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

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

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

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

#include "RuntimeTypePch.h"

#include "cmperr.h"

#include "Language\JavascriptStackWalker.h"

namespace Js

{

bool ActivationObject::Is(void\* instance)

{

return VirtualTableInfo<Js::ActivationObject>::HasVirtualTable(instance) ||

VirtualTableInfo<Js::ActivationObjectEx>::HasVirtualTable(instance) ||

VirtualTableInfo<Js::PseudoActivationObject>::HasVirtualTable(instance) ||

VirtualTableInfo<Js::BlockActivationObject>::HasVirtualTable(instance) ||

VirtualTableInfo<Js::ConsoleScopeActivationObject>::HasVirtualTable(instance);

}

BOOL ActivationObject::HasOwnPropertyCheckNoRedecl(PropertyId propertyId)

{

bool noRedecl = false;

if (!GetTypeHandler()->HasProperty(this, propertyId, &noRedecl))

{

return FALSE;

}

else if (noRedecl)

{

JavascriptError::ThrowReferenceError(GetScriptContext(), ERRRedeclaration);

}

return TRUE;

}

BOOL ActivationObject::SetProperty(PropertyId propertyId, Var value, PropertyOperationFlags flags, PropertyValueInfo\* info)

{

return DynamicObject::SetProperty(propertyId, value, (PropertyOperationFlags)(flags | PropertyOperation\_NonFixedValue), info);

}

BOOL ActivationObject::SetProperty(JavascriptString\* propertyNameString, Var value, PropertyOperationFlags flags, PropertyValueInfo\* info)

{

return DynamicObject::SetProperty(propertyNameString, value, (PropertyOperationFlags)(flags | PropertyOperation\_NonFixedValue), info);

}

BOOL ActivationObject::SetInternalProperty(PropertyId propertyId, Var value, PropertyOperationFlags flags, PropertyValueInfo\* info)

{

return DynamicObject::SetProperty(propertyId, value, (PropertyOperationFlags)(flags | PropertyOperation\_NonFixedValue), info);

}

BOOL ActivationObject::InitProperty(PropertyId propertyId, Var value, PropertyOperationFlags flags, PropertyValueInfo\* info)

{

return DynamicObject::SetPropertyWithAttributes(propertyId, value, PropertyWritable|PropertyEnumerable, info, (PropertyOperationFlags)(flags | PropertyOperation\_NonFixedValue));

}

BOOL ActivationObject::InitPropertyScoped(PropertyId propertyId, Var value)

{

DynamicObject::InitProperty(propertyId, value, PropertyOperation\_NonFixedValue);

return true;

}

BOOL ActivationObject::InitFuncScoped(PropertyId propertyId, Var value)

{

// Var binding of functions declared in eval are elided when conflicting

// with function scope let/const variables, so do not actually set the

// property if it exists and is a let/const variable.

bool noRedecl = false;

if (!GetTypeHandler()->HasProperty(this, propertyId, &noRedecl) || !noRedecl)

{

DynamicObject::InitProperty(propertyId, value, PropertyOperation\_NonFixedValue);

}

return true;

}

BOOL ActivationObject::EnsureProperty(PropertyId propertyId)

{

if (!DynamicObject::HasOwnProperty(propertyId))

{

DynamicObject::SetPropertyWithAttributes(propertyId, this->GetLibrary()->GetUndefined(), PropertyDynamicTypeDefaults, nullptr, PropertyOperation\_NonFixedValue);

}

return true;

}

BOOL ActivationObject::EnsureNoRedeclProperty(PropertyId propertyId)

{

ActivationObject::HasOwnPropertyCheckNoRedecl(propertyId);

return true;

}

BOOL ActivationObject::DeleteItem(uint32 index, PropertyOperationFlags flags)

{

return false;

}

BOOL ActivationObject::GetDiagValueString(StringBuilder<ArenaAllocator>\* stringBuilder, ScriptContext\* requestContext)

{

stringBuilder->AppendCppLiteral(L"{ActivationObject}");

return TRUE;

}

BOOL ActivationObject::GetDiagTypeString(StringBuilder<ArenaAllocator>\* stringBuilder, ScriptContext\* requestContext)

{

stringBuilder->AppendCppLiteral(L"Object, (ActivationObject)");

return TRUE;

}

BOOL BlockActivationObject::InitPropertyScoped(PropertyId propertyId, Var value)

{

// eval, etc., should not create var properties on block scope

return false;

}

BOOL BlockActivationObject::InitFuncScoped(PropertyId propertyId, Var value)

{

// eval, etc., should not create function var properties on block scope

return false;

}

BOOL BlockActivationObject::EnsureProperty(PropertyId propertyId)

{

// eval, etc., should not create function var properties on block scope

return false;

}

BOOL BlockActivationObject::EnsureNoRedeclProperty(PropertyId propertyId)

{

// eval, etc., should not create function var properties on block scope

return false;

}

BlockActivationObject\* BlockActivationObject::Clone(ScriptContext \*scriptContext)

{

DynamicType\* type = this->GetDynamicType();

BlockActivationObject\* blockScopeClone = DynamicObject::NewObject<BlockActivationObject>(scriptContext->GetRecycler(), type);

int slotCapacity = this->GetTypeHandler()->GetSlotCapacity();

for (int i = 0; i < slotCapacity; i += 1)

{

DebugOnly(PropertyId propId = this->GetPropertyId(i));

Var value = this->GetSlot(i);

blockScopeClone->SetSlot(SetSlotArguments(propId, i, value));

}

return blockScopeClone;

}

BOOL PseudoActivationObject::InitPropertyScoped(PropertyId propertyId, Var value)

{

// eval, etc., should not create function properties on something like a "catch" scope

return false;

}

BOOL PseudoActivationObject::InitFuncScoped(PropertyId propertyId, Var value)

{

// eval, etc., should not create function properties on something like a "catch" scope

return false;

}

BOOL PseudoActivationObject::EnsureProperty(PropertyId propertyId)

{

return false;

}

BOOL PseudoActivationObject::EnsureNoRedeclProperty(PropertyId propertyId)

{

return false;

}

/\* static \*/

const PropertyId \* ActivationObjectEx::GetCachedScopeInfo(const PropertyIdArray \*propIds)

{

// Cached scope info is appended to the "normal" prop ID array elements.

return &propIds->elements[propIds->count];

}

BOOL ActivationObjectEx::GetPropertyReference(Var originalInstance, PropertyId propertyId, Var \*value, PropertyValueInfo \*info, ScriptContext \*requestContext)

{

// No need to invalidate the cached scope even if the property is a cached function object.

// The caller won't be using the object itself.

return \_\_super::GetProperty(originalInstance, propertyId, value, info, requestContext);

}

void ActivationObjectEx::GetPropertyCore(PropertyValueInfo \*info, ScriptContext \*requestContext)

{

if (info)

{

PropertyIndex slot = info->GetPropertyIndex();

if (slot >= this->firstFuncSlot && slot <= this->lastFuncSlot)

{

this->parentFunc->InvalidateCachedScopeChain();

// If the caller is an eval, then each time we execute the eval we need to invalidate the

// cached scope chain. We can't rely on detecting the escape each time, because inline

// cache hits may keep us from entering the runtime. So set a flag to make sure the

// invalidation always happens.

