAllocator; }

#ifdef ENABLE\_BASIC\_TELEMETRY

ArenaAllocator\* TelemetryAllocator() { return &telemetryAllocator; }

#endif

#ifdef SEPARATE\_ARENA

ArenaAllocator\* SourceCodeAllocator() { return &sourceCodeAllocator; }

ArenaAllocator\* RegexAllocator() { return &regexAllocator; }

#else

ArenaAllocator\* SourceCodeAllocator() { return &generalAllocator; }

ArenaAllocator\* RegexAllocator() { return &generalAllocator; }

#endif

#ifdef NEED\_MISC\_ALLOCATOR

ArenaAllocator\* MiscAllocator() { return &miscAllocator; }

#endif

InlineCacheAllocator\* GetInlineCacheAllocator() { return &inlineCacheAllocator; }

IsInstInlineCacheAllocator\* GetIsInstInlineCacheAllocator() { return &isInstInlineCacheAllocator; }

ArenaAllocator\* DynamicProfileInfoAllocator() { return &dynamicProfileInfoAllocator; }

ArenaAllocator\* AllocatorForDiagnostics();

Js::TempArenaAllocatorObject\* GetTemporaryAllocator(LPCWSTR name);

void ReleaseTemporaryAllocator(Js::TempArenaAllocatorObject\* tempAllocator);

Js::TempGuestArenaAllocatorObject\* GetTemporaryGuestAllocator(LPCWSTR name);

void ReleaseTemporaryGuestAllocator(Js::TempGuestArenaAllocatorObject\* tempAllocator);

bool EnsureInterpreterArena(ArenaAllocator \*\*);

void ReleaseInterpreterArena();

ArenaAllocator\* GetGuestArena() const

{

return guestArena;

}

void ReleaseGuestArena();

Recycler\* GetRecycler() const { return recycler; }

RecyclerJavascriptNumberAllocator \* GetNumberAllocator() { return &numberAllocator; }

#if ENABLE\_NATIVE\_CODEGEN

NativeCodeGenerator \* GetNativeCodeGenerator() const { return nativeCodeGen; }

#endif

#if ENABLE\_BACKGROUND\_PARSING

BackgroundParser \* GetBackgroundParser() const { return backgroundParser; }

#endif

void OnScriptStart(bool isRoot, bool isScript);

void OnScriptEnd(bool isRoot, bool isForcedEnd);

template <bool stackProbe, bool leaveForHost>

bool LeaveScriptStart(void \* frameAddress);

template <bool leaveForHost>

void LeaveScriptEnd(void \* frameAddress);

HostScriptContext \* GetHostScriptContext() const { return hostScriptContext; }

void SetHostScriptContext(HostScriptContext \* hostScriptContext);

void SetScriptEngineHaltCallback(HaltCallback\* scriptEngine);

void ClearHostScriptContext();

IActiveScriptProfilerHeapEnum\* GetHeapEnum();

void SetHeapEnum(IActiveScriptProfilerHeapEnum\* newHeapEnum);

void ClearHeapEnum();

void SetScriptStartEventHandler(EventHandler eventHandler);

void SetScriptEndEventHandler(EventHandler eventHandler);

#ifdef FAULT\_INJECTION

void DisposeScriptContextByFaultInjection();

void SetDisposeDisposeByFaultInjectionEventHandler(EventHandler eventHandler);

#endif

EnumeratedObjectCache\* GetEnumeratedObjectCache() { return &(cache->enumObjCache); }

PropertyString\* GetPropertyString(PropertyId propertyId);

void InvalidatePropertyStringCache(PropertyId propertyId, Type\* type);

JavascriptString\* GetIntegerString(Var aValue);

JavascriptString\* GetIntegerString(int value);

JavascriptString\* GetIntegerString(uint value);

void CheckEvalRestriction();

RecyclableObject\* GetMissingPropertyResult(Js::RecyclableObject \*instance, Js::PropertyId id);

RecyclableObject\* GetMissingItemResult(Js::RecyclableObject \*instance, uint32 index);

RecyclableObject\* GetMissingParameterValue(Js::JavascriptFunction \*function, uint32 paramIndex);

RecyclableObject \*GetNullPropertyResult(Js::RecyclableObject \*instance, Js::PropertyId id);

RecyclableObject \*GetNullItemResult(Js::RecyclableObject \*instance, uint32 index);

bool HasRecordedException() const { return threadContext->GetRecordedException() != nullptr; }

Js::JavascriptExceptionObject \* GetAndClearRecordedException(bool \*considerPassingToDebugger = nullptr);

void RecordException(Js::JavascriptExceptionObject \* exceptionObject, bool propagateToDebugger = false);

\_\_declspec(noreturn) void RethrowRecordedException(JavascriptExceptionObject::HostWrapperCreateFuncType hostWrapperCreateFunc);

#if ENABLE\_NATIVE\_CODEGEN

BOOL IsNativeAddress(void \* codeAddr);

#endif

uint SaveSourceCopy(Utf8SourceInfo\* sourceInfo, int cchLength, bool isCesu8);

bool SaveSourceCopy(Utf8SourceInfo\* const sourceInfo, int cchLength, bool isCesu8, uint \* index);

uint SaveSourceNoCopy(Utf8SourceInfo\* sourceInfo, int cchLength, bool isCesu8);

Utf8SourceInfo\* CloneSourceCrossContext(Utf8SourceInfo\* crossContextSourceInfo, SRCINFO const\* srcInfo = nullptr);

void CloneSources(ScriptContext\* sourceContext);

Utf8SourceInfo\* GetSource(uint sourceIndex);

uint SourceCount() const { return (uint)sourceList->Count(); }

void CleanSourceList() { CleanSourceListInternal(false); }

SourceList\* GetSourceList() const { return sourceList; }

bool IsItemValidInSourceList(int index);

template <typename TFunction>

void MapScript(TFunction mapper)

{

this->sourceList->Map([mapper] (int, RecyclerWeakReference<Utf8SourceInfo>\* sourceInfoWeakReference)

{

Utf8SourceInfo\* strongRef = sourceInfoWeakReference->Get();

if (strongRef)

{

mapper(strongRef);

}

});

}

#ifdef CHECK\_STACKWALK\_EXCEPTION

void SetIgnoreStackWalkException() {threadContext->GetScriptEntryExit()->ignoreStackWalkException = true; }

#endif

// For debugging scenarios where execution will go to debugging manager and come back to engine again, enforce the current EER to have

// 'hasCaller' property set, which will enable the stack walking across frames.

// Do not call this directly, look for ENFORCE\_ENTRYEXITRECORD\_HASCALLER macro.

void EnforceEERHasCaller() { threadContext->GetScriptEntryExit()->hasCaller = true; }

void SetRaiseMessageToDebuggerFunction(RaiseMessageToDebuggerFunctionType function)

{

raiseMessageToDebuggerFunctionType = function;

}

void RaiseMessageToDebugger(DEBUG\_EVENT\_INFO\_TYPE messageType, LPCWSTR message, LPCWSTR url)

{

if (raiseMessageToDebuggerFunctionType != nullptr)

{

raiseMessageToDebuggerFunctionType(this, messageType, message, url);

}

}

void SetTransitionToDebugModeIfFirstSourceFn(TransitionToDebugModeIfFirstSourceFn function)

{

transitionToDebugModeIfFirstSourceFn = function;

}

void TransitionToDebugModeIfFirstSource(Utf8SourceInfo \*sourceInfo)

{

if (transitionToDebugModeIfFirstSourceFn != nullptr)

{

transitionToDebugModeIfFirstSourceFn(this, sourceInfo);

}

}

void AddSourceSize(size\_t sourceSize)

{

this->sourceSize += sourceSize;

this->threadContext->AddSourceSize(sourceSize);

}

size\_t GetSourceSize()

{

return this->sourceSize;

}

BOOL SetDeferredBody(BOOL set)

{

bool old = this->deferredBody;

this->deferredBody = !!set;

return old;

}

BOOL GetDeferredBody(void) const

{

return this->deferredBody;

}

public:

void RegisterAsScriptContextWithInlineCaches();

void RegisterAsScriptContextWithIsInstInlineCaches();

bool IsRegisteredAsScriptContextWithIsInstInlineCaches();

void FreeLoopBody(void\* codeAddress);

void FreeFunctionEntryPoint(Js::JavascriptMethod method);

private:

void DoRegisterAsScriptContextWithInlineCaches();

void DoRegisterAsScriptContextWithIsInstInlineCaches();

uint CloneSource(Utf8SourceInfo\* info);

public:

void RegisterProtoInlineCache(InlineCache \*pCache, PropertyId propId);

void InvalidateProtoCaches(const PropertyId propertyId);

void InvalidateAllProtoCaches();

void RegisterStoreFieldInlineCache(InlineCache \*pCache, PropertyId propId);

void InvalidateStoreFieldCaches(const PropertyId propertyId);

void InvalidateAllStoreFieldCaches();

void RegisterIsInstInlineCache(Js::IsInstInlineCache \* cache, Js::Var function);

#if DBG

bool IsIsInstInlineCacheRegistered(Js::IsInstInlineCache \* cache, Js::Var function);

#endif

void ClearInlineCaches();

void ClearIsInstInlineCaches();

#ifdef PERSISTENT\_INLINE\_CACHES

void ClearInlineCachesWithDeadWeakRefs();

#endif

void ClearScriptContextCaches();

#if ENABLE\_NATIVE\_CODEGEN

void RegisterConstructorCache(Js::PropertyId propertyId, Js::ConstructorCache\* cache);

#endif

public:

void RegisterPrototypeChainEnsuredToHaveOnlyWritableDataPropertiesScriptContext();

private:

void DoRegisterPrototypeChainEnsuredToHaveOnlyWritableDataPropertiesScriptContext();

public:

void ClearPrototypeChainEnsuredToHaveOnlyWritableDataPropertiesCaches();

public:

JavascriptString \* GetLastNumberToStringRadix10(double value);

void SetLastNumberToStringRadix10(double value, JavascriptString \* str);

bool GetLastUtcTimeFromStr(JavascriptString \* str, double& dbl);

void SetLastUtcTimeFromStr(JavascriptString \* str, double value);

bool IsNoContextSourceContextInfo(SourceContextInfo \*sourceContextInfo) const

{

return sourceContextInfo == cache->noContextSourceContextInfo;

}

BOOL IsProfiling()

{

return (m\_pProfileCallback != nullptr);

}

BOOL IsInProfileCallback()

{

return m\_inProfileCallback;

}

#if DBG

SourceContextInfo const \* GetNoContextSourceContextInfo() const { return cache->noContextSourceContextInfo; }

int GetProfileSession()

{

AssertMsg(m\_pProfileCallback != nullptr, "Asking for profile session when we aren't in one.");

return m\_iProfileSession;

}

void StartNewProfileSession()

{

AssertMsg(m\_pProfileCallback != nullptr, "New Session when the profiler isn't set to any callback.");

m\_iProfileSession++;

}

void StopProfileSession()

{

AssertMsg(m\_pProfileCallback == nullptr, "How to stop when there is still the callback out there");

}

bool hadProfiled;

bool HadProfiled() const { return hadProfiled; }

#endif

SRCINFO \*AddHostSrcInfo(SRCINFO const \*pSrcInfo);

inline void CoreSetProfileEventMask(DWORD dwEventMask);

typedef HRESULT (\*RegisterExternalLibraryType)(Js::ScriptContext \*pScriptContext);

HRESULT RegisterProfileProbe(IActiveScriptProfilerCallback \*pProfileCallback, DWORD dwEventMask, DWORD dwContext, RegisterExternalLibraryType RegisterExternalLibrary, JavascriptMethod dispatchInvoke);

HRESULT SetProfileEventMask(DWORD dwEventMask);

HRESULT DeRegisterProfileProbe(HRESULT hrReason, JavascriptMethod dispatchInvoke);

HRESULT RegisterScript(Js::FunctionProxy \*pFunctionBody, BOOL fRegisterScript = TRUE);

// Register static and dynamic scripts

HRESULT RegisterAllScripts();

// Iterate through utf8sourceinfo and clear debug document if they are there.

void EnsureClearDebugDocument();

// To be called directly only when the thread context is shutting down

void ShutdownClearSourceLists();

HRESULT RegisterLibraryFunction(const wchar\_t \*pwszObjectName, const wchar\_t \*pwszFunctionName, Js::PropertyId functionPropertyId, JavascriptMethod entryPoint);

HRESULT RegisterBuiltinFunctions(RegisterExternalLibraryType RegisterExternalLibrary);

void RegisterDebugThunk(bool calledDuringAttach = true);

void UnRegisterDebugThunk();

void UpdateRecyclerFunctionEntryPointsForDebugger();

void SetFunctionInRecyclerToProfileMode(bool enumerateNonUserFunctionsOnly = false);

static void SetEntryPointToProfileThunk(JavascriptFunction\* function);

static void RestoreEntryPointFromProfileThunk(JavascriptFunction\* function);

static void RecyclerEnumClassEnumeratorCallback(void \*address, size\_t size);

static void RecyclerFunctionCallbackForDebugger(void \*address, size\_t size);

static ushort ProcessNameAndGetLength(Js::StringBuilder<ArenaAllocator>\* nameBuffer, const WCHAR\* name);

#ifdef ASMJS\_PLAT

void TransitionEnvironmentForDebugger(ScriptFunction \* scriptFunction);

#endif

#if ENABLE\_NATIVE\_CODEGEN

HRESULT RecreateNativeCodeGenerator();

#endif

HRESULT OnDebuggerAttached();

HRESULT OnDebuggerDetached();

HRESULT OnDebuggerAttachedDetached(bool attach);

void InitializeDebugging();

bool IsForceNoNative();

bool IsEnumeratingRecyclerObjects() const { return isEnumeratingRecyclerObjects; }

private:

class AutoEnumeratingRecyclerObjects

{

public:

AutoEnumeratingRecyclerObjects(ScriptContext\* scriptContext):

m\_scriptContext(scriptContext)

{

Assert(!m\_scriptContext->IsEnumeratingRecyclerObjects());

m\_scriptContext->isEnumeratingRecyclerObjects = true;

}

~AutoEnumeratingRecyclerObjects()

{

Assert(m\_scriptContext->IsEnumeratingRecyclerObjects());

m\_scriptContext->isEnumeratingRecyclerObjects = false;

}

private:

ScriptContext\* m\_scriptContext;

};

#ifdef EDIT\_AND\_CONTINUE

private:

ScriptEditQuery\* activeScriptEditQuery;

void BeginScriptEditEnumFunctions(ScriptEditQuery\* scriptEditQuery) { Assert(!activeScriptEditQuery); activeScriptEditQuery = scriptEditQuery; }

void EndScriptEditEnumFunctions() { Assert(activeScriptEditQuery); activeScriptEditQuery = nullptr; }

public:

ScriptEditQuery\* GetActiveScriptEditQuery() const { return activeScriptEditQuery; }

class AutoScriptEditEnumFunctions

{

public:

AutoScriptEditEnumFunctions(ScriptContext\* scriptContext, ScriptEditQuery\* scriptEditQuery) : m\_scriptContext(scriptContext)

{

scriptContext->BeginScriptEditEnumFunctions(scriptEditQuery);

}

~AutoScriptEditEnumFunctions() { m\_scriptContext->EndScriptEditEnumFunctions(); }

private:

ScriptContext\* m\_scriptContext;

};

#endif

private:

typedef JsUtil::BaseDictionary<JavascriptMethod, Js::PropertyId, ArenaAllocator, PrimeSizePolicy> BuiltinFunctionIdDictionary;

BuiltinFunctionIdDictionary \*m\_pBuiltinFunctionIdMap;

Js::PropertyId GetFunctionNumber(JavascriptMethod entryPoint);

static const wchar\_t\* CopyString(const wchar\_t\* str, size\_t charCount, ArenaAllocator\* alloc);

static charcount\_t AppendWithEscapeCharacters(Js::StringBuilder<ArenaAllocator>\* stringBuilder, const WCHAR\* sourceString, charcount\_t sourceStringLen, WCHAR escapeChar, WCHAR charToEscape);

public:

#if DYNAMIC\_INTERPRETER\_THUNK

JavascriptMethod GetNextDynamicAsmJsInterpreterThunk(PVOID\* ppDynamicInterpreterThunk);

JavascriptMethod GetNextDynamicInterpreterThunk(PVOID\* ppDynamicInterpreterThunk);

BOOL IsDynamicInterpreterThunk(void\* address);

void ReleaseDynamicInterpreterThunk(BYTE\* address, bool addtoFreeList);

void ReleaseDynamicAsmJsInterpreterThunk(BYTE\* address, bool addtoFreeList);

#endif

void SetProfileMode(BOOL fSet);

static JavascriptMethod GetProfileModeThunk(JavascriptMethod entryPoint);

static Var ProfileModeThunk\_DebugModeWrapper(JavascriptFunction\* function, ScriptContext\* scriptContext, JavascriptMethod entryPoint, Arguments& args);

BOOL GetProfileInfo(

JavascriptFunction\* function,

PROFILER\_TOKEN &scriptId,

PROFILER\_TOKEN &functionId);

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

static JavascriptMethod ProfileModeDeferredParse(ScriptFunction \*\*function);

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

// Thunks for deferred deserialization of function bodies from the byte code cache

static JavascriptMethod ProfileModeDeferredDeserialize(ScriptFunction\* function);

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

HRESULT OnScriptCompiled(PROFILER\_TOKEN scriptId, PROFILER\_SCRIPT\_TYPE type, IUnknown \*pIDebugDocumentContext);

HRESULT OnFunctionCompiled(

PROFILER\_TOKEN functionId,

PROFILER\_TOKEN scriptId,

const WCHAR \*pwszFunctionName,

const WCHAR \*pwszFunctionNameHint,

IUnknown \*pIDebugDocumentContext);

HRESULT OnFunctionEnter(PROFILER\_TOKEN scriptId, PROFILER\_TOKEN functionId);

HRESULT OnFunctionExit(PROFILER\_TOKEN scriptId, PROFILER\_TOKEN functionId);

bool SetDispatchProfile(bool fSet, JavascriptMethod dispatchInvoke);

HRESULT OnDispatchFunctionEnter(const WCHAR \*pwszFunctionName);

HRESULT OnDispatchFunctionExit(const WCHAR \*pwszFunctionName);

void OnStartupComplete();

void SaveStartupProfileAndRelease(bool isSaveOnClose = false);

static HRESULT FunctionExitSenderThunk(PROFILER\_TOKEN functionId, PROFILER\_TOKEN scriptId, ScriptContext \*pScriptContext);

static HRESULT FunctionExitByNameSenderThunk(const wchar\_t \*pwszFunctionName, ScriptContext \*pScriptContext);

#if ENABLE\_PROFILE\_INFO

void AddDynamicProfileInfo(FunctionBody \* functionBody, WriteBarrierPtr<DynamicProfileInfo>\* dynamicProfileInfo);

#endif

#if DBG || defined(RUNTIME\_DATA\_COLLECTION)

uint allocId;

#endif

#ifdef PROFILE\_EXEC

void DisableProfiler();

void SetRecyclerProfiler();

void SetProfilerFromScriptContext(ScriptContext \* scriptContext);

void ProfileBegin(Js::Phase);

void ProfileEnd(Js::Phase);

void ProfileSuspend(Js::Phase, Js::Profiler::SuspendRecord \* suspendRecord);

void ProfileResume(Js::Profiler::SuspendRecord \* suspendRecord);

void ProfilePrint();

bool IsProfilerCreated() const { return isProfilerCreated; }

#endif

#ifdef PROFILE\_MEM

void DisableProfileMemoryDumpOnDelete() { profileMemoryDump = false; }

#endif

#ifdef PROFILE\_STRINGS

StringProfiler \* GetStringProfiler(); // May be null if string profiling not enabled

#endif

public:

void SetBuiltInLibraryFunction(JavascriptMethod entryPoint, JavascriptFunction\* function);

JavascriptFunction\* GetBuiltInLibraryFunction(JavascriptMethod entryPoint);

private:

typedef JsUtil::BaseDictionary<JavascriptMethod, JavascriptFunction\*, Recycler, PowerOf2SizePolicy> BuiltInLibraryFunctionMap;

BuiltInLibraryFunctionMap\* builtInLibraryFunctions;

#ifdef RECYCLER\_PERF\_COUNTERS

size\_t bindReferenceCount;

#endif

ScriptContext \* nextPendingClose;

public:

void SetNextPendingClose(ScriptContext \* nextPendingClose);

inline ScriptContext \* GetNextPendingClose() const { return nextPendingClose; }

#ifdef ENABLE\_MUTATION\_BREAKPOINT

// Keep track of all breakpoints in order to properly clean up on debugger detach

bool HasMutationBreakpoints();

void InsertMutationBreakpoint(Js::MutationBreakpoint \*mutationBreakpoint);

#endif

};

class AutoDynamicCodeReference

{

public:

AutoDynamicCodeReference(ScriptContext\* scriptContext):

m\_scriptContext(scriptContext)

{

scriptContext->BeginDynamicFunctionReferences();

}

~AutoDynamicCodeReference()

{

m\_scriptContext->EndDynamicFunctionReferences();

}

private:

ScriptContext\* m\_scriptContext;

};

template <typename TCacheType>

void ScriptContext::CleanDynamicFunctionCache(TCacheType\* cacheType)

{

// Remove eval map functions that haven't been recently used

// TODO: Metric based on allocation size too? So don't clean if there hasn't been much allocated?

cacheType->Clean([this](const TCacheType::KeyType& key, TCacheType::ValueType value) {

#ifdef ENABLE\_DEBUG\_CONFIG\_OPTIONS

if (CONFIG\_FLAG(DumpEvalStringOnRemoval))

{

Output::Print(L"EvalMap: Removing Dynamic Function String from dynamic function cache: %s\n", key.str.GetBuffer()); Output::Flush();

}

#endif

});

}

template <class TDelegate>

void ScriptContext::MapFunction(TDelegate mapper)

{

if (this->sourceList)

{

this->sourceList->Map([&mapper](int, RecyclerWeakReference<Js::Utf8SourceInfo>\* sourceInfo)

{

Utf8SourceInfo\* sourceInfoStrongRef = sourceInfo->Get();

if (sourceInfoStrongRef)

{

sourceInfoStrongRef->MapFunction(mapper);

}

});

}

}

template <class TDelegate>

FunctionBody\* ScriptContext::FindFunction(TDelegate predicate)

{

FunctionBody\* functionBody = nullptr;

this->sourceList->MapUntil([&functionBody, &predicate](int, RecyclerWeakReference<Js::Utf8SourceInfo>\* sourceInfo) -> bool

{

Utf8SourceInfo\* sourceInfoStrongRef = sourceInfo->Get();

if (sourceInfoStrongRef)

{

functionBody = sourceInfoStrongRef->FindFunction(predicate);

if (functionBody)

{

return true;

}

}

return false;

});

return functionBody;

}

}

#define BEGIN\_TEMP\_ALLOCATOR(allocator, scriptContext, name) \

Js::TempArenaAllocatorObject \*temp##allocator = scriptContext->GetTemporaryAllocator(name); \

ArenaAllocator \* allocator = temp##allocator->GetAllocator();

#define END\_TEMP\_ALLOCATOR(allocator, scriptContext) \

scriptContext->ReleaseTemporaryAllocator(temp##allocator);

#define DECLARE\_TEMP\_ALLOCATOR(allocator) \

Js::TempArenaAllocatorObject \*temp##allocator = nullptr; \

ArenaAllocator \* allocator = nullptr;

#define ACQUIRE\_TEMP\_ALLOCATOR(allocator, scriptContext, name) \

temp##allocator = scriptContext->GetTemporaryAllocator(name); \

allocator = temp##allocator->GetAllocator();

#define RELEASE\_TEMP\_ALLOCATOR(allocator, scriptContext) \

if (temp##allocator) \

scriptContext->ReleaseTemporaryAllocator(temp##allocator);

#define DECLARE\_TEMP\_GUEST\_ALLOCATOR(allocator) \

Js::TempGuestArenaAllocatorObject \*tempGuest##allocator = nullptr; \

ArenaAllocator \* allocator = nullptr;

#define ACQUIRE\_TEMP\_GUEST\_ALLOCATOR(allocator, scriptContext, name) \

tempGuest##allocator = scriptContext->GetTemporaryGuestAllocator(name); \

allocator = tempGuest##allocator->GetAllocator();

#define RELEASE\_TEMP\_GUEST\_ALLOCATOR(allocator, scriptContext) \

if (tempGuest##allocator) \

scriptContext->ReleaseTemporaryGuestAllocator(tempGuest##allocator);

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

// 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

{

inline PropertyString\* ScriptContext::GetPropertyString2(wchar\_t ch1, wchar\_t ch2) {

if (ch1 < '0' || ch1 > 'z' || ch2 < '0' || ch2 > 'z')

return NULL;

const uint i=PropertyStringMap::PStrMapIndex(ch1);

if (propertyStrings[i]==NULL)

return NULL;

const uint j=PropertyStringMap::PStrMapIndex(ch2);

return propertyStrings[i]->strLen2[j];

}

inline void ScriptContext::FindPropertyRecord(JavascriptString \*pstName, PropertyRecord const \*\* propertyRecord)

{

threadContext->FindPropertyRecord(pstName, propertyRecord);

}

inline void ScriptContext::FindPropertyRecord(\_\_in LPCWSTR propertyName, \_\_in int propertyNameLength, PropertyRecord const \*\* propertyRecord)

{

threadContext->FindPropertyRecord(propertyName, propertyNameLength, propertyRecord);

}

inline JsUtil::List<const RecyclerWeakReference<Js::PropertyRecord const>\*>\* ScriptContext::FindPropertyIdNoCase(\_\_in LPCWSTR propertyName,\_\_in int propertyNameLength)

{

return threadContext->FindPropertyIdNoCase(this, propertyName, propertyNameLength);

}

inline PropertyId ScriptContext::GetOrAddPropertyIdTracked(JsUtil::CharacterBuffer<WCHAR> const& propName)

{

Js::PropertyRecord const \* propertyRecord;

threadContext->GetOrAddPropertyId(propName, &propertyRecord);

this->TrackPid(propertyRecord);

return propertyRecord->GetPropertyId();

}

template <size\_t N>

inline PropertyId ScriptContext::GetOrAddPropertyIdTracked(const wchar\_t(&propertyName)[N])

{

return GetOrAddPropertyIdTracked(propertyName, N - 1);

}

inline void ScriptContext::GetOrAddPropertyRecord(JsUtil::CharacterBuffer<WCHAR> const& propertyName, PropertyRecord const \*\* propertyRecord)

{

threadContext->GetOrAddPropertyId(propertyName, propertyRecord);

}

template <size\_t N>

inline void ScriptContext::GetOrAddPropertyRecord(const wchar\_t(&propertyName)[N], PropertyRecord const\*\* propertyRecord)

{

GetOrAddPropertyRecord(propertyName, N - 1, propertyRecord);

}

inline PropertyId ScriptContext::GetOrAddPropertyIdTracked(\_\_in\_ecount(propertyNameLength) LPCWSTR propertyName, \_\_in int propertyNameLength)

{

Js::PropertyRecord const \* propertyRecord;

threadContext->GetOrAddPropertyId(propertyName, propertyNameLength, &propertyRecord);

if (propertyNameLength == 2)

{

CachePropertyString2(propertyRecord);

}

this->TrackPid(propertyRecord);

return propertyRecord->GetPropertyId();

}

inline void ScriptContext::GetOrAddPropertyRecord(\_\_in\_ecount(propertyNameLength) LPCWSTR propertyName, \_\_in int propertyNameLength, PropertyRecord const \*\* propertyRecord)

{

threadContext->GetOrAddPropertyId(propertyName, propertyNameLength, propertyRecord);

if (propertyNameLength == 2)

{

CachePropertyString2(\*propertyRecord);

}

}

template <typename TCacheType>

void ScriptContext::CleanDynamicFunctionCache(TCacheType\* cacheType)

{

// Remove eval map functions that haven't been recently used

cacheType->Clean([this](const TCacheType::KeyType& key, TCacheType::ValueType value) {

#ifdef ENABLE\_DEBUG\_CONFIG\_OPTIONS

if (CONFIG\_FLAG(DumpEvalStringOnRemoval))

{

Output::Print(L"EvalMap: Removing Dynamic Function String from dynamic function cache: %s\n", key.str.GetBuffer()); Output::Flush();

}

#endif

});

}

template <class TDelegate>

void ScriptContext::MapFunction(TDelegate mapper)

{

if (this->sourceList)

{

this->sourceList->Map([&mapper] (int, RecyclerWeakReference<Js::Utf8SourceInfo>\* sourceInfo)

{

Utf8SourceInfo\* sourceInfoStrongRef = sourceInfo->Get();

if (sourceInfoStrongRef)

{

sourceInfoStrongRef->MapFunction(mapper);

}

});

}

}

template <class TDelegate>

FunctionBody\* ScriptContext::FindFunction(TDelegate predicate)

{

FunctionBody\* functionBody = nullptr;

this->sourceList->MapUntil([&functionBody, &predicate] (int, RecyclerWeakReference<Js::Utf8SourceInfo>\* sourceInfo) -> bool

{

Utf8SourceInfo\* sourceInfoStrongRef = sourceInfo->Get();

if (sourceInfoStrongRef)

{

functionBody = sourceInfoStrongRef->FindFunction(predicate);

if (functionBody)

{

return true;

}

}

return false;

});

return functionBody;

}

inline BOOL ScriptContext::IsNumericPropertyId(PropertyId propertyId, uint32\* value)

{

BOOL isNumericPropertyId = threadContext->IsNumericPropertyId(propertyId, value);

#if DEBUG

PropertyRecord const \* name = this->GetPropertyName(propertyId);

if (name != nullptr)

{

// Symbol properties are not numeric - description should not be used.

if (name->IsSymbol())

{

return false;

}

ulong index;

BOOL isIndex = JavascriptArray::GetIndex(name->GetBuffer(), &index);

if (isNumericPropertyId != isIndex)

{

// WOOB 1137798: JavascriptArray::GetIndex does not handle embeded NULLs. So if we have a property

// name "1234\0", JavascriptArray::GetIndex would incorrectly accepts it as an array index property

// name.

Assert((size\_t)(name->GetLength()) != wcslen(name->GetBuffer()));

}

else if (isNumericPropertyId)

{

Assert((ulong)\*value == index);

}

}

#endif

return isNumericPropertyId;

}

inline void ScriptContext::RegisterWeakReferenceDictionary(JsUtil::IWeakReferenceDictionary\* weakReferenceDictionary)

{

this->weakReferenceDictionaryList.Prepend(this->GeneralAllocator(), weakReferenceDictionary);

}

\_\_inline RecyclableObject \*ScriptContext::GetMissingPropertyResult(Js::RecyclableObject \*instance, Js::PropertyId id)

{

return GetLibrary()->GetUndefined();

}

\_\_inline RecyclableObject \*ScriptContext::GetMissingItemResult(Js::RecyclableObject \*instance, uint32 index)

{

return GetLibrary()->GetUndefined();

}

\_\_inline RecyclableObject \*ScriptContext::GetMissingParameterValue(Js::JavascriptFunction \*function, uint32 paramIndex)

{

return GetLibrary()->GetUndefined();

}

\_\_inline RecyclableObject \*ScriptContext::GetNullPropertyResult(Js::RecyclableObject \*instance, Js::PropertyId id)

{

return GetLibrary()->GetNull();

}

\_\_inline RecyclableObject \*ScriptContext::GetNullItemResult(Js::RecyclableObject \*instance, uint32 index)

{

return GetLibrary()->GetUndefined();

}

\_\_inline SRCINFO \*ScriptContext::AddHostSrcInfo(SRCINFO const \*pSrcInfo)

{

Assert(pSrcInfo != nullptr);

return RecyclerNewZ(this->GetRecycler(), SRCINFO, \*pSrcInfo);

}

}

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

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

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

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

//////////////////////////////////////////////////////////

// Should we remove this??

// ScriptContextBase.h is used by static lib shared between host and chakra. We need to keep

// the size consistent and try not to change its size. We need to have matching mshtml.dll

// if the size changed here.

/////////////////////////////////////////////////////////

#pragma once

interface IActiveScriptDirect;

namespace Js

{

class GlobalObject;

class JavascriptLibrary;

class ScriptContextBase

{

public:

ScriptContextBase():

javascriptLibrary(nullptr),

globalObject(nullptr),

isClosed(false),

pActiveScriptDirect(nullptr) {}

JavascriptLibrary\* GetLibrary() const { return javascriptLibrary; }

void SetLibrary(JavascriptLibrary\* library) { javascriptLibrary = library;}

void ClearGlobalObject();

void SetGlobalObject(GlobalObject \*globalObject);

GlobalObject\* GetGlobalObject() const { return globalObject; }

IActiveScriptDirect\* GetActiveScriptDirect()

{

return this->pActiveScriptDirect;

}

void SetActiveScriptDirect(IActiveScriptDirect\* pActiveScriptDirect)

{

this->pActiveScriptDirect = pActiveScriptDirect;

}

protected:

JavascriptLibrary\* javascriptLibrary;

GlobalObject\* globalObject;

bool isClosed;

IActiveScriptDirect\* pActiveScriptDirect;

};

}

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

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

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

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

#include "RuntimeBasePch.h"

namespace Js

{

ScriptContextOptimizationOverrideInfo::ScriptContextOptimizationOverrideInfo()

: sideEffects(SideEffects\_None),

arraySetElementFastPathVtable(VirtualTableInfo<Js::JavascriptArray>::Address),

intArraySetElementFastPathVtable(VirtualTableInfo<Js::JavascriptNativeIntArray>::Address),

floatArraySetElementFastPathVtable(VirtualTableInfo<Js::JavascriptNativeFloatArray>::Address),

crossSiteRoot(nullptr), crossSitePrev(nullptr), crossSiteNext(nullptr)

{

}

ScriptContextOptimizationOverrideInfo::~ScriptContextOptimizationOverrideInfo()

{

ScriptContextOptimizationOverrideInfo \* next = crossSiteNext;

if (next != nullptr)

{

ScriptContextOptimizationOverrideInfo \* root = crossSiteRoot;

Assert(root != nullptr);

if (this == root)

{

// Change the root

ForEachCrossSiteInfo([next](ScriptContextOptimizationOverrideInfo \* info)

{

info->crossSiteRoot = next;

});

}

ScriptContextOptimizationOverrideInfo \* prev = crossSitePrev;

Assert(prev != nullptr);

next->crossSitePrev = prev;

prev->crossSiteNext = next;

}

}

template <typename Fn>

void ScriptContextOptimizationOverrideInfo::ForEachCrossSiteInfo(Fn fn)

{

Assert(crossSiteRoot != nullptr);

ScriptContextOptimizationOverrideInfo \* current = this;

do

{

fn(current);

current = current->crossSiteNext;

}

while (current != this);

}

template <typename Fn>

void ScriptContextOptimizationOverrideInfo::ForEachEditingCrossSiteInfo(Fn fn)

{

Assert(crossSiteRoot != nullptr);

ScriptContextOptimizationOverrideInfo \* current = this;

do

{

ScriptContextOptimizationOverrideInfo \* next = current->crossSiteNext;

fn(current);

current = next;

}

while (current != this);

}

void

ScriptContextOptimizationOverrideInfo::Merge(ScriptContextOptimizationOverrideInfo \* info)

{

ScriptContextOptimizationOverrideInfo \* thisRoot = this->crossSiteRoot;

ScriptContextOptimizationOverrideInfo \* infoRoot = info->crossSiteRoot;

if (thisRoot == infoRoot)

{

if (thisRoot != nullptr)

{

// Both info is already in the same info group

return;

}

// Both of them are null, just group them

// Update this to be the template

this->Update(info);

// Initialize the cross site list

this->crossSiteRoot = this;

this->crossSitePrev = this;

this->crossSiteNext = this;

// Insert the info to the list

this->Insert(info);

}

else

{

if (thisRoot == nullptr)

{

thisRoot = infoRoot;

infoRoot = nullptr;

info = this;

}

thisRoot->Update(info);

// Spread the information on the current group

thisRoot->ForEachCrossSiteInfo([thisRoot](ScriptContextOptimizationOverrideInfo \* i)

{

thisRoot->CopyTo(i);

});

if (infoRoot == nullptr)

{

thisRoot->Insert(info);

}

else

{

// Insert the other group

info->ForEachEditingCrossSiteInfo([thisRoot](ScriptContextOptimizationOverrideInfo \* i)

{

thisRoot->Insert(i);

});

}

}

DebugOnly(Verify());

}

void

ScriptContextOptimizationOverrideInfo::CopyTo(ScriptContextOptimizationOverrideInfo \* info)

{

info->arraySetElementFastPathVtable = this->arraySetElementFastPathVtable;

info->intArraySetElementFastPathVtable = this->intArraySetElementFastPathVtable;

info->floatArraySetElementFastPathVtable = this->floatArraySetElementFastPathVtable;

info->sideEffects = this->sideEffects;

}

void

ScriptContextOptimizationOverrideInfo::Insert(ScriptContextOptimizationOverrideInfo \* info)

{

// Copy the information

this->CopyTo(info);

// Insert

// Only insert at the root

Assert(this == this->crossSiteRoot);

info->crossSiteRoot = this;

info->crossSiteNext = this;

info->crossSitePrev = this->crossSitePrev;

this->crossSitePrev->crossSiteNext = info;

this->crossSitePrev = info;

}

void

ScriptContextOptimizationOverrideInfo::Update(ScriptContextOptimizationOverrideInfo \* info)

{

if (!info->IsEnabledArraySetElementFastPath())

{

this->DisableArraySetElementFastPath();

}

this->sideEffects = (SideEffects)(this->sideEffects | info->sideEffects);

}

void

ScriptContextOptimizationOverrideInfo::SetSideEffects(SideEffects se)

{

if (this->crossSiteRoot == nullptr)

{

sideEffects = (SideEffects)(sideEffects | se);

}

else if ((sideEffects & se) != se)

{

ForEachCrossSiteInfo([se](ScriptContextOptimizationOverrideInfo \* info)

{

Assert((info->sideEffects & se) != se);

info->sideEffects = (SideEffects)(info->sideEffects | se);

});

}

}

bool

ScriptContextOptimizationOverrideInfo::IsEnabledArraySetElementFastPath() const

{

return arraySetElementFastPathVtable != InvalidVtable;

}

void

ScriptContextOptimizationOverrideInfo::DisableArraySetElementFastPath()

{

if (this->crossSiteRoot == nullptr)

{

arraySetElementFastPathVtable = InvalidVtable;

intArraySetElementFastPathVtable = InvalidVtable;

floatArraySetElementFastPathVtable = InvalidVtable;

}

else if (IsEnabledArraySetElementFastPath())

{

// disable for all script context in the cross site group

ForEachCrossSiteInfo([](ScriptContextOptimizationOverrideInfo \* info)

{

Assert(info->IsEnabledArraySetElementFastPath());

info->arraySetElementFastPathVtable = InvalidVtable;

info->intArraySetElementFastPathVtable = InvalidVtable;

info->floatArraySetElementFastPathVtable = InvalidVtable;

});

}

}

INT\_PTR

ScriptContextOptimizationOverrideInfo::GetArraySetElementFastPathVtable() const

{

return arraySetElementFastPathVtable;

}

INT\_PTR

ScriptContextOptimizationOverrideInfo::GetIntArraySetElementFastPathVtable() const

{

return intArraySetElementFastPathVtable;

}

INT\_PTR

ScriptContextOptimizationOverrideInfo::GetFloatArraySetElementFastPathVtable() const

{

return floatArraySetElementFastPathVtable;

}

void \*

ScriptContextOptimizationOverrideInfo::GetAddressOfArraySetElementFastPathVtable()

{

return &arraySetElementFastPathVtable;

}

void \*

ScriptContextOptimizationOverrideInfo::GetAddressOfIntArraySetElementFastPathVtable()

{

return &intArraySetElementFastPathVtable;

}

void \*

ScriptContextOptimizationOverrideInfo::GetAddressOfFloatArraySetElementFastPathVtable()

{

return &floatArraySetElementFastPathVtable;

}

#if DBG

void

ScriptContextOptimizationOverrideInfo::Verify()

{

if (this->crossSiteRoot == nullptr)

{

return;

}

this->ForEachCrossSiteInfo([this](ScriptContextOptimizationOverrideInfo \* i)

{

Assert(i->crossSiteRoot == this->crossSiteRoot);

Assert(i->sideEffects == this->sideEffects);

Assert(i->arraySetElementFastPathVtable == this->arraySetElementFastPathVtable);

Assert(i->intArraySetElementFastPathVtable == this->intArraySetElementFastPathVtable);

Assert(i->floatArraySetElementFastPathVtable == this->floatArraySetElementFastPathVtable);

});

}

#endif

};

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

// 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 ScriptContextOptimizationOverrideInfo

{

public:

ScriptContextOptimizationOverrideInfo();

~ScriptContextOptimizationOverrideInfo();

static DWORD GetSideEffectsOffset() { return offsetof(ScriptContextOptimizationOverrideInfo, sideEffects); }

static DWORD GetArraySetElementFastPathVtableOffset() { return offsetof(ScriptContextOptimizationOverrideInfo, arraySetElementFastPathVtable); }

static DWORD GetIntArraySetElementFastPathVtableOffset() { return offsetof(ScriptContextOptimizationOverrideInfo, intArraySetElementFastPathVtable); }

static DWORD GetFloatArraySetElementFastPathVtableOffset() { return offsetof(ScriptContextOptimizationOverrideInfo, floatArraySetElementFastPathVtable); }

void SetSideEffects(SideEffects se);

SideEffects GetSideEffects() { return sideEffects; }

SideEffects \* GetAddressOfSideEffects() { return &sideEffects; }

bool IsEnabledArraySetElementFastPath() const;

void DisableArraySetElementFastPath();

INT\_PTR GetArraySetElementFastPathVtable() const;

INT\_PTR GetIntArraySetElementFastPathVtable() const;

INT\_PTR GetFloatArraySetElementFastPathVtable() const;

void \* GetAddressOfArraySetElementFastPathVtable();

void \* GetAddressOfIntArraySetElementFastPathVtable();

void \* GetAddressOfFloatArraySetElementFastPathVtable();

void Merge(ScriptContextOptimizationOverrideInfo \* info);

// Use a small integer so JIT'ed code can encode in a smaller instruction

static const INT\_PTR InvalidVtable = (INT\_PTR)1;

private:

// Optimization overrides

SideEffects sideEffects;

INT\_PTR arraySetElementFastPathVtable;

INT\_PTR intArraySetElementFastPathVtable;

INT\_PTR floatArraySetElementFastPathVtable;

// Cross site tracking

ScriptContextOptimizationOverrideInfo \* crossSiteRoot;

ScriptContextOptimizationOverrideInfo \* crossSitePrev;

ScriptContextOptimizationOverrideInfo \* crossSiteNext;

template <typename Fn>

void ForEachCrossSiteInfo(Fn fn);

template <typename Fn>

void ForEachEditingCrossSiteInfo(Fn fn);

void Update(ScriptContextOptimizationOverrideInfo \* info);

void CopyTo(ScriptContextOptimizationOverrideInfo \* info);

void Insert(ScriptContextOptimizationOverrideInfo \* info);

#if DBG

void Verify();

#endif

};

};

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

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

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

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

#include "RuntimeBasePch.h"

#ifdef PROFILE\_EXEC

#include "Base\ScriptContextProfiler.h"

namespace Js

{

ULONG

ScriptContextProfiler::AddRef()

{

return refcount++;

}

ULONG

ScriptContextProfiler::Release()

{

ULONG count = --refcount;

if (count == 0)

{

if (recycler != nullptr && this->profiler == recycler->GetProfiler())

{

recycler->SetProfiler(nullptr, nullptr);

}

NoCheckHeapDelete(this);

}

return count;

}

ScriptContextProfiler::ScriptContextProfiler() :

refcount(1), profilerArena(nullptr), profiler(nullptr), backgroundRecyclerProfilerArena(nullptr), backgroundRecyclerProfiler(nullptr), recycler(nullptr), pageAllocator(nullptr), next(nullptr)

{

}

void

ScriptContextProfiler::Initialize(PageAllocator \* pageAllocator, Recycler \* recycler)

{

Assert(!IsInitialized());

profilerArena = HeapNew(ArenaAllocator, L"Profiler", pageAllocator, Js::Throw::OutOfMemory);

profiler = Anew(profilerArena, Profiler, profilerArena);

if (recycler)

{

backgroundRecyclerProfilerArena = recycler->AddBackgroundProfilerArena();

backgroundRecyclerProfiler = Anew(profilerArena, Profiler, backgroundRecyclerProfilerArena);

#if DBG

//backgroundRecyclerProfiler is allocated from background and its guaranteed to assert below if we don't disable thread access check.

backgroundRecyclerProfiler->alloc->GetPageAllocator()->SetDisableThreadAccessCheck();

#endif

backgroundRecyclerProfiler->Begin(Js::AllPhase);

#if DBG

backgroundRecyclerProfiler->alloc->GetPageAllocator()->SetEnableThreadAccessCheck();

#endif

}

profiler->Begin(Js::AllPhase);

this->recycler = recycler;

}

void

ScriptContextProfiler::ProfilePrint(Js::Phase phase)

{

if (!IsInitialized())

{

return;

}

profiler->End(Js::AllPhase);

profiler->Print(phase);

if (this->backgroundRecyclerProfiler)

{

this->backgroundRecyclerProfiler->End(Js::AllPhase);

this->backgroundRecyclerProfiler->Print(phase);

this->backgroundRecyclerProfiler->Begin(Js::AllPhase);

}

profiler->Begin(Js::AllPhase);

}

ScriptContextProfiler::~ScriptContextProfiler()

{

if (profilerArena)

{

HeapDelete(profilerArena);

}

if (recycler && backgroundRecyclerProfilerArena)

{

#if DBG

//We are freeing from main thread, disable thread check assert.

backgroundRecyclerProfilerArena->GetPageAllocator()->SetDisableThreadAccessCheck();

#endif

recycler->ReleaseBackgroundProfilerArena(backgroundRecyclerProfilerArena);

}

}

void

ScriptContextProfiler::ProfileBegin(Js::Phase phase)

{

Assert(IsInitialized());

this->profiler->Begin(phase);

}

void

ScriptContextProfiler::ProfileEnd(Js::Phase phase)

{

Assert(IsInitialized());

this->profiler->End(phase);

}

void

ScriptContextProfiler::ProfileSuspend(Js::Phase phase, Js::Profiler::SuspendRecord \* suspendRecord)

{

Assert(IsInitialized());

this->profiler->Suspend(phase, suspendRecord);

}

void

ScriptContextProfiler::ProfileResume(Js::Profiler::SuspendRecord \* suspendRecord)

{

Assert(IsInitialized());

this->profiler->Resume(suspendRecord);

}

void

ScriptContextProfiler::ProfileMerge(ScriptContextProfiler \* profiler)

{

Assert(IsInitialized());

this->profiler->Merge(profiler->profiler);

}

}

#endif

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

// 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 ScriptContextProfiler

{

#ifdef PROFILE\_EXEC

friend class NativeCodeGenerator;

public:

ScriptContextProfiler();

~ScriptContextProfiler();

bool IsInitialized() const { return profilerArena != nullptr; }

void Initialize(PageAllocator \* pageAllocator, Recycler \* recycler);

ULONG AddRef();

ULONG Release();

Profiler \* GetProfiler() { return profiler; }

Profiler \* GetBackgroundRecyclerProfiler() { return backgroundRecyclerProfiler; }

void ProfileBegin(Js::Phase phase);

void ProfileEnd(Js::Phase phase);

void ProfileSuspend(Js::Phase, Js::Profiler::SuspendRecord \* suspendRecord);

void ProfileResume(Js::Profiler::SuspendRecord \* suspendRecord);

void ProfilePrint(Js::Phase phase);

void ProfileMerge(ScriptContextProfiler \* profiler);

private:

ArenaAllocator \* profilerArena;

ArenaAllocator \* backgroundRecyclerProfilerArena;

Profiler \* profiler;

Profiler \* backgroundRecyclerProfiler;

uint refcount;

Recycler \* recycler;

PageAllocator \*pageAllocator;

ScriptContextProfiler \*next;

#endif

};

};

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

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

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

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

#include "RuntimeBasePch.h"

#ifdef ENABLE\_DEBUG\_CONFIG\_OPTIONS

#include "Base\ScriptMemoryDumper.h"

ScriptMemoryDumper::ScriptMemoryDumper(Js::ScriptContext\* scriptContext)

:scriptContext(scriptContext)

{

memset(&current, 0, sizeof(current));

memset(&total, 0, sizeof(total));

Init();

}

void ScriptMemoryDumper::Init()

{

pageCountId = scriptContext->GetOrAddPropertyIdTracked(L"pageCount");

objectSizeId = scriptContext->GetOrAddPropertyIdTracked(L"objectSize");

freeObjectCountId = scriptContext->GetOrAddPropertyIdTracked(L"freeObjectCount");

activeObjectCountId = scriptContext->GetOrAddPropertyIdTracked(L"activeObjectCount");

totalByteCountId = scriptContext->GetOrAddPropertyIdTracked(L"totalByteCount");

finalizeCountId = scriptContext->GetOrAddPropertyIdTracked(L"finalizeCount");

weakReferenceCountId = scriptContext->GetOrAddPropertyIdTracked(L"weakReferenceCount");

largeObjectsId = scriptContext->GetOrAddPropertyIdTracked(L"largeObjects");

activeObjectByteSizeId = scriptContext->GetOrAddPropertyIdTracked(L"activeObjectByteSize");

summaryId = scriptContext->GetOrAddPropertyIdTracked(L"summary");

dumpObject = scriptContext->GetLibrary()->CreateObject();

}

// Export script related memory to javascript object containing related information.

