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

// 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.

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

#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

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);

}

}