JavascriptFunction \*currentFunc = nullptr;

JavascriptStackWalker walker(requestContext);

while (walker.GetCaller(&currentFunc))

{

if (walker.IsEvalCaller())

{

//We are walking the stack, so the function body must have been deserialized by this point.

currentFunc->GetFunctionBody()->SetFuncEscapes(true);

break;

}

}

}

}

}

BOOL ActivationObjectEx::GetProperty(Var originalInstance, PropertyId propertyId, Var \*value, PropertyValueInfo \*info, ScriptContext \*requestContext)

{

if (\_\_super::GetProperty(originalInstance, propertyId, value, info, requestContext))

{

GetPropertyCore(info, requestContext);

return TRUE;

}

return FALSE;

}

BOOL ActivationObjectEx::GetProperty(Var originalInstance, JavascriptString\* propertyNameString, Var \*value, PropertyValueInfo \*info, ScriptContext \*requestContext)

{

if (\_\_super::GetProperty(originalInstance, propertyNameString, value, info, requestContext))

{

GetPropertyCore(info, requestContext);

return TRUE;

}

return FALSE;

}

void ActivationObjectEx::InvalidateCachedScope()

{

if (this->cachedFuncCount != 0)

{

// Clearing the cached functions and types isn't strictly necessary for correctness,

// but we want those objects to be collected even if the scope object is part of someone's

// closure environment.

memset(this->cache, 0, this->cachedFuncCount \* sizeof(FuncCacheEntry));

}

this->parentFunc->SetCachedScope(nullptr);

}

void ActivationObjectEx::SetCachedFunc(uint i, ScriptFunction \*func)

{

Assert(i < cachedFuncCount);

cache[i].func = func;

cache[i].type = (DynamicType\*)func->GetType();

}

};

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

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

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

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

#pragma once

namespace Js

{

struct FuncCacheEntry

{

ScriptFunction \*func;

DynamicType \*type;

};

class ActivationObject : public DynamicObject

{

protected:

DEFINE\_VTABLE\_CTOR(ActivationObject, DynamicObject);

DEFINE\_MARSHAL\_OBJECT\_TO\_SCRIPT\_CONTEXT(ActivationObject);

public:

ActivationObject(DynamicType \* type) : DynamicObject(type)

{}

virtual BOOL HasOwnPropertyCheckNoRedecl(PropertyId propertyId) override;

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

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

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

virtual BOOL EnsureProperty(PropertyId propertyId) override;

virtual BOOL EnsureNoRedeclProperty(PropertyId propertyId) override;

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

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

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

virtual BOOL DeleteItem(uint32 index, PropertyOperationFlags flags) override;

virtual BOOL GetDiagValueString(StringBuilder<ArenaAllocator>\* stringBuilder, ScriptContext\* requestContext) override;

virtual BOOL GetDiagTypeString(StringBuilder<ArenaAllocator>\* stringBuilder, ScriptContext\* requestContext) override;

static bool Is(void\* instance);

};

// A block-ActivationObject is a scope for an ES6 block that should only receive block-scoped inits,

// including function, let, and const.

class BlockActivationObject : public ActivationObject

{

private:

DEFINE\_VTABLE\_CTOR(BlockActivationObject, ActivationObject);

DEFINE\_MARSHAL\_OBJECT\_TO\_SCRIPT\_CONTEXT(BlockActivationObject);

public:

BlockActivationObject(DynamicType \* type) : ActivationObject(type) {}

virtual BOOL EnsureProperty(PropertyId propertyId) override;

virtual BOOL EnsureNoRedeclProperty(PropertyId propertyId) override;

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

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

static bool Is(void\* instance)

{

return VirtualTableInfo<Js::BlockActivationObject>::HasVirtualTable(instance);

}

static BlockActivationObject\* FromVar(Var value)

{

Assert(BlockActivationObject::Is(value));

return static\_cast<BlockActivationObject\*>(DynamicObject::FromVar(value));

}

BlockActivationObject\* Clone(ScriptContext \*scriptContext);

};

// A pseudo-ActivationObject is a scope like a "catch" scope that shouldn't receive var inits.

class PseudoActivationObject : public ActivationObject

{

private:

DEFINE\_VTABLE\_CTOR(PseudoActivationObject, ActivationObject);

DEFINE\_MARSHAL\_OBJECT\_TO\_SCRIPT\_CONTEXT(PseudoActivationObject);

public:

PseudoActivationObject(DynamicType \* type) : ActivationObject(type) {}

virtual BOOL EnsureProperty(PropertyId propertyId) override;

virtual BOOL EnsureNoRedeclProperty(PropertyId propertyId) override;

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

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

static bool Is(void\* instance)

{

return VirtualTableInfo<Js::PseudoActivationObject>::HasVirtualTable(instance);

}

};

class ConsoleScopeActivationObject : public ActivationObject

{

private:

DEFINE\_VTABLE\_CTOR(ConsoleScopeActivationObject, ActivationObject);

DEFINE\_MARSHAL\_OBJECT\_TO\_SCRIPT\_CONTEXT(ConsoleScopeActivationObject);

public:

ConsoleScopeActivationObject(DynamicType \* type) : ActivationObject(type) {}

// A dummy function to have a different vtable

virtual void DummyVirtualFunc(void)

{

AssertMsg(false, "ConsoleScopeActivationObject::DummyVirtualFunc function should never be called");

}

static bool Is(void\* instance)

{

return VirtualTableInfo<Js::ConsoleScopeActivationObject>::HasVirtualTable(instance);

}

};

class ActivationObjectEx : public ActivationObject

{

private:

DEFINE\_VTABLE\_CTOR(ActivationObjectEx, ActivationObject);

DEFINE\_MARSHAL\_OBJECT\_TO\_SCRIPT\_CONTEXT(ActivationObjectEx);

void GetPropertyCore(PropertyValueInfo \*info, ScriptContext \*requestContext);

public:

ActivationObjectEx(

DynamicType \* type, ScriptFunction \*func, uint cachedFuncCount, uint firstFuncSlot, uint lastFuncSlot)

: ActivationObject(type),

parentFunc(func),

cachedFuncCount(cachedFuncCount),

firstFuncSlot(firstFuncSlot),

lastFuncSlot(lastFuncSlot),

committed(false)

{

if (cachedFuncCount != 0)

{

cache[0].func = nullptr;

}

}

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

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

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

virtual void InvalidateCachedScope() override sealed;

bool IsCommitted() const { return committed; }

void SetCommit(bool set) { committed = set; }

ScriptFunction \*GetParentFunc() const { return parentFunc; }

uint GetFirstFuncSlot() const { return firstFuncSlot; }

uint GetLastFuncSlot() const { return lastFuncSlot; }

bool HasCachedFuncs() const { return cachedFuncCount != 0 && cache[0].func != nullptr; }

void SetCachedFunc(uint i, ScriptFunction \*func);

FuncCacheEntry \*GetFuncCacheEntry(uint i)

{

Assert(i < cachedFuncCount);

return &cache[i];

}

static uint32 GetOffsetOfCache() { return offsetof(ActivationObjectEx, cache); }

static uint32 GetOffsetOfCommitFlag() { return offsetof(ActivationObjectEx, committed); }

static uint32 GetOffsetOfParentFunc() { return offsetof(ActivationObjectEx, parentFunc); }

static const PropertyId \*GetCachedScopeInfo(const PropertyIdArray \*propIds);

// Cached scope info:

// [0] - cached func count

// [1] - first func slot

// [2] - first var slot

// [3] - literal object reference

static PropertyId GetCachedFuncCount(const PropertyIdArray \*propIds)

{

return ActivationObjectEx::GetCachedScopeInfo(propIds)[0];

}

static PropertyId GetFirstFuncSlot(const PropertyIdArray \*propIds)

{

return ActivationObjectEx::GetCachedScopeInfo(propIds)[1];

}

static PropertyId GetFirstVarSlot(const PropertyIdArray \*propIds)

{

return ActivationObjectEx::GetCachedScopeInfo(propIds)[2];

}

static PropertyId GetLiteralObjectRef(const PropertyIdArray \*propIds)

{

return ActivationObjectEx::GetCachedScopeInfo(propIds)[3];

}

static uint32 ExtraSlotCount() { return 4; }

static bool Is(void\* instance)

{

return VirtualTableInfo<Js::ActivationObjectEx>::HasVirtualTable(instance);

}

private:

ScriptFunction \*parentFunc;

uint cachedFuncCount;

uint firstFuncSlot;

uint lastFuncSlot;

bool committed;

FuncCacheEntry cache[1];

};

};

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

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

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

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

// Implementation for typed arrays based on ArrayBuffer.

// There is one nested ArrayBuffer for each typed array. Multiple typed array

// can share the same array buffer.