Js::Var ScriptMemoryDumper::Dump()

{

Recycler\* recycler = scriptContext->GetRecycler();

HeapInfo\* heapInfo = recycler->GetAutoHeap();

for (uint32 i = 0 ; i < HeapConstants::BucketCount; i++)

{

ResetCurrentStats();

size\_t sizeCat = (i + 1) \* HeapConstants::ObjectGranularity;

DumpHeapBucket(i, &heapInfo->GetBucket<LeafBit>(sizeCat));

DumpHeapBucket(i, &heapInfo->GetBucket<NoBit>(sizeCat));

DumpHeapBucket(i, (SmallFinalizableHeapBucket \*)&heapInfo->GetBucket<FinalizeBit>(sizeCat));

SaveCurrentAtIndex(i);

MergeCurrentStats();

}

#ifdef BUCKETIZE\_MEDIUM\_ALLOCATIONS

for (uint32 i = 0 ; i < HeapConstants::MediumBucketCount; i++)

{

ResetCurrentStats();

size\_t sizeCat = HeapConstants::MaxSmallObjectSize + ((i + 1) \* HeapConstants::ObjectGranularity);

#if SMALLBLOCK\_MEDIUM\_ALLOC

DumpHeapBucket(i, &heapInfo->GetMediumBucket<LeafBit>(sizeCat));

DumpHeapBucket(i, &heapInfo->GetMediumBucket<NoBit>(sizeCat));

DumpHeapBucket(i, (MediumFinalizableHeapBucket \*)&heapInfo->GetMediumBucket<FinalizeBit>(sizeCat));

#else

DumpLargeBucket(&heapInfo->GetMediumBucket(sizeCat));

#endif

SaveCurrentAtIndex(i + HeapConstants::BucketCount);

MergeCurrentStats();

}

#endif

ResetCurrentStats();

DumpLargeBucket(&heapInfo->largeObjectBucket);

SaveCurrentAsLargeBlock();

MergeCurrentStats();

SaveSummary();

return dumpObject;

}

template <typename TBlockType>

void ScriptMemoryDumper::DumpHeapBucket(uint index, HeapBucketT<TBlockType>\* heapBucket)

{

SmallHeapBlockAllocator<TBlockType> \* currentAllocator = heapBucket->GetAllocator();

do

{

DumpSmallHeapBlock(currentAllocator->GetHeapBlock());

currentAllocator = currentAllocator->GetNext();

}

while (currentAllocator != heapBucket->GetAllocator());

DumpSmallHeapBlockList(heapBucket->fullBlockList);

DumpSmallHeapBlockList(heapBucket->heapBlockList);

}

template <typename TBlockType, typename TBlockAttributes>

void ScriptMemoryDumper::DumpHeapBucket(uint index, SmallNormalHeapBucketBase<TBlockType>\* heapBucket)

{

DumpHeapBucket(index, (HeapBucketT<TBlockType> \*)heapBucket);

DumpSmallHeapBlockList(((SmallNormalHeapBucketBase<TBlockType, TBlockAttributes> \*)heapBucket)->partialHeapBlockList);

DumpSmallHeapBlockList(((SmallNormalHeapBucketBase<TBlockType, TBlockAttributes> \*)heapBucket)->partialSweptHeapBlockList);

}

template <typename TBlockAttributes>

void ScriptMemoryDumper::DumpHeapBucket(uint index, SmallFinalizableHeapBucketT<TBlockAttributes>\* heapBucket)

{

DumpHeapBucket(index, (SmallNormalHeapBucketBase<SmallFinalizableHeapBlockT<TBlockAttributes>> \*)heapBucket);

DumpSmallHeapBlockList(heapBucket->pendingDisposeList);

}

template <class TBlockAttributes>

void ScriptMemoryDumper::DumpSmallHeapBlockList(SmallHeapBlockT<TBlockAttributes>\* heapBlockHead)

{

HeapBlockList::ForEach(heapBlockHead, [this](SmallHeapBlockT<TBlockAttributes> \* heapBlock)

{

DumpSmallHeapBlock(heapBlock);

});

}

template <class TBlockAttributes>

void ScriptMemoryDumper::DumpSmallHeapBlock(SmallHeapBlockT<TBlockAttributes>\* heapBlock)

{

if (heapBlock == nullptr)

return;

if (current.objectSize == 0)

{

current.objectSize = heapBlock->objectSize;

}

Assert(current.objectSize == heapBlock->GetObjectSize());

current.freeObjectCount = heapBlock->freeCount;

current.activeObjectCount += heapBlock->objectCount - heapBlock->freeCount;

if (heapBlock->IsAnyFinalizableBlock())

{

current.finalizeCount += heapBlock->AsFinalizableBlock<TBlockAttributes>()->finalizeCount;

}

current.pageCount += heapBlock->GetPageCount();

current.totalByteCount += heapBlock->GetPageCount() \* AutoSystemInfo::PageSize;

current.activeObjectByteSize += (heapBlock->objectCount - heapBlock->freeCount)\* current.objectSize;

}

void ScriptMemoryDumper::DumpLargeHeapBlockList(LargeHeapBlock\* heapBlockHead)

{

HeapBlockList::ForEach(heapBlockHead, [this](LargeHeapBlock \* heapBlock)

{

DumpLargeHeapBlock(heapBlock);

});

}

void ScriptMemoryDumper::DumpLargeBucket(LargeHeapBucket\* heapBucket)

{

DumpLargeHeapBlockList(heapBucket->fullLargeBlockList);

DumpLargeHeapBlockList(heapBucket->largeBlockList);

#ifdef RECYCLER\_PAGE\_HEAP

DumpLargeHeapBlockList(heapBucket->largePageHeapBlockList);

#endif

DumpLargeHeapBlockList(heapBucket->pendingDisposeLargeBlockList);

DumpLargeHeapBlockList(heapBucket->pendingSweepLargeBlockList);

#if defined(PARTIAL\_GC\_ENABLED) && defined(CONCURRENT\_GC\_ENABLED)

DumpLargeHeapBlockList(heapBucket->partialSweptLargeBlockList);

#endif

}

struct LargeObjectHeader;

void ScriptMemoryDumper::DumpLargeHeapBlock(LargeHeapBlock\* heapBlock)

{

if (heapBlock == nullptr)

return;

current.finalizeCount += heapBlock->finalizeCount;

current.pageCount += heapBlock->GetPageCount();

current.totalByteCount += heapBlock->GetPageCount() \* AutoSystemInfo::PageSize;

for (uint32 i = 0; i < heapBlock->allocCount; i++)

{

Memory::LargeObjectHeader\* heapHeader = heapBlock->GetHeader(i);

if (heapHeader != nullptr)

{

current.activeObjectCount++;

current.activeObjectByteSize += heapHeader->objectSize;

}

}

}

inline void ScriptMemoryDumper::ResetCurrentStats()

{

memset(&current, 0, sizeof(current));

}

inline void ScriptMemoryDumper::MergeCurrentStats()

{

total.pageCount += current.pageCount;

total.activeObjectCount += current.activeObjectCount;

total.activeObjectByteSize += current.activeObjectByteSize;

total.finalizeCount += current.finalizeCount;

total.totalByteCount += current.totalByteCount;

total.freeObjectCount += current.freeObjectCount;

}

void ScriptMemoryDumper::SaveCurrentAtIndex(uint32 index)

{

Js::DynamicObject\* currentBucket = scriptContext->GetLibrary()->CreateObject();

FillObjectWithStats(currentBucket, current);

dumpObject->SetItem(index, currentBucket, Js::PropertyOperation\_None);

}

void ScriptMemoryDumper::SaveCurrentAsLargeBlock()

{

Js::DynamicObject\* largeObjectStat = scriptContext->GetLibrary()->CreateObject();

FillObjectWithStats(largeObjectStat, current);

dumpObject->SetProperty(largeObjectsId, largeObjectStat, Js::PropertyOperation\_None, NULL);

}

void ScriptMemoryDumper::SaveSummary()

{

Js::DynamicObject\* summaryStat = scriptContext->GetLibrary()->CreateObject();

FillObjectWithStats(summaryStat, total);

dumpObject->SetProperty(summaryId, summaryStat, Js::PropertyOperation\_None, NULL);

}

void ScriptMemoryDumper::FillObjectWithStats(Js::DynamicObject\* dynamicObject, HeapStats stats)

{

dynamicObject->SetProperty(pageCountId, Js::JavascriptUInt64Number::ToVar(stats.pageCount, scriptContext), Js::PropertyOperation\_None, NULL);

dynamicObject->SetProperty(objectSizeId, Js::JavascriptNumber::New(stats.objectSize, scriptContext), Js::PropertyOperation\_None, NULL);

dynamicObject->SetProperty(freeObjectCountId, Js::JavascriptNumber::New(stats.freeObjectCount, scriptContext), Js::PropertyOperation\_None, NULL);

dynamicObject->SetProperty(activeObjectCountId, Js::JavascriptNumber::New(stats.activeObjectCount, scriptContext), Js::PropertyOperation\_None, NULL);

dynamicObject->SetProperty(activeObjectByteSizeId, Js::JavascriptUInt64Number::ToVar(stats.activeObjectByteSize, scriptContext), Js::PropertyOperation\_None, NULL);

dynamicObject->SetProperty(totalByteCountId, Js::JavascriptUInt64Number::ToVar(stats.totalByteCount, scriptContext), Js::PropertyOperation\_None, NULL);

dynamicObject->SetProperty(finalizeCountId, Js::JavascriptNumber::New(stats.finalizeCount, scriptContext), Js::PropertyOperation\_None, NULL);

}

#endif ENABLE\_DEBUG\_CONFIG\_OPTIONS

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

// 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 ENABLE\_DEBUG\_CONFIG\_OPTIONS

class ScriptMemoryDumper

{

public:

typedef struct HeapStats

{

size\_t pageCount;

uint32 objectSize;

uint32 freeObjectCount;

uint32 activeObjectCount;

size\_t activeObjectByteSize;

size\_t totalByteCount;

uint32 finalizeCount; // hostdispatch etc.

};

ScriptMemoryDumper(Js::ScriptContext\* scriptContext);

~ScriptMemoryDumper()

{

// just to reduce the noise on the stack.

dumpObject = nullptr;

}

Js::Var Dump();

private:

template <typename TBlockType>

void DumpHeapBucket(uint index, HeapBucketT<TBlockType> \* heapBucket);

template <typename TBlockType, typename TBlockAttributes>

void DumpHeapBucket(uint index, SmallNormalHeapBucketBase<TBlockType>\* heapBucket);

template <typename TBlockAttributes>

void DumpHeapBucket(uint index, SmallFinalizableHeapBucketT<TBlockAttributes>\* heapBucket);

void DumpLargeBucket(LargeHeapBucket\* bucket);

void ResetCurrentStats();

void MergeCurrentStats();

void SaveCurrentAtIndex(uint32 index);

void SaveCurrentAsLargeBlock();

void SaveSummary();

void FillObjectWithStats(Js::DynamicObject\* dynamicObject, HeapStats stats);

template <class TBlockAttributes>

void DumpSmallHeapBlockList(SmallHeapBlockT<TBlockAttributes>\* heapBlockHead);

template <class TBlockAttributes>

void DumpSmallHeapBlock(SmallHeapBlockT<TBlockAttributes>\* heapBlock);

void DumpLargeHeapBlockList(LargeHeapBlock\* heapBlockHead);

void DumpLargeHeapBlock(LargeHeapBlock\* heapBlock);

void Init();

Js::ScriptContext\* scriptContext;

Js::DynamicObject\* dumpObject;

HeapStats current;

HeapStats total;

Js::PropertyId pageCountId;

Js::PropertyId objectSizeId;

Js::PropertyId freeObjectCountId;

Js::PropertyId activeObjectCountId;

Js::PropertyId activeObjectByteSizeId;

Js::PropertyId totalByteCountId;

Js::PropertyId finalizeCountId;

Js::PropertyId weakReferenceCountId;

Js::PropertyId largeObjectsId;

Js::PropertyId summaryId;

};

#endif

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

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

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

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

#include "RuntimeBasePch.h"

namespace Js

{

LPCUTF8 const ISourceHolder::emptyString = (LPCUTF8)"\0";

SimpleSourceHolder const ISourceHolder::emptySourceHolder(emptyString, 0, true);

ISourceHolder\* SimpleSourceHolder::Clone(ScriptContext\* scriptContext)

{

if(this == ISourceHolder::GetEmptySourceHolder())

{

return this;

}

utf8char\_t \* newUtf8String = RecyclerNewArrayLeaf(scriptContext->GetRecycler(), utf8char\_t, byteLength + 1);

js\_memcpy\_s(newUtf8String, byteLength + 1, this->source, byteLength + 1);

return RecyclerNew(scriptContext->GetRecycler(), SimpleSourceHolder, newUtf8String, byteLength);

}

}

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

// 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 SimpleSourceHolder;

class ISourceHolder : public FinalizableObject

{

private:

static SimpleSourceHolder const emptySourceHolder;

static LPCUTF8 const emptyString;

public:

static ISourceHolder \*GetEmptySourceHolder()

{

return (ISourceHolder \*)&emptySourceHolder;

}

virtual LPCUTF8 GetSource(const wchar\_t\* reasonString) = 0;

virtual size\_t GetByteLength(const wchar\_t\* reasonString) = 0;

virtual ISourceHolder\* Clone(ScriptContext\* scriptContext) = 0;

virtual bool Equals(ISourceHolder\* other) = 0;

virtual int GetHashCode() = 0;

virtual bool IsEmpty() = 0;

virtual bool IsDeferrable() = 0;

};

class SimpleSourceHolder sealed : public ISourceHolder

{

friend class ISourceHolder;

private:

LPCUTF8 source;

size\_t byteLength;

bool isEmpty;

SimpleSourceHolder(LPCUTF8 source, size\_t byteLength, bool isEmpty)

: source(source),

byteLength(byteLength),

isEmpty(isEmpty)

{

}

public:

SimpleSourceHolder(LPCUTF8 source, size\_t byteLength)

: source(source),

byteLength(byteLength),

isEmpty(false)

{

}

virtual LPCUTF8 GetSource(const wchar\_t\* reasonString) override

{

return source;

}

virtual size\_t GetByteLength(const wchar\_t\* reasonString) override { return byteLength; }

virtual ISourceHolder\* Clone(ScriptContext\* scriptContext) override;

virtual bool Equals(ISourceHolder\* other) override

{

const wchar\_t\* reason = L"Equal Comparison";

return this == other ||

(this->GetByteLength(reason) == other->GetByteLength(reason)

&& (this->GetSource(reason) == other->GetSource(reason)

|| memcmp(this->GetSource(reason), other->GetSource(reason), this->GetByteLength(reason)) == 0 ));

}

virtual bool IsEmpty() override

{

return this->isEmpty;

}

virtual int GetHashCode() override

{

Assert(byteLength < MAXUINT32);

return JsUtil::CharacterBuffer<utf8char\_t>::StaticGetHashCode(source, (charcount\_t)byteLength);

}

virtual void Finalize(bool isShutdown) override

{

}

virtual void Dispose(bool isShutdown) override

{

}

virtual void Mark(Recycler \* recycler) override

{

}

virtual bool IsDeferrable() override

{

return CONFIG\_FLAG(DeferLoadingAvailableSource);

}

};

}

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

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

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

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

#include "RuntimeBasePch.h"

/\*

\* The following diagram depicts the layout of the user-mode stack. To the left is what the stack

\* looks like to the OS. To the right are the logical partitions Chakra divides the usable stack into.

\* The goal of SO checks in the runtime is to never touch a guard page. i.e. we always detect

\* that we might run out of stack space (at specific points in code) and raise a stack overflow

\* exception. We thus guarantee that our data structures are consistent even when a SO exception is raised.

\*

\* All stack probes check if the current stack pointer - requested size > current stack limit.

\*

\* stack start (low) +----------------------+

\* | MEM\_RESERVE |

\* +----------------------+

\* | Thread stack |

\* ~ guarantee ~

\* | (PAGE\_GUARD) |

\* +----------------------+ <--------------> +----------------------+

\* | MEM\_RESERVE | | |

\* ~ ~ ~ overflow handling | (B)

\* | | | buffer |

\* +----------------------+ +----------------------+ <---- stackLimit

\* | PAGE\_GUARD | | |

\* +----------------------+ | |

\* | MEM\_COMMIT | | | script region |

\* ~ PAGE\_READWRITE ~ ~ ~ (A)

\* | | | |

\* (high) +----------------------+ <--------------> +----------------------+

\*/

void

StackProber::Initialize()

{

// NumGuardPages is 2 on x86/x86-64

// 1 MEM\_RESERVE page at the bottom of the stack

// 1 PAGE\_GUARD | PAGE\_READWRITE page that serves as the guard page

const size\_t guardPageSize = Js::Constants::NumGuardPages \* AutoSystemInfo::PageSize;

const size\_t stackOverflowBuffer = Js::Constants::StackOverflowHandlingBufferPages \* AutoSystemInfo::PageSize;

PBYTE stackBottom = 0; // This is the low address limit (here we consider stack growing down).

ULONG stackGuarantee = 0;

#if defined(\_M\_IX86)

stackBottom = (PBYTE)\_\_readfsdword(0xE0C); // points to the DeAllocationStack on the TEB - which turns to be the stack bottom.

#elif defined(\_M\_AMD64)

stackBottom = (PBYTE)\_\_readgsqword(0x1478);

#elif defined(\_M\_ARM)

ULONG lowLimit, highLimit;

::GetCurrentThreadStackLimits(&lowLimit, &highLimit);

stackBottom = (PBYTE)lowLimit;

#elif defined(\_M\_ARM64)

ULONG64 lowLimit, highLimit;

::GetCurrentThreadStackLimits(&lowLimit, &highLimit);

stackBottom = (PBYTE) lowLimit;

#else

stackBottom = NULL;

Js::Throw::NotImplemented();

#endif

Assert(stackBottom);

// Calling this API with stackGuarantee == 0 \*gets\* current stack guarantee.

SetThreadStackGuarantee(&stackGuarantee);

stackLimit = stackBottom + guardPageSize + stackGuarantee + stackOverflowBuffer;

}

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

// 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

class StackProber

{

public:

void Initialize();

PBYTE GetScriptStackLimit() const { return stackLimit; }

#if DBG

void AdjustKnownStackLimit(PBYTE sp, size\_t size)

{

if (knownStackLimit == 0) knownStackLimit = sp - size;

knownStackLimit = ((sp - size) < knownStackLimit) ? (sp - size) : knownStackLimit;

}

#endif

private:

PBYTE stackLimit;

#if DBG

PBYTE knownStackLimit;

#endif

};

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

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

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

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

#include "RuntimeBasePch.h"

namespace Js

{

template <bool isGuestArena>

TempArenaAllocatorWrapper<isGuestArena>\* TempArenaAllocatorWrapper<isGuestArena>::Create(ThreadContext \* threadContext)

{

Recycler \* recycler = threadContext->GetRecycler();

TempArenaAllocatorWrapper<isGuestArena> \* wrapper = RecyclerNewFinalizedLeaf(recycler, Js::TempArenaAllocatorWrapper<isGuestArena>,

L"temp", threadContext->GetPageAllocator(), Js::Throw::OutOfMemory);

if (isGuestArena)

{

wrapper->externalGuestArenaRef = recycler->RegisterExternalGuestArena(wrapper->GetAllocator());

wrapper->recycler = recycler;

if (wrapper->externalGuestArenaRef == nullptr)

{

Js::Throw::OutOfMemory();

}

}

return wrapper;

}

template <bool isGuestArena>

TempArenaAllocatorWrapper<isGuestArena>::TempArenaAllocatorWrapper(\_\_in LPCWSTR name, PageAllocator \* pageAllocator, void (\*outOfMemoryFunc)()) :

allocator(name, pageAllocator, outOfMemoryFunc), recycler(nullptr), externalGuestArenaRef(nullptr)

{

}

template <bool isGuestArena>

void TempArenaAllocatorWrapper<isGuestArena>::Dispose(bool isShutdown)

{

allocator.Clear();

if (isGuestArena && externalGuestArenaRef != nullptr)

{

this->recycler->UnregisterExternalGuestArena(externalGuestArenaRef);

externalGuestArenaRef = nullptr;

}

Assert(allocator.AllocatedSize() == 0);

}

// Explicit instantiation

template class TempArenaAllocatorWrapper<true>;

template class TempArenaAllocatorWrapper<false>;

} // 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.

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

#pragma once

namespace Js

{

template <bool isGuestArena>

class TempArenaAllocatorWrapper sealed : public FinalizableObject

{

private:

ArenaAllocator allocator;

ArenaData \*\* externalGuestArenaRef;

Recycler \* recycler;

TempArenaAllocatorWrapper(\_\_in LPCWSTR name, PageAllocator \* pageAllocator, void (\*outOfMemoryFunc)());

public:

static TempArenaAllocatorWrapper\* Create(ThreadContext \* threadContext);

virtual void Finalize(bool isShutdown) override

{

}

virtual void Dispose(bool isShutdown) override;

virtual void Mark(Recycler \*recycler) override { AssertMsg(false, "Mark called on object that isn't TrackableObject"); }

ArenaAllocator \*GetAllocator()

{

return &allocator;

}

};

typedef TempArenaAllocatorWrapper<true> TempGuestArenaAllocatorObject;

typedef TempArenaAllocatorWrapper<false> TempArenaAllocatorObject;

}

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

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

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

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

#include "RuntimeBasePch.h"

#ifdef TEST\_ETW\_EVENTS

#include "Base\EtwTrace.h"

char const \* const TestEtwEventSink::CreateEventSinkProcName = STRINGIZE(CREATE\_EVENTSINK\_PROC\_NAME);

TestEtwEventSink\* TestEtwEventSink::Instance = NULL;

bool TestEtwEventSink::Load()

{

wchar\_t const \* dllname = Js::Configuration::Global.flags.TestEtwDll;

if(!dllname)

{

return false;

}

HMODULE hModule = ::LoadLibraryW(dllname);

if (hModule == nullptr)

{

Output::Print(L"ERROR: Unable to load ETW event sink %s\n", dllname);

Js::Throw::FatalInternalError();

}

CreateEventSink procAddress = (CreateEventSink)::GetProcAddress(hModule, CreateEventSinkProcName);

if (procAddress == nullptr)

{

Output::Print(L"ERROR: Unable to get function %S from dll %s\n", CreateEventSinkProcName, dllname);

Js::Throw::FatalInternalError();

}

// CONSIDER: pass null and skip rundown testing (if a command line switch is present).

Instance = procAddress(&EtwTrace::PerformRundown, PHASE\_TRACE1(Js::EtwPhase));

if (Instance == nullptr)

{

Output::Print(L"ERROR: Failed to create ETW event sink from dll %s\n", dllname);

Js::Throw::FatalInternalError();

}

return true;

}

bool TestEtwEventSink::IsLoaded()

{

return Instance != NULL;

}

void TestEtwEventSink::Unload()

{

if(Instance != NULL)

{

Instance->UnloadInstance();

Instance = NULL;

}

}

#endif

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

// 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 TEST\_ETW\_EVENTS

#define CREATE\_EVENTSINK\_PROC\_NAME CreateEventSink

// Abstract class to be inherited to receive ETW event notifications

class TestEtwEventSink

{

public:

virtual void WriteMethodEvent(const wchar\_t\* eventName,

void\* scriptContextId,

void\* methodStartAddress,

uint64 methodSize,

uint methodID,

uint16 methodFlags,

uint16 methodAddressRangeID,

DWORD\_PTR sourceID,

uint line,

uint column,

const wchar\_t\* methodName) = 0;

virtual void WriteSourceEvent(const wchar\_t\* eventName, uint64 sourceContext, void\* scriptContextId, uint sourceFlags, const wchar\_t\* url) = 0;

virtual void UnloadInstance() = 0;

static bool IsLoaded();

static bool Load();

static void Unload();

static TestEtwEventSink\* Instance;

typedef void (\*RundownFunc)(bool start);

private:

static char const \* const TestEtwEventSink::CreateEventSinkProcName;

typedef TestEtwEventSink \* (\*CreateEventSink)(RundownFunc rundown, bool trace);

};

#endif

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

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

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

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

#include "RuntimeBasePch.h"

#include "Base\ThreadContextTLSEntry.h"

#include "Base\ThreadBoundThreadContextManager.h"

ThreadBoundThreadContextManager::EntryList ThreadBoundThreadContextManager::entries(&HeapAllocator::Instance);

JsUtil::BackgroundJobProcessor \* ThreadBoundThreadContextManager::s\_sharedJobProcessor = NULL;

CriticalSection ThreadBoundThreadContextManager::s\_sharedJobProcessorCreationLock;

uint ThreadBoundThreadContextManager::s\_maxNumberActiveThreadContexts = 0;

ThreadContext \* ThreadBoundThreadContextManager::EnsureContextForCurrentThread()

{

AutoCriticalSection lock(ThreadContext::GetCriticalSection());

ThreadContextTLSEntry \* entry = ThreadContextTLSEntry::GetEntryForCurrentThread();

if (entry == NULL)

{

ThreadContextTLSEntry::CreateEntryForCurrentThread();

entry = ThreadContextTLSEntry::GetEntryForCurrentThread();

entries.Prepend(entry);

}

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

// An existing TLS entry may have a null ThreadContext

// DllCanUnload may have cleaned out all the TLS entry when the module lock count is 0,

// but the library didn't get unloaded because someone is holding onto ref count via LoadLibrary.

// Just reinitialize the thread context.

if (threadContext == nullptr)

{

threadContext = HeapNew(ThreadContext);

threadContext->SetIsThreadBound();

if (!ThreadContextTLSEntry::TrySetThreadContext(threadContext))

{

HeapDelete(threadContext);

return NULL;

}

}

Assert(threadContext != NULL);

s\_maxNumberActiveThreadContexts = max(s\_maxNumberActiveThreadContexts, GetActiveThreadContextCount());

return threadContext;

}

void ThreadBoundThreadContextManager::DestroyContextAndEntryForCurrentThread()

{

AutoCriticalSection lock(ThreadContext::GetCriticalSection());

ThreadContextTLSEntry \* entry = ThreadContextTLSEntry::GetEntryForCurrentThread();

if (entry == NULL)

{

return;

}

ThreadContext \* threadContext = static\_cast<ThreadContext \*>(entry->GetThreadContext());

entries.Remove(entry);

if (threadContext != NULL && threadContext->GetIsThreadBound())

{

ShutdownThreadContext(threadContext);

}

ThreadContextTLSEntry::CleanupThread();

}

void ThreadBoundThreadContextManager::DestroyAllContexts()

{

JsUtil::BackgroundJobProcessor \* jobProcessor = NULL;

{

AutoCriticalSection lock(ThreadContext::GetCriticalSection());

ThreadContextTLSEntry \* currentEntry = ThreadContextTLSEntry::GetEntryForCurrentThread();

if (currentEntry == NULL)

{

// We need a current thread entry so that we can use it to release any thread contexts

// we find below.

try

{

AUTO\_NESTED\_HANDLED\_EXCEPTION\_TYPE(ExceptionType\_OutOfMemory);

currentEntry = ThreadContextTLSEntry::CreateEntryForCurrentThread();

entries.Prepend(currentEntry);

}

catch (Js::OutOfMemoryException)

{

return;

}

}

else

{

// We need to clear out the current thread entry so that we can use it to release any

// thread contexts we find below.

ThreadContext \* threadContext = static\_cast<ThreadContext \*>(currentEntry->GetThreadContext());

if (threadContext != NULL)

{

if (threadContext->GetIsThreadBound())

{

ShutdownThreadContext(threadContext);

ThreadContextTLSEntry::ClearThreadContext(currentEntry, false);

}

else

{

ThreadContextTLSEntry::ClearThreadContext(currentEntry, true);

}

}

}

EntryList::Iterator iter(&entries);

while (iter.Next())

{

ThreadContextTLSEntry \* entry = iter.Data();

ThreadContext \* threadContext = static\_cast<ThreadContext \*>(entry->GetThreadContext());

if (threadContext != nullptr)

{

// Found a thread context. Remove it from the containing entry.

ThreadContextTLSEntry::ClearThreadContext(entry, true);

// Now set it to our thread's entry.

ThreadContextTLSEntry::SetThreadContext(currentEntry, threadContext);

// Clear it out.

ShutdownThreadContext(threadContext);

// Now clear it out of our entry.

ThreadContextTLSEntry::ClearThreadContext(currentEntry, false);

}

}

// We can only clean up our own TLS entry, so we're going to go ahead and do that here.

entries.Remove(currentEntry);

ThreadContextTLSEntry::CleanupThread();

if (s\_sharedJobProcessor != NULL)

{

jobProcessor = s\_sharedJobProcessor;

s\_sharedJobProcessor = NULL;

jobProcessor->Close();

}

}

if (jobProcessor != NULL)

{

HeapDelete(jobProcessor);

}

}

void ThreadBoundThreadContextManager::DestroyAllContextsAndEntries()

{

AutoCriticalSection lock(ThreadContext::GetCriticalSection());

while (!entries.Empty())

{

ThreadContextTLSEntry \* entry = entries.Head();

ThreadContext \* threadContext = static\_cast<ThreadContext \*>(entry->GetThreadContext());

entries.RemoveHead();

if (threadContext != nullptr)

{

#if DBG

PageAllocator\* pageAllocator = threadContext->GetPageAllocator();

if (pageAllocator)

{

pageAllocator->SetConcurrentThreadId(::GetCurrentThreadId());

}

#endif

threadContext->ShutdownThreads();

HeapDelete(threadContext);

}

ThreadContextTLSEntry::Delete(entry);

}

if (s\_sharedJobProcessor != NULL)

{

s\_sharedJobProcessor->Close();

HeapDelete(s\_sharedJobProcessor);

s\_sharedJobProcessor = NULL;

}

}

JsUtil::JobProcessor \* ThreadBoundThreadContextManager::GetSharedJobProcessor()

{

if (s\_sharedJobProcessor == NULL)

{

// Don't use ThreadContext::GetCriticalSection() because it's also locked during thread detach while the loader lock is

// held, and that may prevent the background job processor's thread from being started due to contention on the loader

// lock, leading to a deadlock

AutoCriticalSection lock(&s\_sharedJobProcessorCreationLock);

if (s\_sharedJobProcessor == NULL)

{

// We don't need to have allocation policy manager for web worker.

s\_sharedJobProcessor = HeapNew(JsUtil::BackgroundJobProcessor, NULL, NULL, false /\*disableParallelThreads\*/);

}

}

return s\_sharedJobProcessor;

}

void RentalThreadContextManager::DestroyThreadContext(ThreadContext\* threadContext)

{

ShutdownThreadContext(threadContext);

}

void ThreadContextManagerBase::ShutdownThreadContext(ThreadContext\* threadContext)

{

#if DBG

PageAllocator\* pageAllocator = threadContext->GetPageAllocator();

if (pageAllocator)

{

pageAllocator->SetConcurrentThreadId(::GetCurrentThreadId());

}

#endif

threadContext->ShutdownThreads();

HeapDelete(threadContext);

}

uint ThreadBoundThreadContextManager::GetActiveThreadContextCount()

{

return entries.Count();

}

void ThreadBoundThreadContextManager::ResetMaxNumberActiveThreadContexts()

{

s\_maxNumberActiveThreadContexts = GetActiveThreadContextCount();

}

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

// 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

class ThreadContextManagerBase

{

protected:

static void ShutdownThreadContext(ThreadContext\* threadContext);

};

class ThreadBoundThreadContextManager : public ThreadContextManagerBase

{

friend class ThreadContext;

public:

typedef DListCounted<ThreadContextTLSEntry \*, HeapAllocator> EntryList;

static ThreadContext \* EnsureContextForCurrentThread();

static void DestroyContextAndEntryForCurrentThread();

static void DestroyAllContexts();

static void DestroyAllContextsAndEntries();

static JsUtil::JobProcessor \* GetSharedJobProcessor();

static uint GetActiveThreadContextCount();

static void ResetMaxNumberActiveThreadContexts();

static uint s\_maxNumberActiveThreadContexts;

private:

static EntryList entries;

static JsUtil::BackgroundJobProcessor \* s\_sharedJobProcessor;

static CriticalSection s\_sharedJobProcessorCreationLock;

};

class RentalThreadContextManager : public ThreadContextManagerBase

{

public:

static void DestroyThreadContext(ThreadContext\* threadContext);

};

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

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

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

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

// Format: FLAG\*(ThreadFlag, GlobalFlag)

FLAG(IsCollectGarbageEnabled, CollectGarbage)

FLAG(IsErrorStackTraceEnabled, errorStackTrace)

FLAG(Is\_\_proto\_\_Enabled, \_\_proto\_\_)

FLAG(IsES6MapEnabled, Map)

FLAG(IsES6SetEnabled, Set)

FLAG(IsES6WeakMapEnabled, WeakMap)

FLAG(IsDefineGetterSetterEnabled, DefineGetterSetter)

FLAG(IsLetAndConstEnabled, LetConst)

FLAG(IsES6UnicodeVerboseEnabled, ES6UnicodeVerbose)

FLAG\_RELEASE(IsIntlEnabled, Intl)

FLAG\_RELEASE(IsES6SpeciesEnabled, ES6Species)

FLAG\_RELEASE(IsES6ClassAndExtendsEnabled, ES6Classes)

FLAG\_RELEASE(IsES6DateParseFixEnabled, ES6DateParseFix)

FLAG\_RELEASE(IsES6DefaultArgsEnabled, ES6DefaultArgs)

FLAG\_RELEASE(IsES6DestructuringEnabled, ES6Destructuring)

FLAG\_RELEASE(IsES6ForLoopSemanticsEnabled, ES6ForLoopSemantics)

FLAG\_RELEASE(IsES6FunctionNameEnabled, ES6FunctionName)

FLAG\_RELEASE(IsES6FunctionNameFullEnabled, ES6FunctionNameFull)

FLAG\_RELEASE(IsES6GeneratorsEnabled, ES6Generators)

FLAG\_RELEASE(IsES7ExponentiationOperatorEnabled, ES7ExponentiationOperator)

FLAG\_RELEASE(IsES7TrailingCommaEnabled, ES7TrailingComma)

FLAG\_RELEASE(IsES7BuiltinsEnabled, ES7Builtins)

FLAG\_RELEASE(IsES6IteratorsEnabled, ES6Iterators)

FLAG\_RELEASE(IsES6IsConcatSpreadableEnabled, ES6IsConcatSpreadable)

FLAG\_RELEASE(IsES6LambdaEnabled, ES6Lambda)

FLAG\_RELEASE(IsES6MathExtensionsEnabled, ES6Math)

FLAG\_RELEASE(IsES6ObjectExtensionsEnabled, ES6Object)

FLAG\_RELEASE(IsES6NumberExtensionsEnabled, ES6Number)

FLAG\_RELEASE(IsES6NumericLiteralEnabled, ES6NumericLiterals)

FLAG\_RELEASE(IsES6ObjectLiteralsEnabled, ES6ObjectLiterals)

FLAG\_RELEASE(IsES6PromiseEnabled, ES6Promise)

FLAG\_RELEASE(IsES6ProxyEnabled, ES6Proxy)

FLAG\_RELEASE(IsES6RestEnabled, ES6Rest)

FLAG\_RELEASE(IsES6SpreadEnabled, ES6Spread)

FLAG\_RELEASE(IsES6StringExtensionsEnabled, ES6String)

FLAG\_RELEASE(IsES6StringPrototypeFixEnabled, ES6StringPrototypeFixes)

FLAG\_RELEASE(IsES6StringTemplateEnabled, ES6StringTemplate)

FLAG\_RELEASE(IsES6PrototypeChain, ES6PrototypeChain)

FLAG\_RELEASE(IsES6SymbolEnabled, ES6Symbol)

FLAG\_RELEASE(IsES6ToPrimitiveEnabled, ES6ToPrimitive)

FLAG\_RELEASE(IsES6ToLengthEnabled, ES6ToLength)

FLAG\_RELEASE(IsES6ToStringTagEnabled, ES6ToStringTag)

FLAG\_RELEASE(IsES6TypedArrayExtensionsEnabled, ES6TypedArrayExtensions)

FLAG\_RELEASE(IsES6UnicodeExtensionsEnabled, ES6Unicode)

FLAG\_RELEASE(IsES6UnscopablesEnabled, ES6Unscopables)

FLAG\_RELEASE(IsES6WeakSetEnabled, ES6WeakSet)

FLAG\_RELEASE(IsES6RegExStickyEnabled, ES6RegExSticky)

FLAG\_RELEASE(IsES6RegExPrototypePropertiesEnabled, ES6RegExPrototypeProperties)

FLAG\_RELEASE(IsES6HasInstanceEnabled, ES6HasInstance)

FLAG\_RELEASE(SkipSplitOnNoResult, SkipSplitOnNoResult)

FLAG\_RELEASE(IsES7AsyncAndAwaitEnabled, ES7AsyncAwait)

FLAG\_RELEASE(IsArrayBufferTransferEnabled, ArrayBufferTransfer)

#ifdef ENABLE\_PROJECTION

FLAG(AreWinRTDelegatesInterfaces, WinRTDelegateInterfaces)

FLAG\_RELEASE(IsWinRTAdaptiveAppsEnabled, WinRTAdaptiveApps)

#endif

//SIMD\_JS

FLAG\_RELEASE(IsSimdjsEnabled, Simdjs)

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

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

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

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

#include "RuntimeBasePch.h"

#include "BackEndAPI.h"

#include "ThreadServiceWrapper.h"

#include "Types\TypePropertyCache.h"

#include "Debug\DebuggingFlags.h"

#include "Debug\DiagProbe.h"

#include "Debug\DebugManager.h"

#include "Chars.h"

#include "CaseInsensitive.h"

#include "CharSet.h"

#include "CharMap.h"

#include "StandardChars.h"

#include "Base\ThreadContextTLSEntry.h"

#include "Base\ThreadBoundThreadContextManager.h"

#include "Language\SourceDynamicProfileManager.h"

#include "Language\CodeGenRecyclableData.h"

#include "Language\InterpreterStackFrame.h"

#include "Language\JavascriptStackWalker.h"

#include "Base\ScriptMemoryDumper.h"

// SIMD\_JS

#include "Library\SimdLib.h"

#if DBG

#include "Memory\StressTest.h"

#endif

#ifdef DYNAMIC\_PROFILE\_MUTATOR

#include "Language\DynamicProfileMutator.h"

#endif

#ifdef ENABLE\_BASIC\_TELEMETRY

#include "Telemetry.h"

#endif

int TotalNumberOfBuiltInProperties = Js::PropertyIds::\_countJSOnlyProperty;

/\*

\* When we aren't adding any additional properties

\*/

void DefaultInitializeAdditionalProperties(ThreadContext \*threadContext)

{

}

/\*

\*

\*/

void (\*InitializeAdditionalProperties)(ThreadContext \*threadContext) = DefaultInitializeAdditionalProperties;

// To make sure the marker function doesn't get inlined, optimized away, or merged with other functions we disable optimization.

// If this method ends up causing a perf problem in the future, we should replace it with asm versions which should be lighter.

#pragma optimize("g", off)

extern "C" void\* MarkerForExternalDebugStep()

{

// We need to return something here to prevent this function from being merged with other empty functions by the linker.

static int \_\_dummy;

return &\_\_dummy;

}

#pragma optimize("", on)

CriticalSection ThreadContext::s\_csThreadContext;

size\_t ThreadContext::processNativeCodeSize = 0;

ThreadContext \* ThreadContext::globalListFirst = nullptr;

ThreadContext \* ThreadContext::globalListLast = nullptr;

uint ThreadContext::activeScriptSiteCount = 0;

#if !\_M\_X64\_OR\_ARM64 && \_CONTROL\_FLOW\_GUARD

uint ThreadContext::numOfThreadContextsWithPreReserveSegment = 0;

#endif

const Js::PropertyRecord \* const ThreadContext::builtInPropertyRecords[] =

{

Js::BuiltInPropertyRecords::EMPTY,

#define ENTRY\_INTERNAL\_SYMBOL(n) Js::BuiltInPropertyRecords::n,

#define ENTRY\_SYMBOL(n, d) Js::BuiltInPropertyRecords::n,

#define ENTRY(n) Js::BuiltInPropertyRecords::n,

#define ENTRY2(n, s) ENTRY(n)

#include "Base\JnDirectFields.h"

};

ThreadContext::RecyclableData::RecyclableData(Recycler \*const recycler) :

soErrorObject(nullptr, nullptr, nullptr, true),

oomErrorObject(nullptr, nullptr, nullptr, true),

terminatedErrorObject(nullptr, nullptr, nullptr),

typesWithProtoPropertyCache(recycler),

propertyGuards(recycler, 128),

oldEntryPointInfo(nullptr),

returnedValueList(nullptr)

{

}

ThreadContext::ThreadContext(AllocationPolicyManager \* allocationPolicyManager, JsUtil::ThreadService::ThreadServiceCallback threadServiceCallback, bool enableExperimentalFeatures) :

currentThreadId(::GetCurrentThreadId()),

stackLimitForCurrentThread(0),

stackProber(nullptr),

isThreadBound(false),

hasThrownPendingException(false),

noScriptScope(false),

heapEnum(nullptr),

threadContextFlags(ThreadContextFlagNoFlag),

JsUtil::DoublyLinkedListElement<ThreadContext>(),

allocationPolicyManager(allocationPolicyManager),

threadService(threadServiceCallback),

isOptimizedForManyInstances(Js::Configuration::Global.flags.OptimizeForManyInstances),

bgJit(Js::Configuration::Global.flags.BgJit),

pageAllocator(allocationPolicyManager, PageAllocatorType\_Thread, Js::Configuration::Global.flags, 0, PageAllocator::DefaultMaxFreePageCount,

false, &backgroundPageQueue),

recycler(nullptr),

hasCollectionCallBack(false),

callDispose(true),

#if ENABLE\_NATIVE\_CODEGEN

jobProcessor(nullptr),

#endif

interruptPoller(nullptr),

expirableCollectModeGcCount(-1),

expirableObjectList(nullptr),

expirableObjectDisposeList(nullptr),

numExpirableObjects(0),

disableExpiration(false),

callRootLevel(0),

nextTypeId((Js::TypeId)Js::Constants::ReservedTypeIds),

entryExitRecord(nullptr),

leafInterpreterFrame(nullptr),

threadServiceWrapper(nullptr),

temporaryArenaAllocatorCount(0),

temporaryGuestArenaAllocatorCount(0),

crefSContextForDiag(0),

scriptContextList(nullptr),

scriptContextEverRegistered(false),

#if DBG\_DUMP || defined(PROFILE\_EXEC)

topLevelScriptSite(nullptr),

#endif

polymorphicCacheState(0),

stackProbeCount(0),

#ifdef BAILOUT\_INJECTION

bailOutByteCodeLocationCount(0),

#endif

sourceCodeSize(0),

nativeCodeSize(0),

threadAlloc(L"TC", GetPageAllocator(), Js::Throw::OutOfMemory),

inlineCacheThreadInfoAllocator(L"TC-InlineCacheInfo", GetPageAllocator(), Js::Throw::OutOfMemory),

isInstInlineCacheThreadInfoAllocator(L"TC-IsInstInlineCacheInfo", GetPageAllocator(), Js::Throw::OutOfMemory),

equivalentTypeCacheInfoAllocator(L"TC-EquivalentTypeCacheInfo", GetPageAllocator(), Js::Throw::OutOfMemory),

protoInlineCacheByPropId(&inlineCacheThreadInfoAllocator, 512),

storeFieldInlineCacheByPropId(&inlineCacheThreadInfoAllocator, 256),

isInstInlineCacheByFunction(&isInstInlineCacheThreadInfoAllocator, 128),

registeredInlineCacheCount(0),

unregisteredInlineCacheCount(0),

prototypeChainEnsuredToHaveOnlyWritableDataPropertiesAllocator(L"TC-ProtoWritableProp", GetPageAllocator(), Js::Throw::OutOfMemory),

standardUTF8Chars(0),

standardUnicodeChars(0),

hasUnhandledException(FALSE),

hasCatchHandler(FALSE),

disableImplicitFlags(DisableImplicitNoFlag),

hasCatchHandlerToUserCode(false),

propertyMap(nullptr),

caseInvariantPropertySet(nullptr),

entryPointToBuiltInOperationIdCache(&threadAlloc, 0),

#if ENABLE\_NATIVE\_CODEGEN

codeGenNumberThreadAllocator(nullptr),

#endif

dynamicObjectEnumeratorCacheMap(&HeapAllocator::Instance, 16),

//threadContextFlags(ThreadContextFlagNoFlag),

telemetryBlock(&localTelemetryBlock),

configuration(enableExperimentalFeatures),

jsrtRuntime(nullptr),

rootPendingClose(nullptr),

wellKnownHostTypeHTMLAllCollectionTypeId(Js::TypeIds\_Undefined),

isProfilingUserCode(true),

loopDepth(0),

maxGlobalFunctionExecTime(0.0),

isAllJITCodeInPreReservedRegion(true),

activityId(GUID\_NULL),

tridentLoadAddress(nullptr),

debugManager(nullptr)

#ifdef ENABLE\_DIRECTCALL\_TELEMETRY

, directCallTelemetry(this)

#endif

{

pendingProjectionContextCloseList = JsUtil::List<IProjectionContext\*, ArenaAllocator>::New(GetThreadAlloc());

hostScriptContextStack = Anew(GetThreadAlloc(), JsUtil::Stack<HostScriptContext\*>, GetThreadAlloc());

functionCount = 0;

sourceInfoCount = 0;

scriptContextCount=0;

isScriptActive = false;

entropy.Initialize();

#if ENABLE\_NATIVE\_CODEGEN

this->bailOutRegisterSaveSpace = AnewArrayZ(this->GetThreadAlloc(), Js::Var, GetBailOutRegisterSaveSlotCount());

#endif

// SIMD\_JS

#if ENABLE\_NATIVE\_CODEGEN

simdFuncInfoToOpcodeMap = Anew(this->GetThreadAlloc(), FuncInfoToOpcodeMap, this->GetThreadAlloc());

simdOpcodeToSignatureMap = AnewArrayZ(this->GetThreadAlloc(), SimdFuncSignature, Js::Simd128OpcodeCount());

{

#define MACRO\_SIMD\_WMS(op, LayoutAsmJs, OpCodeAttrAsmJs, OpCodeAttr, ...) \

AddSimdFuncToMaps(Js::OpCode::##op, \_\_VA\_ARGS\_\_);

#define MACRO\_SIMD\_EXTEND\_WMS(op, LayoutAsmJs, OpCodeAttrAsmJs, OpCodeAttr, ...) MACRO\_SIMD\_WMS(op, LayoutAsmJs, OpCodeAttrAsmJs, OpCodeAttr, \_\_VA\_ARGS\_\_)

#include "ByteCode\OpCodesSimd.h"

}

#endif

#if DBG\_DUMP

scriptSiteCount = 0;

pageAllocator.debugName = L"Thread";

#endif

#ifdef DYNAMIC\_PROFILE\_MUTATOR

this->dynamicProfileMutator = DynamicProfileMutator::GetMutator();

#endif

PERF\_COUNTER\_INC(Basic, ThreadContext);

#ifdef LEAK\_REPORT

this->rootTrackerScriptContext = nullptr;

this->threadId = ::GetCurrentThreadId();

#endif

memset(&localTelemetryBlock, 0, sizeof(localTelemetryBlock));

AutoCriticalSection autocs(ThreadContext::GetCriticalSection());

ThreadContext::LinkToBeginning(this, &ThreadContext::globalListFirst, &ThreadContext::globalListLast);

#if DBG

// Since we created our page allocator while we were constructing this thread context

// it will pick up the thread context id that is current on the thread. We need to update

// that now.

pageAllocator.UpdateThreadContextHandle((ThreadContextId)this);

#endif

#ifdef ENABLE\_PROJECTION

#if DBG\_DUMP

this->projectionMemoryInformation = nullptr;

#endif

#endif

this->InitAvailableCommit();

}

void ThreadContext::InitAvailableCommit()

{

// Once per process: get the available commit for the process from the OS and push it to the AutoSystemInfo.

// (This must be done lazily, outside DllMain. And it must be done from the Runtime, since the common lib

// doesn't have access to the DelayLoadLibrary stuff.)

ULONG64 commit;

BOOL success = AutoSystemInfo::Data.GetAvailableCommit(&commit);

if (!success)

{

commit = (ULONG64)-1;

#ifdef NTBUILD

APP\_MEMORY\_INFORMATION AppMemInfo;

success = GetWinCoreProcessThreads()->GetProcessInformation(

GetCurrentProcess(),

ProcessAppMemoryInfo,

&AppMemInfo,

sizeof(AppMemInfo));

if (success)

{

commit = AppMemInfo.AvailableCommit;

}

#endif

AutoSystemInfo::Data.SetAvailableCommit(commit);

}

}

void ThreadContext::SetStackProber(StackProber \* stackProber)

{

this->stackProber = stackProber;

if (stackProber != NULL && this->stackLimitForCurrentThread != Js::Constants::StackLimitForScriptInterrupt)

{

this->stackLimitForCurrentThread = stackProber->GetScriptStackLimit();

}

}

PBYTE ThreadContext::GetScriptStackLimit() const

{

return stackProber->GetScriptStackLimit();

}

IActiveScriptProfilerHeapEnum\* ThreadContext::GetHeapEnum()

{

return heapEnum;

}

void ThreadContext::SetHeapEnum(IActiveScriptProfilerHeapEnum\* newHeapEnum)

{

Assert((newHeapEnum != nullptr && heapEnum == nullptr) || (newHeapEnum == nullptr && heapEnum != nullptr));

heapEnum = newHeapEnum;

}

void ThreadContext::ClearHeapEnum()

{

Assert(heapEnum != nullptr);

heapEnum = nullptr;

}

void ThreadContext::GlobalInitialize()

{

for (int i = 0; i < \_countof(builtInPropertyRecords); i++)

{

builtInPropertyRecords[i]->SetHash(JsUtil::CharacterBuffer<WCHAR>::StaticGetHashCode(builtInPropertyRecords[i]->GetBuffer(), builtInPropertyRecords[i]->GetLength()));

}

}

void ThreadContext::IncrementThreadContextsWithPreReservedSegment()

{

#if !\_M\_X64\_OR\_ARM64 && \_CONTROL\_FLOW\_GUARD

InterlockedIncrement(&ThreadContext::numOfThreadContextsWithPreReserveSegment);

#endif

}

void ThreadContext::ReleasePreReservedSegment()

{

BOOL success = preReservedVirtualAllocator.Shutdown();

if (success)

{

#if !\_M\_X64\_OR\_ARM64 && \_CONTROL\_FLOW\_GUARD

Assert(numOfThreadContextsWithPreReserveSegment > 0);

InterlockedDecrement(&numOfThreadContextsWithPreReserveSegment);

#endif

}

}

ThreadContext::~ThreadContext()

{

{

AutoCriticalSection autocs(ThreadContext::GetCriticalSection());

ThreadContext::Unlink(this, &ThreadContext::globalListFirst, &ThreadContext::globalListLast);

}

#ifdef LEAK\_REPORT

if (Js::Configuration::Global.flags.IsEnabled(Js::LeakReportFlag))

{

AUTO\_LEAK\_REPORT\_SECTION(Js::Configuration::Global.flags, L"Thread Context (%p): %s (TID: %d)", this,

this->GetRecycler()->IsInDllCanUnloadNow()? L"DllCanUnloadNow" :

this->GetRecycler()->IsInDetachProcess()? L"DetachProcess" : L"Destructor", this->threadId);

LeakReport::DumpUrl(this->threadId);

}

#endif

if (interruptPoller)

{

HeapDelete(interruptPoller);

interruptPoller = nullptr;

}

#if DBG

// ThreadContext dtor may be running on a different thread.