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

#include "RuntimeTypePch.h"

namespace Js

{

ArrayObject::ArrayObject(ArrayObject \* instance)

: DynamicObject(instance),

length(instance->length)

{

}

void ArrayObject::ThrowItemNotConfigurableError(PropertyId propId /\*= Constants::NoProperty\*/)

{

ScriptContext\* scriptContext = GetScriptContext();

JavascriptError::ThrowTypeError(scriptContext, JSERR\_DefineProperty\_NotConfigurable,

propId != Constants::NoProperty ?

scriptContext->GetThreadContext()->GetPropertyName(propId)->GetBuffer() : nullptr);

}

void ArrayObject::VerifySetItemAttributes(PropertyId propId, PropertyAttributes attributes)

{

if (attributes != (PropertyEnumerable | PropertyWritable))

{

ThrowItemNotConfigurableError(propId);

}

}

BOOL ArrayObject::SetItemAttributes(uint32 index, PropertyAttributes attributes)

{

VerifySetItemAttributes(Constants::NoProperty, attributes);

return TRUE;

}

BOOL ArrayObject::SetItemAccessors(uint32 index, Var getter, Var setter)

{

ThrowItemNotConfigurableError();

}

BOOL ArrayObject::IsObjectArrayFrozen()

{

return this->IsFrozen();

}

BOOL ArrayObject::GetEnumerator(BOOL enumNonEnumerable, Var\* enumerator, ScriptContext\* requestContext, bool preferSnapshotSemantics /\*= true\*/, bool enumSymbols /\*= false\*/)

{

return \_\_super::GetEnumerator(enumNonEnumerable, enumerator, requestContext, preferSnapshotSemantics, enumSymbols);

}

BOOL ArrayObject::GetEnumerator(Var originalInstance, BOOL enumNonEnumerable, Var\* enumerator, ScriptContext\* requestContext, bool preferSnapshotSemantics /\*= true\*/, bool enumSymbols /\*= false\*/)

{

return GetEnumerator(enumNonEnumerable, enumerator, requestContext, preferSnapshotSemantics, enumSymbols);

}

} // namespace Js

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

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

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

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

#pragma once

namespace Js

{

#ifdef \_M\_X64\_OR\_ARM64

// This base class has a 4-byte length field. Change struct pack to 4 on 64bit to avoid 4 padding bytes here.

#pragma pack(push, 4)

#endif

//

// A base class for all array-like objects types, that can serve as an Object's internal array:

// JavascriptArray/ES5Array or TypedArray.

//

class ArrayObject : public DynamicObject

{

protected:

uint32 length;

protected:

DEFINE\_VTABLE\_CTOR\_ABSTRACT(ArrayObject, DynamicObject);

ArrayObject(DynamicType \* type, bool initSlots = true, uint32 length = 0)

: DynamicObject(type, initSlots), length(length)

{

}

ArrayObject(DynamicType \* type, ScriptContext \* scriptContext)

: DynamicObject(type, scriptContext), length(0)

{

}

// For boxing stack instance

ArrayObject(ArrayObject \* instance);

void \_\_declspec(noreturn) ThrowItemNotConfigurableError(PropertyId propId = Constants::NoProperty);

void VerifySetItemAttributes(PropertyId propId, PropertyAttributes attributes);

public:

uint32 GetLength() const { return length; }

static uint32 GetOffsetOfLength() { return offsetof(ArrayObject, length); }

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

// objectArray support

virtual BOOL SetItemWithAttributes(uint32 index, Var value, PropertyAttributes attributes) = 0;

virtual BOOL SetItemAttributes(uint32 index, PropertyAttributes attributes);

virtual BOOL SetItemAccessors(uint32 index, Var getter, Var setter);

virtual BOOL IsObjectArrayFrozen();

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

};

#ifdef \_M\_X64\_OR\_ARM64

#pragma pack(pop)

#endif

} // namespace Js

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

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

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

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

<PropertyGroup Label="Globals">

<TargetName>Chakra.Runtime.Types</TargetName>

<ProjectGuid>{706083F7-6AA4-4558-A153-6352EF9110F6}</ProjectGuid>

<RootNamespace>JS</RootNamespace>

<Keyword>Win32Proj</Keyword>

</PropertyGroup>

<PropertyGroup Label="Configuration">

<ConfigurationType>StaticLibrary</ConfigurationType>

</PropertyGroup>

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

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

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

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

<PropertyGroup>

<\_ProjectFileVersion>10.0.30319.1</\_ProjectFileVersion>

</PropertyGroup>

<ItemDefinitionGroup>

<ClCompile>

<AdditionalIncludeDirectories>

$(MSBuildThisFileDirectory)..;

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

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

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

%(AdditionalIncludeDirectories)

</AdditionalIncludeDirectories>

<PrecompiledHeader>Use</PrecompiledHeader>

<PrecompiledHeaderFile>RuntimeTypePch.h</PrecompiledHeaderFile>

</ClCompile>

</ItemDefinitionGroup>

<ItemGroup>

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

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

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

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

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

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

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

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

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

<ClCompile Include="$(MSBuildThisFileDirectory)ES5ArrayTypeHandler.cpp" />

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

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

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

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

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

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

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

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

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

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

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

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

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

<PrecompiledHeader>Create</PrecompiledHeader>

</ClCompile>

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

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

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

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

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

</ItemGroup>

<ItemGroup>

<ClInclude Include="ActivationObject.h" />

<ClInclude Include="ArrayObject.h" />

<ClInclude Include="DeferredTypeHandler.h" />

<ClInclude Include="DictionaryPropertyDescriptor.h" />

<ClInclude Include="DictionaryTypeHandler.h" />

<ClInclude Include="DynamicObject.h" />

<ClInclude Include="DynamicObjectEnumerator.h" />

<ClInclude Include="DynamicObjectSnapshotEnumerator.h" />

<ClInclude Include="DynamicObjectSnapshotEnumeratorWPCache.h" />

<ClInclude Include="DynamicType.h" />

<ClInclude Include="EdgeJavascriptTypeId.h" />

<ClInclude Include="ES5ArrayTypeHandler.h" />

<ClInclude Include="JavascriptEnumerator.h" />

<ClInclude Include="MissingPropertyTypeHandler.h" />

<ClInclude Include="NullTypeHandler.h" />

<ClInclude Include="PathTypeHandler.h" />

<ClInclude Include="PropertyDescriptor.h" />

<ClInclude Include="PropertyIndexRanges.h" />

<ClInclude Include="RecyclableObject.h" />

<ClInclude Include="RuntimeTypePch.h" />

<ClInclude Include="ScriptFunctionType.h" />

<ClInclude Include="SimpleDictionaryPropertyDescriptor.h" />

<ClInclude Include="SimpleDictionaryTypeHandler.h" />

<ClInclude Include="SimpleDictionaryUnorderedTypeHandler.h" />

<ClInclude Include="SimplePropertyDescriptor.h" />

<ClInclude Include="SimpleTypeHandler.h" />

<ClInclude Include="SpreadArgument.h" />

<ClInclude Include="StaticType.h" />

<ClInclude Include="Type.h" />

<ClInclude Include="TypeHandler.h" />

<ClInclude Include="TypeId.h" />

<ClInclude Include="TypePath.h" />

<ClInclude Include="TypePropertyCache.h" />

<ClInclude Include="WithScopeObject.h" />

</ItemGroup>

<ItemGroup>

<None Include="RecyclableObject.inl" />

<None Include="DynamicObject.inl" />

</ItemGroup>

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

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

</Project>

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

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

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

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

#include "RuntimeTypePch.h"

#include "Types\DeferredTypeHandler.h"

namespace Js

{

void DeferredTypeHandlerBase::Convert(DynamicObject \* instance, DynamicTypeHandler \* typeHandler)

{

Assert(instance->GetDynamicType()->GetTypeHandler() == this);

Assert(this->inlineSlotCapacity == typeHandler->inlineSlotCapacity);

Assert(this->offsetOfInlineSlots == typeHandler->offsetOfInlineSlots);

Assert(this->GetIsInlineSlotCapacityLocked() == typeHandler->GetIsInlineSlotCapacityLocked());

// Since the caller owns the typeHandler the instance is transitioning to, the caller should have

// set up the singleton instance on that handler, if fixed fields are desired. The caller is

// also responsible for populating PropertyTypes to indicate whether there are any read-only

// properties unknown to the type handler.

BOOL isProto = (GetFlags() & IsPrototypeFlag);

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

instance->EnsureSlots(0, typeHandler->GetSlotCapacity(), scriptContext, typeHandler);

typeHandler->SetInstanceTypeHandler(instance);

// We may be changing to a type handler that already has some properties. Initialize those to undefined.

const Var undefined = scriptContext->GetLibrary()->GetUndefined();

const BigPropertyIndex propertyCount = typeHandler->GetPropertyCount();

Assert(propertyCount <= typeHandler->GetSlotCapacity());

for(BigPropertyIndex i = 0; i < propertyCount; ++i)

{

typeHandler->SetSlotUnchecked(instance, i, undefined);

}

if (isProto)

{

instance->GetDynamicType()->GetTypeHandler()->SetIsPrototype(instance);

}

}

void DeferredTypeHandlerBase::Convert(DynamicObject \* instance, DeferredInitializeMode mode, int initSlotCapacity, BOOL hasAccessor)

{

Assert(instance->GetDynamicType()->GetTypeHandler() == this);

BOOL isProto = (GetFlags() & IsPrototypeFlag);

BOOL isSimple = !hasAccessor;

switch (mode)

{

case DeferredInitializeMode\_Set:

initSlotCapacity++;

break;

case DeferredInitializeMode\_SetAccessors:

initSlotCapacity += 2;

// fall-through

case DeferredInitializeMode\_Extensions:

isSimple = FALSE;

break;

}

DynamicTypeHandler\* newTypeHandler;

if (isSimple)

{

newTypeHandler = ConvertToSimpleDictionaryType(instance, initSlotCapacity);

}

else

{

newTypeHandler = ConvertToDictionaryType(instance, initSlotCapacity);

}

AssertMsg(!instance->HasSharedType(), "Expect the instance to have a non-shared type and handler after conversion.");

if (isProto)

{

newTypeHandler->SetIsPrototype(instance);

AssertMsg(instance->GetDynamicType()->GetTypeHandler() == newTypeHandler, "Why did SetIsPrototype force a type handler change on a non-shared type handler?");

}

}

template <typename T>

T\* DeferredTypeHandlerBase::ConvertToTypeHandler(DynamicObject\* instance, int initSlotCapacity)

{

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

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

// Create new type handler, allowing slotCapacity round up here. We'll allocate instance slots below.

T\* newTypeHandler = T::New(recycler, initSlotCapacity, GetInlineSlotCapacity(), GetOffsetOfInlineSlots());

newTypeHandler->SetSingletonInstanceIfNeeded(instance);

// EnsureSlots before updating the type handler and instance, as EnsureSlots allocates and may throw.

instance->EnsureSlots(0, newTypeHandler->GetSlotCapacity(), scriptContext, newTypeHandler);

newTypeHandler->SetFlags(IsPrototypeFlag, this->GetFlags());

newTypeHandler->SetPropertyTypes(PropertyTypesWritableDataOnly | PropertyTypesWritableDataOnlyDetection | PropertyTypesInlineSlotCapacityLocked , this->GetPropertyTypes());

if (instance->HasReadOnlyPropertiesInvisibleToTypeHandler())

{

newTypeHandler->ClearHasOnlyWritableDataProperties();

}

newTypeHandler->SetInstanceTypeHandler(instance);

return newTypeHandler;

}

SimpleDictionaryTypeHandler\* DeferredTypeHandlerBase::ConvertToSimpleDictionaryType(DynamicObject\* instance, int initSlotCapacity)

{

// DeferredTypeHandler is only used internally by the type system. "initSlotCapacity" should be a tiny number.

Assert(initSlotCapacity <= SimpleDictionaryTypeHandler::MaxPropertyIndexSize);

SimpleDictionaryTypeHandler\* newTypeHandler = ConvertToTypeHandler<SimpleDictionaryTypeHandler>(instance, initSlotCapacity);

#ifdef PROFILE\_TYPES

instance->GetScriptContext()->convertDeferredToSimpleDictionaryCount++;

#endif

return newTypeHandler;

}

DictionaryTypeHandler\* DeferredTypeHandlerBase::ConvertToDictionaryType(DynamicObject\* instance, int initSlotCapacity)

{

// DeferredTypeHandler is only used internally by the type system. "initSlotCapacity" should be a tiny number.

Assert(initSlotCapacity <= DictionaryTypeHandler::MaxPropertyIndexSize);

DictionaryTypeHandler\* newTypeHandler = ConvertToTypeHandler<DictionaryTypeHandler>(instance, initSlotCapacity);

#ifdef PROFILE\_TYPES

instance->GetScriptContext()->convertDeferredToDictionaryCount++;

#endif

return newTypeHandler;

}

ES5ArrayTypeHandler\* DeferredTypeHandlerBase::ConvertToES5ArrayType(DynamicObject\* instance, int initSlotCapacity)

{

// DeferredTypeHandler is only used internally by the type system. "initSlotCapacity" should be a tiny number.

Assert(initSlotCapacity <= ES5ArrayTypeHandler::MaxPropertyIndexSize);

ES5ArrayTypeHandler\* newTypeHandler = ConvertToTypeHandler<ES5ArrayTypeHandler>(instance, initSlotCapacity);

#ifdef PROFILE\_TYPES

instance->GetScriptContext()->convertDeferredToDictionaryCount++;

#endif

return newTypeHandler;

}

};

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

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

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

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

#pragma once

namespace Js

{

class DeferredTypeHandlerBase : public DynamicTypeHandler

{

public:

DEFINE\_GETCPPNAME();

protected:

DeferredTypeHandlerBase(bool isPrototype, uint16 inlineSlotCapacity, uint16 offsetOfInlineSlots) :

DynamicTypeHandler(0, inlineSlotCapacity, offsetOfInlineSlots, DefaultFlags | IsLockedFlag | MayBecomeSharedFlag | IsSharedFlag | (isPrototype ? IsPrototypeFlag : 0))

{

SetIsInlineSlotCapacityLocked();

this->ClearHasOnlyWritableDataProperties(); // Until the type handler is initialized, we cannot

// be certain that the type has only writable data properties.

}

public:

void Convert(DynamicObject \* instance, DynamicTypeHandler \* handler);

void Convert(DynamicObject \* instance, DeferredInitializeMode mode, int initSlotCapacity, BOOL hasAccessor = false);

virtual void SetAllPropertiesToUndefined(DynamicObject\* instance, bool invalidateFixedFields) override {};

virtual void MarshalAllPropertiesToScriptContext(DynamicObject\* instance, ScriptContext\* targetScriptContext, bool invalidateFixedFields) override {};

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

virtual void SetIsPrototype(DynamicObject\* instance) override { Assert(false); }

#if DBG

virtual bool SupportsPrototypeInstances() const { Assert(false); return false; }

virtual bool RespectsIsolatePrototypes() const { return false; }

virtual bool RespectsChangeTypeOnProto() const { return false; }

#endif

private:

template <typename T>

T\* ConvertToTypeHandler(DynamicObject\* instance, int initSlotCapacity);

DictionaryTypeHandler \* ConvertToDictionaryType(DynamicObject\* instance, int initSlotCapacity);

SimpleDictionaryTypeHandler \* ConvertToSimpleDictionaryType(DynamicObject\* instance, int initSlotCapacity);

ES5ArrayTypeHandler \* ConvertToES5ArrayType(DynamicObject\* instance, int initSlotCapacity);

};

class DefaultDeferredTypeFilter

{

public:

static bool HasFilter() { return false; }

static bool HasProperty(PropertyId propertyId) { Assert(false); return false; }

};

template <DeferredTypeInitializer initializer, typename DeferredTypeFilter = DefaultDeferredTypeFilter, bool isPrototypeTemplate = false, uint16 \_inlineSlotCapacity = 0, uint16 \_offsetOfInlineSlots = 0>

class DeferredTypeHandler : public DeferredTypeHandlerBase

{

friend class DynamicTypeHandler;

public:

DEFINE\_GETCPPNAME();

private:

DeferredTypeHandler() : DeferredTypeHandlerBase(isPrototypeTemplate, \_inlineSlotCapacity, \_offsetOfInlineSlots) { }

public:

static DeferredTypeHandler \*GetDefaultInstance() { return &defaultInstance; }

virtual BOOL IsLockable() const override { return true; }

virtual BOOL IsSharable() const override { return true; }

virtual int GetPropertyCount() override;

virtual PropertyId GetPropertyId(ScriptContext\* scriptContext, PropertyIndex index) override;

virtual PropertyId GetPropertyId(ScriptContext\* scriptContext, BigPropertyIndex index) override;

virtual BOOL FindNextProperty(ScriptContext\* scriptContext, PropertyIndex& index, JavascriptString\*\* propertyString,

PropertyId\* propertyId, PropertyAttributes\* attributes, Type\* type, DynamicType \*typeToEnumerate, bool requireEnumerable, bool enumSymbols = false) override;

virtual PropertyIndex GetPropertyIndex(PropertyRecord const\* propertyRecord) override;

virtual bool GetPropertyEquivalenceInfo(PropertyRecord const\* propertyRecord, PropertyEquivalenceInfo& info) override;

virtual bool IsObjTypeSpecEquivalent(const Type\* type, const TypeEquivalenceRecord& record, uint& failedPropertyIndex) override;

virtual bool IsObjTypeSpecEquivalent(const Type\* type, const EquivalentPropertyEntry\* entry) override;

virtual bool EnsureObjectReady(DynamicObject\* instance) override;

virtual BOOL HasProperty(DynamicObject\* instance, PropertyId propertyId, \_\_out\_opt bool \*noRedecl = nullptr) override;

virtual BOOL HasProperty(DynamicObject\* instance, JavascriptString\* propertyNameString) override;