// Recycler may call finalizer that free temp Arenas, which will free pages back to

// the page Allocator, which will try to suspend idle on a different thread.

// So we need to disable idle decommit asserts.

pageAllocator.ShutdownIdleDecommit();

#endif

// Allocating memory during the shutdown codepath is not preferred

// so we'll close the page allocator before we release the GC

// If any dispose is allocating memory during shutdown, that is a bug

pageAllocator.Close();

// The recycler need to delete before the background code gen thread

// because that might run finalizer which need access to the background code gen thread.

if (recycler != nullptr)

{

for (Js::ScriptContext \*scriptContext = scriptContextList; scriptContext; scriptContext = scriptContext->next)

{

if (!scriptContext->IsClosed())

{

// We close ScriptContext here because anyhow HeapDelete(recycler) when disposing the

// JavaScriptLibrary will close ScriptContext. Explicit close gives us chance to clear

// other things to which ScriptContext holds reference to

AssertMsg(!IsInScript(), "Can we be in script here?");

scriptContext->MarkForClose();

}

}

// If all scriptContext's have been closed, then the sourceProfileManagersByUrl

// should have been released

AssertMsg(this->recyclableData->sourceProfileManagersByUrl == nullptr ||

this->recyclableData->sourceProfileManagersByUrl->Count() == 0, "There seems to have been a refcounting imbalance.");

this->recyclableData->sourceProfileManagersByUrl = nullptr;

this->recyclableData->oldEntryPointInfo = nullptr;

if (this->recyclableData->symbolRegistrationMap != nullptr)

{

this->recyclableData->symbolRegistrationMap->Clear();

this->recyclableData->symbolRegistrationMap = nullptr;

}

if (this->recyclableData->returnedValueList != nullptr)

{

this->recyclableData->returnedValueList->Clear();

this->recyclableData->returnedValueList = nullptr;

}

if (this->propertyMap != nullptr)

{

HeapDelete(this->propertyMap);

this->propertyMap = nullptr;

}

// Unpin the memory for leak report so we don't report this as a leak.

recyclableData.Unroot(recycler);

#if defined(LEAK\_REPORT) || defined(CHECK\_MEMORY\_LEAK)

for (Js::ScriptContext \*scriptContext = scriptContextList; scriptContext; scriptContext = scriptContext->next)

{

scriptContext->ClearSourceContextInfoMaps();

scriptContext->ShutdownClearSourceLists();

}

#ifdef LEAK\_REPORT

// heuristically figure out which one is the root tracker script engine

// and force close on it

if (this->rootTrackerScriptContext != nullptr)

{

this->rootTrackerScriptContext->Close(false);

}

#endif

#endif

#if ENABLE\_NATIVE\_CODEGEN

HeapDelete(this->codeGenNumberThreadAllocator);

this->codeGenNumberThreadAllocator = nullptr;

#endif

if (this->debugManager != nullptr)

{

this->debugManager->Close();

HeapDelete(this->debugManager);

this->debugManager = nullptr;

}

HeapDelete(recycler);

}

#if ENABLE\_NATIVE\_CODEGEN

if(jobProcessor)

{

if(this->bgJit)

{

HeapDelete(static\_cast<JsUtil::BackgroundJobProcessor \*>(jobProcessor));

}

else

{

HeapDelete(static\_cast<JsUtil::ForegroundJobProcessor \*>(jobProcessor));

}

jobProcessor = nullptr;

}

#endif

// Do not require all GC callbacks to be revoked, because Trident may not revoke if there

// is a leak, and we don't want the leak to be masked by an assert

#ifdef ENABLE\_PROJECTION

externalWeakReferenceCacheList.Clear(&HeapAllocator::Instance);

#endif

this->collectCallBackList.Clear(&HeapAllocator::Instance);

this->protoInlineCacheByPropId.Reset();

this->storeFieldInlineCacheByPropId.Reset();

this->isInstInlineCacheByFunction.Reset();

this->inlineCacheScriptContexts.Reset();

this->isInstInlineCacheScriptContexts.Reset();

this->equivalentTypeCacheEntryPoints.Reset();

this->prototypeChainEnsuredToHaveOnlyWritableDataPropertiesScriptContext.Reset();

this->registeredInlineCacheCount = 0;

this->unregisteredInlineCacheCount = 0;

AssertMsg(this->GetHeapEnum() == nullptr, "Heap enumeration should have been cleared/closed by the ScriptSite.");

if (this->GetHeapEnum() != nullptr)

{

this->ClearHeapEnum();

}

#ifdef BAILOUT\_INJECTION

if (Js::Configuration::Global.flags.IsEnabled(Js::BailOutByteCodeFlag)

&& Js::Configuration::Global.flags.BailOutByteCode.Empty())

{

Output::Print(L"Bail out byte code location count: %d", this->bailOutByteCodeLocationCount);

}

#endif

Assert(processNativeCodeSize >= nativeCodeSize);

::InterlockedExchangeSubtract(&processNativeCodeSize, nativeCodeSize);

PERF\_COUNTER\_DEC(Basic, ThreadContext);

#ifdef DYNAMIC\_PROFILE\_MUTATOR

if (this->dynamicProfileMutator != nullptr)

{

this->dynamicProfileMutator->Delete();

}

#endif

#ifdef ENABLE\_PROJECTION

#if DBG\_DUMP

if (this->projectionMemoryInformation)

{

this->projectionMemoryInformation->Release();

this->projectionMemoryInformation = nullptr;

}

#endif

#endif

ReleasePreReservedSegment();

}

void

ThreadContext::SetJSRTRuntime(void\* runtime)

{

Assert(jsrtRuntime == nullptr); jsrtRuntime = runtime;

#ifdef ENABLE\_BASIC\_TELEMETRY

Telemetry::EnsureInitializeForJSRT();

#endif

}

void ThreadContext::CloseForJSRT()

{

// This is used for JSRT APIs only.

Assert(this->jsrtRuntime);

#ifdef ENABLE\_BASIC\_TELEMETRY

// log any relevant telemetry before disposing the current thread for cases which are properly shutdown

Telemetry::OnJSRTThreadContextClose();

#endif

ShutdownThreads();

}

ThreadContext\* ThreadContext::GetContextForCurrentThread()

{

ThreadContextTLSEntry \* tlsEntry = ThreadContextTLSEntry::GetEntryForCurrentThread();

if (tlsEntry != nullptr)

{

return static\_cast<ThreadContext \*>(tlsEntry->GetThreadContext());

}

return nullptr;

}

void ThreadContext::ValidateThreadContext()

{

#if DBG

// verify the runtime pointer is valid.

{

BOOL found = FALSE;

AutoCriticalSection autocs(ThreadContext::GetCriticalSection());

ThreadContext\* currentThreadContext = GetThreadContextList();

while (currentThreadContext)

{

if (currentThreadContext == this)

{

return;

}

currentThreadContext = currentThreadContext->Next();

}

AssertMsg(found, "invalid thread context");

}

#endif

}

#if ENABLE\_NATIVE\_CODEGEN

void ThreadContext::AddSimdFuncToMaps(Js::OpCode op, ...)

{

Assert(simdFuncInfoToOpcodeMap != nullptr);

Assert(simdOpcodeToSignatureMap != nullptr);

va\_list arguments;

va\_start(arguments, op);

int argumentsCount = va\_arg(arguments, int);

if (argumentsCount == 0)

{

// no info to add

return;

}

Js::FunctionInfo \*funcInfo = va\_arg(arguments, Js::FunctionInfo\*);

simdFuncInfoToOpcodeMap->AddNew(funcInfo, op);

SimdFuncSignature simdFuncSignature;

simdFuncSignature.valid = true;

simdFuncSignature.argCount = argumentsCount - 2; // arg count to Simd func = argumentsCount - FuncInfo and return Type fields.

simdFuncSignature.returnType = va\_arg(arguments, ValueType);

simdFuncSignature.args = AnewArrayZ(this->GetThreadAlloc(), ValueType, simdFuncSignature.argCount);

for (uint iArg = 0; iArg < simdFuncSignature.argCount; iArg++)

{

simdFuncSignature.args[iArg] = va\_arg(arguments, ValueType);

}

simdOpcodeToSignatureMap[Js::SimdOpcodeAsIndex(op)] = simdFuncSignature;

va\_end(arguments);

}

Js::OpCode ThreadContext::GetSimdOpcodeFromFuncInfo(Js::FunctionInfo \* funcInfo)

{

Assert(simdFuncInfoToOpcodeMap != nullptr);

if (simdFuncInfoToOpcodeMap->ContainsKey(funcInfo))

{

return simdFuncInfoToOpcodeMap->Item(funcInfo);

}

return (Js::OpCode) 0;

}

void ThreadContext::GetSimdFuncSignatureFromOpcode(Js::OpCode op, SimdFuncSignature &funcSignature)

{

Assert(simdOpcodeToSignatureMap != nullptr);

funcSignature = simdOpcodeToSignatureMap[SimdOpcodeAsIndex(op)];

}

#endif

class AutoRecyclerPtr : public AutoPtr<Recycler>

{

public:

AutoRecyclerPtr(Recycler \* ptr) : AutoPtr<Recycler>(ptr) {}

~AutoRecyclerPtr()

{

#ifdef CONCURRENT\_GC\_ENABLED

if (ptr != nullptr)

{

ptr->ShutdownThread();

}

#endif

}

};

Recycler\* ThreadContext::EnsureRecycler()

{

if (recycler == NULL)

{

AutoRecyclerPtr newRecycler(HeapNew(Recycler, GetAllocationPolicyManager(), &pageAllocator, Js::Throw::OutOfMemory, Js::Configuration::Global.flags));

newRecycler->Initialize(isOptimizedForManyInstances, &threadService); // use in-thread GC when optimizing for many instances

newRecycler->SetCollectionWrapper(this);

#if ENABLE\_NATIVE\_CODEGEN

// This may throw, so it needs to be after the recycler is initialized,

// otherwise, the recycler dtor may encounter problems

AutoPtr<CodeGenNumberThreadAllocator> localCodeGenNumberThreadAllocator(

HeapNew(CodeGenNumberThreadAllocator, newRecycler));

#endif

this->recyclableData.Root(RecyclerNewZ(newRecycler, RecyclableData, newRecycler), newRecycler);

if (this->GetIsThreadBound())

{

newRecycler->SetIsThreadBound();

}

// Assign the recycler to the ThreadContext after everything is initialized, because an OOM during initialization would

// result in only partial initialization, so the 'recycler' member variable should remain null to cause full

// reinitialization when requested later. Anything that happens after the Detach must have special cleanup code.

this->recycler = newRecycler.Detach();

try

{

#ifdef RECYCLER\_WRITE\_BARRIER

#ifdef \_M\_X64\_OR\_ARM64

if (!RecyclerWriteBarrierManager::OnThreadInit())

{

Js::Throw::OutOfMemory();

}

#endif

#endif

this->expirableObjectList = Anew(&this->threadAlloc, ExpirableObjectList, &this->threadAlloc);

this->expirableObjectDisposeList = Anew(&this->threadAlloc, ExpirableObjectList, &this->threadAlloc);

InitializePropertyMaps(); // has many dependencies on the recycler and other members of the thread context

#if ENABLE\_NATIVE\_CODEGEN

this->codeGenNumberThreadAllocator = localCodeGenNumberThreadAllocator.Detach();

#endif

}

catch(...)

{

// Initialization failed, undo what was done above. Callees that throw must clean up after themselves.

if (this->recyclableData != nullptr)

{

this->recyclableData.Unroot(this->recycler);

}

{

// AutoRecyclerPtr's destructor takes care of shutting down the background thread and deleting the recycler

AutoRecyclerPtr recyclerToDelete(this->recycler);

this->recycler = nullptr;

}

throw;

}

JS\_ETW(EventWriteJSCRIPT\_GC\_INIT(this->recycler, this->GetHiResTimer()->Now()));

}

#if DBG

if (CONFIG\_FLAG(RecyclerTest))

{

StressTester test(recycler);

test.Run();

}

#endif

return recycler;

}

Js::PropertyRecord const \*

ThreadContext::GetPropertyName(Js::PropertyId propertyId)

{

// This API should only be use on the main thread

Assert(GetCurrentThreadContextId() == (ThreadContextId)this);

return this->GetPropertyNameImpl<false>(propertyId);

}

Js::PropertyRecord const \*

ThreadContext::GetPropertyNameLocked(Js::PropertyId propertyId)

{

return GetPropertyNameImpl<true>(propertyId);

}

template <bool locked>

Js::PropertyRecord const \*

ThreadContext::GetPropertyNameImpl(Js::PropertyId propertyId)

{

//TODO: Remove this when completely transformed to use PropertyRecord\*. Currently this is only partially done,

// and there are calls to GetPropertyName with InternalPropertyId.

if (propertyId >= 0 && Js::IsInternalPropertyId(propertyId))

{

return Js::InternalPropertyRecords::GetInternalPropertyName(propertyId);

}

int propertyIndex = propertyId - Js::PropertyIds::\_none;

if (propertyIndex < 0 || propertyIndex > propertyMap->GetLastIndex())

{

propertyIndex = 0;

}

const Js::PropertyRecord \* propertyRecord = nullptr;

if (locked) { propertyMap->LockResize(); }

bool found = propertyMap->TryGetValueAt(propertyIndex, &propertyRecord);

if (locked) { propertyMap->UnlockResize(); }

AssertMsg(found && propertyRecord != nullptr, "using invalid propertyid");

return propertyRecord;

}

void

ThreadContext::FindPropertyRecord(Js::JavascriptString \*pstName, Js::PropertyRecord const \*\* propertyRecord)

{

LPCWSTR psz = pstName->GetSz();

FindPropertyRecord(psz, pstName->GetLength(), propertyRecord);

}

void

ThreadContext::FindPropertyRecord(\_\_in LPCWSTR propertyName, \_\_in int propertyNameLength, Js::PropertyRecord const \*\* propertyRecord)

{

EnterPinnedScope((volatile void \*\*)propertyRecord);

\*propertyRecord = FindPropertyRecord(propertyName, propertyNameLength);

LeavePinnedScope();

}

const Js::PropertyRecord \*

ThreadContext::FindPropertyRecord(const wchar\_t \* propertyName, int propertyNameLength)

{

Js::PropertyRecord const \* propertyRecord = nullptr;

if (IsDirectPropertyName(propertyName, propertyNameLength))

{

propertyRecord = propertyNamesDirect[propertyName[0]];

Assert(propertyRecord == propertyMap->LookupWithKey(Js::HashedCharacterBuffer<wchar\_t>(propertyName, propertyNameLength)));

}

else

{

propertyRecord = propertyMap->LookupWithKey(Js::HashedCharacterBuffer<wchar\_t>(propertyName, propertyNameLength));

}

return propertyRecord;

}

Js::PropertyRecord const \*

ThreadContext::UncheckedAddPropertyId(\_\_in LPCWSTR propertyName, \_\_in int propertyNameLength, bool bind, bool isSymbol)

{

return UncheckedAddPropertyId(JsUtil::CharacterBuffer<WCHAR>(propertyName, propertyNameLength), bind, isSymbol);

}

void ThreadContext::InitializePropertyMaps()

{

Assert(this->recycler != nullptr);

Assert(this->recyclableData != nullptr);

Assert(this->propertyMap == nullptr);

Assert(this->caseInvariantPropertySet == nullptr);

try

{

this->propertyMap = HeapNew(PropertyMap, &HeapAllocator::Instance, TotalNumberOfBuiltInProperties + 700);

this->recyclableData->boundPropertyStrings = RecyclerNew(this->recycler, JsUtil::List<Js::PropertyRecord const\*>, this->recycler);

memset(propertyNamesDirect, 0, 128\*sizeof(Js::PropertyRecord \*));

Js::JavascriptLibrary::InitializeProperties(this);

InitializeAdditionalProperties(this);

//Js::JavascriptLibrary::InitializeDOMProperties(this);

}

catch(...)

{

// Initialization failed, undo what was done above. Callees that throw must clean up after themselves. The recycler will

// be trashed, so clear members that point to recyclable memory. Stuff in 'recyclableData' will be taken care of by the

// recycler, and the 'recyclableData' instance will be trashed as well.

if (this->propertyMap != nullptr)

{

HeapDelete(this->propertyMap);

}

this->propertyMap = nullptr;

this->caseInvariantPropertySet = nullptr;

memset(propertyNamesDirect, 0, 128\*sizeof(Js::PropertyRecord \*));

throw;

}

}

void ThreadContext::UncheckedAddBuiltInPropertyId()

{

for (int i = 0; i < \_countof(builtInPropertyRecords); i++)

{

AddPropertyRecordInternal(builtInPropertyRecords[i]);

}

}

bool

ThreadContext::IsDirectPropertyName(const wchar\_t \* propertyName, int propertyNameLength)

{

return ((propertyNameLength == 1) && ((propertyName[0] & 0xFF80) == 0));

}

RecyclerWeakReference<const Js::PropertyRecord> \*

ThreadContext::CreatePropertyRecordWeakRef(const Js::PropertyRecord \* propertyRecord)

{

RecyclerWeakReference<const Js::PropertyRecord> \* propertyRecordWeakRef;

if (propertyRecord->IsBound())

{

// Create a fake weak ref

propertyRecordWeakRef = RecyclerNewLeaf(this->recycler, StaticPropertyRecordReference, propertyRecord);

}

else

{

propertyRecordWeakRef = recycler->CreateWeakReferenceHandle(propertyRecord);

}

return propertyRecordWeakRef;

}

Js::PropertyRecord const \*

ThreadContext::UncheckedAddPropertyId(JsUtil::CharacterBuffer<WCHAR> const& propertyName, bool bind, bool isSymbol)

{

this->propertyMap->EnsureCapacity();

// Automatically bind direct (single-character) property names, so that they can be

// stored in the direct property table

if (IsDirectPropertyName(propertyName.GetBuffer(), propertyName.GetLength()))

{

bind = true;

}

// Create the PropertyRecord

int length = propertyName.GetLength();

uint bytelength = sizeof(wchar\_t) \* length;

uint32 indexVal = 0;

// Symbol properties cannot be numeric since their description is not to be used!

bool isNumeric = !isSymbol && Js::PropertyRecord::IsPropertyNameNumeric(propertyName.GetBuffer(), propertyName.GetLength(), &indexVal);

uint hash = JsUtil::CharacterBuffer<WCHAR>::StaticGetHashCode(propertyName.GetBuffer(), propertyName.GetLength());

size\_t allocLength = bytelength + sizeof(wchar\_t) + (isNumeric ? sizeof(uint32) : 0);

// If it's bound, create it in the thread arena, along with a fake weak ref

Js::PropertyRecord \* propertyRecord;

if (bind)

{

propertyRecord = AnewPlus(GetThreadAlloc(), allocLength, Js::PropertyRecord, bytelength, isNumeric, hash, isSymbol);

propertyRecord->isBound = true;

}

else

{

propertyRecord = RecyclerNewFinalizedLeafPlus(recycler, allocLength, Js::PropertyRecord, bytelength, isNumeric, hash, isSymbol);

}

// Copy string and numeric info

wchar\_t\* buffer = (wchar\_t \*)(propertyRecord + 1);

js\_memcpy\_s(buffer, bytelength, propertyName.GetBuffer(), bytelength);

buffer[length] = L'\0';

if (isNumeric)

{

\*(uint32 \*)(buffer + length + 1) = indexVal;

Assert(propertyRecord->GetNumericValue() == indexVal);

}

Js::PropertyId propertyId = this->GetNextPropertyId();

propertyRecord->pid = propertyId;

AddPropertyRecordInternal(propertyRecord);

return propertyRecord;

}

void

ThreadContext::AddPropertyRecordInternal(const Js::PropertyRecord \* propertyRecord)

{

// At this point the PropertyRecord is constructed but not added to the map.

const wchar\_t \* propertyName = propertyRecord->GetBuffer();

int propertyNameLength = propertyRecord->GetLength();

Js::PropertyId propertyId = propertyRecord->GetPropertyId();

Assert(propertyId == GetNextPropertyId());

Assert(!IsActivePropertyId(propertyId));

#if DBG

// Only Assert we can't find the property if we are not adding a symbol.

// For a symbol, the propertyName is not used and may collide with something in the map already.

if (!propertyRecord->IsSymbol())

{

Assert(FindPropertyRecord(propertyName, propertyNameLength) == nullptr);

}

#endif

// Add to the map

propertyMap->Add(propertyRecord);

PropertyRecordTrace(L"Added property '%s' at 0x%08x, pid = %d\n", propertyName, propertyRecord, propertyId);

// Do not store the pid for symbols in the direct property name table.

// We don't want property ids for symbols to be searchable anyway.

if (!propertyRecord->IsSymbol() && IsDirectPropertyName(propertyName, propertyNameLength))

{

// Store the pids for single character properties in the propertyNamesDirect array.

// This property record should have been created as bound by the caller.

Assert(propertyRecord->IsBound());

Assert(propertyNamesDirect[propertyName[0]] == nullptr);

propertyNamesDirect[propertyName[0]] = propertyRecord;

}

if (caseInvariantPropertySet)

{

AddCaseInvariantPropertyRecord(propertyRecord);

}

// Check that everything was added correctly

#if DBG

// Only Assert we can find the property if we are not adding a symbol.

// For a symbol, the propertyName is not used and we won't be able to look the pid up via name.

if (!propertyRecord->IsSymbol())

{

Assert(FindPropertyRecord(propertyName, propertyNameLength) == propertyRecord);

}

// We will still be able to lookup the symbol property by the property id, so go ahead and check that.

Assert(GetPropertyName(propertyRecord->GetPropertyId()) == propertyRecord);

#endif

JS\_ETW(EventWriteJSCRIPT\_HOSTING\_PROPERTYID\_LIST(propertyRecord, propertyRecord->GetBuffer()));

}

void

ThreadContext::AddCaseInvariantPropertyRecord(const Js::PropertyRecord \* propertyRecord)

{

Assert(this->caseInvariantPropertySet != nullptr);

// Create a weak reference to the property record here (since we no longer use weak refs in the property map)

RecyclerWeakReference<const Js::PropertyRecord> \* propertyRecordWeakRef = CreatePropertyRecordWeakRef(propertyRecord);

JsUtil::CharacterBuffer<WCHAR> newPropertyName(propertyRecord->GetBuffer(), propertyRecord->GetLength());

Js::CaseInvariantPropertyListWithHashCode\* list;

if (!FindExistingPropertyRecord(newPropertyName, &list))

{

// This binds all the property string that is key in this map with no hope of reclaiming them

// TODO: do better

list = RecyclerNew(recycler, Js::CaseInvariantPropertyListWithHashCode, recycler, 1);

// Do the add first so that the list is non-empty and we can calculate its hashcode correctly

list->Add(propertyRecordWeakRef);

// This will calculate the hashcode

caseInvariantPropertySet->Add(list);

}

else

{

list->Add(propertyRecordWeakRef);

}

}

void

ThreadContext::BindPropertyRecord(const Js::PropertyRecord \* propertyRecord)

{

if (!propertyRecord->IsBound())

{

Assert(!this->recyclableData->boundPropertyStrings->Contains(propertyRecord));

this->recyclableData->boundPropertyStrings->Add(propertyRecord);

// Cast around constness to set propertyRecord as bound

const\_cast<Js::PropertyRecord \*>(propertyRecord)->isBound = true;

}

}

void ThreadContext::GetOrAddPropertyId(\_\_in LPCWSTR propertyName, \_\_in int propertyNameLength, Js::PropertyRecord const \*\* propertyRecord)

{

GetOrAddPropertyId(JsUtil::CharacterBuffer<WCHAR>(propertyName, propertyNameLength), propertyRecord);

}

void ThreadContext::GetOrAddPropertyId(JsUtil::CharacterBuffer<WCHAR> const& propertyName, Js::PropertyRecord const \*\* propRecord)

{

EnterPinnedScope((volatile void \*\*)propRecord);

\*propRecord = GetOrAddPropertyRecord(propertyName);

LeavePinnedScope();

}

const Js::PropertyRecord \*

ThreadContext::GetOrAddPropertyRecordImpl(JsUtil::CharacterBuffer<wchar\_t> propertyName, bool bind)

{

// Make sure the recyclers around so that we can take weak references to the property strings

EnsureRecycler();

const Js::PropertyRecord \* propertyRecord;

FindPropertyRecord(propertyName.GetBuffer(), propertyName.GetLength(), &propertyRecord);

if (propertyRecord == nullptr)

{

propertyRecord = UncheckedAddPropertyId(propertyName, bind);

}

else

{

// PropertyRecord exists, but may not be bound. Bind now if requested.

if (bind)

{

BindPropertyRecord(propertyRecord);

}

}

Assert(propertyRecord != nullptr);

Assert(!bind || propertyRecord->IsBound());

return propertyRecord;

}

void ThreadContext::AddBuiltInPropertyRecord(const Js::PropertyRecord \*propertyRecord)

{

this->AddPropertyRecordInternal(propertyRecord);

}

BOOL ThreadContext::IsNumericPropertyId(Js::PropertyId propertyId, uint32\* value)

{

Js::PropertyRecord const \* propertyRecord = this->GetPropertyName(propertyId);

Assert(propertyRecord != nullptr);

if (propertyRecord == nullptr || !propertyRecord->IsNumeric())

{

return false;

}

\*value = propertyRecord->GetNumericValue();

return true;

}

bool ThreadContext::IsActivePropertyId(Js::PropertyId pid)

{

Assert(pid != Js::Constants::NoProperty);

if (Js::IsInternalPropertyId(pid))

{

return true;

}

int propertyIndex = pid - Js::PropertyIds::\_none;

const Js::PropertyRecord \* propertyRecord;

if (propertyMap->TryGetValueAt(propertyIndex, &propertyRecord) && propertyRecord != nullptr)

{

return true;

}

return false;

}

void ThreadContext::InvalidatePropertyRecord(const Js::PropertyRecord \* propertyRecord)

{

InternalInvalidateProtoTypePropertyCaches(propertyRecord->GetPropertyId()); // use the internal version so we don't check for active property id

this->propertyMap->Remove(propertyRecord);

PropertyRecordTrace(L"Reclaimed property '%s' at 0x%08x, pid = %d\n",

propertyRecord->GetBuffer(), propertyRecord, propertyRecord->GetPropertyId());

}

Js::PropertyId ThreadContext::GetNextPropertyId()

{

return this->propertyMap->GetNextIndex() + Js::PropertyIds::\_none;

}

Js::PropertyId ThreadContext::GetMaxPropertyId()

{

auto maxPropertyId = this->propertyMap->Count() + Js::InternalPropertyIds::Count;

return maxPropertyId;

}

void ThreadContext::CreateNoCasePropertyMap()

{

Assert(caseInvariantPropertySet == nullptr);

caseInvariantPropertySet = RecyclerNew(recycler, PropertyNoCaseSetType, recycler, 173);

// Prevent the set from being reclaimed

// Individual items in the set can be reclaimed though since they're lists of weak references

// The lists themselves can be reclaimed when all the weak references in them are cleared

this->recyclableData->caseInvariantPropertySet = caseInvariantPropertySet;

// Note that we are allocating from the recycler below, so we may cause a GC at any time, which

// could cause PropertyRecords to be collected and removed from the propertyMap.

// Thus, don't use BaseDictionary::Map here, as it cannot tolerate changes while mapping.

// Instead, walk the PropertyRecord entries in index order. This will work even if a GC occurs.

for (int propertyIndex = 0; propertyIndex <= this->propertyMap->GetLastIndex(); propertyIndex++)

{

const Js::PropertyRecord \* propertyRecord;

if (this->propertyMap->TryGetValueAt(propertyIndex, &propertyRecord) && propertyRecord != nullptr)

{

AddCaseInvariantPropertyRecord(propertyRecord);

}

}

}

JsUtil::List<const RecyclerWeakReference<Js::PropertyRecord const>\*>\*

ThreadContext::FindPropertyIdNoCase(Js::ScriptContext \* scriptContext, LPCWSTR propertyName, int propertyNameLength)

{

return ThreadContext::FindPropertyIdNoCase(scriptContext, JsUtil::CharacterBuffer<WCHAR>(propertyName, propertyNameLength));

}

JsUtil::List<const RecyclerWeakReference<Js::PropertyRecord const>\*>\*

ThreadContext::FindPropertyIdNoCase(Js::ScriptContext \* scriptContext, JsUtil::CharacterBuffer<WCHAR> const& propertyName)

{

if (caseInvariantPropertySet == nullptr)

{

this->CreateNoCasePropertyMap();

}

Js::CaseInvariantPropertyListWithHashCode\* list;

if (FindExistingPropertyRecord(propertyName, &list))

{

return list;

}

return nullptr;

}

bool

ThreadContext::FindExistingPropertyRecord(\_In\_ JsUtil::CharacterBuffer<WCHAR> const& propertyName, Js::CaseInvariantPropertyListWithHashCode\*\* list)

{

Js::CaseInvariantPropertyListWithHashCode\* l = this->caseInvariantPropertySet->LookupWithKey(propertyName);

(\*list) = l;

return (l != nullptr);

}

void ThreadContext::CleanNoCasePropertyMap()

{

if (this->caseInvariantPropertySet != nullptr)

{

this->caseInvariantPropertySet->MapAndRemoveIf([](Js::CaseInvariantPropertyListWithHashCode\* value) -> bool {

if (value && value->Count() == 0)

{

// Remove entry

return true;

}

// Keep entry

return false;

});

}

}

void

ThreadContext::ForceCleanPropertyMap()

{

// No-op now that we no longer use weak refs

}

#if ENABLE\_NATIVE\_CODEGEN

JsUtil::JobProcessor \*

ThreadContext::GetJobProcessor()

{

if(bgJit && isOptimizedForManyInstances)

{

return ThreadBoundThreadContextManager::GetSharedJobProcessor();

}

if (!jobProcessor)

{

if(bgJit && !isOptimizedForManyInstances)

{

jobProcessor = HeapNew(JsUtil::BackgroundJobProcessor, GetAllocationPolicyManager(), &threadService, false /\*disableParallelThreads\*/);

}

else

{

jobProcessor = HeapNew(JsUtil::ForegroundJobProcessor);

}

}

return jobProcessor;

}

#endif

void

ThreadContext::RegisterCodeGenRecyclableData(Js::CodeGenRecyclableData \*const codeGenRecyclableData)

{

Assert(codeGenRecyclableData);

Assert(recyclableData);

// Linking must not be done concurrently with unlinking (caller must use lock)

recyclableData->codeGenRecyclableDatas.LinkToEnd(codeGenRecyclableData);

}

void

ThreadContext::UnregisterCodeGenRecyclableData(Js::CodeGenRecyclableData \*const codeGenRecyclableData)

{

Assert(codeGenRecyclableData);

if(!recyclableData)

{

// The thread context's recyclable data may have already been released to the recycler if we're shutting down

return;

}

// Unlinking may be done from a background thread, but not concurrently with linking (caller must use lock). Partial unlink

// does not zero the previous and next links for the unlinked node so that the recycler can scan through the node from the

// main thread.

recyclableData->codeGenRecyclableDatas.UnlinkPartial(codeGenRecyclableData);

}

uint

ThreadContext::EnterScriptStart(Js::ScriptEntryExitRecord \* record, bool doCleanup)

{

Recycler \* recycler = this->GetRecycler();

Assert(recycler->IsReentrantState());

JS\_ETW(EventWriteJSCRIPT\_RUN\_START(this,0));

// Increment the callRootLevel early so that Dispose ran during FinishConcurrent will not close the current scriptContext

uint oldCallRootLevel = this->callRootLevel++;

if (oldCallRootLevel == 0)

{

Assert(!this->hasThrownPendingException);

RECORD\_TIMESTAMP(lastScriptStartTime);

InterruptPoller \*poller = this->interruptPoller;

if (poller)

{

poller->StartScript();

}

recycler->SetIsInScript(true);

if (doCleanup)

{

recycler->EnterIdleDecommit();

#ifdef CONCURRENT\_GC\_ENABLED

recycler->FinishConcurrent<FinishConcurrentOnEnterScript>();

#endif

if (threadServiceWrapper == NULL)

{

// Reschedule the next collection at the start of the script.

recycler->ScheduleNextCollection();

}

}

}

this->PushEntryExitRecord(record);

AssertMsg(!this->IsScriptActive(),

"Missing EnterScriptEnd or LeaveScriptStart");

this->isScriptActive = true;

recycler->SetIsScriptActive(true);

#if DBG\_DUMP

if (Js::Configuration::Global.flags.Trace.IsEnabled(Js::RunPhase))

{

Output::Trace(Js::RunPhase, L"%p> EnterScriptStart(%p): Level %d\n", ::GetCurrentThreadId(), this, this->callRootLevel);

Output::Flush();

}

#endif

return oldCallRootLevel;

}

void

ThreadContext::EnterScriptEnd(Js::ScriptEntryExitRecord \* record, bool doCleanup)

{

#if DBG\_DUMP

if (Js::Configuration::Global.flags.Trace.IsEnabled(Js::RunPhase))

{

Output::Trace(Js::RunPhase, L"%p> EnterScriptEnd (%p): Level %d\n", ::GetCurrentThreadId(), this, this->callRootLevel);

Output::Flush();

}

#endif

this->PopEntryExitRecord(record);

AssertMsg(this->IsScriptActive(),

"Missing EnterScriptStart or LeaveScriptEnd");

this->isScriptActive = false;

this->GetRecycler()->SetIsScriptActive(false);

this->callRootLevel--;

#ifdef EXCEPTION\_CHECK

ExceptionCheck::SetHandledExceptionType(record->handledExceptionType);

#endif

#ifdef RECYCLER\_MEMORY\_VERIFY

recycler->Verify(Js::RunPhase);

#endif

if (this->callRootLevel == 0)

{

RECORD\_TIMESTAMP(lastScriptEndTime);

this->GetRecycler()->SetIsInScript(false);

InterruptPoller \*poller = this->interruptPoller;

if (poller)

{

poller->EndScript();

}

ClosePendingProjectionContexts();

ClosePendingScriptContexts();

Assert(rootPendingClose == nullptr);

if (this->hasThrownPendingException)

{

// We have some cases where the thread instant of JavascriptExceptionObject

// are ignored and not clear. To avoid leaks, clear it here when

// we are not in script, where no one should be using these JavascriptExceptionObject

this->ClearPendingOOMError();

this->ClearPendingSOError();

this->hasThrownPendingException = false;

}

#ifdef ENABLE\_DEBUG\_CONFIG\_OPTIONS

if (Js::Configuration::Global.flags.FreeRejittedCode)

#endif

{

// Since we're no longer in script, old entry points can now be collected

Js::FunctionEntryPointInfo\* current = this->recyclableData->oldEntryPointInfo;

this->recyclableData->oldEntryPointInfo = nullptr;

// Clear out the next pointers so older entry points wont be held on

// as a false positive

while (current != nullptr)

{

Js::FunctionEntryPointInfo\* next = current->nextEntryPoint;

current->nextEntryPoint = nullptr;

current = next;

}

}

if (doCleanup)

{

ThreadServiceWrapper\* threadServiceWrapper = GetThreadServiceWrapper();

if (!threadServiceWrapper || !threadServiceWrapper->ScheduleNextCollectOnExit())

{

// Do the idle GC now if we fail schedule one.

recycler->CollectNow<CollectOnScriptExit>();

}

recycler->LeaveIdleDecommit();

}

}

if (doCleanup)

{

PHASE\_PRINT\_TRACE1(Js::DisposePhase, L"[Dispose] NeedDispose in EnterScriptEnd: %d\n", this->recycler->NeedDispose());

if (this->recycler->NeedDispose())

{

this->recycler->FinishDisposeObjectsNow<FinishDispose>();

}

}

JS\_ETW(EventWriteJSCRIPT\_RUN\_STOP(this,0));

}

void

ThreadContext::SetForceOneIdleCollection()

{

ThreadServiceWrapper\* threadServiceWrapper = GetThreadServiceWrapper();

if (threadServiceWrapper)

{

threadServiceWrapper->SetForceOneIdleCollection();

}

}

BOOLEAN

ThreadContext::IsOnStack(void const \*ptr)

{

#if defined(\_M\_IX86)

return ptr < (void\*)\_\_readfsdword(0x4) && ptr >= (void\*)\_\_readfsdword(0xE0C);

#elif defined(\_M\_AMD64)

return ptr < (void\*)\_\_readgsqword(0x8) && ptr >= (void\*)\_\_readgsqword(0x1478);

#elif defined(\_M\_ARM)

ULONG lowLimit, highLimit;

::GetCurrentThreadStackLimits(&lowLimit, &highLimit);

bool isOnStack = (void\*)lowLimit <= ptr && ptr < (void\*)highLimit;

return isOnStack;

#elif defined(\_M\_ARM64)

ULONG64 lowLimit, highLimit;

::GetCurrentThreadStackLimits(&lowLimit, &highLimit);

bool isOnStack = (void\*)lowLimit <= ptr && ptr < (void\*)highLimit;

return isOnStack;

#else

AssertMsg(FALSE, "IsOnStack -- not implemented yet case");

Js::Throw::NotImplemented();

return false;

#endif

}

PBYTE

ThreadContext::GetStackLimitForCurrentThread() const

{

FAULTINJECT\_SCRIPT\_TERMINATION;

PBYTE limit = this->stackLimitForCurrentThread;

Assert(limit == Js::Constants::StackLimitForScriptInterrupt

|| !this->GetStackProber()

|| limit == this->GetStackProber()->GetScriptStackLimit());

return limit;

}

void

ThreadContext::SetStackLimitForCurrentThread(PBYTE limit)

{

this->stackLimitForCurrentThread = limit;

}

bool

ThreadContext::CanPreReserveSegmentForCustomHeap()

{

#if \_M\_IX86 && \_CONTROL\_FLOW\_GUARD

return numOfThreadContextsWithPreReserveSegment <= Js::Constants::MaxThreadContextsWithPreReserveSegment;

#elif \_M\_X64\_OR\_ARM64 && \_CONTROL\_FLOW\_GUARD

return true;

#else

return false;

#endif

}

\_\_declspec(noinline) //Win8 947081: might use wrong \_AddressOfReturnAddress() if this and caller are inlined

bool

ThreadContext::IsStackAvailable(size\_t size)

{

PBYTE sp = (PBYTE)\_AddressOfReturnAddress();

PBYTE stackLimit = this->GetStackLimitForCurrentThread();

bool stackAvailable = ((size\_t)sp > size && (sp - size) > stackLimit);

// Verify that JIT'd frames didn't mess up the ABI stack alignment

Assert(((uintptr\_t)sp & (AutoSystemInfo::StackAlign - 1)) == (sizeof(void\*) & (AutoSystemInfo::StackAlign - 1)));

#if DBG

this->GetStackProber()->AdjustKnownStackLimit(sp, size);

#endif

FAULTINJECT\_STACK\_PROBE

if (stackAvailable)

{

return true;

}

if (sp <= stackLimit)

{

if (stackLimit == Js::Constants::StackLimitForScriptInterrupt)

{

if (sp <= this->GetStackProber()->GetScriptStackLimit())

{

// Take down the process if we cant recover from the stack overflow

Js::Throw::FatalInternalError();

}

}

}

return false;

}

\_\_declspec(noinline) //Win8 947081: might use wrong \_AddressOfReturnAddress() if this and caller are inlined

bool

ThreadContext::IsStackAvailableNoThrow(size\_t size)

{

PBYTE sp = (PBYTE)\_AddressOfReturnAddress();

PBYTE stackLimit = this->GetStackLimitForCurrentThread();

bool stackAvailable = (sp > stackLimit) && ((size\_t)sp > size) && ((sp - size) > stackLimit);

FAULTINJECT\_STACK\_PROBE

return stackAvailable;

}

/\* static \*/ bool

ThreadContext::IsCurrentStackAvailable(size\_t size)

{

ThreadContext \*currentContext = GetContextForCurrentThread();

Assert(currentContext);

return currentContext->IsStackAvailable(size);

}

/\*

returnAddress will be passed in the stackprobe call at the beginning of interpreter frame.

We need to probe the stack before we link up the InterpreterFrame structure in threadcontext,

and if we throw there, the stack walker might get confused when trying to identify a frame

is interpreter frame by comparing the current ebp in ebp chain with return address specified

in the last InterpreterFrame linked in threadcontext. We need to pass in the return address

of the probing frame to skip the right one (we need to skip first match in a->a->a recursion,

but not in a->b->a recursion).

\*/

void

ThreadContext::ProbeStackNoDispose(size\_t size, Js::ScriptContext \*scriptContext, PVOID returnAddress)

{

AssertCanHandleStackOverflow();

if (!this->IsStackAvailable(size))

{

if (this->IsExecutionDisabled())

{

// The probe failed because we hammered the stack limit to trigger script interrupt.

Assert(this->DoInterruptProbe());

throw Js::ScriptAbortException();

}

Js::Throw::StackOverflow(scriptContext, returnAddress);

}

// Use every Nth stack probe as a QC trigger.

if (AutoSystemInfo::ShouldQCMoreFrequently() && this->HasInterruptPoller() && this->IsScriptActive())

{

++(this->stackProbeCount);

if (this->stackProbeCount > ThreadContext::StackProbePollThreshold)

{

this->stackProbeCount = 0;

this->CheckInterruptPoll();

}

}

}

void

ThreadContext::ProbeStack(size\_t size, Js::ScriptContext \*scriptContext, PVOID returnAddress)

{

this->ProbeStackNoDispose(size, scriptContext, returnAddress);

// BACKGROUND-GC TODO: If we're stuck purely in JITted code, we should have the

// background GC thread modify the threads stack limit to trigger the runtime stack probe

if (this->callDispose && this->recycler->NeedDispose())

{

PHASE\_PRINT\_TRACE1(Js::DisposePhase, L"[Dispose] NeedDispose in ProbeStack: %d\n", this->recycler->NeedDispose());

this->recycler->FinishDisposeObjectsNow<FinishDisposeTimed>();

}

}

void

ThreadContext::ProbeStack(size\_t size, Js::RecyclableObject \* obj, Js::ScriptContext \*scriptContext)

{

AssertCanHandleStackOverflowCall(obj->IsExternal() ||

Js::JavascriptOperators::GetTypeId(obj) == Js::TypeIds\_Function &&

Js::JavascriptFunction::FromVar(obj)->IsExternalFunction());

if (!this->IsStackAvailable(size))

{

if (this->IsExecutionDisabled())

{

// The probe failed because we hammered the stack limit to trigger script interrupt.

Assert(this->DoInterruptProbe());

throw Js::ScriptAbortException();

}

if (obj->IsExternal() ||

Js::JavascriptOperators::GetTypeId(obj) == Js::TypeIds\_Function &&

Js::JavascriptFunction::FromVar(obj)->IsExternalFunction())

{

Js::JavascriptError::ThrowStackOverflowError(scriptContext);

}

Js::Throw::StackOverflow(scriptContext, NULL);

}

}

void

ThreadContext::ProbeStack(size\_t size)

{

Assert(this->IsScriptActive());

Js::ScriptEntryExitRecord \*entryExitRecord = this->GetScriptEntryExit();

Assert(entryExitRecord);

Js::ScriptContext \*scriptContext = entryExitRecord->scriptContext;

Assert(scriptContext);

this->ProbeStack(size, scriptContext);

}

/\* static \*/ void

ThreadContext::ProbeCurrentStack(size\_t size, Js::ScriptContext \*scriptContext)

{

Assert(scriptContext != nullptr);

Assert(scriptContext->GetThreadContext() == GetContextForCurrentThread());

scriptContext->GetThreadContext()->ProbeStack(size, scriptContext);

}

/\* static \*/ void

ThreadContext::ProbeCurrentStackNoDispose(size\_t size, Js::ScriptContext \*scriptContext)

{

Assert(scriptContext != nullptr);

Assert(scriptContext->GetThreadContext() == GetContextForCurrentThread());

scriptContext->GetThreadContext()->ProbeStackNoDispose(size, scriptContext);

}

template <bool leaveForHost>

void

ThreadContext::LeaveScriptStart(void \* frameAddress)

{

Assert(this->IsScriptActive());

#if DBG\_DUMP

if (Js::Configuration::Global.flags.Trace.IsEnabled(Js::RunPhase))

{

Output::Trace(Js::RunPhase, L"%p> LeaveScriptStart(%p): Level %d\n", ::GetCurrentThreadId(), this, this->callRootLevel);

Output::Flush();

}

#endif

JS\_ETW(EventWriteJSCRIPT\_CALL\_OUT\_START(this,0));

Js::ScriptEntryExitRecord \* entryExitRecord = this->GetScriptEntryExit();

AssertMsg(entryExitRecord && entryExitRecord->frameIdOfScriptExitFunction == nullptr,

"Missing LeaveScriptEnd or EnterScriptStart");

entryExitRecord->frameIdOfScriptExitFunction = frameAddress;

this->isScriptActive = false;

this->GetRecycler()->SetIsScriptActive(false);

AssertMsg(!(leaveForHost && this->IsDisableImplicitCall()),

"Disable implicit call should have been caught before leaving script for host");

// Save the implicit call flags

entryExitRecord->savedImplicitCallFlags = this->GetImplicitCallFlags();

// clear the hasReentered to detect if we have reentered into script

entryExitRecord->hasReentered = false;

#if DBG || defined(PROFILE\_EXEC)

entryExitRecord->leaveForHost = leaveForHost;

#endif

#if DBG

entryExitRecord->leaveForAsyncHostOperation = false;

#endif

#ifdef PROFILE\_EXEC

if (leaveForHost)

{

entryExitRecord->scriptContext->ProfileEnd(Js::RunPhase);

}

#endif

}

void ThreadContext::DisposeOnLeaveScript()

{

PHASE\_PRINT\_TRACE1(Js::DisposePhase, L"[Dispose] NeedDispose in LeaveScriptStart: %d\n", this->recycler->NeedDispose());

if (this->callDispose && this->recycler->NeedDispose())

{

this->recycler->FinishDisposeObjectsNow<FinishDispose>();

}

}

template <bool leaveForHost>

void

ThreadContext::LeaveScriptEnd(void \* frameAddress)

{

Assert(!this->IsScriptActive());

#if DBG\_DUMP

if (Js::Configuration::Global.flags.Trace.IsEnabled(Js::RunPhase))

{

Output::Trace(Js::RunPhase, L"%p> LeaveScriptEnd(%p): Level %d\n", ::GetCurrentThreadId(), this, this->callRootLevel);

Output::Flush();

}

#endif

JS\_ETW(EventWriteJSCRIPT\_CALL\_OUT\_STOP(this,0));

Js::ScriptEntryExitRecord \* entryExitRecord = this->GetScriptEntryExit();

AssertMsg(entryExitRecord && entryExitRecord->frameIdOfScriptExitFunction,

"LeaveScriptEnd without LeaveScriptStart");

AssertMsg(frameAddress == nullptr || frameAddress == entryExitRecord->frameIdOfScriptExitFunction,

"Mismatched script exit frames");

Assert(!!entryExitRecord->leaveForHost == leaveForHost);

entryExitRecord->frameIdOfScriptExitFunction = nullptr;

AssertMsg(!this->IsScriptActive(), "Missing LeaveScriptStart or LeaveScriptStart");

this->isScriptActive = true;

this->GetRecycler()->SetIsScriptActive(true);

Js::ImplicitCallFlags savedImplicitCallFlags = entryExitRecord->savedImplicitCallFlags;

if (leaveForHost)

{

savedImplicitCallFlags = (Js::ImplicitCallFlags)(savedImplicitCallFlags | Js::ImplicitCall\_External);

}

else if (entryExitRecord->hasReentered)

{

savedImplicitCallFlags = (Js::ImplicitCallFlags)(savedImplicitCallFlags | Js::ImplicitCall\_AsyncHostOperation);

}

// Restore the implicit call flags

this->SetImplicitCallFlags(savedImplicitCallFlags);

#ifdef PROFILE\_EXEC

if (leaveForHost)

{

entryExitRecord->scriptContext->ProfileBegin(Js::RunPhase);

}

#endif

}

// explicit instantiations

template void ThreadContext::LeaveScriptStart<true>(void \* frameAddress);

template void ThreadContext::LeaveScriptStart<false>(void \* frameAddress);

template void ThreadContext::LeaveScriptEnd<true>(void \* frameAddress);

template void ThreadContext::LeaveScriptEnd<false>(void \* frameAddress);

void

ThreadContext::PushInterpreterFrame(Js::InterpreterStackFrame \*interpreterFrame)

{

interpreterFrame->SetPreviousFrame(this->leafInterpreterFrame);

this->leafInterpreterFrame = interpreterFrame;

}

Js::InterpreterStackFrame \*

ThreadContext::PopInterpreterFrame()

{

Js::InterpreterStackFrame \*interpreterFrame = this->leafInterpreterFrame;

Assert(interpreterFrame);

this->leafInterpreterFrame = interpreterFrame->GetPreviousFrame();

return interpreterFrame;

}

BOOL

ThreadContext::ExecuteRecyclerCollectionFunctionCommon(Recycler \* recycler, CollectionFunction function, CollectionFlags flags)

{

return \_\_super::ExecuteRecyclerCollectionFunction(recycler, function, flags);

}

#if DBG

bool

ThreadContext::IsInAsyncHostOperation() const

{

if (!this->IsScriptActive())

{

Js::ScriptEntryExitRecord \* lastRecord = this->entryExitRecord;

if (lastRecord != NULL)

{

return !!lastRecord->leaveForAsyncHostOperation;

}

}

return false;

}

#endif

BOOL

ThreadContext::ExecuteRecyclerCollectionFunction(Recycler \* recycler, CollectionFunction function, CollectionFlags flags)

{

// If the thread context doesn't have an associated Recycler set, don't do anything

if (this->recycler == nullptr)

{

return FALSE;

}

// Take etw rundown lock on this thread context. We can't collect entryPoints if we are in etw rundown.