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

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

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

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

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

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

virtual BOOL DeleteProperty(DynamicObject\* instance, PropertyId propertyId, PropertyOperationFlags flags) override;

virtual BOOL HasItem(DynamicObject\* instance, uint32 index);

virtual BOOL SetItem(DynamicObject\* instance, uint32 index, Var value, PropertyOperationFlags flags) override;

virtual BOOL SetItemWithAttributes(DynamicObject\* instance, uint32 index, Var value, PropertyAttributes attributes) override;

virtual BOOL SetItemAttributes(DynamicObject\* instance, uint32 index, PropertyAttributes attributes) override;

virtual BOOL SetItemAccessors(DynamicObject\* instance, uint32 index, Var getter, Var setter) override;

virtual BOOL DeleteItem(DynamicObject\* instance, uint32 index, PropertyOperationFlags flags) override;

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

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

virtual BOOL IsEnumerable(DynamicObject\* instance, PropertyId propertyId) override;

virtual BOOL IsWritable(DynamicObject\* instance, PropertyId propertyId) override;

virtual BOOL IsConfigurable(DynamicObject\* instance, PropertyId propertyId) override;

virtual BOOL IsFrozen(DynamicObject \*instance) override;

virtual BOOL IsSealed(DynamicObject \*instance) override;

virtual BOOL SetEnumerable(DynamicObject\* instance, PropertyId propertyId, BOOL value) override;

virtual BOOL SetWritable(DynamicObject\* instance, PropertyId propertyId, BOOL value) override;

virtual BOOL SetConfigurable(DynamicObject\* instance, PropertyId propertyId, BOOL value) override;

virtual BOOL SetAccessors(DynamicObject\* instance, PropertyId propertyId, Var getter, Var setter, PropertyOperationFlags flags = PropertyOperation\_None) override;

virtual BOOL GetAccessors(DynamicObject\* instance, PropertyId propertyId, Var \*getter, Var \*setter) override;

virtual BOOL PreventExtensions(DynamicObject \*instance) override;

virtual BOOL Seal(DynamicObject \*instance) override;

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

virtual BOOL SetAttributes(DynamicObject\* instance, PropertyId propertyId, PropertyAttributes attributes) override;

virtual BOOL GetAttributesWithPropertyIndex(DynamicObject \* instance, PropertyId propertyId, BigPropertyIndex index, PropertyAttributes \* attributes) override;

virtual DynamicTypeHandler\* ConvertToTypeWithItemAttributes(DynamicObject\* instance) override;

virtual void SetIsPrototype(DynamicObject\* instance) override;

#if DBG

virtual bool SupportsPrototypeInstances() const { return isPrototypeTemplate; }

#endif

private:

static DeferredTypeHandler defaultInstance;

bool EnsureObjectReady(DynamicObject\* instance, DeferredInitializeMode mode);

virtual BOOL FreezeImpl(DynamicObject \*instance, bool isConvertedType) override;

};

template <DeferredTypeInitializer initializer, typename DeferredTypeFilter, bool isPrototypeTemplate, uint16 \_inlineSlotCapacity, uint16 \_offsetOfInlineSlots>

DeferredTypeHandler<initializer, DeferredTypeFilter, isPrototypeTemplate, \_inlineSlotCapacity, \_offsetOfInlineSlots> DeferredTypeHandler<initializer, DeferredTypeFilter, isPrototypeTemplate, \_inlineSlotCapacity, \_offsetOfInlineSlots>::defaultInstance;

template <DeferredTypeInitializer initializer, typename DeferredTypeFilter, bool isPrototypeTemplate, uint16 \_inlineSlotCapacity, uint16 \_offsetOfInlineSlots>

int DeferredTypeHandler<initializer, DeferredTypeFilter, isPrototypeTemplate, \_inlineSlotCapacity, \_offsetOfInlineSlots>::GetPropertyCount()

{

return 0;

}

template <DeferredTypeInitializer initializer, typename DeferredTypeFilter, bool isPrototypeTemplate, uint16 \_inlineSlotCapacity, uint16 \_offsetOfInlineSlots>

PropertyId DeferredTypeHandler<initializer, DeferredTypeFilter, isPrototypeTemplate, \_inlineSlotCapacity, \_offsetOfInlineSlots>::GetPropertyId(ScriptContext\* scriptContext, PropertyIndex index)

{

Assert(false);

return Constants::NoProperty;

}

template <DeferredTypeInitializer initializer, typename DeferredTypeFilter, bool isPrototypeTemplate, uint16 \_inlineSlotCapacity, uint16 \_offsetOfInlineSlots>

PropertyId DeferredTypeHandler<initializer, DeferredTypeFilter, isPrototypeTemplate, \_inlineSlotCapacity, \_offsetOfInlineSlots>::GetPropertyId(ScriptContext\* scriptContext, BigPropertyIndex index)

{

Assert(false);

return Constants::NoProperty;

}

template <DeferredTypeInitializer initializer, typename DeferredTypeFilter, bool isPrototypeTemplate, uint16 \_inlineSlotCapacity, uint16 \_offsetOfInlineSlots>

BOOL DeferredTypeHandler<initializer, DeferredTypeFilter, isPrototypeTemplate, \_inlineSlotCapacity, \_offsetOfInlineSlots>::FindNextProperty(ScriptContext\* scriptContext, PropertyIndex& index,

\_\_out JavascriptString\*\* propertyString, \_\_out PropertyId\* propertyId, \_\_out\_opt PropertyAttributes\* attributes, Type\* type, DynamicType \*typeToEnumerate, bool requireEnumerable, bool enumSymbols)

{

Assert(false);

return FALSE;

}

template <DeferredTypeInitializer initializer, typename DeferredTypeFilter, bool isPrototypeTemplate, uint16 \_inlineSlotCapacity, uint16 \_offsetOfInlineSlots>

PropertyIndex DeferredTypeHandler<initializer, DeferredTypeFilter, isPrototypeTemplate, \_inlineSlotCapacity, \_offsetOfInlineSlots>::GetPropertyIndex(PropertyRecord const\* propertyRecord)

{

return Constants::NoSlot;

}

template <DeferredTypeInitializer initializer, typename DeferredTypeFilter, bool isPrototypeTemplate, uint16 \_inlineSlotCapacity, uint16 \_offsetOfInlineSlots>

bool DeferredTypeHandler<initializer, DeferredTypeFilter, isPrototypeTemplate, \_inlineSlotCapacity, \_offsetOfInlineSlots>::GetPropertyEquivalenceInfo(PropertyRecord const\* propertyRecord, PropertyEquivalenceInfo& info)

{

info.slotIndex = Constants::NoSlot;

info.isAuxSlot = false;

info.isWritable = false;

return false;

}

template <DeferredTypeInitializer initializer, typename DeferredTypeFilter, bool isPrototypeTemplate, uint16 \_inlineSlotCapacity, uint16 \_offsetOfInlineSlots>

bool DeferredTypeHandler<initializer, DeferredTypeFilter, isPrototypeTemplate, \_inlineSlotCapacity, \_offsetOfInlineSlots>::IsObjTypeSpecEquivalent(const Type\* type, const TypeEquivalenceRecord& record, uint& failedPropertyIndex)

{

uint propertyCount = record.propertyCount;

EquivalentPropertyEntry\* properties = record.properties;

for (uint pi = 0; pi < propertyCount; pi++)

{

const EquivalentPropertyEntry\* refInfo = &properties[pi];

if (!this->DeferredTypeHandler<initializer, DeferredTypeFilter, isPrototypeTemplate, \_inlineSlotCapacity, \_offsetOfInlineSlots>::IsObjTypeSpecEquivalent(type, refInfo))

{

failedPropertyIndex = pi;

return false;

}

}

return true;

}

template <DeferredTypeInitializer initializer, typename DeferredTypeFilter, bool isPrototypeTemplate, uint16 \_inlineSlotCapacity, uint16 \_offsetOfInlineSlots>

bool DeferredTypeHandler<initializer, DeferredTypeFilter, isPrototypeTemplate, \_inlineSlotCapacity, \_offsetOfInlineSlots>::IsObjTypeSpecEquivalent(const Type\* type, const EquivalentPropertyEntry\* entry)

{

if (!DeferredTypeFilter::HasFilter())

{

return false;

}

if (entry->slotIndex != Constants::NoSlot || entry->mustBeWritable || DeferredTypeFilter::HasProperty(entry->propertyId))