AutoCriticalSection autocs(this->GetEtwRundownCriticalSection());

// Disable calling dispose from leave script or the stack probe

// while we're executing the recycler wrapper

AutoRestoreValue<bool> callDispose(&this->callDispose, false);

BOOL ret = FALSE;

if (!this->IsScriptActive())

{

Assert(!this->IsDisableImplicitCall() || this->IsInAsyncHostOperation());

ret = this->ExecuteRecyclerCollectionFunctionCommon(recycler, function, flags);

// Make sure that we finish a collect that is activated outside of script, since

// we won't have exit script to schedule it

if (!this->IsInScript() && recycler->CollectionInProgress()

&& ((flags & CollectOverride\_DisableIdleFinish) == 0) && threadServiceWrapper)

{

threadServiceWrapper->ScheduleFinishConcurrent();

}

}

else

{

void \* frameAddr = nullptr;

GET\_CURRENT\_FRAME\_ID(frameAddr);

// We may need stack to call out from Dispose or QC

if (!this->IsDisableImplicitCall()) // otherwise Dispose/QC disabled

{

// If we don't have stack space to call out from Dispose or QC,

// don't throw, simply return false. This gives SnailAlloc a better

// chance of allocating in low stack-space situations (like allocating

// a StackOverflowException object)

if (!this->IsStackAvailableNoThrow(Js::Constants::MinStackCallout))

{

return false;

}

}

this->LeaveScriptStart<false>(frameAddr);

ret = this->ExecuteRecyclerCollectionFunctionCommon(recycler, function, flags);

this->LeaveScriptEnd<false>(frameAddr);

if (this->callRootLevel != 0)

{

this->CheckScriptInterrupt();

}

}

return ret;

}

void

ThreadContext::DisposeObjects(Recycler \* recycler)

{

if(this->IsDisableImplicitCall())

{

// Don't dispose objects when implicit calls are disabled, since disposing may cause implicit calls. Objects will remain

// in the dispose queue and will be disposed later when implicit calls are not disabled.

return;

}

if (!this->IsScriptActive())

{

\_\_super::DisposeObjects(recycler);

}

else

{

void \* frameAddr = nullptr;

GET\_CURRENT\_FRAME\_ID(frameAddr);

// We may need stack to call out from Dispose

this->ProbeStack(Js::Constants::MinStackCallout);

this->LeaveScriptStart<false>(frameAddr);

\_\_super::DisposeObjects(recycler);

this->LeaveScriptEnd<false>(frameAddr);

}

}

void

ThreadContext::PushEntryExitRecord(Js::ScriptEntryExitRecord \* record)

{

AssertMsg(record, "Didn't provide a script entry record to push");

Assert(record->next == nullptr);

Js::ScriptEntryExitRecord \* lastRecord = this->entryExitRecord;

if (lastRecord != nullptr)

{

// If we enter script again, we should have leave with leaveForHost or leave for dispose.

Assert(lastRecord->leaveForHost || lastRecord->leaveForAsyncHostOperation);

lastRecord->hasReentered = true;

record->next = lastRecord;

}

this->entryExitRecord = record;

}

void ThreadContext::PopEntryExitRecord(Js::ScriptEntryExitRecord \* record)

{

AssertMsg(record && record == this->entryExitRecord, "Mismatch script entry/exit");

this->entryExitRecord = this->entryExitRecord->next;

}

BOOL ThreadContext::ReserveStaticTypeIds(\_\_in int first, \_\_in int last)

{

if ( nextTypeId <= first )

{

nextTypeId = (Js::TypeId) last;

return TRUE;

}

else

{

return FALSE;

}

}

Js::TypeId ThreadContext::ReserveTypeIds(int count)

{

Js::TypeId firstTypeId = nextTypeId;

nextTypeId = (Js::TypeId)(nextTypeId + count);

return firstTypeId;

}

Js::TypeId ThreadContext::CreateTypeId()

{

return nextTypeId = (Js::TypeId)(nextTypeId + 1);

}

WellKnownHostType ThreadContext::GetWellKnownHostType(Js::TypeId typeId)

{

if (this->wellKnownHostTypeHTMLAllCollectionTypeId == typeId)

{

return WellKnownHostType\_HTMLAllCollection;

}

return WellKnownHostType\_Invalid;

}

void ThreadContext::SetWellKnownHostTypeId(WellKnownHostType wellKnownType, Js::TypeId typeId)

{

AssertMsg(WellKnownHostType\_HTMLAllCollection == wellKnownType, "ThreadContext::SetWellKnownHostTypeId called on type other than HTMLAllCollection");

if (WellKnownHostType\_HTMLAllCollection == wellKnownType)

{

this->wellKnownHostTypeHTMLAllCollectionTypeId = typeId;

}

}

bool

ThreadContext::CanBeFalsy(Js::TypeId typeId)

{

// Declare all the type ID that can be falsy so we can avoid the falsy check in the JIT

return typeId == this->wellKnownHostTypeHTMLAllCollectionTypeId;

}

void ThreadContext::EnsureDebugManager()

{

if (this->debugManager == nullptr)

{

this->debugManager = HeapNew(Js::DebugManager, this, this->GetAllocationPolicyManager());

}

InterlockedIncrement(&crefSContextForDiag);

Assert(this->debugManager != nullptr);

}

void ThreadContext::ReleaseDebugManager()

{

Assert(crefSContextForDiag > 0);

long lref = InterlockedDecrement(&crefSContextForDiag);

if (lref == 0)

{

if (this->recyclableData != nullptr)

{

this->recyclableData->returnedValueList = nullptr;

}

this->debugManager->Close();

HeapDelete(this->debugManager);

this->debugManager = nullptr;

}

}

Js::TempArenaAllocatorObject \*

ThreadContext::GetTemporaryAllocator(LPCWSTR name)

{

AssertCanHandleOutOfMemory();

if (temporaryArenaAllocatorCount != 0)

{

temporaryArenaAllocatorCount--;

Js::TempArenaAllocatorObject \* allocator = recyclableData->temporaryArenaAllocators[temporaryArenaAllocatorCount];

recyclableData->temporaryArenaAllocators[temporaryArenaAllocatorCount] = nullptr;

return allocator;

}

return Js::TempArenaAllocatorObject::Create(this);

}

void

ThreadContext::ReleaseTemporaryAllocator(Js::TempArenaAllocatorObject \* tempAllocator)

{

if (temporaryArenaAllocatorCount < MaxTemporaryArenaAllocators)

{

tempAllocator->GetAllocator()->Reset();

recyclableData->temporaryArenaAllocators[temporaryArenaAllocatorCount] = tempAllocator;

temporaryArenaAllocatorCount++;

return;

}

tempAllocator->Dispose(false);

}

Js::TempGuestArenaAllocatorObject \*

ThreadContext::GetTemporaryGuestAllocator(LPCWSTR name)

{

AssertCanHandleOutOfMemory();

if (temporaryGuestArenaAllocatorCount != 0)

{

temporaryGuestArenaAllocatorCount--;

Js::TempGuestArenaAllocatorObject \* allocator = recyclableData->temporaryGuestArenaAllocators[temporaryGuestArenaAllocatorCount];

recyclableData->temporaryGuestArenaAllocators[temporaryGuestArenaAllocatorCount] = nullptr;

return allocator;

}

return Js::TempGuestArenaAllocatorObject::Create(this);

}

void

ThreadContext::ReleaseTemporaryGuestAllocator(Js::TempGuestArenaAllocatorObject \* tempGuestAllocator)

{

if (temporaryGuestArenaAllocatorCount < MaxTemporaryArenaAllocators)

{

tempGuestAllocator->GetAllocator()->Reset();

recyclableData->temporaryGuestArenaAllocators[temporaryGuestArenaAllocatorCount] = tempGuestAllocator;

temporaryGuestArenaAllocatorCount++;

return;

}

tempGuestAllocator->Dispose(false);

}

void

ThreadContext::AddToPendingScriptContextCloseList(Js::ScriptContext \* scriptContext)

{

Assert(scriptContext != nullptr);

if (rootPendingClose == nullptr)

{

rootPendingClose = scriptContext;

return;

}

// Prepend to the list.

scriptContext->SetNextPendingClose(rootPendingClose);

rootPendingClose = scriptContext;

}

void

ThreadContext::RemoveFromPendingClose(Js::ScriptContext \* scriptContext)

{

Assert(scriptContext != nullptr);

if (rootPendingClose == nullptr)

{

// We already sent a close message, ignore the notification.

return;

}

// Base case: The root is being removed. Move the root along.

if (scriptContext == rootPendingClose)

{

rootPendingClose = rootPendingClose->GetNextPendingClose();

return;

}

Js::ScriptContext \* currScriptContext = rootPendingClose;

Js::ScriptContext \* nextScriptContext = nullptr;

while (currScriptContext)

{

nextScriptContext = currScriptContext->GetNextPendingClose();

if (!nextScriptContext)

{

break;

}

if (nextScriptContext == scriptContext) {

// The next pending close ScriptContext is the one to be removed - set prev->next to next->next

currScriptContext->SetNextPendingClose(nextScriptContext->GetNextPendingClose());

return;

}

currScriptContext = nextScriptContext;

}

// We expect to find scriptContext in the pending close list.

Assert(false);

}

void ThreadContext::ClosePendingScriptContexts()

{

Js::ScriptContext \* scriptContext = rootPendingClose;

if (scriptContext == nullptr)

{

return;

}

Js::ScriptContext \* nextScriptContext;

do

{

nextScriptContext = scriptContext->GetNextPendingClose();

scriptContext->Close(false);

scriptContext = nextScriptContext;

}

while (scriptContext);

rootPendingClose = nullptr;

}

void

ThreadContext::AddToPendingProjectionContextCloseList(IProjectionContext \*projectionContext)

{

pendingProjectionContextCloseList->Add(projectionContext);

}

void

ThreadContext::RemoveFromPendingClose(IProjectionContext\* projectionContext)

{

pendingProjectionContextCloseList->Remove(projectionContext);

}

void ThreadContext::ClosePendingProjectionContexts()

{

IProjectionContext\* projectionContext;

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

{

projectionContext = pendingProjectionContextCloseList->Item(i);

projectionContext->Close();

}

pendingProjectionContextCloseList->Clear();

}

void

ThreadContext::RegisterScriptContext(Js::ScriptContext \*scriptContext)

{

// NOTE: ETW rundown thread may be reading the scriptContextList concurrently. We don't need to

// lock access because we only insert to the front here.

scriptContext->next = this->scriptContextList;

if (this->scriptContextList)

{

Assert(this->scriptContextList->prev == NULL);

this->scriptContextList->prev = scriptContext;

}

scriptContext->prev = NULL;

this->scriptContextList = scriptContext;

if(NoJIT())

{

scriptContext->ForceNoNative();

}

scriptContextCount++;

scriptContextEverRegistered = true;

}

void

ThreadContext::UnregisterScriptContext(Js::ScriptContext \*scriptContext)

{

// NOTE: ETW rundown thread may be reading the scriptContextList concurrently. Since this function

// is only called by ~ScriptContext() which already synchronized to ETW rundown, we are safe here.

if (scriptContext == this->scriptContextList)

{

Assert(scriptContext->prev == NULL);

this->scriptContextList = scriptContext->next;

}

else

{

scriptContext->prev->next = scriptContext->next;

}

if (scriptContext->next)

{

scriptContext->next->prev = scriptContext->prev;

}

scriptContextCount--;

}

ThreadContext::CollectCallBack \*

ThreadContext::AddRecyclerCollectCallBack(RecyclerCollectCallBackFunction callback, void \* context)

{

AutoCriticalSection autocs(&csCollectionCallBack);

CollectCallBack \* collectCallBack = this->collectCallBackList.PrependNode(&HeapAllocator::Instance);

collectCallBack->callback = callback;

collectCallBack->context = context;

this->hasCollectionCallBack = true;

return collectCallBack;

}

void

ThreadContext::RemoveRecyclerCollectCallBack(ThreadContext::CollectCallBack \* collectCallBack)

{

AutoCriticalSection autocs(&csCollectionCallBack);

this->collectCallBackList.RemoveElement(&HeapAllocator::Instance, collectCallBack);

this->hasCollectionCallBack = !this->collectCallBackList.Empty();

}

void

ThreadContext::PreCollectionCallBack(CollectionFlags flags)

{

#ifdef PERF\_COUNTERS

PHASE\_PRINT\_TESTTRACE1(Js::DeferParsePhase, L"TestTrace: deferparse - # of func: %d # deferparsed: %d\n", PerfCounter::CodeCounterSet::GetTotalFunctionCounter().GetValue(), PerfCounter::CodeCounterSet::GetDeferedFunctionCounter().GetValue());

#endif

// This needs to be done before ClearInlineCaches since that method can empty the list of

// script contexts with inline caches

this->ClearScriptContextCaches();

// Clear up references to types to avoid keep them alive

this->ClearPrototypeChainEnsuredToHaveOnlyWritableDataPropertiesCaches();

// Clean up unused memory before we start collecting

this->CleanNoCasePropertyMap();

this->TryEnterExpirableCollectMode();

const BOOL concurrent = flags & CollectMode\_Concurrent;

const BOOL partial = flags & CollectMode\_Partial;

if (!partial)

{

// Integrate allocated pages from background JIT threads

#if ENABLE\_NATIVE\_CODEGEN

if (codeGenNumberThreadAllocator)

{

codeGenNumberThreadAllocator->Integrate();

}

#endif

}

RecyclerCollectCallBackFlags callBackFlags = (RecyclerCollectCallBackFlags)

((concurrent ? Collect\_Begin\_Concurrent : Collect\_Begin) | (partial? Collect\_Begin\_Partial : Collect\_Begin));

CollectionCallBack(callBackFlags);

}

void

ThreadContext::PreSweepCallback()

{

#ifdef PERSISTENT\_INLINE\_CACHES

ClearInlineCachesWithDeadWeakRefs();

#else

ClearInlineCaches();

#endif

ClearIsInstInlineCaches();

ClearEquivalentTypeCaches();

this->dynamicObjectEnumeratorCacheMap.Clear();

}

void

ThreadContext::CollectionCallBack(RecyclerCollectCallBackFlags flags)

{

DListBase<CollectCallBack>::Iterator i(&this->collectCallBackList);

while (i.Next())

{

i.Data().callback(i.Data().context, flags);

}

}

void

ThreadContext::WaitCollectionCallBack()

{

// Avoid taking the lock if there are no call back

if (hasCollectionCallBack)

{

AutoCriticalSection autocs(&csCollectionCallBack);

CollectionCallBack(Collect\_Wait);

}

}

void

ThreadContext::PostCollectionCallBack()

{

CollectionCallBack(Collect\_End);

TryExitExpirableCollectMode();

// Recycler is null in the case where the ThreadContext is in the process of creating the recycler and

// we have a GC triggered (say because the -recyclerStress flag is passed in)

if (this->recycler != NULL && this->recycler->InCacheCleanupCollection())

{

this->recycler->ClearCacheCleanupCollection();

for (Js::ScriptContext \*scriptContext = scriptContextList; scriptContext; scriptContext = scriptContext->next)

{

scriptContext->CleanupWeakReferenceDictionaries();

}

}

}

#ifdef FAULT\_INJECTION

void

ThreadContext::DisposeScriptContextByFaultInjectionCallBack()

{

if (FAULTINJECT\_SCRIPT\_TERMINATION\_ON\_DISPOSE) {

int scriptContextToClose = -1;

/\* inject only if we have more than 1 script context\*/

uint totalScriptCount = GetScriptContextCount();

if (totalScriptCount > 1) {

if (Js::Configuration::Global.flags.FaultInjectionScriptContextToTerminateCount > 0)

{

scriptContextToClose = Js::Configuration::Global.flags.FaultInjectionScriptContextToTerminateCount % totalScriptCount;

for (Js::ScriptContext \*scriptContext = GetScriptContextList(); scriptContext; scriptContext = scriptContext->next)

{

if (scriptContextToClose-- == 0)

{

scriptContext->DisposeScriptContextByFaultInjection();

break;

}

}

}

else

{

fwprintf(stderr, L"\*\*\*FI: FaultInjectionScriptContextToTerminateCount Failed, Value should be > 0. \n");

}

}

}

}

#endif

#pragma region "Expirable Object Methods"

void

ThreadContext::TryExitExpirableCollectMode()

{

// If this feature is turned off or if we're already in profile collection mode, do nothing

// We also do nothing if expiration is explicitly disabled by someone lower down the stack

if (PHASE\_OFF1(Js::ExpirableCollectPhase) || !InExpirableCollectMode() || this->disableExpiration)

{

return;

}

if (InExpirableCollectMode())

{

OUTPUT\_TRACE(Js::ExpirableCollectPhase, L"Checking to see whether to complete Expirable Object Collection: GC Count is %d\n", this->expirableCollectModeGcCount);

if (this->expirableCollectModeGcCount > 0)

{

this->expirableCollectModeGcCount--;

}

if (this->expirableCollectModeGcCount == 0 &&

(this->recycler->InCacheCleanupCollection() || CONFIG\_FLAG(ForceExpireOnNonCacheCollect)))

{

OUTPUT\_TRACE(Js::ExpirableCollectPhase, L"Completing Expirable Object Collection\n");

ExpirableObjectList::Iterator expirableObjectIterator(this->expirableObjectList);

while (expirableObjectIterator.Next())

{

ExpirableObject\* object = expirableObjectIterator.Data();

Assert(object);

if (!object->IsObjectUsed())

{

object->Expire();

}

}

// Leave expirable collection mode

expirableCollectModeGcCount = -1;

}

}

}

bool

ThreadContext::InExpirableCollectMode()

{

// We're in expirable collect if we have expirable objects registered,

// and expirableCollectModeGcCount is not negative

// and when debugger is attaching, it might have set the function to deferredParse.

return (expirableObjectList != nullptr &&

numExpirableObjects > 0 &&

expirableCollectModeGcCount >= 0 &&

(this->GetDebugManager() != nullptr &&

!this->GetDebugManager()->IsDebuggerAttaching()));

}

void

ThreadContext::TryEnterExpirableCollectMode()

{

// If this feature is turned off or if we're already in profile collection mode, do nothing

if (PHASE\_OFF1(Js::ExpirableCollectPhase) || InExpirableCollectMode())

{

OUTPUT\_TRACE(Js::ExpirableCollectPhase, L"Not running Expirable Object Collection\n");

return;

}

double entryPointCollectionThreshold = Js::Configuration::Global.flags.ExpirableCollectionTriggerThreshold / 100.0;

double currentThreadNativeCodeRatio = ((double) GetCodeSize()) / Js::Constants::MaxThreadJITCodeHeapSize;

OUTPUT\_TRACE(Js::ExpirableCollectPhase, L"Current native code ratio: %f\n", currentThreadNativeCodeRatio);

if (currentThreadNativeCodeRatio > entryPointCollectionThreshold)

{

OUTPUT\_TRACE(Js::ExpirableCollectPhase, L"Setting up Expirable Object Collection\n");

this->expirableCollectModeGcCount = Js::Configuration::Global.flags.ExpirableCollectionGCCount;

ExpirableObjectList::Iterator expirableObjectIterator(this->expirableObjectList);

while (expirableObjectIterator.Next())

{

ExpirableObject\* object = expirableObjectIterator.Data();

Assert(object);

object->EnterExpirableCollectMode();

}

if (this->entryExitRecord != nullptr)

{

// If we're in script, we will do a stack walk, find the JavascriptFunction's on the stack

// and mark their entry points as being used so that we don't prematurely expire them

Js::ScriptContext\* topScriptContext = this->entryExitRecord->scriptContext;

Js::JavascriptStackWalker walker(topScriptContext, TRUE);

Js::JavascriptFunction\* javascriptFunction = nullptr;

while (walker.GetCallerWithoutInlinedFrames(&javascriptFunction))

{

if (javascriptFunction != nullptr && Js::ScriptFunction::Is(javascriptFunction))

{

Js::ScriptFunction\* scriptFunction = (Js::ScriptFunction\*) javascriptFunction;

Js::FunctionEntryPointInfo\* entryPointInfo = scriptFunction->GetFunctionEntryPointInfo();

entryPointInfo->SetIsObjectUsed();

scriptFunction->GetFunctionBody()->MapEntryPoints([](int index, Js::FunctionEntryPointInfo\* entryPoint){

entryPoint->SetIsObjectUsed();

});

}

}

}

}

}

void

ThreadContext::RegisterExpirableObject(ExpirableObject\* object)

{

Assert(this->expirableObjectList);

Assert(object->registrationHandle == nullptr);

ExpirableObject\*\* registrationData = this->expirableObjectList->PrependNode();

(\*registrationData) = object;

object->registrationHandle = (void\*) registrationData;

OUTPUT\_VERBOSE\_TRACE(Js::ExpirableCollectPhase, L"Registered 0x%p\n", object);

numExpirableObjects++;

}

void

ThreadContext::UnregisterExpirableObject(ExpirableObject\* object)

{

Assert(this->expirableObjectList);

Assert(object->registrationHandle != nullptr);

Assert(this->expirableObjectList->HasElement((ExpirableObject\* const \*) object->registrationHandle));

ExpirableObject\*\* registrationData = (ExpirableObject\*\*) object->registrationHandle;

Assert(\*registrationData == object);

this->expirableObjectList->MoveElementTo(registrationData, this->expirableObjectDisposeList);

object->registrationHandle = nullptr;

OUTPUT\_VERBOSE\_TRACE(Js::ExpirableCollectPhase, L"Unregistered 0x%p\n", object);

numExpirableObjects--;

}

void

ThreadContext::DisposeExpirableObject(ExpirableObject\* object)

{

Assert(this->expirableObjectDisposeList);

Assert(object->registrationHandle == nullptr);

this->expirableObjectDisposeList->Remove(object);

OUTPUT\_VERBOSE\_TRACE(Js::ExpirableCollectPhase, L"Disposed 0x%p\n", object);

}

#pragma endregion

void

ThreadContext::ClearScriptContextCaches()

{

// We go through just the inline script context list since if there is no script context

// registered on this list, we think that there's no script running on that script context

// so we can skip clearing its caches

FOREACH\_DLISTBASE\_ENTRY(Js::ScriptContext \*, scriptContext, &inlineCacheScriptContexts)

{

scriptContext->ClearScriptContextCaches();

}

NEXT\_DLISTBASE\_ENTRY;

}

#ifdef PERSISTENT\_INLINE\_CACHES

void

ThreadContext::ClearInlineCachesWithDeadWeakRefs()

{

FOREACH\_DLISTBASE\_ENTRY(Js::ScriptContext \*, scriptContext, &inlineCacheScriptContexts)

{

scriptContext->ClearInlineCachesWithDeadWeakRefs();

}

NEXT\_DLISTBASE\_ENTRY;

if (PHASE\_TRACE1(Js::InlineCachePhase))

{

size\_t size = 0;

size\_t freeListSize = 0;

size\_t polyInlineCacheSize = 0;

uint scriptContextCount = 0;

for (Js::ScriptContext \*scriptContext = scriptContextList;

scriptContext;

scriptContext = scriptContext->next)

{

scriptContextCount++;

size += scriptContext->GetInlineCacheAllocator()->AllocatedSize();

freeListSize += scriptContext->GetInlineCacheAllocator()->FreeListSize();

#ifdef POLY\_INLINE\_CACHE\_SIZE\_STATS

polyInlineCacheSize += scriptContext->GetInlineCacheAllocator()->GetPolyInlineCacheSize();

#endif

};

printf("Inline cache arena: total = %5I64u KB, free list = %5I64u KB, poly caches = %5I64u KB, script contexts = %u\n",

static\_cast<uint64>(size / 1024), static\_cast<uint64>(freeListSize / 1024), static\_cast<uint64>(polyInlineCacheSize / 1024), scriptContextCount);

}

}

void

ThreadContext::ClearInlineCaches()

{

BOOL hasItem = FALSE;

FOREACH\_DLISTBASE\_ENTRY(Js::ScriptContext \*, scriptContext, &inlineCacheScriptContexts)

{

scriptContext->ClearInlineCaches();

hasItem = TRUE;

}

NEXT\_DLISTBASE\_ENTRY;

#if DBG

for (Js::ScriptContext \*scriptContext = scriptContextList;

scriptContext;

scriptContext = scriptContext->next)

{

Assert(scriptContext->GetInlineCacheAllocator()->IsAllZero());

};

#endif

if (!hasItem)

{

return;

}

inlineCacheScriptContexts.Reset();

inlineCacheThreadInfoAllocator.Reset();

protoInlineCacheByPropId.ResetNoDelete();

storeFieldInlineCacheByPropId.ResetNoDelete();

registeredInlineCacheCount = 0;

unregisteredInlineCacheCount = 0;

if (PHASE\_TRACE1(Js::InlineCachePhase))

{

size\_t size = 0;

size\_t freeListSize = 0;

size\_t polyInlineCacheSize = 0;

uint scriptContextCount = 0;

for (Js::ScriptContext \*scriptContext = scriptContextList;

scriptContext;

scriptContext = scriptContext->next)

{

scriptContextCount++;

size += scriptContext->GetInlineCacheAllocator()->AllocatedSize();

freeListSize += scriptContext->GetInlineCacheAllocator()->FreeListSize();

#ifdef POLY\_INLINE\_CACHE\_SIZE\_STATS

polyInlineCacheSize += scriptContext->GetInlineCacheAllocator()->GetPolyInlineCacheSize();

#endif

};

printf("Inline cache arena: total = %5I64u KB, free list = %5I64u KB, poly caches = %5I64u KB, script contexts = %u\n",

static\_cast<uint64>(size / 1024), static\_cast<uint64>(freeListSize / 1024), static\_cast<uint64>(polyInlineCacheSize / 1024), scriptContextCount);

}

}

void

ThreadContext::ClearIsInstInlineCaches()

{

FOREACH\_DLISTBASE\_ENTRY(Js::ScriptContext \*, scriptContext, &isInstInlineCacheScriptContexts)

{

scriptContext->ClearIsInstInlineCaches();

}

NEXT\_DLISTBASE\_ENTRY;

#if DBG

for (Js::ScriptContext \*scriptContext = scriptContextList;

scriptContext;

scriptContext = scriptContext->next)

{

Assert(scriptContext->GetIsInstInlineCacheAllocator()->IsAllZero());

};

#endif

isInstInlineCacheScriptContexts.Reset();

isInstInlineCacheThreadInfoAllocator.Reset();

isInstInlineCacheByFunction.ResetNoDelete();

}

#endif

void

ThreadContext::ClearEquivalentTypeCaches()

{

#if ENABLE\_NATIVE\_CODEGEN

// Called from PreSweepCallback to clear pointers to types that have no live object references left.

// The EquivalentTypeCache used to keep these types alive, but this caused memory growth in cases where

// entry points stayed around for a long time.

// In future we may want to pin the reference/guard type to the entry point, but that choice will depend

// on a use case where pinning the type helps us optimize. Lacking that, clearing the guard type is a

// simpler short-term solution.

// Note that clearing unmarked types from the cache and guard is needed for correctness if the cache doesn't keep

// the types alive.

FOREACH\_DLISTBASE\_ENTRY\_EDITING(Js::EntryPointInfo \*, entryPoint, &equivalentTypeCacheEntryPoints, iter)

{

bool isLive = entryPoint->ClearEquivalentTypeCaches();

if (!isLive)

{

iter.RemoveCurrent(&equivalentTypeCacheInfoAllocator);

}

}

NEXT\_DLISTBASE\_ENTRY\_EDITING;

// Note: Don't reset the list, because we're only clearing the dead types from these caches.

// There may still be type references we need to keep an eye on.

#endif

}

Js::ScriptContext \*\*

ThreadContext::RegisterInlineCacheScriptContext(Js::ScriptContext \* scriptContext)

{

return inlineCacheScriptContexts.PrependNode(&inlineCacheThreadInfoAllocator, scriptContext);

}

void

ThreadContext::UnregisterInlineCacheScriptContext(Js::ScriptContext \*\* scriptContext)

{

inlineCacheScriptContexts.RemoveElement(&inlineCacheThreadInfoAllocator, scriptContext);

}

Js::ScriptContext \*\*

ThreadContext::RegisterIsInstInlineCacheScriptContext(Js::ScriptContext \* scriptContext)

{

return isInstInlineCacheScriptContexts.PrependNode(&isInstInlineCacheThreadInfoAllocator, scriptContext);

}

void

ThreadContext::UnregisterIsInstInlineCacheScriptContext(Js::ScriptContext \*\* scriptContext)

{

isInstInlineCacheScriptContexts.RemoveElement(&isInstInlineCacheThreadInfoAllocator, scriptContext);

}

Js::EntryPointInfo \*\*

ThreadContext::RegisterEquivalentTypeCacheEntryPoint(Js::EntryPointInfo \* entryPoint)

{

return equivalentTypeCacheEntryPoints.PrependNode(&equivalentTypeCacheInfoAllocator, entryPoint);

}

void

ThreadContext::UnregisterEquivalentTypeCacheEntryPoint(Js::EntryPointInfo \*\* entryPoint)

{

equivalentTypeCacheEntryPoints.RemoveElement(&equivalentTypeCacheInfoAllocator, entryPoint);

}

void

ThreadContext::RegisterProtoInlineCache(Js::InlineCache \* inlineCache, Js::PropertyId propertyId)

{

if (PHASE\_TRACE1(Js::TraceInlineCacheInvalidationPhase))

{

Output::Print(L"InlineCacheInvalidation: registering proto cache 0x%p for property %s(%u)\n",

inlineCache, GetPropertyName(propertyId)->GetBuffer(), propertyId);

Output::Flush();

}

RegisterInlineCache(protoInlineCacheByPropId, inlineCache, propertyId);

}

void

ThreadContext::RegisterStoreFieldInlineCache(Js::InlineCache \* inlineCache, Js::PropertyId propertyId)

{

if (PHASE\_TRACE1(Js::TraceInlineCacheInvalidationPhase))

{

Output::Print(L"InlineCacheInvalidation: registering store field cache 0x%p for property %s(%u)\n",

inlineCache, GetPropertyName(propertyId)->GetBuffer(), propertyId);

Output::Flush();

}

RegisterInlineCache(storeFieldInlineCacheByPropId, inlineCache, propertyId);

}

void

ThreadContext::RegisterInlineCache(InlineCacheListMapByPropertyId& inlineCacheMap, Js::InlineCache \* inlineCache, Js::PropertyId propertyId)

{

InlineCacheList\* inlineCacheList;

if (!inlineCacheMap.TryGetValue(propertyId, &inlineCacheList))

{

inlineCacheList = Anew(&this->inlineCacheThreadInfoAllocator, InlineCacheList, &this->inlineCacheThreadInfoAllocator);

inlineCacheMap.AddNew(propertyId, inlineCacheList);

}

Js::InlineCache\*\* inlineCacheRef = inlineCacheList->PrependNode();

Assert(inlineCacheRef != nullptr);

\*inlineCacheRef = inlineCache;

inlineCache->invalidationListSlotPtr = inlineCacheRef;

this->registeredInlineCacheCount++;

}

void ThreadContext::NotifyInlineCacheBatchUnregistered(uint count)

{

this->unregisteredInlineCacheCount += count;

// Negative or 0 InlineCacheInvalidationListCompactionThreshold forces compaction all the time.

if (CONFIG\_FLAG(InlineCacheInvalidationListCompactionThreshold) <= 0 ||

this->registeredInlineCacheCount / this->unregisteredInlineCacheCount < (uint)CONFIG\_FLAG(InlineCacheInvalidationListCompactionThreshold))

{

CompactInlineCacheInvalidationLists();

}

}

void

ThreadContext::InvalidateProtoInlineCaches(Js::PropertyId propertyId)

{

if (PHASE\_TRACE1(Js::TraceInlineCacheInvalidationPhase))

{

Output::Print(L"InlineCacheInvalidation: invalidating proto caches for property %s(%u)\n",

GetPropertyName(propertyId)->GetBuffer(), propertyId);

Output::Flush();

}

InlineCacheList\* inlineCacheList;

if (protoInlineCacheByPropId.TryGetValueAndRemove(propertyId, &inlineCacheList))

{

InvalidateInlineCacheList(inlineCacheList);

}

}

void

ThreadContext::InvalidateStoreFieldInlineCaches(Js::PropertyId propertyId)

{

if (PHASE\_TRACE1(Js::TraceInlineCacheInvalidationPhase))

{

Output::Print(L"InlineCacheInvalidation: invalidating store field caches for property %s(%u)\n",

GetPropertyName(propertyId)->GetBuffer(), propertyId);

Output::Flush();

}

InlineCacheList\* inlineCacheList;

if (storeFieldInlineCacheByPropId.TryGetValueAndRemove(propertyId, &inlineCacheList))

{

InvalidateInlineCacheList(inlineCacheList);

}

}

void

ThreadContext::InvalidateInlineCacheList(InlineCacheList\* inlineCacheList)

{

Assert(inlineCacheList != nullptr);

uint cacheCount = 0;

FOREACH\_SLISTBASE\_ENTRY(Js::InlineCache\*, inlineCache, inlineCacheList)

{

cacheCount++;

if (inlineCache != nullptr)

{

if (PHASE\_VERBOSE\_TRACE1(Js::TraceInlineCacheInvalidationPhase))

{

Output::Print(L"InlineCacheInvalidation: invalidating cache 0x%p\n", inlineCache);

Output::Flush();

}

memset(inlineCache, 0, sizeof(Js::InlineCache));

}

}

NEXT\_SLISTBASE\_ENTRY;

inlineCacheList->Clear();

this->registeredInlineCacheCount = this->registeredInlineCacheCount > cacheCount ? this->registeredInlineCacheCount - cacheCount : 0;

}

void

ThreadContext::CompactInlineCacheInvalidationLists()

{

Assert(this->unregisteredInlineCacheCount > 0);

CompactProtoInlineCaches();

if (this->unregisteredInlineCacheCount > 0)

{

CompactStoreFieldInlineCaches();

}

}

void

ThreadContext::CompactProtoInlineCaches()

{

protoInlineCacheByPropId.MapUntil([this](Js::PropertyId propertyId, InlineCacheList\* inlineCacheList)

{

CompactInlineCacheList(inlineCacheList);

return this->unregisteredInlineCacheCount == 0;

});

}

void

ThreadContext::CompactStoreFieldInlineCaches()

{

storeFieldInlineCacheByPropId.MapUntil([this](Js::PropertyId propertyId, InlineCacheList\* inlineCacheList)

{

CompactInlineCacheList(inlineCacheList);

return this->unregisteredInlineCacheCount == 0;

});

}

void

ThreadContext::CompactInlineCacheList(InlineCacheList\* inlineCacheList)

{

Assert(inlineCacheList != nullptr);

uint cacheCount = 0;

FOREACH\_SLISTBASE\_ENTRY\_EDITING(Js::InlineCache\*, inlineCache, inlineCacheList, iterator)

{

if (inlineCache == nullptr)

{

iterator.RemoveCurrent(&this->inlineCacheThreadInfoAllocator);

cacheCount++;

}

}

NEXT\_SLISTBASE\_ENTRY\_EDITING;

if (cacheCount > 0)

{

this->unregisteredInlineCacheCount = this->unregisteredInlineCacheCount > cacheCount ?

this->unregisteredInlineCacheCount - cacheCount : 0;

}

}

#if DBG

bool

ThreadContext::IsProtoInlineCacheRegistered(const Js::InlineCache\* inlineCache, Js::PropertyId propertyId)

{

return IsInlineCacheRegistered(protoInlineCacheByPropId, inlineCache, propertyId);

}

bool

ThreadContext::IsStoreFieldInlineCacheRegistered(const Js::InlineCache\* inlineCache, Js::PropertyId propertyId)

{

return IsInlineCacheRegistered(storeFieldInlineCacheByPropId, inlineCache, propertyId);

}

bool

ThreadContext::IsInlineCacheRegistered(InlineCacheListMapByPropertyId& inlineCacheMap, const Js::InlineCache\* inlineCache, Js::PropertyId propertyId)

{

InlineCacheList\* inlineCacheList;

if (inlineCacheMap.TryGetValue(propertyId, &inlineCacheList))

{

return IsInlineCacheInList(inlineCache, inlineCacheList);

}

else

{

return false;

}

}

bool

ThreadContext::IsInlineCacheInList(const Js::InlineCache\* inlineCache, const InlineCacheList\* inlineCacheList)

{

Assert(inlineCache != nullptr);

Assert(inlineCacheList != nullptr);

FOREACH\_SLISTBASE\_ENTRY(Js::InlineCache\*, curInlineCache, inlineCacheList)

{

if (curInlineCache == inlineCache)

{

return true;

}

}

NEXT\_SLISTBASE\_ENTRY;

return false;

}

#endif

#if ENABLE\_NATIVE\_CODEGEN

ThreadContext::PropertyGuardEntry\*

ThreadContext::EnsurePropertyGuardEntry(const Js::PropertyRecord\* propertyRecord, bool& foundExistingEntry)

{

PropertyGuardDictionary &guards = this->recyclableData->propertyGuards;

PropertyGuardEntry\* entry;

foundExistingEntry = guards.TryGetValue(propertyRecord, &entry);

if (!foundExistingEntry)

{

entry = RecyclerNew(GetRecycler(), PropertyGuardEntry, GetRecycler());

guards.UncheckedAdd(CreatePropertyRecordWeakRef(propertyRecord), entry);

}

return entry;

}

Js::PropertyGuard\*

ThreadContext::RegisterSharedPropertyGuard(Js::PropertyId propertyId)

{

Assert(IsActivePropertyId(propertyId));

const Js::PropertyRecord \* propertyRecord = GetPropertyName(propertyId);

bool foundExistingGuard;

PropertyGuardEntry\* entry = EnsurePropertyGuardEntry(propertyRecord, foundExistingGuard);

if (entry->sharedGuard == nullptr)

{

entry->sharedGuard = Js::PropertyGuard::New(GetRecycler());

}

Js::PropertyGuard\* guard = entry->sharedGuard;

PHASE\_PRINT\_VERBOSE\_TRACE1(Js::FixedMethodsPhase, L"FixedFields: registered shared guard: name: %s, address: 0x%p, value: 0x%p, value address: 0x%p, %s\n",

propertyRecord->GetBuffer(), guard, guard->GetValue(), guard->GetAddressOfValue(), foundExistingGuard ? L"existing" : L"new");

PHASE\_PRINT\_TESTTRACE1(Js::FixedMethodsPhase, L"FixedFields: registered shared guard: name: %s, value: 0x%p, %s\n",

propertyRecord->GetBuffer(), guard->GetValue(), foundExistingGuard ? L"existing" : L"new");

return guard;

}

void

ThreadContext::RegisterLazyBailout(Js::PropertyId propertyId, Js::EntryPointInfo\* entryPoint)

{

const Js::PropertyRecord \* propertyRecord = GetPropertyName(propertyId);

bool foundExistingGuard;

PropertyGuardEntry\* entry = EnsurePropertyGuardEntry(propertyRecord, foundExistingGuard);

if (!entry->entryPoints)

{

entry->entryPoints = RecyclerNew(recycler, PropertyGuardEntry::EntryPointDictionary, recycler, /\*capacity\*/ 3);

}

entry->entryPoints->UncheckedAdd(entryPoint, NULL);

}

void

ThreadContext::RegisterUniquePropertyGuard(Js::PropertyId propertyId, Js::PropertyGuard\* guard)

{

Assert(IsActivePropertyId(propertyId));

Assert(guard != nullptr);

RecyclerWeakReference<Js::PropertyGuard>\* guardWeakRef = this->recycler->CreateWeakReferenceHandle(guard);

RegisterUniquePropertyGuard(propertyId, guardWeakRef);

}

void

ThreadContext::RegisterUniquePropertyGuard(Js::PropertyId propertyId, RecyclerWeakReference<Js::PropertyGuard>\* guardWeakRef)

{

Assert(IsActivePropertyId(propertyId));

Assert(guardWeakRef != nullptr);

Js::PropertyGuard\* guard = guardWeakRef->Get();

Assert(guard != nullptr);

const Js::PropertyRecord \* propertyRecord = GetPropertyName(propertyId);

bool foundExistingGuard;

PropertyGuardEntry\* entry = EnsurePropertyGuardEntry(propertyRecord, foundExistingGuard);

entry->uniqueGuards.Item(guardWeakRef);

if (PHASE\_TRACE1(Js::TracePropertyGuardsPhase) || PHASE\_VERBOSE\_TRACE1(Js::FixedMethodsPhase))

{

Output::Print(L"FixedFields: registered unique guard: name: %s, address: 0x%p, value: 0x%p, value address: 0x%p, %s entry\n",

propertyRecord->GetBuffer(), guard, guard->GetValue(), guard->GetAddressOfValue(), foundExistingGuard ? L"existing" : L"new");

Output::Flush();

}

if (PHASE\_TESTTRACE1(Js::TracePropertyGuardsPhase) || PHASE\_VERBOSE\_TESTTRACE1(Js::FixedMethodsPhase))

{

Output::Print(L"FixedFields: registered unique guard: name: %s, value: 0x%p, %s entry\n",

propertyRecord->GetBuffer(), guard->GetValue(), foundExistingGuard ? L"existing" : L"new");

Output::Flush();

}

}

void

ThreadContext::RegisterConstructorCache(Js::PropertyId propertyId, Js::ConstructorCache\* cache)

{

Assert(Js::ConstructorCache::GetOffsetOfGuardValue() == Js::PropertyGuard::GetOffsetOfValue());

Assert(Js::ConstructorCache::GetSizeOfGuardValue() == Js::PropertyGuard::GetSizeOfValue());

RegisterUniquePropertyGuard(propertyId, reinterpret\_cast<Js::PropertyGuard\*>(cache));

}

void

ThreadContext::InvalidatePropertyGuardEntry(const Js::PropertyRecord\* propertyRecord, PropertyGuardEntry\* entry)

{

Assert(entry != nullptr);

if (entry->sharedGuard != nullptr)

{

Js::PropertyGuard\* guard = entry->sharedGuard;

if (PHASE\_TRACE1(Js::TracePropertyGuardsPhase) || PHASE\_VERBOSE\_TRACE1(Js::FixedMethodsPhase))

{

Output::Print(L"FixedFields: invalidating guard: name: %s, address: 0x%p, value: 0x%p, value address: 0x%p\n",

propertyRecord->GetBuffer(), guard, guard->GetValue(), guard->GetAddressOfValue());

Output::Flush();

}

if (PHASE\_TESTTRACE1(Js::TracePropertyGuardsPhase) || PHASE\_VERBOSE\_TESTTRACE1(Js::FixedMethodsPhase))

{

Output::Print(L"FixedFields: invalidating guard: name: %s, value: 0x%p\n", propertyRecord->GetBuffer(), guard->GetValue());

Output::Flush();

}

guard->Invalidate();

}

entry->uniqueGuards.Map([propertyRecord](RecyclerWeakReference<Js::PropertyGuard>\* guardWeakRef)

{

Js::PropertyGuard\* guard = guardWeakRef->Get();

if (guard != nullptr)

{

if (PHASE\_TRACE1(Js::TracePropertyGuardsPhase) || PHASE\_VERBOSE\_TRACE1(Js::FixedMethodsPhase))

{

Output::Print(L"FixedFields: invalidating guard: name: %s, address: 0x%p, value: 0x%p, value address: 0x%p\n",

propertyRecord->GetBuffer(), guard, guard->GetValue(), guard->GetAddressOfValue());

Output::Flush();

}

if (PHASE\_TESTTRACE1(Js::TracePropertyGuardsPhase) || PHASE\_VERBOSE\_TESTTRACE1(Js::FixedMethodsPhase))

{

Output::Print(L"FixedFields: invalidating guard: name: %s, value: 0x%p\n",

propertyRecord->GetBuffer(), guard->GetValue());

Output::Flush();

}

guard->Invalidate();

}

});

entry->uniqueGuards.Clear();

if (entry->entryPoints && entry->entryPoints->Count() > 0)

{

Js::JavascriptStackWalker stackWalker(this->GetScriptContextList());

Js::JavascriptFunction\* caller;

while (stackWalker.GetCaller(&caller, /\*includeInlineFrames\*/ false))

{

// If the current frame is already from a bailout - we do not need to do on stack invalidation

if (caller != nullptr && Js::ScriptFunction::Is(caller) && !stackWalker.GetCurrentFrameFromBailout())

{

BYTE dummy;

Js::FunctionEntryPointInfo\* functionEntryPoint = caller->GetFunctionBody()->GetDefaultFunctionEntryPointInfo();

if (functionEntryPoint->IsInNativeAddressRange((DWORD\_PTR)stackWalker.GetInstructionPointer()))

{

if (entry->entryPoints->TryGetValue(functionEntryPoint, &dummy))

{

functionEntryPoint->DoLazyBailout(stackWalker.GetCurrentAddressOfInstructionPointer(),

caller->GetFunctionBody(), propertyRecord);

}

}

}

}

entry->entryPoints->Map([=](Js::EntryPointInfo\* info, BYTE& dummy, const RecyclerWeakReference<Js::EntryPointInfo>\* infoWeakRef)

{

OUTPUT\_TRACE2(Js::LazyBailoutPhase, info->GetFunctionBody(), L"Lazy bailout - Invalidation due to property: %s \n", propertyRecord->GetBuffer());

info->Invalidate(true);

});

entry->entryPoints->Clear();

}

}

void

ThreadContext::InvalidatePropertyGuards(Js::PropertyId propertyId)

{

const Js::PropertyRecord\* propertyRecord = GetPropertyName(propertyId);

PropertyGuardDictionary &guards = this->recyclableData->propertyGuards;

PropertyGuardEntry\* entry;

if (guards.TryGetValueAndRemove(propertyRecord, &entry))

{

InvalidatePropertyGuardEntry(propertyRecord, entry);

}

}

void

ThreadContext::InvalidateAllPropertyGuards()

{

PropertyGuardDictionary &guards = this->recyclableData->propertyGuards;

if (guards.Count() > 0)

{

guards.Map([this](Js::PropertyRecord const \* propertyRecord, PropertyGuardEntry\* entry, const RecyclerWeakReference<const Js::PropertyRecord>\* weakRef)

{

InvalidatePropertyGuardEntry(propertyRecord, entry);

});

guards.Clear();

}

}

#endif

void

ThreadContext::InvalidateAllProtoInlineCaches()

{

protoInlineCacheByPropId.Map([this](Js::PropertyId propertyId, InlineCacheList\* inlineCacheList)

{

InvalidateInlineCacheList(inlineCacheList);

});

protoInlineCacheByPropId.ResetNoDelete();

}

bool

ThreadContext::AreAllProtoInlineCachesInvalidated()

{

return protoInlineCacheByPropId.Count() == 0;

}

void

ThreadContext::InvalidateAllStoreFieldInlineCaches()

{

storeFieldInlineCacheByPropId.Map([this](Js::PropertyId propertyId, InlineCacheList\* inlineCacheList)

{

InvalidateInlineCacheList(inlineCacheList);

});

storeFieldInlineCacheByPropId.ResetNoDelete();

}

bool

ThreadContext::AreAllStoreFieldInlineCachesInvalidated()

{

return storeFieldInlineCacheByPropId.Count() == 0;

}

#if DBG

bool

ThreadContext::IsIsInstInlineCacheRegistered(Js::IsInstInlineCache \* inlineCache, Js::Var function)

{

Assert(inlineCache != nullptr);

Assert(function != nullptr);

Js::IsInstInlineCache\* firstInlineCache;

if (this->isInstInlineCacheByFunction.TryGetValue(function, &firstInlineCache))

{

return IsIsInstInlineCacheInList(inlineCache, firstInlineCache);

}

else

{

return false;

}

}

#endif

void

ThreadContext::RegisterIsInstInlineCache(Js::IsInstInlineCache \* inlineCache, Js::Var function)

{

Assert(function != nullptr);

Assert(inlineCache != nullptr);

// We should never cross-register or re-register a cache that is already on some invalidation list (for its function or some other function).

// Every cache must be first cleared and unregistered before being registered again.

AssertMsg(inlineCache->function == nullptr, "We should only register instance-of caches that have not yet been populated.");

Js::IsInstInlineCache\*\* inlineCacheRef = nullptr;

if (this->isInstInlineCacheByFunction.TryGetReference(function, &inlineCacheRef))

{

AssertMsg(!IsIsInstInlineCacheInList(inlineCache, \*inlineCacheRef), "Why are we registering a cache that is already registered?");

inlineCache->next = \*inlineCacheRef;

\*inlineCacheRef = inlineCache;

}

else

{

inlineCache->next = nullptr;

this->isInstInlineCacheByFunction.Add(function, inlineCache);

}

}

void

ThreadContext::UnregisterIsInstInlineCache(Js::IsInstInlineCache \* inlineCache, Js::Var function)

{

Assert(inlineCache != nullptr);

Js::IsInstInlineCache\*\* inlineCacheRef = nullptr;

if (this->isInstInlineCacheByFunction.TryGetReference(function, &inlineCacheRef))

{

Assert(\*inlineCacheRef != nullptr);

if (inlineCache == \*inlineCacheRef)

{

\*inlineCacheRef = (\*inlineCacheRef)->next;

if (\*inlineCacheRef == nullptr)

{

this->isInstInlineCacheByFunction.Remove(function);

}

}

else

{

Js::IsInstInlineCache \* prevInlineCache;

Js::IsInstInlineCache \* curInlineCache;

for (prevInlineCache = \*inlineCacheRef, curInlineCache = (\*inlineCacheRef)->next; curInlineCache != nullptr;

prevInlineCache = curInlineCache, curInlineCache = curInlineCache->next)

{

if (curInlineCache == inlineCache)

{

prevInlineCache->next = curInlineCache->next;

return;

}

}

AssertMsg(false, "Why are we unregistering a cache that is not registered?");

}

}

}

void

ThreadContext::InvalidateIsInstInlineCacheList(Js::IsInstInlineCache\* inlineCacheList)

{

Assert(inlineCacheList != nullptr);

Js::IsInstInlineCache\* curInlineCache;

Js::IsInstInlineCache\* nextInlineCache;

for (curInlineCache = inlineCacheList; curInlineCache != nullptr; curInlineCache = nextInlineCache)

{

if (PHASE\_VERBOSE\_TRACE1(Js::TraceInlineCacheInvalidationPhase))

{

Output::Print(L"InlineCacheInvalidation: invalidating instanceof cache 0x%p\n", curInlineCache);

Output::Flush();

}

// Stash away the next cache before we zero out the current one (including its next pointer).

nextInlineCache = curInlineCache->next;

// Clear the current cache to invalidate it.

memset(curInlineCache, 0, sizeof(Js::IsInstInlineCache));

}

}

void

ThreadContext::InvalidateIsInstInlineCachesForFunction(Js::Var function)

{

Js::IsInstInlineCache\* inlineCacheList;

if (this->isInstInlineCacheByFunction.TryGetValueAndRemove(function, &inlineCacheList))

{

InvalidateIsInstInlineCacheList(inlineCacheList);

}

}

void

ThreadContext::InvalidateAllIsInstInlineCaches()

{

isInstInlineCacheByFunction.Map([this](const Js::Var function, Js::IsInstInlineCache\* inlineCacheList)

{

InvalidateIsInstInlineCacheList(inlineCacheList);

});

isInstInlineCacheByFunction.Clear();

}

bool

ThreadContext::AreAllIsInstInlineCachesInvalidated() const

{

return isInstInlineCacheByFunction.Count() == 0;

}

#if DBG

bool

ThreadContext::IsIsInstInlineCacheInList(const Js::IsInstInlineCache\* inlineCache, const Js::IsInstInlineCache\* inlineCacheList)

{

Assert(inlineCache != nullptr);

Assert(inlineCacheList != nullptr);

for (const Js::IsInstInlineCache\* curInlineCache = inlineCacheList; curInlineCache != nullptr; curInlineCache = curInlineCache->next)

{

if (curInlineCache == inlineCache)

{

return true;

}

}

return false;

}

#endif

void ThreadContext::RegisterTypeWithProtoPropertyCache(const Js::PropertyId propertyId, Js::Type \*const type)

{

Assert(propertyId != Js::Constants::NoProperty);

Assert(IsActivePropertyId(propertyId));

Assert(type);

PropertyIdToTypeHashSetDictionary &typesWithProtoPropertyCache = recyclableData->typesWithProtoPropertyCache;

TypeHashSet \*typeHashSet;

if(!typesWithProtoPropertyCache.TryGetValue(propertyId, &typeHashSet))

{

typeHashSet = RecyclerNew(recycler, TypeHashSet, recycler);

typesWithProtoPropertyCache.Add(propertyId, typeHashSet);

}

typeHashSet->Item(type, false);

}

void ThreadContext::InvalidateProtoTypePropertyCaches(const Js::PropertyId propertyId)

{

Assert(propertyId != Js::Constants::NoProperty);

Assert(IsActivePropertyId(propertyId));

InternalInvalidateProtoTypePropertyCaches(propertyId);

}

void ThreadContext::InternalInvalidateProtoTypePropertyCaches(const Js::PropertyId propertyId)

{

// Get the hash set of registered types associated with the property ID, invalidate each type in the hash set, and

// remove the property ID and its hash set from the map

PropertyIdToTypeHashSetDictionary &typesWithProtoPropertyCache = recyclableData->typesWithProtoPropertyCache;

TypeHashSet \*typeHashSet;

if(typesWithProtoPropertyCache.Count() != 0 && typesWithProtoPropertyCache.TryGetValueAndRemove(propertyId, &typeHashSet))

{

DoInvalidateProtoTypePropertyCaches(propertyId, typeHashSet);

}

}

void ThreadContext::InvalidateAllProtoTypePropertyCaches()

{

PropertyIdToTypeHashSetDictionary &typesWithProtoPropertyCache = recyclableData->typesWithProtoPropertyCache;

if (typesWithProtoPropertyCache.Count() > 0)

{

typesWithProtoPropertyCache.Map([this](Js::PropertyId propertyId, TypeHashSet \* typeHashSet)

{

DoInvalidateProtoTypePropertyCaches(propertyId, typeHashSet);

});

typesWithProtoPropertyCache.Clear();

}

}

void ThreadContext::DoInvalidateProtoTypePropertyCaches(const Js::PropertyId propertyId, TypeHashSet \*const typeHashSet)

{

Assert(propertyId != Js::Constants::NoProperty);

Assert(typeHashSet);

typeHashSet->Map(

[propertyId](Js::Type \*const type, const bool unused, const RecyclerWeakReference<Js::Type>\*)

{

type->GetPropertyCache()->ClearIfPropertyIsOnAPrototype(propertyId);

});

}

Js::ScriptContext \*\*

ThreadContext::RegisterPrototypeChainEnsuredToHaveOnlyWritableDataPropertiesScriptContext(Js::ScriptContext \* scriptContext)

{

return prototypeChainEnsuredToHaveOnlyWritableDataPropertiesScriptContext.PrependNode(&prototypeChainEnsuredToHaveOnlyWritableDataPropertiesAllocator, scriptContext);

}

void

ThreadContext::UnregisterPrototypeChainEnsuredToHaveOnlyWritableDataPropertiesScriptContext(Js::ScriptContext \*\* scriptContext)

{

prototypeChainEnsuredToHaveOnlyWritableDataPropertiesScriptContext.RemoveElement(&prototypeChainEnsuredToHaveOnlyWritableDataPropertiesAllocator, scriptContext);

}

void

ThreadContext::ClearPrototypeChainEnsuredToHaveOnlyWritableDataPropertiesCaches()

{

bool hasItem = false;

FOREACH\_DLISTBASE\_ENTRY(Js::ScriptContext \*, scriptContext, &prototypeChainEnsuredToHaveOnlyWritableDataPropertiesScriptContext)

{

scriptContext->ClearPrototypeChainEnsuredToHaveOnlyWritableDataPropertiesCaches();

hasItem = true;

}

NEXT\_DLISTBASE\_ENTRY;

if (!hasItem)

{

return;

}

prototypeChainEnsuredToHaveOnlyWritableDataPropertiesScriptContext.Reset();

prototypeChainEnsuredToHaveOnlyWritableDataPropertiesAllocator.Reset();

}

BOOL ThreadContext::HasPreviousHostScriptContext()

{

return hostScriptContextStack->Count() != 0;

}

HostScriptContext\* ThreadContext::GetPreviousHostScriptContext()

{

return hostScriptContextStack->Peek();

}

void ThreadContext::PushHostScriptContext(HostScriptContext\* topProvider)

{

// script engine can be created coming from GetDispID, so push/pop can be

// happening after the first round of enterscript as well. we might need to

// revisit the whole callRootLevel but probably not now.

// Assert(HasPreviousHostScriptContext() || callRootLevel == 0);

hostScriptContextStack->Push(topProvider);

}

HostScriptContext\* ThreadContext::PopHostScriptContext()

{

return hostScriptContextStack->Pop();

// script engine can be created coming from GetDispID, so push/pop can be

// happening after the first round of enterscript as well. we might need to

// revisit the whole callRootLevel but probably not now.

// Assert(HasPreviousHostScriptContext() || callRootLevel == 0);

}

#if DBG || defined(PROFILE\_EXEC)

bool

ThreadContext::AsyncHostOperationStart(void \* suspendRecord)

{

bool wasInAsync = false;

Assert(!this->IsScriptActive());

Js::ScriptEntryExitRecord \* lastRecord = this->entryExitRecord;

if (lastRecord != NULL)

{

if (!lastRecord->leaveForHost)

{

#if DBG

wasInAsync = !!lastRecord->leaveForAsyncHostOperation;

lastRecord->leaveForAsyncHostOperation = true;

#endif

#ifdef PROFILE\_EXEC

lastRecord->scriptContext->ProfileSuspend(Js::RunPhase, (Js::Profiler::SuspendRecord \*)suspendRecord);

#endif

}

else

{

Assert(!lastRecord->leaveForAsyncHostOperation);

}

}

return wasInAsync;

}

void

ThreadContext::AsyncHostOperationEnd(bool wasInAsync, void \* suspendRecord)

{

Assert(!this->IsScriptActive());

Js::ScriptEntryExitRecord \* lastRecord = this->entryExitRecord;

if (lastRecord != NULL)

{

if (!lastRecord->leaveForHost)

{

Assert(lastRecord->leaveForAsyncHostOperation);

#if DBG

lastRecord->leaveForAsyncHostOperation = wasInAsync;

#endif

#ifdef PROFILE\_EXEC

lastRecord->scriptContext->ProfileResume((Js::Profiler::SuspendRecord \*)suspendRecord);

#endif

}

else

{

Assert(!lastRecord->leaveForAsyncHostOperation);

Assert(!wasInAsync);

}

}

}

#endif

#ifdef RECYCLER\_DUMP\_OBJECT\_GRAPH

void DumpRecyclerObjectGraph()

{

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

if (threadContext == nullptr)

{

Output::Print(L"No thread context");

}

threadContext->GetRecycler()->DumpObjectGraph();

}

#endif

#if ENABLE\_NATIVE\_CODEGEN

BOOL ThreadContext::IsNativeAddress(void \*pCodeAddr)

{

for (Js::ScriptContext \*scriptContext = this->scriptContextList;

scriptContext;

scriptContext = scriptContext->next)

{

if (IsNativeFunctionAddr(scriptContext, pCodeAddr))

{

return TRUE;

}

};

return FALSE;

}

#endif

#if ENABLE\_PROFILE\_INFO

void ThreadContext::EnsureSourceProfileManagersByUrlMap()

{

if(this->recyclableData->sourceProfileManagersByUrl == nullptr)

{

this->EnsureRecycler();

this->recyclableData->sourceProfileManagersByUrl = RecyclerNew(GetRecycler(), SourceProfileManagersByUrlMap, GetRecycler());

}

}

//

// Returns the cache profile manager for the URL and hash combination for a particular dynamic script. There is a ref count added for every script context

// that references the shared profile manager info.

//

Js::SourceDynamicProfileManager\* ThreadContext::GetSourceDynamicProfileManager(\_In\_z\_ const WCHAR\* url, \_In\_ uint hash, \_Inout\_ bool\* addRef)

{

EnsureSourceProfileManagersByUrlMap();

Js::SourceDynamicProfileManager\* profileManager = nullptr;

SourceDynamicProfileManagerCache\* managerCache;

bool newCache = false;

if(!this->recyclableData->sourceProfileManagersByUrl->TryGetValue(url, &managerCache))

{

if(this->recyclableData->sourceProfileManagersByUrl->Count() >= INMEMORY\_CACHE\_MAX\_URL)

{

return nullptr;

}

managerCache = RecyclerNewZ(this->GetRecycler(), SourceDynamicProfileManagerCache);

newCache = true;

}

bool createProfileManager = false;

if(!managerCache->sourceProfileManagerMap)

{

managerCache->sourceProfileManagerMap = RecyclerNew(this->GetRecycler(), SourceDynamicProfileManagerMap, this->GetRecycler());

createProfileManager = true;

}

else

{

createProfileManager = !managerCache->sourceProfileManagerMap->TryGetValue(hash, &profileManager);

}

if(createProfileManager)

{

if(managerCache->sourceProfileManagerMap->Count() < INMEMORY\_CACHE\_MAX\_PROFILE\_MANAGER)

{

profileManager = RecyclerNewZ(this->GetRecycler(), Js::SourceDynamicProfileManager, this->GetRecycler());

managerCache->sourceProfileManagerMap->Add(hash, profileManager);

}

}

else

{

profileManager->Reuse();

}

if(!\*addRef)

{

managerCache->AddRef();

\*addRef = true;

OUTPUT\_VERBOSE\_TRACE(Js::DynamicProfilePhase, L"Addref dynamic source profile manger - Url: %s\n", url);

}

if (newCache)

{

// Let's make a copy of the URL because there is no guarantee this URL will remain alive in the future.

size\_t lengthInChars = wcslen(url) + 1;

WCHAR\* urlCopy = RecyclerNewArrayLeaf(GetRecycler(), WCHAR, lengthInChars);

js\_memcpy\_s(urlCopy, lengthInChars \* sizeof(WCHAR), url, lengthInChars \* sizeof(WCHAR));

this->recyclableData->sourceProfileManagersByUrl->Add(urlCopy, managerCache);

}

return profileManager;

}

//

// Decrement the ref count for this URL and cleanup the corresponding record if there are no other references to it.

//

uint ThreadContext::ReleaseSourceDynamicProfileManagers(const WCHAR\* url)

{

// If we've already freed the recyclable data, we're shutting down the thread context so skip clean up

if (this->recyclableData == nullptr) return 0;

SourceDynamicProfileManagerCache\* managerCache = this->recyclableData->sourceProfileManagersByUrl->Lookup(url, nullptr);

uint refCount = 0;

if(managerCache) // manager cache may be null we exceeded -INMEMORY\_CACHE\_MAX\_URL

{

refCount = managerCache->Release();

OUTPUT\_VERBOSE\_TRACE(Js::DynamicProfilePhase, L"Release dynamic source profile manger %d Url: %s\n", refCount, url);

Output::Flush();

if(refCount == 0)

{

this->recyclableData->sourceProfileManagersByUrl->Remove(url);

}

}

return refCount;

}

#endif

void ThreadContext::EnsureSymbolRegistrationMap()

{

if (this->recyclableData->symbolRegistrationMap == nullptr)

{

this->EnsureRecycler();

this->recyclableData->symbolRegistrationMap = RecyclerNew(GetRecycler(), SymbolRegistrationMap, GetRecycler());

}

}

const Js::PropertyRecord\* ThreadContext::GetSymbolFromRegistrationMap(const wchar\_t\* stringKey)

{

this->EnsureSymbolRegistrationMap();

return this->recyclableData->symbolRegistrationMap->Lookup(stringKey, nullptr);

}

const Js::PropertyRecord\* ThreadContext::AddSymbolToRegistrationMap(const wchar\_t\* stringKey, charcount\_t stringLength)

{

this->EnsureSymbolRegistrationMap();

const Js::PropertyRecord\* propertyRecord = this->UncheckedAddPropertyId(stringKey, stringLength, /\*bind\*/false, /\*isSymbol\*/true);

Assert(propertyRecord);

// The key is the PropertyRecord's buffer (the PropertyRecord itself) which is being pinned as long as it's in this map.

// If that's ever not the case, we'll need to duplicate the key here and put that in the map instead.

this->recyclableData->symbolRegistrationMap->Add(propertyRecord->GetBuffer(), propertyRecord);

return propertyRecord;

}

void ThreadContext::ClearImplicitCallFlags()

{

SetImplicitCallFlags(Js::ImplicitCall\_None);

}

void ThreadContext::ClearImplicitCallFlags(Js::ImplicitCallFlags flags)

{

Assert((flags & Js::ImplicitCall\_None) == 0);

SetImplicitCallFlags((Js::ImplicitCallFlags)(implicitCallFlags & ~flags));

}

void ThreadContext::CheckAndResetImplicitCallAccessorFlag()

{

Js::ImplicitCallFlags accessorCallFlag = (Js::ImplicitCallFlags)(Js::ImplicitCall\_Accessor & ~Js::ImplicitCall\_None);

if ((GetImplicitCallFlags() & accessorCallFlag) != 0)

{

ClearImplicitCallFlags(accessorCallFlag);

AddImplicitCallFlags(Js::ImplicitCall\_NonProfiledAccessor);

}

}

bool ThreadContext::HasNoSideEffect(Js::RecyclableObject \* function) const

{

Js::FunctionInfo::Attributes attributes = Js::FunctionInfo::GetAttributes(function);

return this->HasNoSideEffect(function, attributes);

}

bool ThreadContext::HasNoSideEffect(Js::RecyclableObject \* function, Js::FunctionInfo::Attributes attributes) const

{

if (((attributes & Js::FunctionInfo::CanBeHoisted) != 0)

|| ((attributes & Js::FunctionInfo::HasNoSideEffect) != 0 && !IsDisableImplicitException()))

{

Assert((attributes & Js::FunctionInfo::HasNoSideEffect) != 0);

return true;

}

return false;

}

bool

ThreadContext::RecordImplicitException()

{

AddImplicitCallFlags(Js::ImplicitCall\_Exception);

return !IsDisableImplicitException();

}

void ThreadContext::SetThreadServiceWrapper(ThreadServiceWrapper\* inThreadServiceWrapper)

{

AssertMsg(threadServiceWrapper == NULL || inThreadServiceWrapper == NULL, "double set ThreadServiceWrapper");

threadServiceWrapper = inThreadServiceWrapper;

}

ThreadServiceWrapper\* ThreadContext::GetThreadServiceWrapper()

{

return threadServiceWrapper;

}

uint ThreadContext::GetRandomNumber()

{

return (uint)GetEntropy().GetRand();

}

#ifdef ENABLE\_JS\_ETW

void ThreadContext::EtwLogPropertyIdList()

{

propertyMap->Map([&](const Js::PropertyRecord\* propertyRecord){

EventWriteJSCRIPT\_HOSTING\_PROPERTYID\_LIST(propertyRecord, propertyRecord->GetBuffer());

});

}

#endif

#ifdef ENABLE\_PROJECTION

void ThreadContext::AddExternalWeakReferenceCache(ExternalWeakReferenceCache \*externalWeakReferenceCache)

{

this->externalWeakReferenceCacheList.Prepend(&HeapAllocator::Instance, externalWeakReferenceCache);

}

void ThreadContext::RemoveExternalWeakReferenceCache(ExternalWeakReferenceCache \*externalWeakReferenceCache)

{

Assert(!externalWeakReferenceCacheList.Empty());

this->externalWeakReferenceCacheList.Remove(&HeapAllocator::Instance, externalWeakReferenceCache);

}

void ThreadContext::MarkExternalWeakReferencedObjects(bool inPartialCollect)

{

SListBase<ExternalWeakReferenceCache \*>::Iterator iteratorWeakRefCache(&this->externalWeakReferenceCacheList);

while (iteratorWeakRefCache.Next())

{

iteratorWeakRefCache.Data()->MarkNow(recycler, inPartialCollect);

}

}

void ThreadContext::ResolveExternalWeakReferencedObjects()

{

SListBase<ExternalWeakReferenceCache \*>::Iterator iteratorWeakRefCache(&this->externalWeakReferenceCacheList);

while (iteratorWeakRefCache.Next())

{

iteratorWeakRefCache.Data()->ResolveNow(recycler);

}

}

#if DBG\_DUMP

void ThreadContext::RegisterProjectionMemoryInformation(IProjectionContextMemoryInfo\* projectionContextMemoryInfo)

{

Assert(this->projectionMemoryInformation == nullptr || this->projectionMemoryInformation == projectionContextMemoryInfo);

this->projectionMemoryInformation = projectionContextMemoryInfo;

}

void ThreadContext::DumpProjectionContextMemoryStats(LPCWSTR headerMsg, bool forceDetailed)

{

if (this->projectionMemoryInformation)

{

this->projectionMemoryInformation->DumpCurrentStats(headerMsg, forceDetailed);

}

}

IProjectionContextMemoryInfo\* ThreadContext::GetProjectionContextMemoryInformation()

{

return this->projectionMemoryInformation;

}

#endif

#endif

#ifdef ENABLE\_DEBUG\_CONFIG\_OPTIONS

Js::Var ThreadContext::GetMemoryStat(Js::ScriptContext\* scriptContext)

{

ScriptMemoryDumper dumper(scriptContext);

return dumper.Dump();

}

void ThreadContext::SetAutoProxyName(LPCWSTR objectName)

{

recyclableData->autoProxyName = objectName;

}

#endif

//

// Regex helpers

//

UnifiedRegex::StandardChars<uint8>\* ThreadContext::GetStandardChars(\_\_inout\_opt uint8\* dummy)

{

if (standardUTF8Chars == 0)

{

ArenaAllocator\* allocator = GetThreadAlloc();

standardUTF8Chars = Anew(allocator, UnifiedRegex::UTF8StandardChars, allocator);

}

return standardUTF8Chars;

}

UnifiedRegex::StandardChars<wchar\_t>\* ThreadContext::GetStandardChars(\_\_inout\_opt wchar\_t\* dummy)

{

if (standardUnicodeChars == 0)

{

ArenaAllocator\* allocator = GetThreadAlloc();

standardUnicodeChars = Anew(allocator, UnifiedRegex::UnicodeStandardChars, allocator);

}

return standardUnicodeChars;

}

void ThreadContext::CheckScriptInterrupt()

{

if (TestThreadContextFlag(ThreadContextFlagCanDisableExecution))

{

if (this->IsExecutionDisabled())

{

Assert(this->DoInterruptProbe());

throw Js::ScriptAbortException();

}

}

else

{

this->CheckInterruptPoll();

}

}

void ThreadContext::CheckInterruptPoll()

{

// Disable QC when implicit calls are disabled since the host can do anything before returning back, like servicing the

// message loop, which may in turn cause script code to be executed and implicit calls to be made

if (!IsDisableImplicitCall())

{

InterruptPoller \*poller = this->interruptPoller;

if (poller)

{

poller->CheckInterruptPoll();

}

}

}

void \*

ThreadContext::GetDynamicObjectEnumeratorCache(Js::DynamicType const \* dynamicType)

{

void \* data;

return this->dynamicObjectEnumeratorCacheMap.TryGetValue(dynamicType, &data)? data : nullptr;

}

void

ThreadContext::AddDynamicObjectEnumeratorCache(Js::DynamicType const \* dynamicType, void \* cache)

{

this->dynamicObjectEnumeratorCacheMap.Item(dynamicType, cache);

}

InterruptPoller::InterruptPoller(ThreadContext \*tc) :

threadContext(tc),

lastPollTick(0),

lastResetTick(0),

isDisabled(FALSE)

{

tc->SetInterruptPoller(this);

}

void InterruptPoller::CheckInterruptPoll()

{

if (!isDisabled)

{

Js::ScriptEntryExitRecord \*entryExitRecord = this->threadContext->GetScriptEntryExit();

if (entryExitRecord)

{

Js::ScriptContext \*scriptContext = entryExitRecord->scriptContext;

if (scriptContext)

{

this->TryInterruptPoll(scriptContext);

}

}

}

}

void InterruptPoller::GetStatementCount(ULONG \*pluHi, ULONG \*pluLo)

{

DWORD resetTick = this->lastResetTick;

DWORD pollTick = this->lastPollTick;

DWORD elapsed;

elapsed = pollTick - resetTick;

ULONGLONG statements = (ULONGLONG)elapsed \* InterruptPoller::TicksToStatements;

\*pluLo = (ULONG)statements;

\*pluHi = (ULONG)(statements >> 32);

}

void ThreadContext::DisableExecution()

{

Assert(TestThreadContextFlag(ThreadContextFlagCanDisableExecution));

// Hammer the stack limit with a value that will cause script abort on the next stack probe.

this->SetStackLimitForCurrentThread(Js::Constants::StackLimitForScriptInterrupt);

return;

}

void ThreadContext::EnableExecution()

{

Assert(this->GetStackProber());

// Restore the normal stack limit.

this->SetStackLimitForCurrentThread(this->GetStackProber()->GetScriptStackLimit());

// It's possible that the host disabled execution after the script threw an exception

// of it's own, so we shouldn't clear that. Only exceptions for script termination

// should be cleared.

if (GetRecordedException() == GetPendingTerminatedErrorObject())

{

SetRecordedException(NULL);

}

}

bool ThreadContext::TestThreadContextFlag(ThreadContextFlags contextFlag) const

{

return (this->threadContextFlags & contextFlag) != 0;

}

void ThreadContext::SetThreadContextFlag(ThreadContextFlags contextFlag)

{

this->threadContextFlags = (ThreadContextFlags)(this->threadContextFlags | contextFlag);

}

void ThreadContext::ClearThreadContextFlag(ThreadContextFlags contextFlag)

{

this->threadContextFlags = (ThreadContextFlags)(this->threadContextFlags & ~contextFlag);

}

#ifdef \_CONTROL\_FLOW\_GUARD

Js::DelayLoadWinCoreMemory \* ThreadContext::GetWinCoreMemoryLibrary()

{

delayLoadWinCoreMemoryLibrary.EnsureFromSystemDirOnly();

return &delayLoadWinCoreMemoryLibrary;

}

#endif

Js::DelayLoadWinCoreProcessThreads \* ThreadContext::GetWinCoreProcessThreads()

{

delayLoadWinCoreProcessThreads.EnsureFromSystemDirOnly();

return &delayLoadWinCoreProcessThreads;

}

Js::DelayLoadWinRtString \* ThreadContext::GetWinRTStringLibrary()

{

delayLoadWinRtString.EnsureFromSystemDirOnly();

return &delayLoadWinRtString;

}

#ifdef ENABLE\_PROJECTION

Js::DelayLoadWinRtError \* ThreadContext::GetWinRTErrorLibrary()

{

delayLoadWinRtError.EnsureFromSystemDirOnly();

return &delayLoadWinRtError;

}

Js::DelayLoadWinRtTypeResolution\* ThreadContext::GetWinRTTypeResolutionLibrary()

{

delayLoadWinRtTypeResolution.EnsureFromSystemDirOnly();

return &delayLoadWinRtTypeResolution;

}

Js::DelayLoadWinRtRoParameterizedIID\* ThreadContext::GetWinRTRoParameterizedIIDLibrary()

{

delayLoadWinRtRoParameterizedIID.EnsureFromSystemDirOnly();

return &delayLoadWinRtRoParameterizedIID;

}

#endif

#if defined(ENABLE\_INTL\_OBJECT) || defined(ENABLE\_ES6\_CHAR\_CLASSIFIER)

Js::WindowsGlobalizationAdapter\* ThreadContext::GetWindowsGlobalizationAdapter()

{

return &windowsGlobalizationAdapter;

}

Js::DelayLoadWindowsGlobalization\* ThreadContext::GetWindowsGlobalizationLibrary()

{

delayLoadWindowsGlobalizationLibrary.Ensure(this->GetWinRTStringLibrary());

return &delayLoadWindowsGlobalizationLibrary;

}

#endif

#ifdef ENABLE\_FOUNDATION\_OBJECT

Js::WindowsFoundationAdapter\* ThreadContext::GetWindowsFoundationAdapter()

{

return &windowsFoundationAdapter;

}

Js::DelayLoadWinRtFoundation\* ThreadContext::GetWinRtFoundationLibrary()

{

delayLoadWinRtFoundationLibrary.EnsureFromSystemDirOnly();

return &delayLoadWinRtFoundationLibrary;

}

#endif

bool ThreadContext::IsCFGEnabled()

{

#if defined(\_CONTROL\_FLOW\_GUARD)

PROCESS\_MITIGATION\_CONTROL\_FLOW\_GUARD\_POLICY CfgPolicy;

BOOL isGetMitigationPolicySucceeded = GetWinCoreProcessThreads()->GetMitigationPolicyForProcess(

GetCurrentProcess(),

ProcessControlFlowGuardPolicy,

&CfgPolicy,

sizeof(CfgPolicy));

Assert(isGetMitigationPolicySucceeded || !AutoSystemInfo::Data.IsCFGEnabled());

return CfgPolicy.EnableControlFlowGuard && AutoSystemInfo::Data.IsCFGEnabled();

#else

return false;

#endif

}

//Masking bits according to AutoSystemInfo::PageSize

#define PAGE\_START\_ADDR(address) ((size\_t)(address) & ~(size\_t)(AutoSystemInfo::PageSize - 1))

#define IS\_16BYTE\_ALIGNED(address) (((size\_t)(address) & 0xF) == 0)

#define OFFSET\_ADDR\_WITHIN\_PAGE(address) ((size\_t)(address) & (AutoSystemInfo::PageSize - 1))

void ThreadContext::SetValidCallTargetForCFG(PVOID callTargetAddress, bool isSetValid)

{

#ifdef \_CONTROL\_FLOW\_GUARD

if (IsCFGEnabled())

{

AssertMsg(IS\_16BYTE\_ALIGNED(callTargetAddress), "callTargetAddress is not 16-byte page aligned?");

PVOID startAddressOfPage = (PVOID) (PAGE\_START\_ADDR(callTargetAddress));

size\_t codeOffset = OFFSET\_ADDR\_WITHIN\_PAGE(callTargetAddress);

CFG\_CALL\_TARGET\_INFO callTargetInfo[1];

callTargetInfo[0].Offset = codeOffset;

callTargetInfo[0].Flags = (isSetValid? CFG\_CALL\_TARGET\_VALID : 0);

AssertMsg((size\_t)callTargetAddress - (size\_t)startAddressOfPage <= AutoSystemInfo::PageSize - 1, "Only last bits corresponding to PageSize should be masked");

AssertMsg((size\_t)startAddressOfPage + (size\_t)codeOffset == (size\_t)callTargetAddress, "Wrong masking of address?");

BOOL isCallTargetRegistrationSucceed = GetWinCoreMemoryLibrary()->SetProcessCallTargets(GetCurrentProcess(), startAddressOfPage, AutoSystemInfo::PageSize, 1, callTargetInfo);

if (!isCallTargetRegistrationSucceed)

{

if (GetLastError() == ERROR\_COMMITMENT\_LIMIT)

{

//Throw OOM, if there is not enough virtual memory for paging (required for CFG BitMap)

Js::Throw::OutOfMemory();

}

else

{

Js::Throw::InternalError();

}

}

#if DBG

if (isSetValid)

{

\_guard\_check\_icall((uintptr\_t) callTargetAddress);

}

if (PHASE\_TRACE1(Js::CFGPhase))

{

if (!isSetValid)

{

Output::Print(L"DEREGISTER:");

}

Output::Print(L"CFGRegistration: StartAddr: 0x%p , Offset: 0x%x, TargetAddr: 0x%x \n", (char\*) startAddressOfPage, callTargetInfo[0].Offset, ((size\_t) startAddressOfPage + (size\_t) callTargetInfo[0].Offset));

Output::Flush();

}

#endif

}

#endif // \_CONTROL\_FLOW\_GUARD

}

// Despite the name, callers expect this to return the highest propid + 1.

uint ThreadContext::GetHighestPropertyNameIndex() const

{

return propertyMap->GetLastIndex() + 1 + Js::InternalPropertyIds::Count;

}

#if defined(CHECK\_MEMORY\_LEAK) || defined(LEAK\_REPORT)

void ThreadContext::ReportAndCheckLeaksOnProcessDetach()

{

bool needReportOrCheck = false;

#ifdef LEAK\_REPORT

needReportOrCheck = needReportOrCheck || Js::Configuration::Global.flags.IsEnabled(Js::LeakReportFlag);

#endif

#ifdef CHECK\_MEMORY\_LEAK

needReportOrCheck = needReportOrCheck ||

(Js::Configuration::Global.flags.CheckMemoryLeak && MemoryLeakCheck::IsEnableOutput());

#endif

if (!needReportOrCheck)

{

return;

}

// Report leaks even if this is a force termination and we have not clean up the thread

// This is call during process detach. No one should be creating new thread context.

// So don't need to take the lock

ThreadContext \* current = GetThreadContextList();

while (current)

{

#if DBG

current->pageAllocator.ClearConcurrentThreadId();

#endif

Recycler \* recycler = current->GetRecycler();

#ifdef LEAK\_REPORT

if (Js::Configuration::Global.flags.IsEnabled(Js::LeakReportFlag))

{

AUTO\_LEAK\_REPORT\_SECTION(Js::Configuration::Global.flags, L"Thread Context (%p): Process Termination (TID: %d)", current, current->threadId);

LeakReport::DumpUrl(current->threadId);

// Heuristically figure out which one is the root tracker script engine

// and force close on it

if (current->rootTrackerScriptContext != nullptr)

{

current->rootTrackerScriptContext->Close(false);

}

recycler->ReportLeaksOnProcessDetach();

}

#endif

#ifdef CHECK\_MEMORY\_LEAK

recycler->CheckLeaksOnProcessDetach(L"Process Termination");

#endif

current = current->Next();

}

}

#endif

#ifdef LEAK\_REPORT

void

ThreadContext::SetRootTrackerScriptContext(Js::ScriptContext \* scriptContext)

{

Assert(this->rootTrackerScriptContext == nullptr);

this->rootTrackerScriptContext = scriptContext;

scriptContext->isRootTrackerScriptContext = true;

}

void

ThreadContext::ClearRootTrackerScriptContext(Js::ScriptContext \* scriptContext)

{

Assert(this->rootTrackerScriptContext == scriptContext);

this->rootTrackerScriptContext->isRootTrackerScriptContext = false;

this->rootTrackerScriptContext = nullptr;

}

#endif

JITTimer::JITTimer()

{

Reset();

}

double JITTimer::Now()

{

return timer.Now();

}

JITStats JITTimer::GetStats()

{

return stats;

}

void JITTimer::Reset()

{

stats = { 0 };

}

void JITTimer::LogTime(double ms)

{

if (ms <= 5.0)

{

stats.lessThan5ms++;

}

else if (ms <= 10.0)

{

stats.within5And10ms++;

}

else if (ms <= 20.0)

{

stats.within10And20ms++;

}

else if (ms <= 50.0)

{

stats.within20And50ms++;

}

else if (ms <= 100.0)

{

stats.within50And100ms++;

}

else if (ms <= 300.0)

{

stats.within100And300ms++;

}

else

{

stats.greaterThan300ms++;

}

}

ParserTimer::ParserTimer()

{

Reset();

}

double ParserTimer::Now()

{

return timer.Now();

}

ParserStats ParserTimer::GetStats()

{

return stats;

}

void ParserTimer::Reset()

{

stats = { 0 };

}

void ParserTimer::LogTime(double ms)

{

if (ms <= 1.0)

{

stats.lessThan1ms++;

}

else if (ms <= 3.0)

{

stats.within1And3ms++;

}

else if (ms <= 10.0)

{

stats.within3And10ms++;

}

else if (ms <= 20.0)

{

stats.within10And20ms++;

}

else if (ms <= 50.0)

{

stats.within20And50ms++;

}

else if (ms <= 100.0)

{

stats.within50And100ms++;

}

else if (ms <= 300.0)

{

stats.within100And300ms++;

}

else

{

stats.greaterThan300ms++;

}

}

AutoTagNativeLibraryEntry::AutoTagNativeLibraryEntry(Js::RecyclableObject\* function, Js::CallInfo callInfo, PCWSTR name, void\* addr)

{

// Save function/callInfo values (for StackWalker). Compiler may stackpack/optimize them for built-in native functions.

entry.function = function;

entry.callInfo = callInfo;

entry.name = name;

entry.addr = addr;

ThreadContext\* threadContext = function->GetScriptContext()->GetThreadContext();

threadContext->PushNativeLibraryEntry(&entry);

}

AutoTagNativeLibraryEntry::~AutoTagNativeLibraryEntry()

{

ThreadContext\* threadContext = entry.function->GetScriptContext()->GetThreadContext();

Assert(threadContext->PeekNativeLibraryEntry() == &entry);

threadContext->PopNativeLibraryEntry();

}

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

// 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 ScriptContext;

struct InlineCache;

class DebugManager;

class CodeGenRecyclableData;

struct ReturnedValue;

typedef JsUtil::List<ReturnedValue\*> ReturnedValueList;

}

struct IAuthorFileContext;

class HostScriptContext;

class ScriptSite;

class ThreadServiceWrapper;

struct IActiveScriptProfilerHeapEnum;

class DynamicProfileMutator;

class StackProber;

enum DisableImplicitFlags : BYTE

{

DisableImplicitNoFlag = 0x00,

DisableImplicitCallFlag = 0x01,

DisableImplicitExceptionFlag = 0x02,

DisableImplicitCallAndExceptionFlag = DisableImplicitCallFlag | DisableImplicitExceptionFlag

};

enum ThreadContextFlags

{

ThreadContextFlagNoFlag = 0x00000000,

ThreadContextFlagCanDisableExecution = 0x00000001,

ThreadContextFlagEvalDisabled = 0x00000002,

ThreadContextFlagNoJIT = 0x00000004,

};

const int LS\_MAX\_STACK\_SIZE\_KB = 300;

struct IProjectionContext

{

public:

virtual HRESULT Close() = 0;

};

class ThreadContext;

class InterruptPoller abstract

{

// Interface with a polling object located in the hosting layer.

public:

InterruptPoller(ThreadContext \*tc);

void CheckInterruptPoll();

void GetStatementCount(ULONG \*pluHi, ULONG \*pluLo);

void ResetStatementCount() { lastResetTick = lastPollTick; }

void StartScript() { lastResetTick = lastPollTick = ::GetTickCount(); }

void EndScript() { lastResetTick = lastPollTick = 0;}

bool IsDisabled() const { return isDisabled; }

void SetDisabled(bool disable) { isDisabled = disable; }

virtual void TryInterruptPoll(Js::ScriptContext \*scriptContext) = 0;

// Default: throw up QC dialog after 5M statements == 2 minutes

static const DWORD TicksToStatements = (5000000 / 120000);

protected:

ThreadContext \*threadContext;

DWORD lastPollTick;

DWORD lastResetTick;

bool isDisabled;

};

class AutoDisableInterrupt

{

private:

InterruptPoller\* interruptPoller;

bool previousState;

public:

AutoDisableInterrupt(InterruptPoller\* interruptPoller, bool disable)

: interruptPoller(interruptPoller)

{

if (interruptPoller != nullptr)

{

previousState = interruptPoller->IsDisabled();

interruptPoller->SetDisabled(disable);

}

}

~AutoDisableInterrupt()

{

if (interruptPoller != nullptr)

{

interruptPoller->SetDisabled(previousState);

}

}

};

// This function is called before we step out of script (currently only for WinRT callout).

// Debugger would put a breakpoint on this function if they want to detect the point at which we step

// over the boundary.

// It is intentionally left blank and the next operation should be the callout.

extern "C" void\* MarkerForExternalDebugStep();

#define PROBE\_STACK(scriptContext, size) ((scriptContext)->GetThreadContext()->ProbeStack(size, scriptContext))

#define PROBE\_STACK\_NO\_DISPOSE(scriptContext, size) ((scriptContext)->GetThreadContext()->ProbeStackNoDispose(size, scriptContext))

#define PROBE\_STACK\_PARTIAL\_INITIALIZED\_INTERPRETER\_FRAME(scriptContext, size) ((scriptContext)->GetThreadContext()->ProbeStack(size, scriptContext, \_ReturnAddress()))

#define PROBE\_STACK\_PARTIAL\_INITIALIZED\_BAILOUT\_FRAME(scriptContext, size, returnAddress) ((scriptContext)->GetThreadContext()->ProbeStack(size, scriptContext, returnAddress))

#define PROBE\_STACK\_CALL(scriptContext, obj, size) ((scriptContext)->GetThreadContext()->ProbeStack(size, obj, scriptContext))

#define AssertInScript() Assert(ThreadContext::GetContextForCurrentThread()->IsScriptActive());

#define AssertNotInScript() Assert(!ThreadContext::GetContextForCurrentThread()->IsScriptActive());

#define LEAVE\_SCRIPT\_START\_EX(scriptContext, stackProbe, leaveForHost, isFPUControlRestoreNeeded) \

{ \

void \* \_\_frameAddr = nullptr; \

GET\_CURRENT\_FRAME\_ID(\_\_frameAddr); \

Js::LeaveScriptObject<stackProbe, leaveForHost, isFPUControlRestoreNeeded> \_\_leaveScriptObject(scriptContext, \_\_frameAddr);

#define LEAVE\_SCRIPT\_END\_EX(scriptContext) \

if (scriptContext != nullptr) \

{ \

scriptContext->GetThreadContext()->DisposeOnLeaveScript(); \

}\

}

#define BEGIN\_LEAVE\_SCRIPT(scriptContext) \

LEAVE\_SCRIPT\_START\_EX(scriptContext, /\* stackProbe \*/ true, /\* leaveForHost \*/ true, /\* isFPUControlRestoreNeeded \*/ false)

#define BEGIN\_LEAVE\_SCRIPT\_SAVE\_FPU\_CONTROL(scriptContext) \

LEAVE\_SCRIPT\_START\_EX(scriptContext, /\* stackProbe \*/ true, /\* leaveForHost \*/ true, /\* isFPUControlRestoreNeeded \*/ true)

// BEGIN\_LEAVE\_SCRIPT\_INTERNAL is used when there are no explicit external call after leave script,

// but we might have external call when allocation memory doing QC or GC Dispose, which may enter script again.

// This will record the reentry as an implicit call (ImplicitCall\_AsyncHostOperation)

#define BEGIN\_LEAVE\_SCRIPT\_INTERNAL(scriptContext) \

LEAVE\_SCRIPT\_START\_EX(scriptContext, /\* stackProbe \*/ true, /\* leaveForHost \*/ false, /\* isFPUControlRestoreNeeded \*/ false)

#define BEGIN\_LEAVE\_SCRIPT\_NO\_STACK\_PROBE(scriptContext) \

LEAVE\_SCRIPT\_START\_EX(scriptContext, /\* stackProbe \*/ false, /\* leaveForHost \*/ true, /\* isFPUControlRestoreNeeded \*/ false)

#define END\_LEAVE\_SCRIPT(scriptContext) \

LEAVE\_SCRIPT\_END\_EX(scriptContext)

#define END\_LEAVE\_SCRIPT\_RESTORE\_FPU\_CONTROL(scriptContext) \

LEAVE\_SCRIPT\_END\_EX(scriptContext)

#define END\_LEAVE\_SCRIPT\_INTERNAL(scriptContext) \

LEAVE\_SCRIPT\_END\_EX(scriptContext)

#define END\_LEAVE\_SCRIPT\_NO\_STACK\_PROBE(scriptContext) \

LEAVE\_SCRIPT\_END\_EX(scriptContext)

#define BEGIN\_LEAVE\_SCRIPT\_WITH\_EXCEPTION(scriptContext) \

BEGIN\_LEAVE\_SCRIPT(scriptContext)

#define END\_LEAVE\_SCRIPT\_WITH\_EXCEPTION(scriptContext) \

Assert(!scriptContext->HasRecordedException()); \

END\_LEAVE\_SCRIPT(scriptContext)

// Keep in sync with CollectGarbageCallBackFlags in scriptdirect.idl

enum RecyclerCollectCallBackFlags

{

Collect\_Begin = 0x01,

Collect\_Begin\_Concurrent = 0x11,

Collect\_Begin\_Partial = 0x21,

Collect\_Begin\_Concurrent\_Partial = Collect\_Begin\_Concurrent | Collect\_Begin\_Partial,

Collect\_End = 0x02,

Collect\_Wait = 0x04 // callback can be from another thread

};

typedef void (\_\_cdecl \*RecyclerCollectCallBackFunction)(void \* context, RecyclerCollectCallBackFlags flags);

// Keep in sync with WellKnownType in scriptdirect.idl

typedef enum WellKnownHostType

{

WellKnownHostType\_HTMLAllCollection = 0,

WellKnownHostType\_Last = WellKnownHostType\_HTMLAllCollection,

WellKnownHostType\_Invalid = WellKnownHostType\_Last+1

} WellKnownHostType;

#ifdef ENABLE\_PROJECTION

class ExternalWeakReferenceCache

{

public:

virtual void MarkNow(Recycler \*recycler, bool inPartialCollect) = 0;

virtual void ResolveNow(Recycler \*recycler) = 0;

};

#if DBG\_DUMP

class IProjectionContextMemoryInfo abstract

{

public:

virtual void DumpCurrentStats(LPCWSTR headerMsg, bool forceDetailed) = 0;

virtual void Release() = 0;

};

#endif

#endif

struct ThreadContextWatsonTelemetryBlock

{

FILETIME lastScriptStartTime;

FILETIME lastScriptEndTime;

};

class NativeLibraryEntryRecord

{

public:

struct Entry

{

Js::RecyclableObject\* function;

Js::CallInfo callInfo;

PCWSTR name;

PVOID addr;

Entry\* next;

};

private:

Entry\* head;

public:

NativeLibraryEntryRecord() : head(nullptr)

{

}

const Entry\* Peek() const

{

return head;

}

void Push(\_In\_ Entry\* e)

{

e->next = head;

head = e;

}

void Pop()

{

head = head->next;

}

};

class AutoTagNativeLibraryEntry

{

private:

NativeLibraryEntryRecord::Entry entry;

public:

AutoTagNativeLibraryEntry(Js::RecyclableObject\* function, Js::CallInfo callInfo, PCWSTR name, void\* addr);

~AutoTagNativeLibraryEntry();

};

struct JITStats

{

uint lessThan5ms;

uint within5And10ms;

uint within10And20ms;

uint within20And50ms;

uint within50And100ms;

uint within100And300ms;

uint greaterThan300ms;

};

struct ParserStats

{

uint64 lessThan1ms;

uint64 within1And3ms;

uint64 within3And10ms;

uint64 within10And20ms;

uint64 within20And50ms;

uint64 within50And100ms;

uint64 within100And300ms;

uint64 greaterThan300ms;

};

class ParserTimer

{

private:

Js::HiResTimer timer;

ParserStats stats;

public:

ParserTimer();

ParserStats GetStats();

void Reset();

double Now();

void LogTime(double ms);

};

class JITTimer

{

private:

Js::HiResTimer timer;

JITStats stats;

public:

JITTimer();

JITStats GetStats();

void Reset();

double Now();

void LogTime(double ms);

};

#define AUTO\_TAG\_NATIVE\_LIBRARY\_ENTRY(function, callInfo, name) \

AutoTagNativeLibraryEntry \_\_tag(function, callInfo, name, \_AddressOfReturnAddress())

class ThreadConfiguration

{

public:

ThreadConfiguration(bool enableExperimentalFeatures)

{

CopyGlobalFlags();

if (enableExperimentalFeatures)

{

EnableExperimentalFeatures();

}

}

#define DEFINE\_FLAG(threadFlag, globalFlag) \

public: \

inline bool threadFlag() const { return m\_##globalFlag##; } \

\

private: \

bool m\_##globalFlag##;

#define FLAG(threadFlag, globalFlag) DEFINE\_FLAG(threadFlag, globalFlag)

#define FLAG\_RELEASE(threadFlag, globalFlag) DEFINE\_FLAG(threadFlag, globalFlag)

#include "ThreadConfigFlagsList.h"

#undef FLAG\_RELEASE

#undef FLAG

#undef DEFINE\_FLAG

private:

void CopyGlobalFlags()

{

AutoCriticalSection autocs(&Js::Configuration::Global.flags.csExperimentalFlags);

#define FLAG(threadFlag, globalFlag) m\_##globalFlag## = CONFIG\_FLAG(globalFlag);

#define FLAG\_RELEASE(threadFlag, globalFlag) m\_##globalFlag## = CONFIG\_FLAG\_RELEASE(globalFlag);

#include "ThreadConfigFlagsList.h"

#undef FLAG\_RELEASE

#undef FLAG

}

void EnableExperimentalFeatures()

{

#define FLAG\_REGOVR\_EXP(type, name, ...) m\_##name## = true;

#include "ConfigFlagsList.h"

#undef FLAG\_REGOVR\_EXP

}

};

class ThreadContext sealed :

public DefaultRecyclerCollectionWrapper,

public JsUtil::DoublyLinkedListElement<ThreadContext>

{

public:

static void GlobalInitialize();

static const DWORD NoThread = 0xFFFFFFFF;

struct CollectCallBack

{

RecyclerCollectCallBackFunction callback;

void \* context;

};

struct WorkerThread

{

// Abstract notion to hold onto threadHandle of worker thread

HANDLE threadHandle;

WorkerThread(HANDLE handle = nullptr) :threadHandle(handle){};

};

void ReleasePreReservedSegment();

void IncrementThreadContextsWithPreReservedSegment();

void SetCurrentThreadId(DWORD threadId) { this->currentThreadId = threadId; }

DWORD GetCurrentThreadId() const { return this->currentThreadId; }

void SetIsThreadBound()

{

if (this->recycler)

{

this->recycler->SetIsThreadBound();

}

this->isThreadBound = true;

}

bool GetIsThreadBound() const { return this->isThreadBound; }

void SetStackProber(StackProber \* stackProber);

PBYTE GetScriptStackLimit() const;

static DWORD GetStackLimitForCurrentThreadOffset() { return offsetof(ThreadContext, stackLimitForCurrentThread); }

void \* GetAddressOfStackLimitForCurrentThread()

{

FAULTINJECT\_SCRIPT\_TERMINATION

return &this->stackLimitForCurrentThread;

}

void InitAvailableCommit();

// This is always on for JSRT APIs.

bool IsRentalThreadingEnabledInJSRT() const { return true; }

IActiveScriptProfilerHeapEnum\* GetHeapEnum();

void SetHeapEnum(IActiveScriptProfilerHeapEnum\* newHeapEnum);

void ClearHeapEnum();

#ifdef ENABLE\_BASIC\_TELEMETRY

Js::LanguageStats\* GetLanguageStats()

{

return langTel.GetLanguageStats();

}

void ResetLangStats()

{

this->langTel.Reset();

}

#endif

bool CanPreReserveSegmentForCustomHeap();

#if ENABLE\_NATIVE\_CODEGEN

// used by inliner. Maps Simd FuncInfo (library func) to equivalent opcode.

typedef JsUtil::BaseDictionary<Js::FunctionInfo \*, Js::OpCode, ArenaAllocator> FuncInfoToOpcodeMap;

FuncInfoToOpcodeMap \* simdFuncInfoToOpcodeMap;

struct SimdFuncSignature

{

bool valid;

uint argCount; // actual arguments count (excluding this)

ValueType returnType;

ValueType \*args; // argument types

};

SimdFuncSignature \*simdOpcodeToSignatureMap;

void AddSimdFuncToMaps(Js::OpCode op, ...);

Js::OpCode GetSimdOpcodeFromFuncInfo(Js::FunctionInfo \* funcInfo);

void GetSimdFuncSignatureFromOpcode(Js::OpCode op, SimdFuncSignature &funcSignature);

#endif

private:

bool noScriptScope;

Js::DebugManager \* debugManager;

static uint const MaxTemporaryArenaAllocators = 5;

static CriticalSection s\_csThreadContext;

StackProber \* GetStackProber() const { return this->stackProber; }

PBYTE GetStackLimitForCurrentThread() const;

void SetStackLimitForCurrentThread(PBYTE limit);

#if !\_M\_X64\_OR\_ARM64 && \_CONTROL\_FLOW\_GUARD

static uint numOfThreadContextsWithPreReserveSegment;

#endif

// The current heap enumeration object being used during enumeration.