{

return false;

}

return true;

}

template <DeferredTypeInitializer initializer, typename DeferredTypeFilter, bool isPrototypeTemplate, uint16 \_inlineSlotCapacity, uint16 \_offsetOfInlineSlots>

bool DeferredTypeHandler<initializer, DeferredTypeFilter, isPrototypeTemplate, \_inlineSlotCapacity, \_offsetOfInlineSlots>::EnsureObjectReady(DynamicObject\* instance)

{

return EnsureObjectReady(instance, DeferredInitializeMode\_Default);

}

template <DeferredTypeInitializer initializer, typename DeferredTypeFilter, bool isPrototypeTemplate, uint16 \_inlineSlotCapacity, uint16 \_offsetOfInlineSlots>

bool DeferredTypeHandler<initializer, DeferredTypeFilter, isPrototypeTemplate, \_inlineSlotCapacity, \_offsetOfInlineSlots>::EnsureObjectReady(DynamicObject\* instance, DeferredInitializeMode mode)

{

initializer(instance, this, mode);

ThreadContext\* threadContext = instance->GetScriptContext()->GetThreadContext();

if ((threadContext->GetImplicitCallFlags() > ImplicitCall\_None) && threadContext->IsDisableImplicitCall())

{

return false;

}

return true;

}

template <DeferredTypeInitializer initializer, typename DeferredTypeFilter, bool isPrototypeTemplate, uint16 \_inlineSlotCapacity, uint16 \_offsetOfInlineSlots>

BOOL DeferredTypeHandler<initializer, DeferredTypeFilter, isPrototypeTemplate, \_inlineSlotCapacity, \_offsetOfInlineSlots>::HasProperty(DynamicObject\* instance, PropertyId propertyId, \_\_out\_opt bool \*noRedecl)

{

if (noRedecl != nullptr)

{

\*noRedecl = false;

}

if (DeferredTypeFilter::HasFilter() && DeferredTypeFilter::HasProperty(propertyId))

{

return true;

}

if (!EnsureObjectReady(instance, DeferredInitializeMode\_Default))

{

return FALSE;

}

return GetCurrentTypeHandler(instance)->HasProperty(instance, propertyId, noRedecl);

}

template <DeferredTypeInitializer initializer, typename DeferredTypeFilter, bool isPrototypeTemplate, uint16 \_inlineSlotCapacity, uint16 \_offsetOfInlineSlots>

BOOL DeferredTypeHandler<initializer, DeferredTypeFilter, isPrototypeTemplate, \_inlineSlotCapacity, \_offsetOfInlineSlots>::HasProperty(DynamicObject\* instance, JavascriptString\* propertyNameString)

{

if (!EnsureObjectReady(instance, DeferredInitializeMode\_Default))

{

return FALSE;

}

return GetCurrentTypeHandler(instance)->HasProperty(instance, propertyNameString);

}

template <DeferredTypeInitializer initializer, typename DeferredTypeFilter, bool isPrototypeTemplate, uint16 \_inlineSlotCapacity, uint16 \_offsetOfInlineSlots>

BOOL DeferredTypeHandler<initializer, DeferredTypeFilter, isPrototypeTemplate, \_inlineSlotCapacity, \_offsetOfInlineSlots>::GetProperty(DynamicObject\* instance, Var originalInstance,

PropertyId propertyId, Var\* value, PropertyValueInfo\* info, ScriptContext\* requestContext)

{

if (DeferredTypeFilter::HasFilter() && !DeferredTypeFilter::HasProperty(propertyId))

{

return false;

}

if (!EnsureObjectReady(instance, DeferredInitializeMode\_Default))

{

return FALSE;

}

return GetCurrentTypeHandler(instance)->GetProperty(instance, originalInstance, propertyId, value, info, requestContext);

}

template <DeferredTypeInitializer initializer, typename DeferredTypeFilter, bool isPrototypeTemplate, uint16 \_inlineSlotCapacity, uint16 \_offsetOfInlineSlots>

BOOL DeferredTypeHandler<initializer, DeferredTypeFilter, isPrototypeTemplate, \_inlineSlotCapacity, \_offsetOfInlineSlots>::GetProperty(DynamicObject\* instance, Var originalInstance,

JavascriptString\* propertyNameString, Var\* value, PropertyValueInfo\* info, ScriptContext\* requestContext)

{

if (!EnsureObjectReady(instance, DeferredInitializeMode\_Default))

{

return FALSE;

}

return GetCurrentTypeHandler(instance)->GetProperty(instance, originalInstance, propertyNameString, value, info, requestContext);

}

template <DeferredTypeInitializer initializer, typename DeferredTypeFilter, bool isPrototypeTemplate, uint16 \_inlineSlotCapacity, uint16 \_offsetOfInlineSlots>

BOOL DeferredTypeHandler<initializer, DeferredTypeFilter, isPrototypeTemplate, \_inlineSlotCapacity, \_offsetOfInlineSlots>::SetProperty(DynamicObject\* instance, PropertyId propertyId, Var value, PropertyOperationFlags flags, PropertyValueInfo\* info)

{

if (!EnsureObjectReady(instance, DeferredInitializeMode\_Set))

{

return FALSE;

}

return GetCurrentTypeHandler(instance)->SetProperty(instance, propertyId, value, flags, info);

}

template <DeferredTypeInitializer initializer, typename DeferredTypeFilter, bool isPrototypeTemplate, uint16 \_inlineSlotCapacity, uint16 \_offsetOfInlineSlots>

BOOL DeferredTypeHandler<initializer, DeferredTypeFilter, isPrototypeTemplate, \_inlineSlotCapacity, \_offsetOfInlineSlots>::SetProperty(DynamicObject\* instance, JavascriptString\* propertyNameString, Var value, PropertyOperationFlags flags, PropertyValueInfo\* info)

{

if (!EnsureObjectReady(instance, DeferredInitializeMode\_Set))

{

return FALSE;

}

return GetCurrentTypeHandler(instance)->SetProperty(instance, propertyNameString, value, flags, info);

}

template <DeferredTypeInitializer initializer, typename DeferredTypeFilter, bool isPrototypeTemplate, uint16 \_inlineSlotCapacity, uint16 \_offsetOfInlineSlots>

DescriptorFlags DeferredTypeHandler<initializer, DeferredTypeFilter, isPrototypeTemplate, \_inlineSlotCapacity, \_offsetOfInlineSlots>::GetSetter(DynamicObject\* instance, PropertyId propertyId, Var\* setterValue, PropertyValueInfo\* info, ScriptContext\* requestContext)

{

if (DeferredTypeFilter::HasFilter() && !DeferredTypeFilter::HasProperty(propertyId))

{

return DescriptorFlags::None;

}

if (!EnsureObjectReady(instance, DeferredInitializeMode\_Default))

{

return DescriptorFlags::None;

}

return GetCurrentTypeHandler(instance)->GetSetter(instance, propertyId, setterValue, info, requestContext);

}

template <DeferredTypeInitializer initializer, typename DeferredTypeFilter, bool isPrototypeTemplate, uint16 \_inlineSlotCapacity, uint16 \_offsetOfInlineSlots>

DescriptorFlags DeferredTypeHandler<initializer, DeferredTypeFilter, isPrototypeTemplate, \_inlineSlotCapacity, \_offsetOfInlineSlots>::GetSetter(DynamicObject\* instance, JavascriptString\* propertyNameString, Var\* setterValue, PropertyValueInfo\* info, ScriptContext\* requestContext)

{

if (!EnsureObjectReady(instance, DeferredInitializeMode\_Default))

{

return DescriptorFlags::None;

}

return GetCurrentTypeHandler(instance)->GetSetter(instance, propertyNameString, setterValue, info, requestContext);

}

template <DeferredTypeInitializer initializer, typename DeferredTypeFilter, bool isPrototypeTemplate, uint16 \_inlineSlotCapacity, uint16 \_offsetOfInlineSlots>

BOOL DeferredTypeHandler<initializer, DeferredTypeFilter, isPrototypeTemplate, \_inlineSlotCapacity, \_offsetOfInlineSlots>::DeleteProperty(DynamicObject\* instance, PropertyId propertyId, PropertyOperationFlags flags)

{

if (!EnsureObjectReady(instance, DeferredInitializeMode\_Default))

{

return FALSE;

}

return GetCurrentTypeHandler(instance)->DeleteProperty(instance, propertyId, flags);

}

template <DeferredTypeInitializer initializer, typename DeferredTypeFilter, bool isPrototypeTemplate, uint16 \_inlineSlotCapacity, uint16 \_offsetOfInlineSlots>