IActiveScriptProfilerHeapEnum\* heapEnum;

#ifdef ENABLE\_BASIC\_TELEMETRY

Js::LanguageTelemetry langTel;

#endif

struct PropertyGuardEntry

{

public:

typedef JsUtil::BaseHashSet<RecyclerWeakReference<Js::PropertyGuard>\*, Recycler, PowerOf2SizePolicy> PropertyGuardHashSet;

// we do not have WeaklyReferencedKeyHashSet - hence use BYTE as a dummy value.

typedef JsUtil::WeaklyReferencedKeyDictionary<Js::EntryPointInfo, BYTE> EntryPointDictionary;

// The sharedGuard is strongly referenced and will be kept alive by ThreadContext::propertyGuards until it's invalidated or

// the property record itself is collected. If the code using the guard needs access to it after it's been invalidated, it

// (the code) is responsible for keeping it alive. Each unique guard, is weakly referenced, such that it can be reclaimed

// if not referenced elsewhere even without being invalidated. It's up to the owner of that guard to keep it alive as long

// as necessary.

Js::PropertyGuard\* sharedGuard;

PropertyGuardHashSet uniqueGuards;

EntryPointDictionary\* entryPoints;

PropertyGuardEntry(Recycler\* recycler) : sharedGuard(nullptr), uniqueGuards(recycler), entryPoints(nullptr) {}

};

public:

typedef JsUtil::BaseHashSet<const Js::PropertyRecord \*, HeapAllocator, PrimeSizePolicy, const Js::PropertyRecord \*,

Js::PropertyRecordStringHashComparer, JsUtil::SimpleHashedEntry, JsUtil::AsymetricResizeLock> PropertyMap;

typedef JsUtil::BaseHashSet<Js::CaseInvariantPropertyListWithHashCode\*, Recycler, PowerOf2SizePolicy, Js::CaseInvariantPropertyListWithHashCode\*, JsUtil::NoCaseComparer, JsUtil::SimpleDictionaryEntry>

PropertyNoCaseSetType;

typedef JsUtil::WeaklyReferencedKeyDictionary<Js::Type, bool> TypeHashSet;

typedef JsUtil::BaseDictionary<Js::PropertyId, TypeHashSet \*, Recycler, PowerOf2SizePolicy> PropertyIdToTypeHashSetDictionary;

typedef JsUtil::WeaklyReferencedKeyDictionary<const Js::PropertyRecord, PropertyGuardEntry\*, Js::PropertyRecordPointerComparer> PropertyGuardDictionary;

private:

typedef JsUtil::BaseDictionary<uint, Js::SourceDynamicProfileManager\*, Recycler, PowerOf2SizePolicy> SourceDynamicProfileManagerMap;

typedef JsUtil::BaseDictionary<const wchar\_t\*, const Js::PropertyRecord\*, Recycler, PowerOf2SizePolicy> SymbolRegistrationMap;

class SourceDynamicProfileManagerCache

{

public:

SourceDynamicProfileManagerCache() : refCount(0), sourceProfileManagerMap(nullptr) {}

SourceDynamicProfileManagerMap\* sourceProfileManagerMap;

void AddRef() { refCount++; }

uint Release() { Assert(refCount > 0); return --refCount; }

private:

uint refCount; // For every script context using this cache, there is a ref count added.

};

typedef JsUtil::BaseDictionary<const WCHAR\*, SourceDynamicProfileManagerCache\*, Recycler, PowerOf2SizePolicy> SourceProfileManagersByUrlMap;

struct RecyclableData

{

RecyclableData(Recycler \*const recycler);

Js::TempArenaAllocatorObject \* temporaryArenaAllocators[MaxTemporaryArenaAllocators];

Js::TempGuestArenaAllocatorObject \* temporaryGuestArenaAllocators[MaxTemporaryArenaAllocators];

Js::JavascriptExceptionObject \* exceptionObject;

bool propagateException;

// We throw a JS catchable SO exception if we detect we might overflow the stack. Allocating this (JS)

// object though might really overflow the stack. So use this thread global to identify them from the throw point

// to where they are caught; where the stack has been unwound and it is safer to allocate the real exception

// object and throw.

Js::JavascriptExceptionObject soErrorObject;

// We can't allocate an out of memory object... So use this static as a way to identify

// them from the throw point to where they are caught.

Js::JavascriptExceptionObject oomErrorObject;

// This is for JsRT scenario where a runtime is not usable after a suspend request, before a resume runtime call is made

Js::JavascriptExceptionObject terminatedErrorObject;

Js::JavascriptExceptionObject\* unhandledExceptionObject;

// Contains types that have property caches that need to be tracked, as the caches may need to be cleared. Types that

// contain a property cache for a property that is on a prototype object will be tracked in this map since those caches

// need to be cleared if for instance, the property is deleted from the prototype object.

//

// It is expected that over time, types that are deleted will eventually be removed by the weak reference hash sets when

// they're searching through a bucket while registering a type or enumerating types to invalidate, or when a property ID

// is reclaimed. If none of those happen, then this collection may contain weak reference handles to deleted objects

// that would not get removed, but it would also not get any bigger.

PropertyIdToTypeHashSetDictionary typesWithProtoPropertyCache;

// The property guard dictionary contains property guards which need to be invalidated in response to properties changing

// from writable to read-only and vice versa, properties being shadowed or unshadowed on prototypes, etc. The dictionary

// holds only weak references to property guards and their lifetimes are controlled by their creators (typically entry points).

// When a guard is no longer needed it is garbage collected, but the weak references and dictionary entries remain, until

// the guards for a given property get invalidated.

// TODO: Create and use a self-cleaning weak reference dictionary, which would periodically remove any unused weak references.

PropertyGuardDictionary propertyGuards;

PropertyNoCaseSetType \* caseInvariantPropertySet;

JsUtil::List<Js::PropertyRecord const\*>\* boundPropertyStrings; // Recycler allocated list of property strings that we need to strongly reference so that they're not reclaimed

SourceProfileManagersByUrlMap\* sourceProfileManagersByUrl;

// Used to register recyclable data that needs to be kept alive while jitting

JsUtil::DoublyLinkedList<Js::CodeGenRecyclableData> codeGenRecyclableDatas;

// Used to root old entry points so that they're not prematurely collected

Js::FunctionEntryPointInfo\* oldEntryPointInfo;

// Used to store a mapping of string to Symbol for cross-realm Symbol registration

// See ES6 (draft 22) 19.4.2.2

SymbolRegistrationMap\* symbolRegistrationMap;

// Just holding the reference to the returnedValueList of the stepController. This way that list will not get recycled prematurely.

Js::ReturnedValueList \*returnedValueList;

#ifdef ENABLE\_DEBUG\_CONFIG\_OPTIONS

// use for autoProxy called from Debug.setAutoProxyName. we need to keep the buffer from GetSz() alive.

LPCWSTR autoProxyName;

#endif

};

static ThreadContext \* globalListLast;

ThreadContextFlags threadContextFlags;

DWORD currentThreadId;

mutable PBYTE stackLimitForCurrentThread;

StackProber \* stackProber;

bool isThreadBound;

bool hasThrownPendingException;

bool callDispose;

bool isAllJITCodeInPreReservedRegion;

AllocationPolicyManager \* allocationPolicyManager;

JsUtil::ThreadService threadService;

PreReservedVirtualAllocWrapper preReservedVirtualAllocator;

uint callRootLevel;

// The thread page allocator is used by the recycler and need the background page queue

PageAllocator::BackgroundPageQueue backgroundPageQueue;

IdleDecommitPageAllocator pageAllocator;

Recycler\* recycler;

// Fake RecyclerWeakReference for built-in properties

class StaticPropertyRecordReference : public RecyclerWeakReference<const Js::PropertyRecord>

{

public:

StaticPropertyRecordReference(const Js::PropertyRecord\* propertyRecord)

{

strongRef = (char\*)propertyRecord;

strongRefHeapBlock = &CollectedRecyclerWeakRefHeapBlock::Instance;

}

};

static const Js::PropertyRecord \* const builtInPropertyRecords[];

PropertyMap \* propertyMap;

PropertyNoCaseSetType \* caseInvariantPropertySet;

Js::ScriptContext \* rootPendingClose;

JsUtil::List<IProjectionContext \*, ArenaAllocator>\* pendingProjectionContextCloseList;

Js::ScriptEntryExitRecord \* entryExitRecord;

Js::InterpreterStackFrame\* leafInterpreterFrame;

const Js::PropertyRecord \* propertyNamesDirect[128];

ArenaAllocator threadAlloc;

ThreadServiceWrapper\* threadServiceWrapper;

uint functionCount;

uint sourceInfoCount;

Js::TypeId nextTypeId;

uint32 polymorphicCacheState;

Js::TypeId wellKnownHostTypeHTMLAllCollectionTypeId;

#ifdef ENABLE\_PROJECTION

SListBase<ExternalWeakReferenceCache \*> externalWeakReferenceCacheList;

#if DBG\_DUMP

IProjectionContextMemoryInfo \*projectionMemoryInformation;

#endif

#endif

#if ENABLE\_NATIVE\_CODEGEN

JsUtil::JobProcessor \*jobProcessor;

Js::Var \* bailOutRegisterSaveSpace;

CodeGenNumberThreadAllocator \* codeGenNumberThreadAllocator;

#endif

RecyclerRootPtr<RecyclableData> recyclableData;

uint temporaryArenaAllocatorCount;

uint temporaryGuestArenaAllocatorCount;

#if DBG\_DUMP || defined(PROFILE\_EXEC)

ScriptSite\* topLevelScriptSite;

#endif

Js::ScriptContext \*scriptContextList;

bool scriptContextEverRegistered;

static size\_t processNativeCodeSize;

size\_t nativeCodeSize;

size\_t sourceCodeSize;

Js::HiResTimer hTimer;

int stackProbeCount;

// Count stack probes and poll for continuation every n probes

static const int StackProbePollThreshold = 1000;

ArenaAllocator inlineCacheThreadInfoAllocator;

ArenaAllocator isInstInlineCacheThreadInfoAllocator;

ArenaAllocator equivalentTypeCacheInfoAllocator;

DListBase<Js::ScriptContext \*> inlineCacheScriptContexts;

DListBase<Js::ScriptContext \*> isInstInlineCacheScriptContexts;

DListBase<Js::EntryPointInfo \*> equivalentTypeCacheEntryPoints;

typedef SList<Js::InlineCache\*> InlineCacheList;

typedef JsUtil::BaseDictionary<Js::PropertyId, InlineCacheList\*, ArenaAllocator> InlineCacheListMapByPropertyId;

InlineCacheListMapByPropertyId protoInlineCacheByPropId;

InlineCacheListMapByPropertyId storeFieldInlineCacheByPropId;

uint registeredInlineCacheCount;

uint unregisteredInlineCacheCount;

typedef JsUtil::BaseDictionary<Js::Var, Js::IsInstInlineCache\*, ArenaAllocator> IsInstInlineCacheListMapByFunction;

IsInstInlineCacheListMapByFunction isInstInlineCacheByFunction;

ArenaAllocator prototypeChainEnsuredToHaveOnlyWritableDataPropertiesAllocator;

DListBase<Js::ScriptContext \*> prototypeChainEnsuredToHaveOnlyWritableDataPropertiesScriptContext;

DListBase<CollectCallBack> collectCallBackList;

CriticalSection csCollectionCallBack;

bool hasCollectionCallBack;

bool isOptimizedForManyInstances;

bool bgJit;

// We report library code to profiler only if called directly by user code. Not if called by library implementation.

bool isProfilingUserCode;

void\* jsrtRuntime;

bool hasUnhandledException;

bool hasCatchHandler;

DisableImplicitFlags disableImplicitFlags;

// Used for identifying that any particular time, the caller chain has try/catch blocks belong to the user code.

// If all try/catch blocks in the current stack marked as non-user code then this member will remain false.

bool hasCatchHandlerToUserCode;

Js::DelayLoadWinRtString delayLoadWinRtString;

#ifdef ENABLE\_PROJECTION

Js::DelayLoadWinRtError delayLoadWinRtError;

Js::DelayLoadWinRtTypeResolution delayLoadWinRtTypeResolution;

Js::DelayLoadWinRtRoParameterizedIID delayLoadWinRtRoParameterizedIID;

#endif

#if defined(ENABLE\_INTL\_OBJECT) || defined(ENABLE\_ES6\_CHAR\_CLASSIFIER)

Js::DelayLoadWindowsGlobalization delayLoadWindowsGlobalizationLibrary;

Js::WindowsGlobalizationAdapter windowsGlobalizationAdapter;

#endif

#ifdef ENABLE\_FOUNDATION\_OBJECT

Js::DelayLoadWinRtFoundation delayLoadWinRtFoundationLibrary;

Js::WindowsFoundationAdapter windowsFoundationAdapter;

#endif

#ifdef \_CONTROL\_FLOW\_GUARD

Js::DelayLoadWinCoreMemory delayLoadWinCoreMemoryLibrary;

#endif

Js::DelayLoadWinCoreProcessThreads delayLoadWinCoreProcessThreads;

// Number of script context attached with probe manager.

// This counter will be used as addref when the script context is created, this way we maintain the life of diagnostic object.

// Once no script context available , diagnostic will go away.

long crefSContextForDiag;

Entropy entropy;

JsUtil::Stack<HostScriptContext\*>\* hostScriptContextStack;

//

// Regex globals

//

UnifiedRegex::StandardChars<uint8>\* standardUTF8Chars;

UnifiedRegex::StandardChars<wchar\_t>\* standardUnicodeChars;

Js::ImplicitCallFlags implicitCallFlags;

\_\_declspec(thread) static uint activeScriptSiteCount;

bool isScriptActive;

// To synchronize with ETW rundown, which needs to walk scriptContext/functionBody/entryPoint lists.

CriticalSection csEtwRundown;

#ifdef \_M\_X64

friend class Js::Amd64StackFrame;

Js::Amd64ContextsManager amd64ContextsManager;

Js::Amd64ContextsManager\* GetAmd64ContextsManager() { return &amd64ContextsManager; }

#endif

typedef JsUtil::BaseDictionary<Js::DynamicType const \*, void \*, HeapAllocator, PowerOf2SizePolicy> DynamicObjectEnumeratorCacheMap;

DynamicObjectEnumeratorCacheMap dynamicObjectEnumeratorCacheMap;

ThreadContextWatsonTelemetryBlock localTelemetryBlock;

ThreadContextWatsonTelemetryBlock \* telemetryBlock;

NativeLibraryEntryRecord nativeLibraryEntry;

UCrtC99MathApis ucrtC99MathApis;

// Indicates the current loop depth as observed by the interpreter. The interpreter causes this value to be updated upon

// entering and leaving a loop.

uint8 loopDepth;

const ThreadConfiguration configuration;

public:

static ThreadContext \* globalListFirst;

static uint GetScriptSiteHolderCount() { return activeScriptSiteCount; }

static uint IncrementActiveScriptSiteCount() { return ++activeScriptSiteCount; }

static uint DecrementActiveScriptSiteCount() { return --activeScriptSiteCount; }

static ThreadContext \* GetThreadContextList() { return globalListFirst; }

void ValidateThreadContext();

bool IsInScript() const { return callRootLevel != 0; }

uint GetCallRootLevel() const { return callRootLevel; }

PageAllocator \* GetPageAllocator() { return &pageAllocator; }

AllocationPolicyManager \* GetAllocationPolicyManager() { return allocationPolicyManager; }

PreReservedVirtualAllocWrapper \* GetPreReservedVirtualAllocator() { return &preReservedVirtualAllocator; }

void ResetIsAllJITCodeInPreReservedRegion() { isAllJITCodeInPreReservedRegion = false; }

bool IsAllJITCodeInPreReservedRegion() { return isAllJITCodeInPreReservedRegion; }

CriticalSection\* GetEtwRundownCriticalSection() { return &csEtwRundown; }

UCrtC99MathApis\* GetUCrtC99MathApis() { return &ucrtC99MathApis; }

Js::DelayLoadWinRtString \*GetWinRTStringLibrary();

#ifdef ENABLE\_PROJECTION

Js::DelayLoadWinRtError \*GetWinRTErrorLibrary();

Js::DelayLoadWinRtTypeResolution\* GetWinRTTypeResolutionLibrary();

Js::DelayLoadWinRtRoParameterizedIID\* GetWinRTRoParameterizedIIDLibrary();

#endif

#if defined(ENABLE\_INTL\_OBJECT) || defined(ENABLE\_ES6\_CHAR\_CLASSIFIER)

Js::DelayLoadWindowsGlobalization \*GetWindowsGlobalizationLibrary();

Js::WindowsGlobalizationAdapter \*GetWindowsGlobalizationAdapter();

#endif

#ifdef ENABLE\_FOUNDATION\_OBJECT

Js::DelayLoadWinRtFoundation \*GetWinRtFoundationLibrary();

Js::WindowsFoundationAdapter \*GetWindowsFoundationAdapter();

#endif

#ifdef \_CONTROL\_FLOW\_GUARD

Js::DelayLoadWinCoreMemory \* GetWinCoreMemoryLibrary();

#endif

Js::DelayLoadWinCoreProcessThreads \* GetWinCoreProcessThreads();

JITTimer JITTelemetry;

ParserTimer ParserTelemetry;

GUID activityId;

void \*tridentLoadAddress;

void\* GetTridentLoadAddress() const { return tridentLoadAddress; }

void SetTridentLoadAddress(void \*loadAddress) { tridentLoadAddress = loadAddress; }

#ifdef ENABLE\_DIRECTCALL\_TELEMETRY

DirectCallTelemetry directCallTelemetry;

#endif

JITStats GetJITStats()

{

return JITTelemetry.GetStats();

}

void ResetJITStats()

{

JITTelemetry.Reset();

}

ParserStats GetParserStats()

{

return ParserTelemetry.GetStats();

}

void ResetParserStats()

{

ParserTelemetry.Reset();

}

double maxGlobalFunctionExecTime;

double GetAndResetMaxGlobalFunctionExecTime()

{

double res = maxGlobalFunctionExecTime;

this->maxGlobalFunctionExecTime = 0.0;

return res;

}

bool IsCFGEnabled();

void SetValidCallTargetForCFG(PVOID callTargetAddress, bool isSetValid = true);

BOOL HasPreviousHostScriptContext();

HostScriptContext\* GetPreviousHostScriptContext() ;

void PushHostScriptContext(HostScriptContext\* topProvider);

HostScriptContext\* PopHostScriptContext();

void SetInterruptPoller(InterruptPoller \*poller) { interruptPoller = poller; }

InterruptPoller \*GetInterruptPoller() const { return interruptPoller; }

BOOL HasInterruptPoller() const { return interruptPoller != nullptr; }

void CheckScriptInterrupt();

void CheckInterruptPoll();

bool DoInterruptProbe(Js::FunctionBody \*const func) const

{

return

(this->TestThreadContextFlag(ThreadContextFlagCanDisableExecution) &&

!PHASE\_OFF(Js::InterruptProbePhase, func)) ||

PHASE\_ON(Js::InterruptProbePhase, func);

}

bool DoInterruptProbe() const

{

return

(this->TestThreadContextFlag(ThreadContextFlagCanDisableExecution) &&

!PHASE\_OFF1(Js::InterruptProbePhase)) ||

PHASE\_ON1(Js::InterruptProbePhase);

}

bool EvalDisabled() const

{

return this->TestThreadContextFlag(ThreadContextFlagEvalDisabled);

}

bool NoJIT() const

{

return this->TestThreadContextFlag(ThreadContextFlagNoJIT);

}

#ifdef ENABLE\_DEBUG\_CONFIG\_OPTIONS

Js::Var GetMemoryStat(Js::ScriptContext\* scriptContext);

void SetAutoProxyName(LPCWSTR objectName);

LPCWSTR GetAutoProxyName() const { return recyclableData->autoProxyName; }

Js::PropertyId handlerPropertyId = Js::Constants::NoProperty;

#endif

void SetReturnedValueList(Js::ReturnedValueList \*returnedValueList)

{

Assert(this->recyclableData != nullptr);

this->recyclableData->returnedValueList = returnedValueList;

}

#if DBG

void EnsureNoReturnedValueList()

{

Assert(this->recyclableData == nullptr || this->recyclableData->returnedValueList == nullptr);

}

uint GetScriptContextCount() const { return this->scriptContextCount; }

#endif

Js::ScriptContext\* GetScriptContextList() const { return this->scriptContextList; }

bool WasAnyScriptContextEverRegistered() const { return this->scriptContextEverRegistered; }

#if DBG\_DUMP || defined(PROFILE\_EXEC)

void SetTopLevelScriptSite(ScriptSite\* topScriptSite) { this->topLevelScriptSite = topScriptSite; }

ScriptSite\* GetTopLevelScriptSite () { return this->topLevelScriptSite; }

#endif

#if DBG || defined(PROFILE\_EXEC)

virtual bool AsyncHostOperationStart(void \*) override;

virtual void AsyncHostOperationEnd(bool wasInAsync, void \*) override;

#endif

#if DBG

bool IsInAsyncHostOperation() const;

#endif

BOOL ReserveStaticTypeIds(\_\_in int first, \_\_in int last);

Js::TypeId ReserveTypeIds(int count);

Js::TypeId CreateTypeId();

Js::TypeId GetNextTypeId() { return nextTypeId; }

// Lookup the well known type registered with a Js::TypeId.

// typeId: The type id to match

// returns: The well known type which was previously registered via a call to SetWellKnownHostTypeId

WellKnownHostType GetWellKnownHostType(Js::TypeId typeId);

// Register a well known type to a Js::TypeId.

// wellKnownType: The well known type which we should register

// typeId: The type id which matches to the well known type

void SetWellKnownHostTypeId(WellKnownHostType wellKnownType, Js::TypeId typeId);

uint32 GetNextPolymorphicCacheState()

{

return ++polymorphicCacheState;

};

~ThreadContext();

void CloseForJSRT();

//Call back is called for one or more handles

//It does multiple callbacks (For example: separate call back for GC thread handle & JIT thread handles)

// template<class Fn>

//void ShutdownThreads(Fn callback);

void ShutdownThreads()

{

#if ENABLE\_NATIVE\_CODEGEN

if (jobProcessor)

{

jobProcessor->Close();

}

#endif

#ifdef CONCURRENT\_GC\_ENABLED

if (this->recycler != nullptr)

{

this->recycler->ShutdownThread();

}

#endif

}

Js::HiResTimer \* GetHiResTimer() { return &hTimer; }

ArenaAllocator\* GetThreadAlloc() { return &threadAlloc; }

static CriticalSection \* GetCriticalSection() { return &s\_csThreadContext; }

ThreadContext(AllocationPolicyManager \* allocationPolicyManager = nullptr, JsUtil::ThreadService::ThreadServiceCallback threadServiceCallback = nullptr, bool enableExperimentalFeatures = false);

static void Add(ThreadContext \*threadContext);

ThreadConfiguration const \* GetConfig() const { return &configuration; }

public:

void SetTelemetryBlock(ThreadContextWatsonTelemetryBlock \* telemetryBlock) { this->telemetryBlock = telemetryBlock; }

static ThreadContext\* GetContextForCurrentThread();

Recycler\* GetRecycler() { return recycler; }

Recycler\* EnsureRecycler();

ThreadContext::CollectCallBack \* AddRecyclerCollectCallBack(RecyclerCollectCallBackFunction callback, void \* context);

void RemoveRecyclerCollectCallBack(ThreadContext::CollectCallBack \* collectCallBack);

void AddToPendingProjectionContextCloseList(IProjectionContext \*projectionContext);

void RemoveFromPendingClose(IProjectionContext \*projectionContext);

void ClosePendingProjectionContexts();

void AddToPendingScriptContextCloseList(Js::ScriptContext \* scriptContext);

void RemoveFromPendingClose(Js::ScriptContext \* scriptContext);

void ClosePendingScriptContexts();

Js::PropertyRecord const \* GetPropertyName(Js::PropertyId propertyId);

Js::PropertyRecord const \* GetPropertyNameLocked(Js::PropertyId propertyId);

private:

template <bool locked> Js::PropertyRecord const \* GetPropertyNameImpl(Js::PropertyId propertyId);

public:

void FindPropertyRecord(Js::JavascriptString \*pstName, Js::PropertyRecord const \*\* propertyRecord);

void FindPropertyRecord(\_\_in LPCWSTR propertyName, \_\_in int propertyNameLength, Js::PropertyRecord const \*\* propertyRecord);

const Js::PropertyRecord \* FindPropertyRecord(const wchar\_t \* propertyName, int propertyNameLength);

JsUtil::List<const RecyclerWeakReference<Js::PropertyRecord const>\*>\* FindPropertyIdNoCase(Js::ScriptContext \* scriptContext, LPCWSTR propertyName, int propertyNameLength);

JsUtil::List<const RecyclerWeakReference<Js::PropertyRecord const>\*>\* FindPropertyIdNoCase(Js::ScriptContext \* scriptContext, JsUtil::CharacterBuffer<WCHAR> const& propertyName);

bool FindExistingPropertyRecord(\_In\_ JsUtil::CharacterBuffer<WCHAR> const& propertyName, Js::CaseInvariantPropertyListWithHashCode\*\* propertyRecord);

void CleanNoCasePropertyMap();

void ForceCleanPropertyMap();

const Js::PropertyRecord \* GetOrAddPropertyRecord(JsUtil::CharacterBuffer<wchar\_t> propertyName)

{

return GetOrAddPropertyRecordImpl(propertyName, false);

}

const Js::PropertyRecord \* GetOrAddPropertyRecordBind(JsUtil::CharacterBuffer<wchar\_t> propertyName)

{

return GetOrAddPropertyRecordImpl(propertyName, true);

}

void AddBuiltInPropertyRecord(const Js::PropertyRecord \*propertyRecord);

void GetOrAddPropertyId(\_\_in LPCWSTR propertyName, \_\_in int propertyNameLength, Js::PropertyRecord const\*\* propertyRecord);

void GetOrAddPropertyId(JsUtil::CharacterBuffer<WCHAR> const& propertName, Js::PropertyRecord const\*\* propertyRecord);

Js::PropertyRecord const \* UncheckedAddPropertyId(JsUtil::CharacterBuffer<WCHAR> const& propertyName, bool bind, bool isSymbol = false);

Js::PropertyRecord const \* UncheckedAddPropertyId(\_\_in LPCWSTR propertyName, \_\_in int propertyNameLength, bool bind = false, bool isSymbol = false);

#ifdef ENABLE\_JS\_ETW

void EtwLogPropertyIdList();

#endif

private:

const Js::PropertyRecord \* GetOrAddPropertyRecordImpl(JsUtil::CharacterBuffer<wchar\_t> propertyName, bool bind);

void AddPropertyRecordInternal(const Js::PropertyRecord \* propertyRecord);

void BindPropertyRecord(const Js::PropertyRecord \* propertyRecord);

bool IsDirectPropertyName(const wchar\_t \* propertyName, int propertyNameLength);

RecyclerWeakReference<const Js::PropertyRecord> \* CreatePropertyRecordWeakRef(const Js::PropertyRecord \* propertyRecord);

void AddCaseInvariantPropertyRecord(const Js::PropertyRecord \* propertyRecord);

uint scriptContextCount;

public:

void UncheckedAddBuiltInPropertyId();

BOOL IsNumericPropertyId(Js::PropertyId propertyId, uint32\* value);

bool IsActivePropertyId(Js::PropertyId pid);

void InvalidatePropertyRecord(const Js::PropertyRecord \* propertyRecord);

Js::PropertyId GetNextPropertyId();

Js::PropertyId GetMaxPropertyId();

uint GetHighestPropertyNameIndex() const;

void SetThreadServiceWrapper(ThreadServiceWrapper\*);

ThreadServiceWrapper\* GetThreadServiceWrapper();

uint GetUnreleasedScriptContextCount(){return scriptContextCount;}

#ifdef ENABLE\_PROJECTION

void AddExternalWeakReferenceCache(ExternalWeakReferenceCache \*externalWeakReferenceCache);

void RemoveExternalWeakReferenceCache(ExternalWeakReferenceCache \*externalWeakReferenceCache);

virtual void MarkExternalWeakReferencedObjects(bool inPartialCollect) override;

virtual void ResolveExternalWeakReferencedObjects() override;

#if DBG\_DUMP

void RegisterProjectionMemoryInformation(IProjectionContextMemoryInfo\* projectionContextMemoryInfo);

void DumpProjectionContextMemoryStats(LPCWSTR headerMsg, bool forceDetailed = false);

IProjectionContextMemoryInfo\* GetProjectionContextMemoryInformation();

#endif

#endif

uint NewFunctionNumber() { return ++functionCount; }

uint PeekNewFunctionNumber() { return functionCount + 1; }

uint NewSourceInfoNumber() { return ++sourceInfoCount; }

void AddCodeSize(size\_t newCode)

{

::InterlockedExchangeAdd(&nativeCodeSize, newCode);

::InterlockedExchangeAdd(&processNativeCodeSize, newCode);

}

void AddSourceSize(size\_t newCode) { sourceCodeSize += newCode; }

void SubCodeSize(size\_t deadCode)

{

Assert(nativeCodeSize >= deadCode);

Assert(processNativeCodeSize >= deadCode);

::InterlockedExchangeSubtract(&nativeCodeSize, deadCode);

::InterlockedExchangeSubtract(&processNativeCodeSize, deadCode);

}

void SubSourceSize(size\_t deadCode) { Assert(sourceCodeSize >= deadCode); sourceCodeSize -= deadCode; }

size\_t GetCodeSize() { return nativeCodeSize; }

static size\_t GetProcessCodeSize() { return processNativeCodeSize; }

size\_t GetSourceSize() { return sourceCodeSize; }

Js::ScriptEntryExitRecord \* GetScriptEntryExit() const { return entryExitRecord; }

void RegisterCodeGenRecyclableData(Js::CodeGenRecyclableData \*const codeGenRecyclableData);

void UnregisterCodeGenRecyclableData(Js::CodeGenRecyclableData \*const codeGenRecyclableData);

#if ENABLE\_NATIVE\_CODEGEN

BOOL IsNativeAddress(void \*pCodeAddr);

JsUtil::JobProcessor \*GetJobProcessor();

static void CloseSharedJobProcessor(const bool waitForThread);

Js::Var \* GetBailOutRegisterSaveSpace() const { return bailOutRegisterSaveSpace; }

CodeGenNumberThreadAllocator \* GetCodeGenNumberThreadAllocator() const

{

return codeGenNumberThreadAllocator;

}

#endif

void ResetFunctionCount() { Assert(this->GetScriptSiteHolderCount() == 0); this->functionCount = 0; }

void PushEntryExitRecord(Js::ScriptEntryExitRecord \*);

void PopEntryExitRecord(Js::ScriptEntryExitRecord \*);

uint EnterScriptStart(Js::ScriptEntryExitRecord \*, bool doCleanup);

void EnterScriptEnd(Js::ScriptEntryExitRecord \*, bool doCleanup);

template <bool leaveForHost>

void LeaveScriptStart(void \*);

template <bool leaveForHost>

void LeaveScriptEnd(void \*);

void DisposeOnLeaveScript();

void PushInterpreterFrame(Js::InterpreterStackFrame \*interpreterFrame);

Js::InterpreterStackFrame \*PopInterpreterFrame();

Js::InterpreterStackFrame \*GetLeafInterpreterFrame() const { return leafInterpreterFrame; }

Js::TempArenaAllocatorObject \* GetTemporaryAllocator(LPCWSTR name);

void ReleaseTemporaryAllocator(Js::TempArenaAllocatorObject \* tempAllocator);

Js::TempGuestArenaAllocatorObject \* GetTemporaryGuestAllocator(LPCWSTR name);

void ReleaseTemporaryGuestAllocator(Js::TempGuestArenaAllocatorObject \* tempAllocator);

// Should be called from script context, at the time when construction for scriptcontext is just done.

void EnsureDebugManager();

// Should be called from script context 's destructor,

void ReleaseDebugManager();

void RegisterScriptContext(Js::ScriptContext \*scriptContext);

void UnregisterScriptContext(Js::ScriptContext \*scriptContext);

// NoScriptScope

void SetNoScriptScope(bool noScriptScope) { this->noScriptScope = noScriptScope; }

bool IsNoScriptScope() { return this->noScriptScope; }

Js::ScriptContext \*\* RegisterInlineCacheScriptContext(Js::ScriptContext \* scriptContext);

Js::ScriptContext \*\* RegisterIsInstInlineCacheScriptContext(Js::ScriptContext \* scriptContext);

Js::EntryPointInfo \*\* RegisterEquivalentTypeCacheEntryPoint(Js::EntryPointInfo \* entryPoint);

void UnregisterInlineCacheScriptContext(Js::ScriptContext \*\* scriptContext);

void UnregisterIsInstInlineCacheScriptContext(Js::ScriptContext \*\* scriptContext);

void UnregisterEquivalentTypeCacheEntryPoint(Js::EntryPointInfo \*\* entryPoint);

void RegisterProtoInlineCache(Js::InlineCache \* inlineCache, Js::PropertyId propertyId);

void RegisterStoreFieldInlineCache(Js::InlineCache \* inlineCache, Js::PropertyId propertyId);

void NotifyInlineCacheBatchUnregistered(uint count);

#if DBG

bool IsProtoInlineCacheRegistered(const Js::InlineCache \* inlineCache, Js::PropertyId propertyId);

bool IsStoreFieldInlineCacheRegistered(const Js::InlineCache \* inlineCache, Js::PropertyId propertyId);

#endif

#if ENABLE\_NATIVE\_CODEGEN

Js::PropertyGuard\* RegisterSharedPropertyGuard(Js::PropertyId propertyId);

void RegisterLazyBailout(Js::PropertyId propertyId, Js::EntryPointInfo\* entryPoint);

void RegisterUniquePropertyGuard(Js::PropertyId propertyId, Js::PropertyGuard\* guard);

void RegisterUniquePropertyGuard(Js::PropertyId propertyId, RecyclerWeakReference<Js::PropertyGuard>\* guardWeakRef);

void RegisterConstructorCache(Js::PropertyId propertyId, Js::ConstructorCache\* cache);

#endif

private:

void RegisterInlineCache(InlineCacheListMapByPropertyId& inlineCacheMap, Js::InlineCache\* inlineCache, Js::PropertyId propertyId);

static bool IsInlineCacheRegistered(InlineCacheListMapByPropertyId& inlineCacheMap, const Js::InlineCache\* inlineCache, Js::PropertyId propertyId);

void InvalidateInlineCacheList(InlineCacheList \*inlineCacheList);

void CompactInlineCacheList(InlineCacheList \*inlineCacheList);

void CompactInlineCacheInvalidationLists();

void CompactProtoInlineCaches();

void CompactStoreFieldInlineCaches();

#if DBG

static bool IsInlineCacheInList(const Js::InlineCache\* inlineCache, const InlineCacheList\* inlineCacheChain);

#endif

#if ENABLE\_NATIVE\_CODEGEN

void InvalidateFixedFieldGuard(Js::PropertyGuard\* guard);

PropertyGuardEntry\* EnsurePropertyGuardEntry(const Js::PropertyRecord\* propertyRecord, bool& foundExistingEntry);

void InvalidatePropertyGuardEntry(const Js::PropertyRecord\* propertyRecord, PropertyGuardEntry\* entry);

#endif

public:

class AutoDisableExpiration

{

public:

AutoDisableExpiration(ThreadContext\* threadContext):

\_threadContext(threadContext),

\_oldExpirationDisabled(threadContext->disableExpiration)

{

\_threadContext->disableExpiration = true;

}

~AutoDisableExpiration()

{

\_threadContext->disableExpiration = \_oldExpirationDisabled;

}

private:

ThreadContext\* \_threadContext;

bool \_oldExpirationDisabled;

};

void InvalidateProtoInlineCaches(Js::PropertyId propertyId);

void InvalidateStoreFieldInlineCaches(Js::PropertyId propertyId);

void InvalidateAllProtoInlineCaches();

bool AreAllProtoInlineCachesInvalidated();

void InvalidateAllStoreFieldInlineCaches();

bool AreAllStoreFieldInlineCachesInvalidated();

void InvalidatePropertyGuards(Js::PropertyId propertyId);

void InvalidateAllPropertyGuards();

void RegisterIsInstInlineCache(Js::IsInstInlineCache \* inlineCache, Js::Var function);

void UnregisterIsInstInlineCache(Js::IsInstInlineCache \* inlineCache, Js::Var function);

#if DBG

bool IsIsInstInlineCacheRegistered(Js::IsInstInlineCache \* inlineCache, Js::Var function);

#endif

private:

void InvalidateIsInstInlineCacheList(Js::IsInstInlineCache\* inlineCacheList);

#if DBG

bool IsIsInstInlineCacheInList(const Js::IsInstInlineCache\* inlineCache, const Js::IsInstInlineCache\* inlineCacheList);

#endif

public:

void InvalidateIsInstInlineCachesForFunction(Js::Var function);

void InvalidateAllIsInstInlineCaches();

bool AreAllIsInstInlineCachesInvalidated() const;

#ifdef PERSISTENT\_INLINE\_CACHES

void ClearInlineCachesWithDeadWeakRefs();

#endif

void ClearInlineCaches();

void ClearIsInstInlineCaches();

void ClearEquivalentTypeCaches();

void ClearScriptContextCaches();

void RegisterTypeWithProtoPropertyCache(const Js::PropertyId propertyId, Js::Type \*const type);

void InvalidateProtoTypePropertyCaches(const Js::PropertyId propertyId);

void InternalInvalidateProtoTypePropertyCaches(const Js::PropertyId propertyId);

void InvalidateAllProtoTypePropertyCaches();

Js::ScriptContext \*\* RegisterPrototypeChainEnsuredToHaveOnlyWritableDataPropertiesScriptContext(Js::ScriptContext \* scriptContext);

void UnregisterPrototypeChainEnsuredToHaveOnlyWritableDataPropertiesScriptContext(Js::ScriptContext \*\* scriptContext);

void ClearPrototypeChainEnsuredToHaveOnlyWritableDataPropertiesCaches();

BOOL HasUnhandledException() const { return hasUnhandledException; }

void SetHasUnhandledException() {hasUnhandledException = TRUE; }

void ResetHasUnhandledException() {hasUnhandledException = FALSE; }

void SetUnhandledExceptionObject(Js::JavascriptExceptionObject\* exceptionObject) {recyclableData->unhandledExceptionObject = exceptionObject; }

Js::JavascriptExceptionObject\* GetUnhandledExceptionObject() const { return recyclableData->unhandledExceptionObject; };

bool HasCatchHandler() const { return hasCatchHandler; }

void SetHasCatchHandler(bool hasCatchHandler) { this->hasCatchHandler = hasCatchHandler; }

bool IsUserCode() const { return this->hasCatchHandlerToUserCode; }

void SetIsUserCode(bool set) { this->hasCatchHandlerToUserCode = set; }

void QueueFreeOldEntryPointInfoIfInScript(Js::FunctionEntryPointInfo\* oldEntryPointInfo)

{

if (this->IsInScript())

{

// Add it to the list only if it's not already in it

if (oldEntryPointInfo->nextEntryPoint == nullptr && !oldEntryPointInfo->IsCleanedUp())

{

oldEntryPointInfo->nextEntryPoint = recyclableData->oldEntryPointInfo;

recyclableData->oldEntryPointInfo = oldEntryPointInfo;

}

}

}

static BOOLEAN IsOnStack(void const \*ptr);

\_\_declspec(noinline) bool IsStackAvailable(size\_t size);

\_\_declspec(noinline) bool IsStackAvailableNoThrow(size\_t size = Js::Constants::MinStackDefault);

static bool IsCurrentStackAvailable(size\_t size);

void ProbeStackNoDispose(size\_t size, Js::ScriptContext \*scriptContext, PVOID returnAddress = nullptr);

void ProbeStack(size\_t size, Js::ScriptContext \*scriptContext, PVOID returnAddress = nullptr);

void ProbeStack(size\_t size, Js::RecyclableObject \* obj, Js::ScriptContext \*scriptContext);

void ProbeStack(size\_t size);

static void \_\_stdcall ProbeCurrentStackNoDispose(size\_t size, Js::ScriptContext \*scriptContext);

static void \_\_stdcall ProbeCurrentStack(size\_t size, Js::ScriptContext \*scriptContext);

static void \_\_stdcall ProbeCurrentStack2(size\_t size, Js::ScriptContext \*scriptContext, uint32 u1, uint32 u2)

{

ProbeCurrentStack(size, scriptContext);

}

#if ENABLE\_PROFILE\_INFO

void ThreadContext::EnsureSourceProfileManagersByUrlMap();

Js::SourceDynamicProfileManager\* GetSourceDynamicProfileManager(\_In\_z\_ const WCHAR\* url, \_In\_ uint hash, \_Inout\_ bool\* addref);

uint ReleaseSourceDynamicProfileManagers(const WCHAR\* url);

#endif

void EnsureSymbolRegistrationMap();

const Js::PropertyRecord\* GetSymbolFromRegistrationMap(const wchar\_t\* stringKey);

const Js::PropertyRecord\* AddSymbolToRegistrationMap(const wchar\_t\* stringKey, charcount\_t stringLength);

inline void ClearPendingSOError()

{

this->GetPendingSOErrorObject()->ClearError();

}

inline void ClearPendingOOMError()

{

this->GetPendingOOMErrorObject()->ClearError();

}

Js::JavascriptExceptionObject \*GetPendingSOErrorObject()

{

Assert(recyclableData->soErrorObject.IsPendingExceptionObject());

return &recyclableData->soErrorObject;

}

Js::JavascriptExceptionObject \*GetPendingOOMErrorObject()

{

Assert(recyclableData->oomErrorObject.IsPendingExceptionObject());

return &recyclableData->oomErrorObject;

}

Js::JavascriptExceptionObject \*GetPendingTerminatedErrorObject()

{

return &recyclableData->terminatedErrorObject;

}

Js::JavascriptExceptionObject\* GetRecordedException()

{

return recyclableData->exceptionObject;

}

bool GetPropagateException()

{

return recyclableData->propagateException;

}

void SetHasThrownPendingException()

{

Assert(this->IsInScript());

this->hasThrownPendingException = true;

}

void SetRecordedException(Js::JavascriptExceptionObject\* exceptionObject, bool propagateToDebugger = false)

{

this->recyclableData->exceptionObject = exceptionObject;

this->recyclableData->propagateException = propagateToDebugger;

}

Entropy& GetEntropy()

{

return entropy;

}

Js::ImplicitCallFlags \* GetAddressOfImplicitCallFlags()

{

return &implicitCallFlags;

}

DisableImplicitFlags \* GetAddressOfDisableImplicitFlags()

{

return &disableImplicitFlags;

}

Js::ImplicitCallFlags GetImplicitCallFlags()

{

return implicitCallFlags;

}

void SetImplicitCallFlags(Js::ImplicitCallFlags flags)

{

//Note: this action is inlined into JITed code in Lowerer::GenerateCallProfiling.

// if you change this, you might want to add it there too.

implicitCallFlags = flags;

}

void ClearImplicitCallFlags();

void ClearImplicitCallFlags(Js::ImplicitCallFlags flags);

void AddImplicitCallFlags(Js::ImplicitCallFlags flags)

{

SetImplicitCallFlags((Js::ImplicitCallFlags)(implicitCallFlags | flags));

}

void CheckAndResetImplicitCallAccessorFlag();

template <class Fn>

\_\_inline Js::Var ExecuteImplicitCall(Js::RecyclableObject \* function, Js::ImplicitCallFlags flags, Fn implicitCall)

{

// For now, we will not allow Function that is marked as HasNoSideEffect to be called, and we will just bailout.

// These function may still throw exceptions, so we will need to add checks with RecordImplicitException

// so that we don't throw exception when disableImplicitCall is set before we allow these function to be called

// as an optimization. (These functions are valueOf and toString calls for built-in non primitive types)

Js::FunctionInfo::Attributes attributes = Js::FunctionInfo::GetAttributes(function);

// we can hoist out const method if we know the function doesn't have side effect,

// and the value can be hoisted.

if (this->HasNoSideEffect(function, attributes))

{

// Has no side effect means the function does not change global value or

// will check for implicit call flags

return implicitCall();

}

// Don't call the implicit call if disable implicit call

if (IsDisableImplicitCall())

{

AddImplicitCallFlags(flags);

// Return "undefined" just so we have a valid var, in case subsequent instructions are executed

// before we bail out.

return function->GetScriptContext()->GetLibrary()->GetUndefined();

}

if ((attributes & Js::FunctionInfo::HasNoSideEffect) != 0)

{

// Has no side effect means the function does not change global value or

// will check for implicit call flags

return implicitCall();

}

// Save and restore implicit flags around the implicit call

Js::ImplicitCallFlags saveImplicitCallFlags = this->GetImplicitCallFlags();

Js::Var result = implicitCall();

this->SetImplicitCallFlags((Js::ImplicitCallFlags)(saveImplicitCallFlags | flags));

return result;

}

bool HasNoSideEffect(Js::RecyclableObject \* function) const;

bool HasNoSideEffect(Js::RecyclableObject \* function, Js::FunctionInfo::Attributes attr) const;

bool RecordImplicitException();

DisableImplicitFlags GetDisableImplicitFlags() const { return disableImplicitFlags; }

void SetDisableImplicitFlags(DisableImplicitFlags flags) { disableImplicitFlags = flags; }

bool IsDisableImplicitCall() const { return (disableImplicitFlags & DisableImplicitCallFlag) != 0; }

bool IsDisableImplicitException() const { return (disableImplicitFlags & DisableImplicitExceptionFlag) != 0; }

void DisableImplicitCall() { disableImplicitFlags = (DisableImplicitFlags)(disableImplicitFlags | DisableImplicitCallFlag); }

void ClearDisableImplicitFlags() { disableImplicitFlags = DisableImplicitNoFlag; }

virtual uint GetRandomNumber() override;

// DefaultCollectWrapper

virtual void PreCollectionCallBack(CollectionFlags flags) override;

virtual void PreSweepCallback() override;

virtual void WaitCollectionCallBack() override;

virtual void PostCollectionCallBack() override;

virtual BOOL ExecuteRecyclerCollectionFunction(Recycler \* recycler, CollectionFunction function, CollectionFlags flags) override;

#ifdef FAULT\_INJECTION

virtual void DisposeScriptContextByFaultInjectionCallBack() override;

#endif

virtual void DisposeObjects(Recycler \* recycler) override;

typedef DList<ExpirableObject\*, ArenaAllocator> ExpirableObjectList;

ExpirableObjectList\* expirableObjectList;

ExpirableObjectList\* expirableObjectDisposeList;

int numExpirableObjects;

int expirableCollectModeGcCount;

bool disableExpiration;

bool InExpirableCollectMode();

void TryEnterExpirableCollectMode();

void TryExitExpirableCollectMode();

void RegisterExpirableObject(ExpirableObject\* object);

void UnregisterExpirableObject(ExpirableObject\* object);

void DisposeExpirableObject(ExpirableObject\* object);

void \* GetDynamicObjectEnumeratorCache(Js::DynamicType const \* dynamicType);

void AddDynamicObjectEnumeratorCache(Js::DynamicType const \* dynamicType, void \* cache);

public:

bool IsScriptActive() const { return isScriptActive; }

void SetIsScriptActive(bool isActive) { isScriptActive = isActive; }

bool IsExecutionDisabled() const

{

return this->GetStackLimitForCurrentThread() == Js::Constants::StackLimitForScriptInterrupt;

}

void DisableExecution();

void EnableExecution();

bool TestThreadContextFlag(ThreadContextFlags threadContextFlag) const;

void SetThreadContextFlag(ThreadContextFlags threadContextFlag);

void ClearThreadContextFlag(ThreadContextFlags threadContextFlag);

void SetForceOneIdleCollection();

bool IsInThreadServiceCallback() const { return threadService.IsInCallback(); }

Js::DebugManager \* GetDebugManager() const { return this->debugManager; }

const NativeLibraryEntryRecord::Entry\* PeekNativeLibraryEntry() const { return this->nativeLibraryEntry.Peek(); }

void PushNativeLibraryEntry(\_In\_ NativeLibraryEntryRecord::Entry\* entry) { this->nativeLibraryEntry.Push(entry); }

void PopNativeLibraryEntry() { this->nativeLibraryEntry.Pop(); }

bool IsProfilingUserCode() const { return isProfilingUserCode; }

void SetIsProfilingUserCode(bool value) { isProfilingUserCode = value; }

#if DBG\_DUMP

uint scriptSiteCount;

#endif

#ifdef BAILOUT\_INJECTION

uint bailOutByteCodeLocationCount;

#endif

#ifdef DYNAMIC\_PROFILE\_MUTATOR

DynamicProfileMutator \* dynamicProfileMutator;

#endif

//

// Regex helpers

//

UnifiedRegex::StandardChars<uint8>\* GetStandardChars(\_\_inout\_opt uint8\* dummy);

UnifiedRegex::StandardChars<wchar\_t>\* GetStandardChars(\_\_inout\_opt wchar\_t\* dummy);

bool IsOptimizedForManyInstances() const { return isOptimizedForManyInstances; }

void OptimizeForManyInstances(const bool optimizeForManyInstances)

{

Assert(!recycler || optimizeForManyInstances == isOptimizedForManyInstances); // mode cannot be changed after recycler is created

isOptimizedForManyInstances = optimizeForManyInstances;

}

#if ENABLE\_NATIVE\_CODEGEN

bool IsBgJitEnabled() const { return bgJit; }

void EnableBgJit(const bool enableBgJit)

{

Assert(!jobProcessor || enableBgJit == bgJit);

bgJit = enableBgJit;

}

#endif

void\* GetJSRTRuntime() const { return jsrtRuntime; }

void SetJSRTRuntime(void\* runtime);

bool CanBeFalsy(Js::TypeId typeId);

private:

BOOL ExecuteRecyclerCollectionFunctionCommon(Recycler \* recycler, CollectionFunction function, CollectionFlags flags);

void DoInvalidateProtoTypePropertyCaches(const Js::PropertyId propertyId, TypeHashSet \*const typeHashSet);

void InitializePropertyMaps();

void CreateNoCasePropertyMap();

InterruptPoller \*interruptPoller;

void CollectionCallBack(RecyclerCollectCallBackFlags flags);

// Cache used by HostDispatch::GetBuiltInOperationFromEntryPoint

private:

JsUtil::BaseDictionary<Js::JavascriptMethod, uint, ArenaAllocator, PowerOf2SizePolicy> entryPointToBuiltInOperationIdCache;

public:

bool IsEntryPointToBuiltInOperationIdCacheInitialized()

{

return entryPointToBuiltInOperationIdCache.Count() != 0;

}

bool GetBuiltInOperationIdFromEntryPoint(Js::JavascriptMethod entryPoint, uint \* id)

{

return entryPointToBuiltInOperationIdCache.TryGetValue(entryPoint, id);

}

void SetBuiltInOperationIdForEntryPoint(Js::JavascriptMethod entryPoint, uint id)

{

entryPointToBuiltInOperationIdCache.Add(entryPoint, id);

}

void ResetEntryPointToBuiltInOperationIdCache()

{

entryPointToBuiltInOperationIdCache.ResetNoDelete();

}

uint8 LoopDepth() const

{

return loopDepth;

}

void SetLoopDepth(const uint8 loopDepth)

{

this->loopDepth = loopDepth;

}

void IncrementLoopDepth()

{

if(loopDepth != UCHAR\_MAX)

{

++loopDepth;

}

}

void DecrementLoopDepth()

{

if(loopDepth != 0)

{

--loopDepth;

}

}

#if defined(CHECK\_MEMORY\_LEAK) || defined(LEAK\_REPORT)

static void ReportAndCheckLeaksOnProcessDetach();

#endif

#ifdef LEAK\_REPORT

void SetRootTrackerScriptContext(Js::ScriptContext \* scriptContext);

void ClearRootTrackerScriptContext(Js::ScriptContext \* scriptContext);

private:

Js::ScriptContext \* rootTrackerScriptContext;

DWORD threadId;

#endif

};

extern void(\*InitializeAdditionalProperties)(ThreadContext \*threadContext);

// Temporarily set script profiler isProfilingUserCode state, restore at destructor

class AutoProfilingUserCode

{

private:

ThreadContext\* threadContext;

const bool oldIsProfilingUserCode;

public:

AutoProfilingUserCode(ThreadContext\* threadContext, bool isProfilingUserCode) :

threadContext(threadContext),

oldIsProfilingUserCode(threadContext->IsProfilingUserCode())

{

threadContext->SetIsProfilingUserCode(isProfilingUserCode);

}

~AutoProfilingUserCode()

{

threadContext->SetIsProfilingUserCode(oldIsProfilingUserCode);

}

};

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

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

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

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

#include "RuntimeBasePch.h"

#include "Base\ThreadContextTLSEntry.h"

ulong ThreadContextTLSEntry::s\_tlsSlot = TLS\_OUT\_OF\_INDEXES;

bool ThreadContextTLSEntry::InitializeProcess()

{

Assert(s\_tlsSlot == TLS\_OUT\_OF\_INDEXES);

s\_tlsSlot = TlsAlloc();

return (s\_tlsSlot != TLS\_OUT\_OF\_INDEXES);

}

void ThreadContextTLSEntry::CleanupProcess()

{

Assert(s\_tlsSlot != TLS\_OUT\_OF\_INDEXES);

TlsFree(s\_tlsSlot);

s\_tlsSlot = TLS\_OUT\_OF\_INDEXES;

}

bool ThreadContextTLSEntry::IsProcessInitialized()

{

return s\_tlsSlot != TLS\_OUT\_OF\_INDEXES;

}

void ThreadContextTLSEntry::InitializeThread()

{

Assert(s\_tlsSlot != TLS\_OUT\_OF\_INDEXES);

Assert(!TlsGetValue(s\_tlsSlot));

TlsSetValue(s\_tlsSlot, NULL);

}

void ThreadContextTLSEntry::CleanupThread()

{

Assert(s\_tlsSlot != TLS\_OUT\_OF\_INDEXES);

ThreadContextTLSEntry \* entry = GetEntryForCurrentThread();

if (entry != NULL)

{

HeapDelete(entry);

TlsSetValue(s\_tlsSlot, NULL);

}

}

bool ThreadContextTLSEntry::TrySetThreadContext(ThreadContext \* threadContext)

{

Assert(threadContext != NULL);

Assert(s\_tlsSlot != TLS\_OUT\_OF\_INDEXES);

DWORD threadContextThreadId = threadContext->GetCurrentThreadId();

// If a thread context is current on another thread, then you cannot set it to current on this one.

if (threadContextThreadId != ThreadContext::NoThread && threadContextThreadId != ::GetCurrentThreadId())

{

// the thread doesn't support rental thread and try to set on a different thread???

Assert(!threadContext->GetIsThreadBound());

return false;

}

ThreadContextTLSEntry \* entry = GetEntryForCurrentThread();

if (entry == NULL)

{

Assert(!threadContext->GetIsThreadBound());

entry = CreateEntryForCurrentThread();

}

else if (entry->threadContext != NULL && entry->threadContext != threadContext)

{

// If the thread has an active thread context and either that thread context is thread

// bound (in which case it cannot be moved off this thread), or if the thread context

// is running script, you cannot move it off this thread.

if (entry->threadContext->GetIsThreadBound() || entry->threadContext->IsInScript())

{

return false;

}

ClearThreadContext(entry, true);

}

SetThreadContext(entry, threadContext);

return true;

}

void ThreadContextTLSEntry::SetThreadContext(ThreadContextTLSEntry \* entry, ThreadContext \* threadContext)

{

entry->threadContext = threadContext;

threadContext->SetStackProber(&entry->prober);

threadContext->SetCurrentThreadId(::GetCurrentThreadId());

}

bool ThreadContextTLSEntry::ClearThreadContext(bool isValid)

{

return ClearThreadContext(GetEntryForCurrentThread(), isValid, false);

}

bool ThreadContextTLSEntry::ClearThreadContext(ThreadContextTLSEntry \* entry, bool isThreadContextValid, bool force)

{

Assert(s\_tlsSlot != TLS\_OUT\_OF\_INDEXES);

if (entry != NULL)

{

if (entry->threadContext != NULL && isThreadContextValid)

{

// If the thread has an active thread context and either that thread context is thread

// bound (in which case it cannot be moved off this thread), or if the thread context

// is running script, you cannot move it off this thread.

if (!force && (entry->threadContext->GetIsThreadBound() || entry->threadContext->IsInScript()))

{

return false;

}

entry->threadContext->SetCurrentThreadId(ThreadContext::NoThread);

entry->threadContext->SetStackProber(NULL);

}

entry->threadContext = NULL;

}

return true;

}

void ThreadContextTLSEntry::Delete(ThreadContextTLSEntry \* entry)

{

HeapDelete(entry);

}

\_\_inline ThreadContextTLSEntry \* ThreadContextTLSEntry::GetEntryForCurrentThread()

{

Assert(s\_tlsSlot != TLS\_OUT\_OF\_INDEXES);

return reinterpret\_cast<ThreadContextTLSEntry \*>(TlsGetValue(s\_tlsSlot));

}

ThreadContextTLSEntry \* ThreadContextTLSEntry::CreateEntryForCurrentThread()

{

Assert(s\_tlsSlot != TLS\_OUT\_OF\_INDEXES);

Assert(TlsGetValue(s\_tlsSlot) == NULL);

ThreadContextTLSEntry \* entry = HeapNewStructZ(ThreadContextTLSEntry);

#pragma prefast(suppress:6001, "Memory from HeapNewStructZ are zero initialized")

entry->prober.Initialize();

TlsSetValue(s\_tlsSlot, entry);

return entry;

}

\_\_inline ThreadContext \* ThreadContextTLSEntry::GetThreadContext()

{

return this->threadContext;

}

ThreadContextId ThreadContextTLSEntry::GetCurrentThreadContextId()

{

ThreadContextTLSEntry \* entry = GetEntryForCurrentThread();

if (entry != NULL && entry->GetThreadContext() != NULL)

{

return (ThreadContextId)entry->GetThreadContext();

}

return NoThreadContextId;

}

ThreadContextId ThreadContextTLSEntry::GetThreadContextId(ThreadContext \* threadContext)

{

return threadContext;

}

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

// 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

class ThreadContextTLSEntry

{

public:

static bool InitializeProcess();

static void CleanupProcess();

static bool IsProcessInitialized();

static void InitializeThread();

static void CleanupThread();

static void Delete(ThreadContextTLSEntry \* entry);

static bool TrySetThreadContext(ThreadContext \* threadContext);

static void SetThreadContext(ThreadContextTLSEntry \* entry, ThreadContext \* threadContext);

static bool ClearThreadContext(bool isValid);

static bool ClearThreadContext(ThreadContextTLSEntry \* entry, bool isThreadContextValid, bool force = true);

static ThreadContextTLSEntry \* GetEntryForCurrentThread();

static ThreadContextTLSEntry \* CreateEntryForCurrentThread();

static ThreadContextId GetThreadContextId(ThreadContext \* threadContext);

static ulong s\_tlsSlot;

ThreadContext \* GetThreadContext();

private:

friend JsUtil::ExternalApi;

static ThreadContextId GetCurrentThreadContextId();

private:

ThreadContext \* threadContext;

StackProber prober;

};

class ThreadContextScope

{

public:

ThreadContextScope(ThreadContext \* threadContext)

{

if (!threadContext->GetIsThreadBound())

{

originalContext = ThreadContextTLSEntry::GetEntryForCurrentThread() ?

ThreadContextTLSEntry::GetEntryForCurrentThread()->GetThreadContext() : NULL;

wasInUse = threadContext == originalContext;

isValid = ThreadContextTLSEntry::TrySetThreadContext(threadContext);

doCleanup = !wasInUse && isValid;

}

else

{

Assert(ThreadContext::GetContextForCurrentThread() == threadContext);

isValid = true;

wasInUse = true;

doCleanup = false;

}

}

~ThreadContextScope()

{

if (doCleanup)

{

bool cleared = true;

#if DBG

cleared =

#endif

ThreadContextTLSEntry::ClearThreadContext(this->isValid);

Assert(cleared);

if (originalContext)

{

bool canSetback = true;

#if DBG

canSetback =

#endif

ThreadContextTLSEntry::TrySetThreadContext(originalContext);

Assert(canSetback);

}

}

}

void Invalidate()

{

this->isValid = false;

}

bool IsValid() const

{

return this->isValid;

}

bool WasInUse() const

{

return this->wasInUse;

}

private:

bool doCleanup;

bool isValid;

bool wasInUse;

ThreadContext\* originalContext;

};

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

// 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

// This class is used to communicate between ThreadContext and JavascriptThreadService

class ThreadServiceWrapper abstract

{

public:

virtual bool ScheduleNextCollectOnExit() = 0;

virtual void ScheduleFinishConcurrent() = 0;

virtual void SetForceOneIdleCollection() = 0;

};

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

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

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

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

#include "RuntimeBasePch.h"

#include "Base\ThreadServiceWrapperbase.h"

ThreadServiceWrapperBase::ThreadServiceWrapperBase() :

threadContext(nullptr),

needIdleCollect(false),

inIdleCollect(false),

hasScheduledIdleCollect(false),

shouldScheduleIdleCollectOnExitIdle(false),

forceIdleCollectOnce(false)

{

}

bool ThreadServiceWrapperBase::Initialize(ThreadContext \*newThreadContext)

{

if (newThreadContext == nullptr)

{

return false;

}

threadContext = newThreadContext;

threadContext->SetThreadServiceWrapper(this);

return true;

}

void ThreadServiceWrapperBase::Shutdown()

{

if (hasScheduledIdleCollect)

{

#if DBG

// Fake the inIdleCollect to get pass asserts in FinishIdleCollect

inIdleCollect = true;

#endif

FinishIdleCollect(FinishReason::FinishReasonNormal);

}

}

bool ThreadServiceWrapperBase::ScheduleIdleCollect(uint ticks, bool scheduleAsTask)

{

Assert(!threadContext->IsInScript());

// We should schedule have called this in one of two cases:

// 1) Either needIdleCollect is true- in which case, we should schedule one

// 2) Or ScheduleNextCollectionOnExit was called when needIdleCollect was true, but we didn't schedule

// one because we were at the time in one. Later, as we unwound, we might have set needIdleCollect to false

// but because we had noted that we needed to schedule a collect, we would end up coming into this function

// so allow for that

Assert(needIdleCollect || shouldScheduleIdleCollectOnExitIdle || threadContext->GetRecycler()->CollectionInProgress());

if (!CanScheduleIdleCollect())

{

return false;

}

if (hasScheduledIdleCollect)

{

return true;

}

if (OnScheduleIdleCollect(ticks, scheduleAsTask))

{

JS\_ETW(EventWriteJSCRIPT\_GC\_IDLE\_START(this));

IDLE\_COLLECT\_VERBOSE\_TRACE(L"ScheduledIdleCollect- Set hasScheduledIdleCollect\n");

hasScheduledIdleCollect = true;

return true;

}

else

{

IDLE\_COLLECT\_TRACE(L"Idle timer setup failed\n");

FinishIdleCollect(FinishReason::FinishReasonIdleTimerSetupFailed);

return false;

}

}

bool ThreadServiceWrapperBase::IdleCollect()

{

Assert(hasScheduledIdleCollect);

IDLE\_COLLECT\_VERBOSE\_TRACE(L"IdleCollect- reset hasScheduledIdleCollect\n");

hasScheduledIdleCollect = false;

// Don't do anything and kill the timer if we are called recursively or if we are in script

if (inIdleCollect || threadContext->IsInScript())

{

FinishIdleCollect(FinishReason::FinishReasonNormal);

return hasScheduledIdleCollect;

}

// If during idle collect we determine that we need to schedule another

// idle collect, this gets flipped to true

shouldScheduleIdleCollectOnExitIdle = false;

AutoBooleanToggle autoInIdleCollect(&inIdleCollect);

Recycler\* recycler = threadContext->GetRecycler();

// Finish concurrent on timer heart beat if needed

// We wouldn't try to finish if we need to schedule

// an idle task to finish the collection

if (this->ShouldFinishConcurrentCollectOnIdleCallback() && recycler->FinishConcurrent<FinishConcurrentOnIdle>())

{

IDLE\_COLLECT\_TRACE(L"Idle callback: finish concurrent\n");

JS\_ETW(EventWriteJSCRIPT\_GC\_IDLE\_CALLBACK\_FINISH(this));

}

while (true)

{

// If a GC is still happening, just wait for the next heart beat

if (recycler->CollectionInProgress())

{

ScheduleIdleCollect(IdleTicks, true /\* schedule as task \*/);

break;

}

// If there no more need of idle collect, then cancel the timer

if (!needIdleCollect)

{

FinishIdleCollect(FinishReason::FinishReasonNormal);

break;

}

int timeDiff = tickCountNextIdleCollection - GetTickCount();

// See if we pass the time for the next scheduled Idle GC

if (timeDiff > 0)

{

// Not time yet, wait for the next heart beat

ScheduleIdleCollect(IdleTicks, false /\* not schedule as task \*/);

IDLE\_COLLECT\_TRACE(L"Idle callback: nop until next collection: %d\n", timeDiff);

break;

}

// activate an idle collection

IDLE\_COLLECT\_TRACE(L"Idle callback: collection: %d\n", timeDiff);

JS\_ETW(EventWriteJSCRIPT\_GC\_IDLE\_CALLBACK\_NEWCOLLECT(this));

needIdleCollect = false;

recycler->CollectNow<CollectOnScriptIdle>();

}

if (shouldScheduleIdleCollectOnExitIdle)

{

ScheduleIdleCollect(IdleTicks, false /\* not schedule as task \*/);

}

return hasScheduledIdleCollect;

}

void ThreadServiceWrapperBase::FinishIdleCollect(ThreadServiceWrapperBase::FinishReason reason)

{

Assert(reason == FinishReason::FinishReasonIdleTimerSetupFailed ||

reason == FinishReason::FinishReasonTaskComplete ||

inIdleCollect || threadContext->IsInScript() || !threadContext->GetRecycler()->CollectionInProgress());

IDLE\_COLLECT\_VERBOSE\_TRACE(L"FinishIdleCollect- Reset hasScheduledIdleCollect\n");

hasScheduledIdleCollect = false;

needIdleCollect = false;

OnFinishIdleCollect();

IDLE\_COLLECT\_TRACE(L"Idle timer finished\n");

JS\_ETW(EventWriteJSCRIPT\_GC\_IDLE\_FINISHED(this));

}

bool ThreadServiceWrapperBase::ScheduleNextCollectOnExit()

{

Assert(!threadContext->IsInScript());

Assert(!needIdleCollect || hasScheduledIdleCollect);

Recycler\* recycler = threadContext->GetRecycler();

recycler->FinishConcurrent<FinishConcurrentOnExitScript>();

#ifdef RECYCLER\_TRACE

bool oldNeedIdleCollect = needIdleCollect;

if (forceIdleCollectOnce)

{

IDLE\_COLLECT\_VERBOSE\_TRACE(L"Need to force one idle collection\n");

}

#endif

needIdleCollect = forceIdleCollectOnce || recycler->ShouldIdleCollectOnExit();

if (needIdleCollect)

{

// Set up when we will do the idle decommit

tickCountNextIdleCollection = GetTickCount() + IdleTicks;

IDLE\_COLLECT\_VERBOSE\_TRACE(L"Idle on exit collect %s: %d\n", (oldNeedIdleCollect ? L"rescheduled" : L"scheduled"),

tickCountNextIdleCollection - GetTickCount());

JS\_ETW(EventWriteJSCRIPT\_GC\_IDLE\_SCHEDULED(this));

}

else

{

IDLE\_COLLECT\_VERBOSE\_TRACE(L"Idle on exit collect %s\n", oldNeedIdleCollect ? L"cancelled" : L"not scheduled");

if (!recycler->CollectionInProgress())

{

// We collected and finished, no need to ensure the idle collect call back.

return true;

}

IDLE\_COLLECT\_VERBOSE\_TRACE(L"Idle on exit collect %s\n", hasScheduledIdleCollect || oldNeedIdleCollect ? L"reschedule finish" : L"schedule finish");

}

// Don't schedule the call back if we are already in idle call back, as we don't do anything on recursive call anyways

// IdleCollect will schedule one if necessary

if (inIdleCollect)

{

shouldScheduleIdleCollectOnExitIdle = true;

return true;

}

else

{

return ScheduleIdleCollect(IdleTicks, false /\* not schedule as task \*/);

}

}

void ThreadServiceWrapperBase::ClearForceOneIdleCollection()

{

IDLE\_COLLECT\_VERBOSE\_TRACE(L"Clearing force idle collect flag\n");

this->forceIdleCollectOnce = false;

}

void ThreadServiceWrapperBase::SetForceOneIdleCollection()

{

IDLE\_COLLECT\_VERBOSE\_TRACE(L"Setting force idle collect flag\n");

this->forceIdleCollectOnce = true;

}

void ThreadServiceWrapperBase::ScheduleFinishConcurrent()

{

Assert(!threadContext->IsInScript());

Assert(threadContext->GetRecycler()->CollectionInProgress());

if (!this->inIdleCollect)

{

IDLE\_COLLECT\_VERBOSE\_TRACE(L"Idle collect %s\n", needIdleCollect ? L"reschedule finish" : L"scheduled finish");

this->needIdleCollect = false;

ScheduleIdleCollect(IdleFinishTicks, true /\* schedule as task \*/);

}

}

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

// 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

#include "Base\threadservicewrapper.h"

namespace JsStaticAPI

{

class JavascriptLibrary;

}

class ThreadServiceWrapperBase : public ThreadServiceWrapper

{

friend class JsStaticAPI::JavascriptLibrary;

public:

bool ScheduleNextCollectOnExit() override sealed;

void ScheduleFinishConcurrent() override sealed;

void SetForceOneIdleCollection() override;

protected:

enum FinishReason

{

FinishReasonNormal,

FinishReasonIdleTimerSetupFailed,

FinishReasonTaskComplete

};

ThreadServiceWrapperBase();

bool Initialize(ThreadContext \*newThreadContext);

void Shutdown();

bool IdleCollect();

void FinishIdleCollect(FinishReason reason);

void ClearForceOneIdleCollection();

virtual bool CanScheduleIdleCollect() = 0;

virtual bool OnScheduleIdleCollect(uint delta, bool scheduleAsTask) = 0;

virtual void OnFinishIdleCollect() = 0;

virtual bool ShouldFinishConcurrentCollectOnIdleCallback() = 0;

ThreadContext \*GetThreadContext() { return threadContext; }

private:

static const unsigned int IdleTicks = 1000; // 1 second

static const unsigned int IdleFinishTicks = 100; // 100 ms;

bool ScheduleIdleCollect(uint ticks, bool scheduleAsTask);

ThreadContext\* threadContext;

bool inIdleCollect;

bool needIdleCollect;

bool forceIdleCollectOnce;

unsigned int tickCountNextIdleCollection;

bool hasScheduledIdleCollect;

bool shouldScheduleIdleCollectOnExitIdle;

};

#ifdef RECYCLER\_TRACE

#define IDLE\_COLLECT\_VERBOSE\_TRACE(msg, ...) \

if (Js::Configuration::Global.flags.Verbose) \

{ \

IDLE\_COLLECT\_TRACE(msg, \_\_VA\_ARGS\_\_); \

}

#define IDLE\_COLLECT\_TRACE(msg, ...) \

if (Js::Configuration::Global.flags.Trace.IsEnabled(Js::IdleCollectPhase)) \

{\

Output::Print(L"%04X> " msg, ::GetCurrentThreadId(), \_\_VA\_ARGS\_\_); \

Output::Flush(); \

}

#else

#define IDLE\_COLLECT\_TRACE(...)

#define IDLE\_COLLECT\_VERBOSE\_TRACE(...)

#endif

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

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

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

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

#include "RuntimeBasePch.h"

#include "Debug\DiagProbe.h"

#include "Debug\BreakpointProbe.h"

#include "Debug\DebugDocument.h"

namespace Js

{

// if m\_cchLength < 0 it came from an external source.

// If m\_cbLength > abs(m\_cchLength) then m\_utf8Source contains non-ASCII (multi-byte encoded) characters.

Utf8SourceInfo::Utf8SourceInfo(ISourceHolder\* mappableSource, int32 cchLength, SRCINFO const\* srcInfo, DWORD\_PTR secondaryHostSourceContext, ScriptContext\* scriptContext) :

sourceHolder(mappableSource),

m\_cchLength(cchLength),

m\_pOriginalSourceInfo(nullptr),

m\_srcInfo(srcInfo),

m\_secondaryHostSourceContext(secondaryHostSourceContext),

m\_debugDocument(nullptr),

m\_sourceInfoId(scriptContext->GetThreadContext()->NewSourceInfoNumber()),

m\_hasHostBuffer(false),

m\_isCesu8(false),

m\_isLibraryCode(false),

m\_isXDomain(false),

m\_isXDomainString(false),

m\_scriptContext(scriptContext),

m\_lineOffsetCache(nullptr),

m\_deferredFunctionsDictionary(nullptr),

m\_deferredFunctionsInitialized(false),

debugModeSource(nullptr),

debugModeSourceIsEmpty(false),

debugModeSourceLength(0),

callerUtf8SourceInfo(nullptr)

{

if (!sourceHolder->IsDeferrable())

{

this->debugModeSource = this->sourceHolder->GetSource(L"Entering Debug Mode");

this->debugModeSourceLength = this->sourceHolder->GetByteLength(L"Entering Debug Mode");

this->debugModeSourceIsEmpty = !this->HasSource() || this->debugModeSource == nullptr;

}

}

LPCUTF8 Utf8SourceInfo::GetSource(const wchar\_t \* reason) const

{

AssertMsg(this->sourceHolder != nullptr, "We have no source mapper.");

if (this->m\_scriptContext->IsInDebugMode())

{

AssertMsg(this->debugModeSource != nullptr || this->debugModeSourceIsEmpty, "Debug mode source should have been set by this point.");

return debugModeSource;

}

else

{

return sourceHolder->GetSource(reason == nullptr ? L"Utf8SourceInfo::GetSource" : reason);

}

}

size\_t Utf8SourceInfo::GetCbLength(const wchar\_t \* reason) const

{

AssertMsg(this->sourceHolder != nullptr, "We have no source mapper.");

if (this->m\_scriptContext->IsInDebugMode())

{

AssertMsg(this->debugModeSource != nullptr || this->debugModeSourceIsEmpty, "Debug mode source should have been set by this point.");

return debugModeSourceLength;

}

else

{

return sourceHolder->GetByteLength(reason == nullptr ? L"Utf8SourceInfo::GetSource" : reason);

}

}

void

Utf8SourceInfo::Dispose(bool isShutdown)

{

ClearDebugDocument();

this->debugModeSource = nullptr;

if (this->m\_hasHostBuffer)

{

PERF\_COUNTER\_DEC(Basic, ScriptCodeBufferCount);

HeapFree(GetProcessHeap(), 0 , m\_pHostBuffer);

m\_pHostBuffer = nullptr;

}

};

void

Utf8SourceInfo::SetHostBuffer(BYTE \* pcszCode)

{

Assert(!this->m\_hasHostBuffer);

Assert(this->m\_pHostBuffer == nullptr);

this->m\_hasHostBuffer = true;

this->m\_pHostBuffer = pcszCode;

}

enum

{

fsiHostManaged = 0x01,

fsiScriptlet = 0x02,

fsiDeferredParse = 0x04

};

void Utf8SourceInfo::RemoveFunctionBody(FunctionBody\* functionBody)

{

Assert(this->functionBodyDictionary);

const LocalFunctionId functionId = functionBody->GetLocalFunctionId();

Assert(functionBodyDictionary->Item(functionId) == functionBody);

functionBodyDictionary->Remove(functionId);

functionBody->SetIsFuncRegistered(false);

}

void Utf8SourceInfo::SetFunctionBody(FunctionBody \* functionBody)

{

Assert(this->m\_scriptContext == functionBody->GetScriptContext());

Assert(this->functionBodyDictionary);

// Only register a function body when source info is ready. Note that m\_pUtf8Source can still be null for lib script.

Assert(functionBody->GetSourceIndex() != Js::Constants::InvalidSourceIndex);

Assert(!functionBody->GetIsFuncRegistered());

const LocalFunctionId functionId = functionBody->GetLocalFunctionId();

FunctionBody\* oldFunctionBody = nullptr;

if (functionBodyDictionary->TryGetValue(functionId, &oldFunctionBody)) {

Assert(oldFunctionBody != functionBody);

oldFunctionBody->SetIsFuncRegistered(false);

}

functionBodyDictionary->Item(functionId, functionBody);

functionBody->SetIsFuncRegistered(true);

}

void Utf8SourceInfo::EnsureInitialized(int initialFunctionCount)

{

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

Recycler\* recycler = threadContext->GetRecycler();

if (this->functionBodyDictionary == nullptr)

{

// This collection is allocated with leaf allocation policy. The references to the function body

// here does not keep the function alive. However, the functions remove themselves at finalize

// so if a function actually is in this map, it means that it is alive.

this->functionBodyDictionary = RecyclerNew(recycler, FunctionBodyDictionary, recycler,

initialFunctionCount, threadContext->GetEtwRundownCriticalSection());

}

if (CONFIG\_FLAG(DeferTopLevelTillFirstCall) && !m\_deferredFunctionsInitialized)

{

Assert(this->m\_deferredFunctionsDictionary == nullptr);

this->m\_deferredFunctionsDictionary = RecyclerNew(recycler, DeferredFunctionsDictionary, recycler,

initialFunctionCount, threadContext->GetEtwRundownCriticalSection());

m\_deferredFunctionsInitialized = true;

}

}

Utf8SourceInfo\*

Utf8SourceInfo::NewWithHolder(ScriptContext\* scriptContext, ISourceHolder\* sourceHolder, int32 length, SRCINFO const\* srcInfo)

{

// TODO: make this finalizable? Or have a finalizable version which would HeapDelete the string? Is this needed?

DWORD\_PTR secondaryHostSourceContext = Js::Constants::NoHostSourceContext;

if (srcInfo->sourceContextInfo->IsDynamic())

{

secondaryHostSourceContext = scriptContext->GetThreadContext()->GetDebugManager()->AllocateSecondaryHostSourceContext();

}

Recycler \* recycler = scriptContext->GetRecycler();

Utf8SourceInfo\* toReturn = RecyclerNewFinalized(recycler,

Utf8SourceInfo, sourceHolder, length, SRCINFO::Copy(recycler, srcInfo), secondaryHostSourceContext, scriptContext);

if (scriptContext->IsInDebugMode())

{

toReturn->debugModeSource = sourceHolder->GetSource(L"Debug Mode Loading");

toReturn->debugModeSourceLength = sourceHolder->GetByteLength(L"Debug Mode Loading");

toReturn->debugModeSourceIsEmpty = toReturn->debugModeSource == nullptr || sourceHolder->IsEmpty();

}

return toReturn;

}

Utf8SourceInfo\*

Utf8SourceInfo::New(ScriptContext\* scriptContext, LPCUTF8 utf8String, int32 length, size\_t numBytes, SRCINFO const\* srcInfo)

{

utf8char\_t \* newUtf8String = RecyclerNewArrayLeaf(scriptContext->GetRecycler(), utf8char\_t, numBytes + 1);

js\_memcpy\_s(newUtf8String, numBytes + 1, utf8String, numBytes + 1);

return NewWithNoCopy(scriptContext, newUtf8String, length, numBytes, srcInfo);

}

Utf8SourceInfo\*

Utf8SourceInfo::NewWithNoCopy(ScriptContext\* scriptContext, LPCUTF8 utf8String, int32 length, size\_t numBytes, SRCINFO const \* srcInfo)

{

ISourceHolder\* sourceHolder = RecyclerNew(scriptContext->GetRecycler(), SimpleSourceHolder, utf8String, numBytes);

return NewWithHolder(scriptContext, sourceHolder, length, srcInfo);

}

Utf8SourceInfo\*

Utf8SourceInfo::Clone(ScriptContext\* scriptContext, const Utf8SourceInfo\* sourceInfo)

{

Utf8SourceInfo\* newSourceInfo = Utf8SourceInfo::NewWithHolder(scriptContext, sourceInfo->GetSourceHolder()->Clone(scriptContext), sourceInfo->m\_cchLength,

SRCINFO::Copy(scriptContext->GetRecycler(), sourceInfo->GetSrcInfo()));

newSourceInfo->m\_isXDomain = sourceInfo->m\_isXDomain;

newSourceInfo->m\_isXDomainString = sourceInfo->m\_isXDomainString;

newSourceInfo->m\_isLibraryCode = sourceInfo->m\_isLibraryCode;

newSourceInfo->SetIsCesu8(sourceInfo->GetIsCesu8());

newSourceInfo->m\_lineOffsetCache = sourceInfo->m\_lineOffsetCache;

if (scriptContext->IsInDebugMode())

{

newSourceInfo->SetInDebugMode(true);

}

return newSourceInfo;

}

Utf8SourceInfo\*

Utf8SourceInfo::CloneNoCopy(ScriptContext\* scriptContext, const Utf8SourceInfo\* sourceInfo, SRCINFO const\* srcInfo)

{

Utf8SourceInfo\* newSourceInfo = Utf8SourceInfo::NewWithHolder(scriptContext, sourceInfo->GetSourceHolder(), sourceInfo->m\_cchLength,

srcInfo ? srcInfo : sourceInfo->GetSrcInfo());

newSourceInfo->m\_isXDomain = sourceInfo->m\_isXDomain;

newSourceInfo->m\_isXDomainString = sourceInfo->m\_isXDomainString;

newSourceInfo->m\_isLibraryCode = sourceInfo->m\_isLibraryCode;

newSourceInfo->SetIsCesu8(sourceInfo->GetIsCesu8());

if (sourceInfo->m\_hasHostBuffer)

{

// Keep the host buffer alive via the original source info

newSourceInfo->m\_pOriginalSourceInfo = sourceInfo;

}

newSourceInfo->EnsureInitialized(sourceInfo->GetFunctionBodyCount());

return newSourceInfo;

}

HRESULT Utf8SourceInfo::EnsureLineOffsetCacheNoThrow()

{

HRESULT hr = S\_OK;

// This is a double check, otherwise we would have to have a private function, and add an assert.

// Basically the outer check is for try/catch, inner check (inside EnsureLineOffsetCache) is for that method as its public.

if (this->m\_lineOffsetCache == nullptr)

{

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

{

this->EnsureLineOffsetCache();

}

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

}

return hr;

}

void Utf8SourceInfo::EnsureLineOffsetCache()

{

if (this->m\_lineOffsetCache == nullptr)

{

LPCUTF8 sourceStart = this->GetSource(L"Utf8SourceInfo::AllocateLineOffsetCache");

LPCUTF8 sourceEnd = sourceStart + this->GetCbLength(L"Utf8SourceInfo::AllocateLineOffsetCache");

LPCUTF8 sourceAfterBOM = sourceStart;

charcount\_t startChar = FunctionBody::SkipByteOrderMark(sourceAfterBOM /\* byref \*/);

int64 byteStartOffset = (sourceAfterBOM - sourceStart);

Recycler\* recycler = this->m\_scriptContext->GetRecycler();

this->m\_lineOffsetCache = RecyclerNew(recycler, JsUtil::LineOffsetCache<Recycler>, recycler, sourceAfterBOM, sourceEnd, startChar, (int)byteStartOffset);

}

}

Js::FunctionBody\* Utf8SourceInfo::FindFunction(Js::LocalFunctionId id) const

{

Js::FunctionBody \*matchedFunctionBody = nullptr;

if (this->functionBodyDictionary)

{

// Ignore return value - OK if function is not found.

this->functionBodyDictionary->TryGetValue(id, &matchedFunctionBody);

if (matchedFunctionBody == nullptr || matchedFunctionBody->IsPartialDeserializedFunction())

{

return nullptr;

}

}

return matchedFunctionBody;

}

void Utf8SourceInfo::GetLineInfoForCharPosition(charcount\_t charPosition, charcount\_t \*outLineNumber, charcount\_t \*outColumn, charcount\_t \*outLineByteOffset, bool allowSlowLookup)

{

AssertMsg(this->m\_lineOffsetCache != nullptr || allowSlowLookup, "LineOffsetCache wasn't created, EnsureLineOffsetCache should have been called.");

AssertMsg(outLineNumber != nullptr && outColumn != nullptr && outLineByteOffset != nullptr, "Expected out parameter's can't be a nullptr.");

charcount\_t lineCharOffset = 0;

int line = 0;

if (this->m\_lineOffsetCache == nullptr)

{

LPCUTF8 sourceStart = this->GetSource(L"Utf8SourceInfo::AllocateLineOffsetCache");

LPCUTF8 sourceEnd = sourceStart + this->GetCbLength(L"Utf8SourceInfo::AllocateLineOffsetCache");

LPCUTF8 sourceAfterBOM = sourceStart;

lineCharOffset = FunctionBody::SkipByteOrderMark(sourceAfterBOM /\* byref \*/);

Assert((sourceAfterBOM - sourceStart) < MAXUINT32);

charcount\_t byteStartOffset = (charcount\_t)(sourceAfterBOM - sourceStart);

line = JsUtil::LineOffsetCache<Recycler>::FindLineForCharacterOffset(sourceAfterBOM, sourceEnd, lineCharOffset, byteStartOffset, charPosition);

\*outLineByteOffset = byteStartOffset;

}

else

{

line = this->m\_lineOffsetCache->GetLineForCharacterOffset(charPosition, &lineCharOffset, outLineByteOffset);

}

Assert(charPosition >= lineCharOffset);

\*outLineNumber = line;

\*outColumn = charPosition - lineCharOffset;

}

void Utf8SourceInfo::CreateLineOffsetCache(const JsUtil::LineOffsetCache<Recycler>::LineOffsetCacheItem \*items, charcount\_t numberOfItems)

{

AssertMsg(this->m\_lineOffsetCache == nullptr, "LineOffsetCache is already initialized!");

Recycler\* recycler = this->m\_scriptContext->GetRecycler();

this->m\_lineOffsetCache = RecyclerNew(recycler, JsUtil::LineOffsetCache<Recycler>, recycler, items, numberOfItems);

}

DWORD\_PTR Utf8SourceInfo::GetHostSourceContext() const

{

return m\_srcInfo->sourceContextInfo->dwHostSourceContext;

}

bool Utf8SourceInfo::IsDynamic() const

{

return m\_srcInfo->sourceContextInfo->IsDynamic();

}

SourceContextInfo\* Utf8SourceInfo::GetSourceContextInfo() const

{

return this->m\_srcInfo->sourceContextInfo;

}

// Get's the first function in the function body dictionary

// Used if the caller want's any function in this source info

Js::FunctionBody\* Utf8SourceInfo::GetAnyParsedFunction()

{

if (this->functionBodyDictionary != nullptr && this->functionBodyDictionary->Count() > 0)

{

FunctionBody\* functionBody = nullptr;

int i = 0;

do

{

functionBody = this->functionBodyDictionary->GetValueAt(i);

if (functionBody != nullptr && functionBody->GetByteCode() == nullptr && !functionBody->GetIsFromNativeCodeModule()) functionBody = nullptr;

i++;

}

while (functionBody == nullptr && i < this->functionBodyDictionary->Count());

return functionBody;

}

return nullptr;

}

bool Utf8SourceInfo::IsHostManagedSource() const

{

return ((this->m\_srcInfo->grfsi & fsiHostManaged) == fsiHostManaged);

}

void Utf8SourceInfo::SetCallerUtf8SourceInfo(Utf8SourceInfo\* callerUtf8SourceInfo)

{

this->callerUtf8SourceInfo = callerUtf8SourceInfo;

}

Utf8SourceInfo\* Utf8SourceInfo::GetCallerUtf8SourceInfo() const

{

return this->callerUtf8SourceInfo;

}

void Utf8SourceInfo::TrackDeferredFunction(Js::LocalFunctionId functionID, Js::ParseableFunctionInfo \*function)

{

if (this->m\_scriptContext->DoUndeferGlobalFunctions())

{

Assert(m\_deferredFunctionsInitialized);

if (this->m\_deferredFunctionsDictionary != nullptr)

{

this->m\_deferredFunctionsDictionary->Add(functionID, function);

}

}

}

void Utf8SourceInfo::StopTrackingDeferredFunction(Js::LocalFunctionId functionID)

{

if (this->m\_scriptContext->DoUndeferGlobalFunctions())

{

Assert(m\_deferredFunctionsInitialized);

if (this->m\_deferredFunctionsDictionary != nullptr)

{

this->m\_deferredFunctionsDictionary->Remove(functionID);

}

}

}

void Utf8SourceInfo::ClearDebugDocument(bool close)

{

if (this->m\_debugDocument != nullptr)

{

if (close)

{

m\_debugDocument->CloseDocument();

}

this->m\_debugDocument = nullptr;

}

}

bool Utf8SourceInfo::GetDebugDocumentName(BSTR \* sourceName)

{

if (this->HasDebugDocument() && this->GetDebugDocument()->HasDocumentText())

{

// ToDo (SaAgarwa): Fix for JsRT debugging

IDebugDocumentText \*documentText = static\_cast<IDebugDocumentText \*>(this->GetDebugDocument()->GetDocumentText());

if (documentText->GetName(DOCUMENTNAMETYPE\_URL, sourceName) == S\_OK)

{

return true;

}

}

return false;

}

}

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

// 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

{

struct Utf8SourceInfo : public FinalizableObject

{

// TODO: Change this to LeafValueDictionary

typedef JsUtil::SynchronizedDictionary<Js::LocalFunctionId, Js::FunctionBody\*, Recycler> FunctionBodyDictionary;

typedef JsUtil::SynchronizedDictionary<Js::LocalFunctionId, Js::ParseableFunctionInfo\*, Recycler> DeferredFunctionsDictionary;

friend class RemoteUtf8SourceInfo;

friend class ScriptContext;

public:

bool HasSource() const { return !this->sourceHolder->IsEmpty(); }

LPCUTF8 GetSource(const wchar\_t \* reason = nullptr) const;

size\_t GetCbLength(const wchar\_t \* reason = nullptr) const;

ULONG GetParseFlags()

{

return this->parseFlags;

}

void SetParseFlags(ULONG parseFlags)

{

this->parseFlags = parseFlags;

}

ULONG GetByteCodeGenerationFlags()

{

return this->byteCodeGenerationFlags;

}

void SetByteCodeGenerationFlags(ULONG byteCodeGenerationFlags)

{

this->byteCodeGenerationFlags = byteCodeGenerationFlags;

}

bool IsInDebugMode()

{

return this->debugModeSource != nullptr || this->debugModeSourceIsEmpty;

}

// For Hybrid debugging purposes we need to have the source mapped in because chakra may be in a frozen state when the source would be needed.

void SetInDebugMode(bool inDebugMode)

{

AssertMsg(this->sourceHolder != nullptr, "We have no source holder.");

if (!this->sourceHolder->IsDeferrable())

{

//The source should be already loaded;

AssertMsg(IsInDebugMode(), "Debug mode source should have been loaded as the source holder isn't deferrable.");

return;

}

if (inDebugMode)

{

AssertMsg(!IsInDebugMode(), "Debug mode source should have not yet been set.");

this->debugModeSource = this->sourceHolder->GetSource(L"Entering Debug Mode");

this->debugModeSourceLength = this->sourceHolder->GetByteLength(L"Entering Debug Mode");

this->debugModeSourceIsEmpty = !this->HasSource() || this->debugModeSource == nullptr;

this->EnsureLineOffsetCache();

}

else

{

// If debugModeSourceIsEmpty is false, that means debugModeSource isn't nullptr or we aren't in debug mode.

this->debugModeSourceIsEmpty = false;

this->debugModeSource = nullptr;

this->debugModeSourceLength = 0;

}

}

void RetrieveSourceText(\_\_out\_ecount\_full(cchLim - cchMin) LPOLESTR cpText, charcount\_t cchMin, charcount\_t cchLim) const

{

size\_t cbMin = GetCbLength(L"Utf8SourceInfo::RetrieveSourceText") == GetCchLength() ? cchMin : utf8::CharacterIndexToByteIndex(GetSource(L"Utf8SourceInfo::RetrieveSourceText"), GetCbLength(L"Utf8SourceInfo::RetrieveSourceText"), cchMin, utf8::doAllowThreeByteSurrogates);

utf8::DecodeInto(cpText, GetSource(L"Utf8SourceInfo::RetrieveSourceText") + cbMin, cchLim - cchMin, utf8::doAllowThreeByteSurrogates);

}

size\_t CharacterIndexToByteIndex(charcount\_t cchIndex) const

{

return cchIndex < m\_cchLength ? (GetCbLength(L"CharacterIndexToByteIndex") == m\_cchLength ? cchIndex : utf8::CharacterIndexToByteIndex(this->GetSource(L"CharacterIndexToByteIndex"), GetCbLength(L"CharacterIndexToByteIndex"), cchIndex, utf8::doAllowThreeByteSurrogates)) : m\_cchLength;

}

charcount\_t ByteIndexToCharacterIndex(size\_t cbIndex) const

{

return cbIndex < GetCbLength(L"CharacterIndexToByteIndex") ? static\_cast< charcount\_t>(GetCbLength(L"CharacterIndexToByteIndex") == m\_cchLength ? cbIndex : utf8::ByteIndexIntoCharacterIndex(this->GetSource(L"CharacterIndexToByteIndex"), cbIndex, utf8::doAllowThreeByteSurrogates)) : static\_cast< charcount\_t >(GetCbLength(L"CharacterIndexToByteIndex"));

}

charcount\_t GetCchLength() const

{

return m\_cchLength;

}

void SetCchLength(charcount\_t cchLength)

{

m\_cchLength = cchLength;

}

const SRCINFO\* GetSrcInfo() const

{

return m\_srcInfo;

}

void EnsureInitialized(int initialFunctionCount);

// The following 3 functions are on individual functions that exist within this

// source info- for them to be called, byte code generation should have created

// the function body in question, in which case, functionBodyDictionary needs to have

// been initialized

void SetFunctionBody(FunctionBody \* functionBody);

void RemoveFunctionBody(FunctionBody\* functionBodyBeingRemoved);

// The following functions could get called even if EnsureInitialized hadn't gotten called

// (Namely in the OOM scenario), so we simply guard against that condition rather than

// asserting

int GetFunctionBodyCount() const

{

return (this->functionBodyDictionary ? this->functionBodyDictionary->Count() : 0);

}

// Gets a thread-unique id for this source info

// (Behaves the same as the function number)

uint GetSourceInfoId()

{

return m\_sourceInfoId;

}

void ClearFunctions()

{

if (this->functionBodyDictionary)

{

this->functionBodyDictionary->Clear();

}

}

bool HasFunctions() const

{

return (this->functionBodyDictionary ? this->functionBodyDictionary->Count() > 0 : false);

}

template <typename TDelegate>

void MapFunction(TDelegate mapper) const

{

if (this->functionBodyDictionary)

{

this->functionBodyDictionary->Map([mapper] (Js::LocalFunctionId, Js::FunctionBody\* functionBody) {

Assert(functionBody);

mapper(functionBody);

});

}

}

// Get's the first function in the function body dictionary

// Used if the caller want's any function in this source info

Js::FunctionBody\* GetAnyParsedFunction();

template <typename TDelegate>

void MapFunctionUntil(TDelegate mapper) const

{

if (this->functionBodyDictionary)

{

this->functionBodyDictionary->MapUntil([mapper] (Js::LocalFunctionId, Js::FunctionBody\* functionBody) {

Assert(functionBody);

return mapper(functionBody);

});

}

}

Js::FunctionBody\* FindFunction(Js::LocalFunctionId id) const;

template <typename TDelegate>

Js::FunctionBody\* FindFunction(TDelegate predicate) const

{

Js::FunctionBody\* matchedFunctionBody = nullptr;

// Function body collection could be null if we OOMed

// during byte code generation but the source info didn't get

// collected because of a false positive reference- we should

// skip over these Utf8SourceInfos.

if (this->functionBodyDictionary)

{

this->functionBodyDictionary->Map(

[&matchedFunctionBody, predicate] (Js::LocalFunctionId, Js::FunctionBody\* functionBody)

{

Assert(functionBody);

if (predicate(functionBody)) {

matchedFunctionBody = functionBody;

return true;

}

return false;

});

}

return matchedFunctionBody;

}

void SetHostBuffer(BYTE \* pcszCode);

bool HasDebugDocument() const

{

return m\_debugDocument != nullptr;

}

void SetDebugDocument(DebugDocument \* document)

{

Assert(!HasDebugDocument());

m\_debugDocument = document;

}

DebugDocument\* GetDebugDocument() const

{

Assert(HasDebugDocument());

return m\_debugDocument;

}

void ClearDebugDocument(bool close = true);

void SetIsCesu8(bool isCesu8)

{

this->m\_isCesu8 = isCesu8;

}

bool GetIsCesu8() const

{

return m\_isCesu8;

}

DWORD\_PTR GetSecondaryHostSourceContext() const

{

return m\_secondaryHostSourceContext;

}

void SetIsLibraryCode()

{

m\_isLibraryCode = true;

}

bool GetIsLibraryCode() const

{

return m\_isLibraryCode;

}

bool GetIsXDomain() const { return m\_isXDomain; }

void SetIsXDomain() { m\_isXDomain = true; }

bool GetIsXDomainString() const { return m\_isXDomainString; }

void SetIsXDomainString() { m\_isXDomainString = true; }

DWORD\_PTR GetHostSourceContext() const;

bool IsDynamic() const;

SourceContextInfo\* GetSourceContextInfo() const;

void SetSecondaryHostContext(DWORD\_PTR hostSourceContext);

bool IsHostManagedSource() const;

static int StaticGetHashCode(\_\_in const Utf8SourceInfo\* const si)

{

return si->GetSourceHolder()->GetHashCode();

}

static bool StaticEquals(\_\_in Utf8SourceInfo\* s1, \_\_in Utf8SourceInfo\* s2)

{

if (s1 == nullptr || s2 == nullptr) return false;

//If the source holders have the same pointer, we are expecting the sources to be equal

return (s1 == s2) || s1->GetSourceHolder()->Equals(s2->GetSourceHolder());

}

virtual void Finalize(bool isShutdown) override { /\* nothing \*/ }

virtual void Dispose(bool isShutdown) override;

virtual void Mark(Recycler \*recycler) override { AssertMsg(false, "Mark called on object that isn't TrackableObject"); }

static Utf8SourceInfo\* NewWithHolder(ScriptContext\* scriptContext, ISourceHolder\* sourceHolder, int32 length, SRCINFO const\* srcInfo);

static Utf8SourceInfo\* New(ScriptContext\* scriptContext, LPCUTF8 utf8String, int32 length, size\_t numBytes, SRCINFO const\* srcInfo);

static Utf8SourceInfo\* NewWithNoCopy(ScriptContext\* scriptContext, LPCUTF8 utf8String, int32 length, size\_t numBytes, SRCINFO const\* srcInfo);

static Utf8SourceInfo\* Clone(ScriptContext\* scriptContext, const Utf8SourceInfo\* sourceinfo);

static Utf8SourceInfo\* CloneNoCopy(ScriptContext\* scriptContext, const Utf8SourceInfo\* sourceinfo, SRCINFO const \* srcInfo);

ScriptContext \* GetScriptContext() const

{

return this->m\_scriptContext;

}

void EnsureLineOffsetCache();

HRESULT EnsureLineOffsetCacheNoThrow();

void DeleteLineOffsetCache()

{

this->m\_lineOffsetCache = nullptr;

}

void CreateLineOffsetCache(const JsUtil::LineOffsetCache<Recycler>::LineOffsetCacheItem \*items, charcount\_t numberOfItems);

size\_t GetLineCount()

{

return this->GetLineOffsetCache()->GetLineCount();

}

JsUtil::LineOffsetCache<Recycler> \*GetLineOffsetCache()

{

AssertMsg(this->m\_lineOffsetCache != nullptr, "LineOffsetCache wasn't created, EnsureLineOffsetCache should have been called.");

return m\_lineOffsetCache;

}

void GetLineInfoForCharPosition(charcount\_t charPosition, charcount\_t \*outLineNumber, charcount\_t \*outColumn, charcount\_t \*outLineByteOffset, bool allowSlowLookup = false);

void GetCharPositionForLineInfo(charcount\_t lineNumber, charcount\_t \*outCharPosition, charcount\_t \*outByteOffset)

{

Assert(outCharPosition != nullptr && outByteOffset != nullptr);

\*outCharPosition = this->m\_lineOffsetCache->GetCharacterOffsetForLine(lineNumber, outByteOffset);

}

void TrackDeferredFunction(Js::LocalFunctionId functionID, Js::ParseableFunctionInfo \*function);

void StopTrackingDeferredFunction(Js::LocalFunctionId functionID);

template <class Fn>

void UndeferGlobalFunctions(Fn fn)

{

if (this->m\_scriptContext->DoUndeferGlobalFunctions())

{

Assert(m\_deferredFunctionsInitialized);

if (m\_deferredFunctionsDictionary == nullptr)

{

return;

}

DeferredFunctionsDictionary \*tmp = this->m\_deferredFunctionsDictionary;

this->m\_deferredFunctionsDictionary = nullptr;

tmp->MapAndRemoveIf(fn);

}

}

ISourceHolder\* GetSourceHolder() const

{

return sourceHolder;

}

bool IsCesu8() const

{

return this->m\_isCesu8;

}

void SetCallerUtf8SourceInfo(Utf8SourceInfo\* callerUtf8SourceInfo);

Utf8SourceInfo\* GetCallerUtf8SourceInfo() const;

bool GetDebugDocumentName(BSTR \* sourceName);

private:

charcount\_t m\_cchLength; // The number of characters encoded in m\_utf8Source.