BOOL DeferredTypeHandler<initializer, DeferredTypeFilter, isPrototypeTemplate, \_inlineSlotCapacity, \_offsetOfInlineSlots>::HasItem(DynamicObject\* instance, uint32 index)

{

if (!EnsureObjectReady(instance, DeferredInitializeMode\_Default))

{

return FALSE;

}

return GetCurrentTypeHandler(instance)->HasItem(instance, index);

}

template <DeferredTypeInitializer initializer, typename DeferredTypeFilter, bool isPrototypeTemplate, uint16 \_inlineSlotCapacity, uint16 \_offsetOfInlineSlots>

BOOL DeferredTypeHandler<initializer, DeferredTypeFilter, isPrototypeTemplate, \_inlineSlotCapacity, \_offsetOfInlineSlots>::SetItem(DynamicObject\* instance, uint32 index, Var value, PropertyOperationFlags flags)

{

if (!EnsureObjectReady(instance, DeferredInitializeMode\_Default))

{

return FALSE;

}

return GetCurrentTypeHandler(instance)->SetItem(instance, index, value, flags);

}

template <DeferredTypeInitializer initializer, typename DeferredTypeFilter, bool isPrototypeTemplate, uint16 \_inlineSlotCapacity, uint16 \_offsetOfInlineSlots>

BOOL DeferredTypeHandler<initializer, DeferredTypeFilter, isPrototypeTemplate, \_inlineSlotCapacity, \_offsetOfInlineSlots>::SetItemWithAttributes(DynamicObject\* instance, uint32 index, Var value, PropertyAttributes attributes)

{

EnsureObjectReady(instance, DeferredInitializeMode\_Default);

return GetCurrentTypeHandler(instance)->SetItemWithAttributes(instance, index, value, attributes);

}

template <DeferredTypeInitializer initializer, typename DeferredTypeFilter, bool isPrototypeTemplate, uint16 \_inlineSlotCapacity, uint16 \_offsetOfInlineSlots>

BOOL DeferredTypeHandler<initializer, DeferredTypeFilter, isPrototypeTemplate, \_inlineSlotCapacity, \_offsetOfInlineSlots>::SetItemAttributes(DynamicObject\* instance, uint32 index, PropertyAttributes attributes)

{

if (!EnsureObjectReady(instance, DeferredInitializeMode\_Default))

{

return FALSE;

}

return GetCurrentTypeHandler(instance)->SetItemAttributes(instance, index, attributes);

}

template <DeferredTypeInitializer initializer, typename DeferredTypeFilter, bool isPrototypeTemplate, uint16 \_inlineSlotCapacity, uint16 \_offsetOfInlineSlots>

BOOL DeferredTypeHandler<initializer, DeferredTypeFilter, isPrototypeTemplate, \_inlineSlotCapacity, \_offsetOfInlineSlots>::SetItemAccessors(DynamicObject\* instance, uint32 index, Var getter, Var setter)

{

if (!EnsureObjectReady(instance, DeferredInitializeMode\_Default))

{

return FALSE;

}

return GetCurrentTypeHandler(instance)->SetItemAccessors(instance, index, getter, setter);

}

template <DeferredTypeInitializer initializer, typename DeferredTypeFilter, bool isPrototypeTemplate, uint16 \_inlineSlotCapacity, uint16 \_offsetOfInlineSlots>

BOOL DeferredTypeHandler<initializer, DeferredTypeFilter, isPrototypeTemplate, \_inlineSlotCapacity, \_offsetOfInlineSlots>::DeleteItem(DynamicObject\* instance, uint32 index, PropertyOperationFlags flags)

{

if (!EnsureObjectReady(instance, DeferredInitializeMode\_Default))

{

return FALSE;

}

return GetCurrentTypeHandler(instance)->DeleteItem(instance, index, flags);

}

template <DeferredTypeInitializer initializer, typename DeferredTypeFilter, bool isPrototypeTemplate, uint16 \_inlineSlotCapacity, uint16 \_offsetOfInlineSlots>

BOOL DeferredTypeHandler<initializer, DeferredTypeFilter, isPrototypeTemplate, \_inlineSlotCapacity, \_offsetOfInlineSlots>::GetItem(DynamicObject\* instance, Var originalInstance, uint32 index, Var\* value, ScriptContext \* requestContext)

{

if (!EnsureObjectReady(instance, DeferredInitializeMode\_Default))

{

return FALSE;

}

return GetCurrentTypeHandler(instance)->GetItem(instance, originalInstance, index, value, requestContext);

}

template <DeferredTypeInitializer initializer, typename DeferredTypeFilter, bool isPrototypeTemplate, uint16 \_inlineSlotCapacity, uint16 \_offsetOfInlineSlots>

DescriptorFlags DeferredTypeHandler<initializer, DeferredTypeFilter, isPrototypeTemplate, \_inlineSlotCapacity, \_offsetOfInlineSlots>::GetItemSetter(DynamicObject\* instance, uint32 index, Var\* setterValue, ScriptContext\* requestContext)

{

if (!EnsureObjectReady(instance, DeferredInitializeMode\_Default))

{

return DescriptorFlags::None;

}

return GetCurrentTypeHandler(instance)->GetItemSetter(instance, index, setterValue, requestContext);

}

template <DeferredTypeInitializer initializer, typename DeferredTypeFilter, bool isPrototypeTemplate, uint16 \_inlineSlotCapacity, uint16 \_offsetOfInlineSlots>

BOOL DeferredTypeHandler<initializer, DeferredTypeFilter, isPrototypeTemplate, \_inlineSlotCapacity, \_offsetOfInlineSlots>::IsEnumerable(DynamicObject\* instance, PropertyId propertyId)

{

if (!EnsureObjectReady(instance, DeferredInitializeMode\_Default))

{

return FALSE;

}

return GetCurrentTypeHandler(instance)->IsEnumerable(instance, propertyId);

}

template <DeferredTypeInitializer initializer, typename DeferredTypeFilter, bool isPrototypeTemplate, uint16 \_inlineSlotCapacity, uint16 \_offsetOfInlineSlots>

BOOL DeferredTypeHandler<initializer, DeferredTypeFilter, isPrototypeTemplate, \_inlineSlotCapacity, \_offsetOfInlineSlots>::IsWritable(DynamicObject\* instance, PropertyId propertyId)

{

if (!EnsureObjectReady(instance, DeferredInitializeMode\_Default))

{

return FALSE;

}

return GetCurrentTypeHandler(instance)->IsWritable(instance, propertyId);

}

template <DeferredTypeInitializer initializer, typename DeferredTypeFilter, bool isPrototypeTemplate, uint16 \_inlineSlotCapacity, uint16 \_offsetOfInlineSlots>

BOOL DeferredTypeHandler<initializer, DeferredTypeFilter, isPrototypeTemplate, \_inlineSlotCapacity, \_offsetOfInlineSlots>::IsConfigurable(DynamicObject\* instance, PropertyId propertyId)

{

if (!EnsureObjectReady(instance, DeferredInitializeMode\_Default))

{

return FALSE;

}

return GetCurrentTypeHandler(instance)->IsConfigurable(instance, propertyId);

}

template <DeferredTypeInitializer initializer, typename DeferredTypeFilter, bool isPrototypeTemplate, uint16 \_inlineSlotCapacity, uint16 \_offsetOfInlineSlots>

BOOL DeferredTypeHandler<initializer, DeferredTypeFilter, isPrototypeTemplate, \_inlineSlotCapacity, \_offsetOfInlineSlots>::SetEnumerable(DynamicObject\* instance, PropertyId propertyId, BOOL value)

{

if (!EnsureObjectReady(instance, DeferredInitializeMode\_Default))

{

return FALSE;

}

return GetCurrentTypeHandler(instance)->SetEnumerable(instance, propertyId, value);

}

template <DeferredTypeInitializer initializer, typename DeferredTypeFilter, bool isPrototypeTemplate, uint16 \_inlineSlotCapacity, uint16 \_offsetOfInlineSlots>

BOOL DeferredTypeHandler<initializer, DeferredTypeFilter, isPrototypeTemplate, \_inlineSlotCapacity, \_offsetOfInlineSlots>::SetWritable(DynamicObject\* instance, PropertyId propertyId, BOOL value)

{

if (!EnsureObjectReady(instance, DeferredInitializeMode\_Default))

{

return FALSE;

}

return GetCurrentTypeHandler(instance)->SetWritable(instance, propertyId, value);