ISourceHolder\* sourceHolder;

union

{

BYTE\* m\_pHostBuffer; // Pointer to a host source buffer (null unless this is host code that we need to free)

Utf8SourceInfo const\* m\_pOriginalSourceInfo; // Pointer to source info with original source text, created during cloning

};

FunctionBodyDictionary\* functionBodyDictionary;

DeferredFunctionsDictionary\* m\_deferredFunctionsDictionary;

DebugDocument\* m\_debugDocument;

const SRCINFO\* m\_srcInfo;

DWORD\_PTR m\_secondaryHostSourceContext;

LPCUTF8 debugModeSource;

size\_t debugModeSourceLength;

ScriptContext\* const m\_scriptContext; // Pointer to ScriptContext under which this source info was created

// Line offset cache used for quickly finding line/column offsets.

JsUtil::LineOffsetCache<Recycler>\* m\_lineOffsetCache;

// Utf8SourceInfo of the caller, used for mapping eval/new Function node to its caller node for debugger

Utf8SourceInfo\* callerUtf8SourceInfo;

bool m\_deferredFunctionsInitialized : 1;

bool m\_isCesu8 : 1;

bool m\_hasHostBuffer : 1;

bool m\_isLibraryCode : 1; // true, the current source belongs to the internal library code. Used for debug purpose to not show in debugger

bool m\_isXDomain : 1;

// we found that m\_isXDomain could cause regression without CORS, so the new flag is just for callee.caller in window.onerror

bool m\_isXDomainString : 1;

bool debugModeSourceIsEmpty : 1;

uint m\_sourceInfoId;

// Various flags preserved for Edit-and-Continue re-compile purpose

ULONG parseFlags;

ULONG byteCodeGenerationFlags;

Utf8SourceInfo(ISourceHolder \*sourceHolder, int32 cchLength, SRCINFO const\* srcInfo, DWORD\_PTR secondaryHostSourceContext, ScriptContext\* scriptContext);

};

}

template <>

struct DefaultComparer<Js::Utf8SourceInfo\*>

{

\_\_inline static bool Equals(Js::Utf8SourceInfo\* const& x, Js::Utf8SourceInfo\* const& y)

{

return Js::Utf8SourceInfo::StaticEquals(x, y);

}

\_\_inline static hash\_t GetHashCode(Js::Utf8SourceInfo\* const& s)

{

return Js::Utf8SourceInfo::StaticGetHashCode(s);

}

};

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

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

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

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

#include "RuntimeBasePch.h"

#ifdef ENABLE\_FOUNDATION\_OBJECT

using namespace Windows::Foundation::Diagnostics;

#define IfFailReturnNULL(op) \

if (FAILED(hr=(op))) { return NULL; }; \

namespace Js

{

\_\_inline DelayLoadWinRtString\* WindowsFoundationAdapter::GetWinRtStringLibrary(\_In\_ ScriptContext\* scriptContext)

{

return scriptContext->GetThreadContext()->GetWinRTStringLibrary();

}

\_\_inline DelayLoadWinRtFoundation\* WindowsFoundationAdapter::GetWinRtFoundationLibrary(\_In\_ ScriptContext\* scriptContext)

{

return scriptContext->GetThreadContext()->GetWinRtFoundationLibrary();

}

IActivationFactory\* WindowsFoundationAdapter::GetAsyncCausalityTracerActivationFactory(\_In\_ ScriptContext\* scriptContext)

{

if (! asyncCausalityTracerActivationFactory)

{

HRESULT hr;

HSTRING hString;

HSTRING\_HEADER hStringHdr;

LPCWSTR factoryName = L"Windows.Foundation.Diagnostics.AsyncCausalityTracer";

UINT32 factoryNameLen = \_countof(L"Windows.Foundation.Diagnostics.AsyncCausalityTracer") - 1;

IID factoryIID = \_\_uuidof(IAsyncCausalityTracerStatics);

IfFailReturnNULL(GetWinRtStringLibrary(scriptContext)->WindowsCreateStringReference(factoryName, factoryNameLen, &hStringHdr, &hString));

IfFailReturnNULL(GetWinRtFoundationLibrary(scriptContext)->RoGetActivationFactory(hString, factoryIID, &asyncCausalityTracerActivationFactory));

Assert(asyncCausalityTracerActivationFactory != NULL);

}

return asyncCausalityTracerActivationFactory;

}

IAsyncCausalityTracerStatics\* WindowsFoundationAdapter::GetAsyncCausalityTracerStatics(\_In\_ ScriptContext\* scriptContext)

{

if (! asyncCausalityTracerStatics)

{

IActivationFactory\* factory = GetAsyncCausalityTracerActivationFactory(scriptContext);

if (!factory)

{

return NULL;

}

HRESULT hr;

IfFailReturnNULL(factory->QueryInterface(\_\_uuidof(IAsyncCausalityTracerStatics), reinterpret\_cast<void\*\*>(&asyncCausalityTracerStatics)));

Assert(asyncCausalityTracerStatics != NULL);

}

return asyncCausalityTracerStatics;

}

HRESULT WindowsFoundationAdapter::TraceOperationCreation(

\_In\_ ScriptContext\* scriptContext,

\_In\_ INT traceLevel,

\_In\_ INT source,

\_In\_ GUID platformId,

\_In\_ UINT64 operationId,

\_In\_z\_ PCWSTR operationName,

\_In\_ UINT64 relatedContext)

{

HRESULT hr;

HSTRING hString;

HSTRING\_HEADER hStringHdr;

Assert(traceLevel <= CausalityTraceLevel::CausalityTraceLevel\_Verbose && traceLevel >= CausalityTraceLevel\_Required);

Assert(source <= CausalitySource::CausalitySource\_System && source >= CausalitySource\_Application);

size\_t operationNameLen = wcslen(operationName);

if (operationNameLen > UINT\_MAX)

{

return E\_OUTOFMEMORY;

}

IFFAILRET(GetWinRtStringLibrary(scriptContext)->WindowsCreateStringReference(operationName, static\_cast<UINT32>(operationNameLen), &hStringHdr, &hString));

IAsyncCausalityTracerStatics\* tracerStatics = GetAsyncCausalityTracerStatics(scriptContext);

if (!tracerStatics)

{

return E\_UNEXPECTED;

}

return tracerStatics->TraceOperationCreation((CausalityTraceLevel)traceLevel, (CausalitySource)source, platformId, operationId, hString, relatedContext);

}

HRESULT WindowsFoundationAdapter::TraceOperationCompletion(

\_In\_ ScriptContext\* scriptContext,

\_In\_ INT traceLevel,

\_In\_ INT source,

\_In\_ GUID platformId,

\_In\_ UINT64 operationId,

\_In\_ INT status)

{

Assert(traceLevel <= CausalityTraceLevel::CausalityTraceLevel\_Verbose && traceLevel >= CausalityTraceLevel\_Required);

Assert(source <= CausalitySource::CausalitySource\_System && source >= CausalitySource\_Application);

Assert(status <= (INT)AsyncStatus::Error && status >= (INT)AsyncStatus::Started);

IAsyncCausalityTracerStatics\* tracerStatics = GetAsyncCausalityTracerStatics(scriptContext);

if (!tracerStatics)

{

return E\_UNEXPECTED;

}

return tracerStatics->TraceOperationCompletion((CausalityTraceLevel)traceLevel, (CausalitySource)source, platformId, operationId, (AsyncStatus)status);

}

HRESULT WindowsFoundationAdapter::TraceOperationRelation(

\_In\_ ScriptContext\* scriptContext,

\_In\_ INT traceLevel,

\_In\_ INT source,

\_In\_ GUID platformId,

\_In\_ UINT64 operationId,

\_In\_ INT relation)

{

Assert(traceLevel <= CausalityTraceLevel::CausalityTraceLevel\_Verbose && traceLevel >= CausalityTraceLevel\_Required);

Assert(source <= CausalitySource::CausalitySource\_System && source >= CausalitySource\_Application);

Assert(relation <= CausalityRelation::CausalityRelation\_Error && relation >= CausalityRelation\_AssignDelegate);

IAsyncCausalityTracerStatics\* tracerStatics = GetAsyncCausalityTracerStatics(scriptContext);

if (!tracerStatics)

{

return E\_UNEXPECTED;

}

return tracerStatics->TraceOperationRelation((CausalityTraceLevel)traceLevel, (CausalitySource)source, platformId, operationId, (CausalityRelation)relation);

}

HRESULT WindowsFoundationAdapter::TraceSynchronousWorkStart(

\_In\_ ScriptContext\* scriptContext,

\_In\_ INT traceLevel,

\_In\_ INT source,

\_In\_ GUID platformId,

\_In\_ UINT64 operationId,

\_In\_ INT work)

{

Assert(traceLevel <= CausalityTraceLevel::CausalityTraceLevel\_Verbose && traceLevel >= CausalityTraceLevel\_Required);

Assert(source <= CausalitySource::CausalitySource\_System && source >= CausalitySource\_Application);

Assert(work <= CausalitySynchronousWork::CausalitySynchronousWork\_Execution && work >= CausalitySynchronousWork\_CompletionNotification);

IAsyncCausalityTracerStatics\* tracerStatics = GetAsyncCausalityTracerStatics(scriptContext);

if (!tracerStatics)

{

return E\_UNEXPECTED;

}

return tracerStatics->TraceSynchronousWorkStart((CausalityTraceLevel)traceLevel, (CausalitySource)source, platformId, operationId, (CausalitySynchronousWork)work);

}

HRESULT WindowsFoundationAdapter::TraceSynchronousWorkCompletion(

\_In\_ ScriptContext\* scriptContext,

\_In\_ INT traceLevel,

\_In\_ INT source,

\_In\_ INT work)

{

Assert(traceLevel <= CausalityTraceLevel::CausalityTraceLevel\_Verbose && traceLevel >= CausalityTraceLevel\_Required);

Assert(source <= CausalitySource::CausalitySource\_System && source >= CausalitySource\_Application);

Assert(work <= CausalitySynchronousWork::CausalitySynchronousWork\_Execution && work >= CausalitySynchronousWork\_CompletionNotification);

IAsyncCausalityTracerStatics\* tracerStatics = GetAsyncCausalityTracerStatics(scriptContext);

if (!tracerStatics)

{

return E\_UNEXPECTED;

}

return tracerStatics->TraceSynchronousWorkCompletion((CausalityTraceLevel)traceLevel, (CausalitySource)source, (CausalitySynchronousWork)work);

}

}

#endif

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

// 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 ENABLE\_FOUNDATION\_OBJECT

#include "Windows.Foundation.Diagnostics.h"

using namespace Windows::Foundation::Diagnostics;

namespace Js

{

class WindowsFoundationAdapter

{

private:

AutoCOMPtr<IActivationFactory> asyncCausalityTracerActivationFactory;

AutoCOMPtr<IAsyncCausalityTracerStatics> asyncCausalityTracerStatics;

IActivationFactory\* GetAsyncCausalityTracerActivationFactory(\_In\_ ScriptContext\* scriptContext);

IAsyncCausalityTracerStatics\* GetAsyncCausalityTracerStatics(\_In\_ ScriptContext\* scriptContext);

DelayLoadWinRtString\* GetWinRtStringLibrary(\_In\_ ScriptContext\* scriptContext);

DelayLoadWinRtFoundation\* GetWinRtFoundationLibrary(\_In\_ ScriptContext\* scriptContext);

public:

HRESULT TraceOperationCreation(

\_In\_ ScriptContext\* scriptContext,

\_In\_ INT traceLevel,

\_In\_ INT source,

\_In\_ GUID platformId,

\_In\_ UINT64 operationId,

\_In\_z\_ PCWSTR operationName,

\_In\_ UINT64 relatedContext);

HRESULT TraceOperationCompletion(

\_In\_ ScriptContext\* scriptContext,

\_In\_ INT traceLevel,

\_In\_ INT source,

\_In\_ GUID platformId,

\_In\_ UINT64 operationId,

\_In\_ INT status);

HRESULT TraceOperationRelation(

\_In\_ ScriptContext\* scriptContext,

\_In\_ INT traceLevel,

\_In\_ INT source,

\_In\_ GUID platformId,

\_In\_ UINT64 operationId,

\_In\_ INT relation);

HRESULT TraceSynchronousWorkStart(

\_In\_ ScriptContext\* scriptContext,

\_In\_ INT traceLevel,

\_In\_ INT source,

\_In\_ GUID platformId,

\_In\_ UINT64 operationId,

\_In\_ INT work);

HRESULT TraceSynchronousWorkCompletion(

\_In\_ ScriptContext\* scriptContext,

\_In\_ INT traceLevel,

\_In\_ INT source,

\_In\_ INT work);

};

}

#endif

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

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

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

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

#include "RuntimeBasePch.h"

#if defined(ENABLE\_INTL\_OBJECT) || defined(ENABLE\_ES6\_CHAR\_CLASSIFIER)

#include "strsafe.h"

#define \_\_WRL\_ASSERT\_\_(cond) Assert(cond)

#include <wrl\implements.h>

#ifdef NTBUILD

using namespace Windows::Globalization;

using namespace Windows::Foundation::Collections;

#else

using namespace ABI::Windows::Globalization;

using namespace ABI::Windows::Foundation::Collections;

#endif

#define IfFailThrowHr(op) \

if (FAILED(hr=(op))) \

{ \

JavascriptError::MapAndThrowError(scriptContext, hr);\

} \

#define IfNullReturnError(EXPR, ERROR) do { if (!(EXPR)) { return (ERROR); } } while(FALSE)

#define IfFailedReturn(EXPR) do { hr = (EXPR); if (FAILED(hr)) { return hr; }} while(FALSE)

#define IfFailedSetErrorCodeAndReturn(EXPR, hrVariable) do { hr = (EXPR); if (FAILED(hr)) { hrVariable = hr; return hr; }} while(FALSE)

#define IfFailedGoLabel(expr, label) if (FAILED(expr)) { goto label; }

#define IfFailedGo(expr) IfFailedGoLabel(expr, LReturn)

namespace Js

{

#ifdef ENABLE\_INTL\_OBJECT

class HSTRINGIterator : public Microsoft::WRL::RuntimeClass<IIterator<HSTRING>>

{

HSTRING \*items;

uint32 length;

boolean hasMore;

uint32 currentPosition;

public:

HRESULT RuntimeClassInitialize(HSTRING \*items, uint32 length)

{

this->items = items;

this->currentPosition = 0;

this->length = length;

this->hasMore = currentPosition < this->length;

return S\_OK;

}

~HSTRINGIterator()

{

}

// IIterator

IFACEMETHODIMP get\_Current(\_Out\_ HSTRING \*current)

{

if (current != nullptr)

{

if (hasMore)

{

return WindowsDuplicateString(items[currentPosition], current);

}

else

{

\*current = nullptr;

}

}

return E\_BOUNDS;

}

IFACEMETHODIMP get\_HasCurrent(\_Out\_ boolean \*hasCurrent)

{

if (hasCurrent != nullptr)

{

\*hasCurrent = hasMore;

}

return S\_OK;

}

IFACEMETHODIMP MoveNext(\_Out\_opt\_ boolean \*hasCurrent) sealed

{

this->currentPosition++;

this->hasMore = this->currentPosition < this->length;

if (hasCurrent != nullptr)

{

\*hasCurrent = hasMore;

}

return S\_OK;

}

IFACEMETHODIMP GetMany(\_In\_ unsigned capacity,

\_Out\_writes\_to\_(capacity,\*actual) HSTRING \*value,

\_Out\_ unsigned \*actual)

{

uint count = 0;

while (this->hasMore)

{

if (count == capacity)

{

break;

}

if (value != nullptr)

{

get\_Current(value + count);

}

count ++;

this->MoveNext(nullptr);

}

if (actual != nullptr)

{

\*actual = count;

}

return S\_OK;

}

IFACEMETHOD(GetRuntimeClassName)(\_Out\_ HSTRING\* runtimeName) sealed

{

\*runtimeName = nullptr;

HRESULT hr = S\_OK;

const wchar\_t \*name = L"Js.HSTRINGIterator";

hr = WindowsCreateString(name, static\_cast<UINT32>(wcslen(name)), runtimeName);

return hr;

}

IFACEMETHOD(GetTrustLevel)(\_Out\_ TrustLevel\* trustLvl)

{

\*trustLvl = BaseTrust;

return S\_OK;

}

IFACEMETHOD(GetIids)(\_Out\_ ULONG \*iidCount, \_Outptr\_result\_buffer\_(\*iidCount) IID \*\*)

{

iidCount;

return E\_NOTIMPL;

}

};

class HSTRINGIterable : public Microsoft::WRL::RuntimeClass<IIterable<HSTRING>>

{

HSTRING \*items;

uint32 length;

public:

HRESULT RuntimeClassInitialize(HSTRING \*string, uint32 length)

{

this->items = new HSTRING[length];

if (this->items == nullptr)

{

return E\_OUTOFMEMORY;

}

for(uint32 i = 0; i < length; i++)

{

this->items[i] = string[i];

}

this->length = length;

return S\_OK;

}

~HSTRINGIterable()

{

if(this->items != nullptr)

{

delete [] items;

}

}

IFACEMETHODIMP First(\_Outptr\_result\_maybenull\_ IIterator<HSTRING> \*\*first)

{

return Microsoft::WRL::MakeAndInitialize<HSTRINGIterator>(first, this->items, this->length);

}

IFACEMETHOD(GetRuntimeClassName)(\_Out\_ HSTRING\* runtimeName) sealed

{

\*runtimeName = nullptr;

HRESULT hr = S\_OK;

// Return type that does not exist in metadata

const wchar\_t \*name = L"Js.HSTRINGIterable";

hr = WindowsCreateString(name, static\_cast<UINT32>(wcslen(name)), runtimeName);

return hr;

}

IFACEMETHOD(GetTrustLevel)(\_Out\_ TrustLevel\* trustLvl)

{

\*trustLvl = BaseTrust;

return S\_OK;

}

IFACEMETHOD(GetIids)(\_Out\_ ULONG \*iidCount, \_Outptr\_result\_buffer\_(\*iidCount) IID \*\*)

{

iidCount;

return E\_NOTIMPL;

}

};

#endif

\_\_inline DelayLoadWindowsGlobalization\* WindowsGlobalizationAdapter::GetWindowsGlobalizationLibrary(\_In\_ ScriptContext\* scriptContext)

{

return this->GetWindowsGlobalizationLibrary(scriptContext->GetThreadContext());

}

\_\_inline DelayLoadWindowsGlobalization\* WindowsGlobalizationAdapter::GetWindowsGlobalizationLibrary(\_In\_ ThreadContext\* threadContext)

{

return threadContext->GetWindowsGlobalizationLibrary();

}

template<typename T>

HRESULT WindowsGlobalizationAdapter::GetActivationFactory(DelayLoadWindowsGlobalization \*delayLoadLibrary, LPCWSTR factoryName, T\*\* instance)

{

\*instance = nullptr;

AutoCOMPtr<IActivationFactory> factory;

HSTRING hString;

HSTRING\_HEADER hStringHdr;

HRESULT hr;

// factoryName will never get truncated as the name of interfaces in Windows.globalization are not that long.

IfFailedReturn(delayLoadLibrary->WindowsCreateStringReference(factoryName, static\_cast<UINT32>(wcslen(factoryName)), &hStringHdr, &hString));

AnalysisAssert(hString);

IfFailedReturn(delayLoadLibrary->DllGetActivationFactory(hString, &factory));

return factory->QueryInterface(\_\_uuidof(T), reinterpret\_cast<void\*\*>(instance));

}

#ifdef ENABLE\_INTL\_OBJECT

HRESULT WindowsGlobalizationAdapter::EnsureCommonObjectsInitialized(DelayLoadWindowsGlobalization \*library)

{

HRESULT hr = S\_OK;

if (initializedCommonGlobObjects)

{

AssertMsg(hrForCommonGlobObjectsInit == S\_OK, "If IntlGlobObjects are initialized, we should be returning S\_OK.");

return hrForCommonGlobObjectsInit;

}

else if (hrForCommonGlobObjectsInit != S\_OK)

{

return hrForCommonGlobObjectsInit;

}

IfFailedSetErrorCodeAndReturn(GetActivationFactory(library, RuntimeClass\_Windows\_Globalization\_Language, &languageFactory), hrForCommonGlobObjectsInit);

IfFailedSetErrorCodeAndReturn(GetActivationFactory(library, RuntimeClass\_Windows\_Globalization\_Language, &languageStatics), hrForCommonGlobObjectsInit);

IfFailedSetErrorCodeAndReturn(GetActivationFactory(library, RuntimeClass\_Windows\_Globalization\_DateTimeFormatting\_DateTimeFormatter, &dateTimeFormatterFactory), hrForCommonGlobObjectsInit);

hrForCommonGlobObjectsInit = S\_OK;

initializedCommonGlobObjects = true;

return hr;

}

HRESULT WindowsGlobalizationAdapter::EnsureDateTimeFormatObjectsInitialized(DelayLoadWindowsGlobalization \*library)

{

HRESULT hr = S\_OK;

if (initializedDateTimeFormatObjects)

{

AssertMsg(hrForDateTimeFormatObjectsInit == S\_OK, "If DateTimeFormatObjects are initialized, we should be returning S\_OK.");

return hrForDateTimeFormatObjectsInit;

}

else if (hrForDateTimeFormatObjectsInit != S\_OK)

{

return hrForDateTimeFormatObjectsInit;

}

IfFailedSetErrorCodeAndReturn(GetActivationFactory(library, RuntimeClass\_Windows\_Globalization\_Calendar, &calendarFactory), hrForDateTimeFormatObjectsInit);

IfFailedSetErrorCodeAndReturn(this->CreateTimeZoneOnCalendar(library, &defaultTimeZoneCalendar), hrForDateTimeFormatObjectsInit);

IfFailedSetErrorCodeAndReturn(this->CreateTimeZoneOnCalendar(library, &timeZoneCalendar), hrForDateTimeFormatObjectsInit);

hrForDateTimeFormatObjectsInit = S\_OK;

initializedDateTimeFormatObjects = true;

return hr;

}

HRESULT WindowsGlobalizationAdapter::EnsureNumberFormatObjectsInitialized(DelayLoadWindowsGlobalization \*library)

{

HRESULT hr = S\_OK;

if (initializedNumberFormatObjects)

{

AssertMsg(hrForNumberFormatObjectsInit == S\_OK, "If NumberFormatObjects are initialized, we should be returning S\_OK.");

return hrForNumberFormatObjectsInit;

}

else if (hrForNumberFormatObjectsInit != S\_OK)

{

return hrForNumberFormatObjectsInit;

}

IfFailedSetErrorCodeAndReturn(GetActivationFactory(library, RuntimeClass\_Windows\_Globalization\_NumberFormatting\_CurrencyFormatter, &currencyFormatterFactory), hrForNumberFormatObjectsInit);

IfFailedSetErrorCodeAndReturn(GetActivationFactory(library, RuntimeClass\_Windows\_Globalization\_NumberFormatting\_DecimalFormatter, &decimalFormatterFactory), hrForNumberFormatObjectsInit);

IfFailedSetErrorCodeAndReturn(GetActivationFactory(library, RuntimeClass\_Windows\_Globalization\_NumberFormatting\_PercentFormatter, &percentFormatterFactory), hrForNumberFormatObjectsInit);

IfFailedSetErrorCodeAndReturn(GetActivationFactory(library, RuntimeClass\_Windows\_Globalization\_NumberFormatting\_SignificantDigitsNumberRounder, &significantDigitsRounderActivationFactory), hrForNumberFormatObjectsInit);

IfFailedSetErrorCodeAndReturn(GetActivationFactory(library, RuntimeClass\_Windows\_Globalization\_NumberFormatting\_IncrementNumberRounder, &incrementNumberRounderActivationFactory), hrForNumberFormatObjectsInit);

hrForNumberFormatObjectsInit = S\_OK;

initializedNumberFormatObjects = true;

return hr;

}

#endif

HRESULT WindowsGlobalizationAdapter::EnsureDataTextObjectsInitialized(DelayLoadWindowsGlobalization \*library)

{

HRESULT hr = S\_OK;

if (initializedCharClassifierObjects)

{

AssertMsg(hrForCharClassifierObjectsInit == S\_OK, "If DataTextObjects are initialized, we should be returning S\_OK.");

return hrForCharClassifierObjectsInit;

}

else if (hrForCharClassifierObjectsInit != S\_OK)

{

return hrForCharClassifierObjectsInit;

}

IfFailedSetErrorCodeAndReturn(GetActivationFactory(library, RuntimeClass\_Windows\_Data\_Text\_UnicodeCharacters, &unicodeStatics), hrForCharClassifierObjectsInit);

hrForCharClassifierObjectsInit = S\_OK;

initializedCharClassifierObjects = true;

return hr;

}

#ifdef ENABLE\_INTL\_OBJECT

HRESULT WindowsGlobalizationAdapter::CreateLanguage(\_In\_ ScriptContext\* scriptContext, \_In\_z\_ PCWSTR languageTag, ILanguage\*\* language)

{

HRESULT hr = S\_OK;

HSTRING hString;

HSTRING\_HEADER hStringHdr;

// OK for languageTag to get truncated as it would pass incomplete languageTag below which

// will be rejected by globalization dll

IfFailedReturn(GetWindowsGlobalizationLibrary(scriptContext)->WindowsCreateStringReference(languageTag, static\_cast<UINT32>(wcslen(languageTag)), &hStringHdr, &hString));

AnalysisAssert(hString);

IfFailedReturn(this->languageFactory->CreateLanguage(hString, language));

return hr;

}

boolean WindowsGlobalizationAdapter::IsWellFormedLanguageTag(\_In\_ ScriptContext\* scriptContext, \_In\_z\_ PCWSTR languageTag)

{

boolean retVal;

HRESULT hr;

HSTRING hString;

HSTRING\_HEADER hStringHdr;

// OK for languageTag to get truncated as it would pass incomplete languageTag below which

// will be rejected by globalization dll

IfFailThrowHr(GetWindowsGlobalizationLibrary(scriptContext)->WindowsCreateStringReference(languageTag, static\_cast<UINT32>(wcslen(languageTag)), &hStringHdr, &hString));

AnalysisAssert(hString);

IfFailThrowHr(this->languageStatics->IsWellFormed(hString, &retVal));

return retVal;

}

// OK for timeZoneId to get truncated as it would pass incomplete timeZoneId below which

// will be rejected by globalization dll

HRESULT WindowsGlobalizationAdapter::NormalizeLanguageTag(\_In\_ ScriptContext\* scriptContext, \_In\_z\_ PCWSTR languageTag, HSTRING \*result)

{

HRESULT hr;

AutoCOMPtr<ILanguage> language;

IfFailedReturn(CreateLanguage(scriptContext, languageTag, &language));

IfFailedReturn(language->get\_LanguageTag(result));

return hr;

}

boolean WindowsGlobalizationAdapter::ValidateAndCanonicalizeTimeZone(\_In\_ ScriptContext\* scriptContext, \_In\_z\_ PCWSTR timeZoneId, HSTRING \*result)

{

HRESULT hr = S\_OK;

HSTRING timeZone;

HSTRING\_HEADER timeZoneHeader;

// Construct HSTRING of timeZoneId passed

IfFailThrowHr(GetWindowsGlobalizationLibrary(scriptContext)->WindowsCreateStringReference(timeZoneId, static\_cast<UINT32>(wcslen(timeZoneId)), &timeZoneHeader, &timeZone));

Assert(timeZone);

// The warning is timeZone could be '0'. This is valid scenario and in that case, ChangeTimeZone() would

// return error HR in which case we will throw.

#pragma warning(suppress:6387)

// ChangeTimeZone should fail if this is not a valid time zone

hr = timeZoneCalendar->ChangeTimeZone(timeZone);

if (hr != S\_OK)

{

return false;

}

// Retrieve canonicalize timeZone name

IfFailThrowHr(timeZoneCalendar->GetTimeZone(result));

return true;

}

void WindowsGlobalizationAdapter::GetDefaultTimeZoneId(\_In\_ ScriptContext\* scriptContext, HSTRING \*result)

{

HRESULT hr = S\_OK;

IfFailThrowHr(defaultTimeZoneCalendar->GetTimeZone(result));

}

HRESULT WindowsGlobalizationAdapter::CreateTimeZoneOnCalendar(\_In\_ DelayLoadWindowsGlobalization \*library, \_\_out::ITimeZoneOnCalendar\*\* result)

{

AutoCOMPtr<::ICalendar> calendar;

HRESULT hr = S\_OK;

// initialize hard-coded default languages

AutoArrayPtr<HSTRING> arr(HeapNewArray(HSTRING, 1), 1);

AutoArrayPtr<HSTRING\_HEADER> headers(HeapNewArray(HSTRING\_HEADER, 1), 1);

IfFailedReturn(library->WindowsCreateStringReference(L"en-US", static\_cast<UINT32>(wcslen(L"en-US")), (headers), (arr)));

Microsoft::WRL::ComPtr<IIterable<HSTRING>> defaultLanguages;

IfFailedReturn(Microsoft::WRL::MakeAndInitialize<HSTRINGIterable>(&defaultLanguages, arr, 1));

// Create calendar object

IfFailedReturn(this->calendarFactory->CreateCalendarDefaultCalendarAndClock(defaultLanguages.Get(), &calendar));

// Get ITimeZoneOnCalendar part of calendar object

IfFailedReturn(calendar->QueryInterface(\_\_uuidof(::ITimeZoneOnCalendar), reinterpret\_cast<void\*\*>(result)));

return hr;

}

void WindowsGlobalizationAdapter::ClearTimeZoneCalendars()

{

if (this->timeZoneCalendar)

{

this->timeZoneCalendar.Detach()->Release();

}

if (this->defaultTimeZoneCalendar)

{

this->defaultTimeZoneCalendar.Detach()->Release();

}

}

HRESULT WindowsGlobalizationAdapter::CreateCurrencyFormatterCode(\_In\_ ScriptContext\* scriptContext, \_In\_z\_ PCWSTR currencyCode, NumberFormatting::ICurrencyFormatter\*\* currencyFormatter)

{

HRESULT hr;

HSTRING hString;

HSTRING\_HEADER hStringHdr;

// OK for currencyCode to get truncated as it would pass incomplete currencyCode below which

// will be rejected by globalization dll

IfFailedReturn(GetWindowsGlobalizationLibrary(scriptContext)->WindowsCreateStringReference(currencyCode, static\_cast<UINT32>(wcslen(currencyCode)), &hStringHdr, &hString));

AnalysisAssert(hString);

IfFailedReturn(this->currencyFormatterFactory->CreateCurrencyFormatterCode(hString, currencyFormatter));

return hr;

}

HRESULT WindowsGlobalizationAdapter::CreateCurrencyFormatter(\_In\_ ScriptContext\* scriptContext, PCWSTR\* localeStrings, uint32 numLocaleStrings, \_In\_z\_ PCWSTR currencyCode, NumberFormatting::ICurrencyFormatter\*\* currencyFormatter)

{

HRESULT hr;

HSTRING hString;

HSTRING\_HEADER hStringHdr;

AutoArrayPtr<HSTRING> arr(HeapNewArray(HSTRING, numLocaleStrings), numLocaleStrings);

AutoArrayPtr<HSTRING\_HEADER> headers(HeapNewArray(HSTRING\_HEADER, numLocaleStrings), numLocaleStrings);

for(uint32 i = 0; i< numLocaleStrings; i++)

{

// OK for localeString to get truncated as it would pass incomplete localeString below which

// will be rejected by globalization dll.

IfFailedReturn(GetWindowsGlobalizationLibrary(scriptContext)->WindowsCreateStringReference(localeStrings[i], static\_cast<UINT32>(wcslen(localeStrings[i])), (headers + i), (arr + i)));

}

Microsoft::WRL::ComPtr<IIterable<HSTRING>> languages(nullptr);

IfFailedReturn(Microsoft::WRL::MakeAndInitialize<HSTRINGIterable>(&languages, arr, numLocaleStrings));

HSTRING geoString;

HSTRING\_HEADER geoStringHeader;

IfFailedReturn(GetWindowsGlobalizationLibrary(scriptContext)->WindowsCreateStringReference(L"ZZ", 2, &geoStringHeader, &geoString));

AnalysisAssert(geoString);

IfFailedReturn(GetWindowsGlobalizationLibrary(scriptContext)->WindowsCreateStringReference(currencyCode, static\_cast<UINT32>(wcslen(currencyCode)), &hStringHdr, &hString));

AnalysisAssert(hString);

IfFailedReturn(this->currencyFormatterFactory->CreateCurrencyFormatterCodeContext(hString, languages.Get(), geoString, currencyFormatter));

return hr;

}

HRESULT WindowsGlobalizationAdapter::CreateNumberFormatter(\_In\_ ScriptContext\* scriptContext, PCWSTR\* localeStrings, uint32 numLocaleStrings, NumberFormatting::INumberFormatter\*\* numberFormatter)

{

HRESULT hr = S\_OK;

AutoArrayPtr<HSTRING> arr(HeapNewArray(HSTRING, numLocaleStrings), numLocaleStrings);

AutoArrayPtr<HSTRING\_HEADER> headers(HeapNewArray(HSTRING\_HEADER, numLocaleStrings), numLocaleStrings);

for(uint32 i = 0; i< numLocaleStrings; i++)

{

IfFailedReturn(GetWindowsGlobalizationLibrary(scriptContext)->WindowsCreateStringReference(localeStrings[i], static\_cast<UINT32>(wcslen(localeStrings[i])), (headers + i), (arr + i)));

}

Microsoft::WRL::ComPtr<IIterable<HSTRING>> languages(nullptr);

IfFailedReturn(Microsoft::WRL::MakeAndInitialize<HSTRINGIterable>(&languages, arr, numLocaleStrings));

HSTRING geoString;

HSTRING\_HEADER geoStringHeader;

IfFailedReturn(GetWindowsGlobalizationLibrary(scriptContext)->WindowsCreateStringReference(L"ZZ", 2, &geoStringHeader, &geoString));

AnalysisAssert(geoString);

IfFailedReturn(this->decimalFormatterFactory->CreateDecimalFormatter(languages.Get(), geoString, numberFormatter));

return hr;

}

HRESULT WindowsGlobalizationAdapter::CreatePercentFormatter(\_In\_ ScriptContext\* scriptContext, PCWSTR\* localeStrings, uint32 numLocaleStrings, NumberFormatting::INumberFormatter\*\* numberFormatter)

{

HRESULT hr = S\_OK;

AutoArrayPtr<HSTRING> arr(HeapNewArray(HSTRING, numLocaleStrings), numLocaleStrings);

AutoArrayPtr<HSTRING\_HEADER> headers(HeapNewArray(HSTRING\_HEADER, numLocaleStrings), numLocaleStrings);

for(uint32 i = 0; i< numLocaleStrings; i++)

{

// OK for localeString to get truncated as it would pass incomplete localeString below which

// will be rejected by globalization dll.

IfFailedReturn(GetWindowsGlobalizationLibrary(scriptContext)->WindowsCreateStringReference(localeStrings[i], static\_cast<UINT32>(wcslen(localeStrings[i])), (headers + i), (arr + i)));

}

Microsoft::WRL::ComPtr<IIterable<HSTRING>> languages(nullptr);

IfFailedReturn(Microsoft::WRL::MakeAndInitialize<HSTRINGIterable>(&languages, arr, numLocaleStrings));

HSTRING geoString;

HSTRING\_HEADER geoStringHeader;

IfFailedReturn(GetWindowsGlobalizationLibrary(scriptContext)->WindowsCreateStringReference(L"ZZ", 2, &geoStringHeader, &geoString));

AnalysisAssert(geoString);

IfFailedReturn(this->percentFormatterFactory->CreatePercentFormatter(languages.Get(), geoString, numberFormatter));

return hr;

}

HRESULT WindowsGlobalizationAdapter::CreateDateTimeFormatter(\_In\_ ScriptContext\* scriptContext, \_In\_z\_ PCWSTR formatString, \_In\_z\_ PCWSTR\* localeStrings,

uint32 numLocaleStrings, \_In\_opt\_z\_ PCWSTR calendar, \_In\_opt\_z\_ PCWSTR clock, \_Out\_ DateTimeFormatting::IDateTimeFormatter\*\* result)

{

HRESULT hr = S\_OK;

if(numLocaleStrings == 0) return E\_INVALIDARG;

AnalysisAssert((calendar == nullptr && clock == nullptr) || (calendar != nullptr && clock != nullptr));

HSTRING fsHString;

HSTRING\_HEADER fsHStringHdr;

// OK for formatString to get truncated as it would pass incomplete formatString below which

// will be rejected by globalization dll.

IfFailedReturn(GetWindowsGlobalizationLibrary(scriptContext)->WindowsCreateStringReference(formatString, static\_cast<UINT32>(wcslen(formatString)), &fsHStringHdr, &fsHString));

AnalysisAssert(fsHString);

AutoArrayPtr<HSTRING> arr(HeapNewArray(HSTRING, numLocaleStrings), numLocaleStrings);

AutoArrayPtr<HSTRING\_HEADER> headers(HeapNewArray(HSTRING\_HEADER, numLocaleStrings), numLocaleStrings);

for(uint32 i = 0; i< numLocaleStrings; i++)

{

// OK for localeString to get truncated as it would pass incomplete localeString below which

// will be rejected by globalization dll.

IfFailedReturn(GetWindowsGlobalizationLibrary(scriptContext)->WindowsCreateStringReference(localeStrings[i], static\_cast<UINT32>(wcslen(localeStrings[i])), (headers + i), (arr + i)));

}

Microsoft::WRL::ComPtr<IIterable<HSTRING>> languages(nullptr);

IfFailedReturn(Microsoft::WRL::MakeAndInitialize<HSTRINGIterable>(&languages, arr, numLocaleStrings));

if(clock == nullptr)

{

IfFailedReturn(this->dateTimeFormatterFactory->CreateDateTimeFormatterLanguages(fsHString, languages.Get(), result));

}

else

{

HSTRING geoString;

HSTRING\_HEADER geoStringHeader;

HSTRING calString;

HSTRING\_HEADER calStringHeader;

HSTRING clockString;

HSTRING\_HEADER clockStringHeader;

IfFailedReturn(GetWindowsGlobalizationLibrary(scriptContext)->WindowsCreateStringReference(L"ZZ", 2, &geoStringHeader, &geoString));

AnalysisAssert(geoString);

// OK for calendar/clock to get truncated as it would pass incomplete text below which

// will be rejected by globalization dll.

IfFailedReturn(GetWindowsGlobalizationLibrary(scriptContext)->WindowsCreateStringReference(calendar, static\_cast<UINT32>(wcslen(calendar)), &calStringHeader, &calString));

AnalysisAssert(calString);

IfFailedReturn(GetWindowsGlobalizationLibrary(scriptContext)->WindowsCreateStringReference(clock, static\_cast<UINT32>(wcslen(clock)), &clockStringHeader, &clockString));

AnalysisAssert(clockString);

IfFailedReturn(this->dateTimeFormatterFactory->CreateDateTimeFormatterContext(fsHString, languages.Get(), geoString, calString, clockString, result));

}

return hr;

}

HRESULT WindowsGlobalizationAdapter::CreateIncrementNumberRounder(\_In\_ ScriptContext\* scriptContext, NumberFormatting::INumberRounder\*\* numberRounder)

{

return incrementNumberRounderActivationFactory->ActivateInstance(reinterpret\_cast<IInspectable\*\*>(numberRounder));

}

HRESULT WindowsGlobalizationAdapter::CreateSignificantDigitsRounder(\_In\_ ScriptContext\* scriptContext, NumberFormatting::INumberRounder\*\* numberRounder)

{

return significantDigitsRounderActivationFactory->ActivateInstance(reinterpret\_cast<IInspectable\*\*>(numberRounder));

}

#endif

}

#endif

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

// 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

#if defined(ENABLE\_INTL\_OBJECT) || defined(ENABLE\_ES6\_CHAR\_CLASSIFIER)

#include "Windows.Globalization.h"

#ifndef NTBUILD

#include "windows.globalization.numberformatting.h"

#include "windows.globalization.datetimeformatting.h"

#include "Windows.Data.Text.h"

#include "activation.h"

using namespace ABI;

#endif

class ThreadContext;

namespace Js

{

class WindowsGlobalizationAdapter

{

private:

bool initializedCommonGlobObjects;

HRESULT hrForCommonGlobObjectsInit;

bool initializedDateTimeFormatObjects;

HRESULT hrForDateTimeFormatObjectsInit;

bool initializedNumberFormatObjects;

HRESULT hrForNumberFormatObjectsInit;

bool initializedCharClassifierObjects;

HRESULT hrForCharClassifierObjectsInit;

#ifdef ENABLE\_INTL\_OBJECT

// Common

AutoCOMPtr<Windows::Globalization::ILanguageFactory> languageFactory;

AutoCOMPtr<Windows::Globalization::ILanguageStatics> languageStatics;

AutoCOMPtr<Windows::Globalization::DateTimeFormatting::IDateTimeFormatterFactory> dateTimeFormatterFactory;

// DateTimeFormat

AutoCOMPtr<Windows::Globalization::ICalendarFactory> calendarFactory;

AutoCOMPtr<Windows::Globalization::ITimeZoneOnCalendar> timeZoneCalendar; // use to validate timeZone

AutoCOMPtr<Windows::Globalization::ITimeZoneOnCalendar> defaultTimeZoneCalendar; // default current time zone

// NumberFormat

AutoCOMPtr<Windows::Globalization::NumberFormatting::ICurrencyFormatterFactory> currencyFormatterFactory;

AutoCOMPtr<Windows::Globalization::NumberFormatting::IDecimalFormatterFactory> decimalFormatterFactory;

AutoCOMPtr<Windows::Globalization::NumberFormatting::IPercentFormatterFactory> percentFormatterFactory;

AutoCOMPtr<IActivationFactory> incrementNumberRounderActivationFactory;

AutoCOMPtr<IActivationFactory> significantDigitsRounderActivationFactory;

#endif

// CharClassifier

AutoCOMPtr<Windows::Data::Text::IUnicodeCharactersStatics> unicodeStatics;

DelayLoadWindowsGlobalization\* GetWindowsGlobalizationLibrary(\_In\_ ScriptContext\* scriptContext);

DelayLoadWindowsGlobalization\* GetWindowsGlobalizationLibrary(\_In\_ ThreadContext\* threadContext);

template <typename T>

HRESULT GetActivationFactory(DelayLoadWindowsGlobalization \*library, LPCWSTR factoryName, T\*\* instance);

public:

WindowsGlobalizationAdapter()

: initializedCommonGlobObjects(false),

hrForCommonGlobObjectsInit(S\_OK),

initializedDateTimeFormatObjects(false),

hrForDateTimeFormatObjectsInit(S\_OK),

initializedNumberFormatObjects(false),

hrForNumberFormatObjectsInit(S\_OK),

initializedCharClassifierObjects(false),

hrForCharClassifierObjectsInit(S\_OK),

#ifdef ENABLE\_INTL\_OBJECT

languageFactory(nullptr),

languageStatics(nullptr),

dateTimeFormatterFactory(nullptr),

calendarFactory(nullptr),

timeZoneCalendar(nullptr),

defaultTimeZoneCalendar(nullptr),

currencyFormatterFactory(nullptr),

decimalFormatterFactory(nullptr),

percentFormatterFactory(nullptr),

incrementNumberRounderActivationFactory(nullptr),

significantDigitsRounderActivationFactory(nullptr),

#endif // ENABLE\_INTL\_OBJECT

unicodeStatics(nullptr)

{ }

#ifdef ENABLE\_INTL\_OBJECT

HRESULT EnsureCommonObjectsInitialized(DelayLoadWindowsGlobalization \*library);

HRESULT EnsureDateTimeFormatObjectsInitialized(DelayLoadWindowsGlobalization \*library);

HRESULT EnsureNumberFormatObjectsInitialized(DelayLoadWindowsGlobalization \*library);

#endif

HRESULT EnsureDataTextObjectsInitialized(DelayLoadWindowsGlobalization \*library);

#ifdef ENABLE\_INTL\_OBJECT

HRESULT CreateLanguage(\_In\_ ScriptContext\* scriptContext, \_In\_z\_ PCWSTR languageTag, Windows::Globalization::ILanguage\*\* language);

boolean IsWellFormedLanguageTag(\_In\_ ScriptContext\* scriptContext, \_In\_z\_ PCWSTR languageTag);

HRESULT NormalizeLanguageTag(\_In\_ ScriptContext\* scriptContext, \_In\_z\_ PCWSTR languageTag, HSTRING \*result);

HRESULT CreateCurrencyFormatterCode(\_In\_ ScriptContext\* scriptContext, \_In\_z\_ PCWSTR currencyCode, Windows::Globalization::NumberFormatting::ICurrencyFormatter\*\* currencyFormatter);

HRESULT CreateCurrencyFormatter(\_In\_ ScriptContext\* scriptContext, PCWSTR\* localeStrings, uint32 numLocaleStrings, \_In\_z\_ PCWSTR currencyCode, Windows::Globalization::NumberFormatting::ICurrencyFormatter\*\* currencyFormatter);

HRESULT CreateNumberFormatter(\_In\_ ScriptContext\* scriptContext, PCWSTR\* localeStrings, uint32 numLocaleStrings, Windows::Globalization::NumberFormatting::INumberFormatter\*\* numberFormatter);

HRESULT CreatePercentFormatter(\_In\_ ScriptContext\* scriptContext, PCWSTR\* localeStrings, uint32 numLocaleStrings, Windows::Globalization::NumberFormatting::INumberFormatter\*\* numberFormatter);

HRESULT CreateDateTimeFormatter(\_In\_ ScriptContext\* scriptContext, \_In\_z\_ PCWSTR formatString, \_In\_z\_ PCWSTR\* localeStrings, uint32 numLocaleStrings,

\_In\_opt\_z\_ PCWSTR calendar, \_In\_opt\_z\_ PCWSTR clock, \_Out\_ Windows::Globalization::DateTimeFormatting::IDateTimeFormatter\*\* formatter);

HRESULT CreateIncrementNumberRounder(\_In\_ ScriptContext\* scriptContext, Windows::Globalization::NumberFormatting::INumberRounder\*\* numberRounder);

HRESULT CreateSignificantDigitsRounder(\_In\_ ScriptContext\* scriptContext, Windows::Globalization::NumberFormatting::INumberRounder\*\* numberRounder);

boolean ValidateAndCanonicalizeTimeZone(\_In\_ ScriptContext\* scriptContext, \_In\_z\_ PCWSTR timeZoneId, HSTRING\* result);

void GetDefaultTimeZoneId(\_In\_ ScriptContext\* scriptContext, HSTRING\* result);

void ClearTimeZoneCalendars();

#endif // ENABLE\_INTL\_OBJECT

Windows::Data::Text::IUnicodeCharactersStatics\* GetUnicodeStatics()

{

return unicodeStatics;

}

private:

HRESULT CreateTimeZoneOnCalendar(\_In\_ DelayLoadWindowsGlobalization \*library, \_\_out Windows::Globalization::ITimeZoneOnCalendar\*\* result);

};

}

#endif