}

template <DeferredTypeInitializer initializer, typename DeferredTypeFilter, bool isPrototypeTemplate, uint16 \_inlineSlotCapacity, uint16 \_offsetOfInlineSlots>

BOOL DeferredTypeHandler<initializer, DeferredTypeFilter, isPrototypeTemplate, \_inlineSlotCapacity, \_offsetOfInlineSlots>::SetConfigurable(DynamicObject\* instance, PropertyId propertyId, BOOL value)

{

if (!EnsureObjectReady(instance, DeferredInitializeMode\_Default))

{

return FALSE;

}

return GetCurrentTypeHandler(instance)->SetConfigurable(instance, propertyId, value);

}

template <DeferredTypeInitializer initializer, typename DeferredTypeFilter, bool isPrototypeTemplate, uint16 \_inlineSlotCapacity, uint16 \_offsetOfInlineSlots>

BOOL DeferredTypeHandler<initializer, DeferredTypeFilter, isPrototypeTemplate, \_inlineSlotCapacity, \_offsetOfInlineSlots>::SetAccessors(DynamicObject\* instance, PropertyId propertyId, Var getter, Var setter, PropertyOperationFlags flags)

{

if (!EnsureObjectReady(instance, DeferredInitializeMode\_SetAccessors))

{

return FALSE;

}

return GetCurrentTypeHandler(instance)->SetAccessors(instance, propertyId, getter, setter, flags);

}

template <DeferredTypeInitializer initializer, typename DeferredTypeFilter, bool isPrototypeTemplate, uint16 \_inlineSlotCapacity, uint16 \_offsetOfInlineSlots>

BOOL DeferredTypeHandler<initializer, DeferredTypeFilter, isPrototypeTemplate, \_inlineSlotCapacity, \_offsetOfInlineSlots>::GetAccessors(DynamicObject\* instance, PropertyId propertyId, Var \*getter, Var \*setter)

{

if (!EnsureObjectReady(instance, DeferredInitializeMode\_Default))

{

return FALSE;

}

return GetCurrentTypeHandler(instance)->GetAccessors(instance, propertyId, getter, setter);

}

template <DeferredTypeInitializer initializer, typename DeferredTypeFilter, bool isPrototypeTemplate, uint16 \_inlineSlotCapacity, uint16 \_offsetOfInlineSlots>

BOOL DeferredTypeHandler<initializer, DeferredTypeFilter, isPrototypeTemplate, \_inlineSlotCapacity, \_offsetOfInlineSlots>::IsSealed(DynamicObject \*instance)

{

if (!EnsureObjectReady(instance, DeferredInitializeMode\_Default))

{

return FALSE;

}

return GetCurrentTypeHandler(instance)->IsSealed(instance);

}

template <DeferredTypeInitializer initializer, typename DeferredTypeFilter, bool isPrototypeTemplate, uint16 \_inlineSlotCapacity, uint16 \_offsetOfInlineSlots>

BOOL DeferredTypeHandler<initializer, DeferredTypeFilter, isPrototypeTemplate, \_inlineSlotCapacity, \_offsetOfInlineSlots>::IsFrozen(DynamicObject \*instance)

{

if (!EnsureObjectReady(instance, DeferredInitializeMode\_Default))

{

return FALSE;

}

return GetCurrentTypeHandler(instance)->IsFrozen(instance);

}

template <DeferredTypeInitializer initializer, typename DeferredTypeFilter, bool isPrototypeTemplate, uint16 \_inlineSlotCapacity, uint16 \_offsetOfInlineSlots>

BOOL DeferredTypeHandler<initializer, DeferredTypeFilter, isPrototypeTemplate, \_inlineSlotCapacity, \_offsetOfInlineSlots>::PreventExtensions(DynamicObject\* instance)

{

if (!EnsureObjectReady(instance, DeferredInitializeMode\_Extensions))

{

return FALSE;

}

return GetCurrentTypeHandler(instance)->PreventExtensions(instance);

}

template <DeferredTypeInitializer initializer, typename DeferredTypeFilter, bool isPrototypeTemplate, uint16 \_inlineSlotCapacity, uint16 \_offsetOfInlineSlots>

BOOL DeferredTypeHandler<initializer, DeferredTypeFilter, isPrototypeTemplate, \_inlineSlotCapacity, \_offsetOfInlineSlots>::Seal(DynamicObject\* instance)

{

if (!EnsureObjectReady(instance, DeferredInitializeMode\_Extensions))

{

return FALSE;

}

return GetCurrentTypeHandler(instance)->Seal(instance);

}

template <DeferredTypeInitializer initializer, typename DeferredTypeFilter, bool isPrototypeTemplate, uint16 \_inlineSlotCapacity, uint16 \_offsetOfInlineSlots>

BOOL DeferredTypeHandler<initializer, DeferredTypeFilter, isPrototypeTemplate, \_inlineSlotCapacity, \_offsetOfInlineSlots>::FreezeImpl(DynamicObject\* instance, bool isConvertedType)

{

if (!EnsureObjectReady(instance, DeferredInitializeMode\_Extensions))

{

return FALSE;

}

return GetCurrentTypeHandler(instance)->Freeze(instance, true);

}

template <DeferredTypeInitializer initializer, typename DeferredTypeFilter, bool isPrototypeTemplate, uint16 \_inlineSlotCapacity, uint16 \_offsetOfInlineSlots>

BOOL DeferredTypeHandler<initializer, DeferredTypeFilter, isPrototypeTemplate, \_inlineSlotCapacity, \_offsetOfInlineSlots>::SetPropertyWithAttributes(DynamicObject\* instance, PropertyId propertyId, Var value, PropertyAttributes attributes, PropertyValueInfo\* info, PropertyOperationFlags flags, SideEffects possibleSideEffects)

{

if (!EnsureObjectReady(instance, DeferredInitializeMode\_Set))

{

return FALSE;

}

return GetCurrentTypeHandler(instance)->SetPropertyWithAttributes(instance, propertyId, value, attributes, info, flags, possibleSideEffects);

}

template <DeferredTypeInitializer initializer, typename DeferredTypeFilter, bool isPrototypeTemplate, uint16 \_inlineSlotCapacity, uint16 \_offsetOfInlineSlots>

BOOL DeferredTypeHandler<initializer, DeferredTypeFilter, isPrototypeTemplate, \_inlineSlotCapacity, \_offsetOfInlineSlots>::SetAttributes(DynamicObject\* instance, PropertyId propertyId, PropertyAttributes attributes)

{

if (!EnsureObjectReady(instance, DeferredInitializeMode\_Set))

{

return FALSE;

}

return GetCurrentTypeHandler(instance)->SetAttributes(instance, propertyId, attributes);

}

template <DeferredTypeInitializer initializer, typename DeferredTypeFilter, bool isPrototypeTemplate, uint16 \_inlineSlotCapacity, uint16 \_offsetOfInlineSlots>

BOOL DeferredTypeHandler<initializer, DeferredTypeFilter, isPrototypeTemplate, \_inlineSlotCapacity, \_offsetOfInlineSlots>::GetAttributesWithPropertyIndex(DynamicObject \* instance, PropertyId propertyId, BigPropertyIndex index, PropertyAttributes \* attributes)

{

if (!EnsureObjectReady(instance, DeferredInitializeMode\_Default))

{

return FALSE;

}

return GetCurrentTypeHandler(instance)->GetAttributesWithPropertyIndex(instance, propertyId, index, attributes);

}

template <DeferredTypeInitializer initializer, typename DeferredTypeFilter, bool isPrototypeTemplate, uint16 \_inlineSlotCapacity, uint16 \_offsetOfInlineSlots>

DynamicTypeHandler\* DeferredTypeHandler<initializer, DeferredTypeFilter, isPrototypeTemplate, \_inlineSlotCapacity, \_offsetOfInlineSlots>::ConvertToTypeWithItemAttributes(DynamicObject\* instance)

{

if (!EnsureObjectReady(instance, DeferredInitializeMode\_Set))

{

return nullptr;

}

return GetCurrentTypeHandler(instance)->ConvertToTypeWithItemAttributes(instance);

}

template <DeferredTypeInitializer initializer, typename DeferredTypeFilter, bool isPrototypeTemplate, uint16 \_inlineSlotCapacity, uint16 \_offsetOfInlineSlots>

void DeferredTypeHandler<initializer, DeferredTypeFilter, isPrototypeTemplate, \_inlineSlotCapacity, \_offsetOfInlineSlots>::SetIsPrototype(DynamicObject\* instance)

{

if (!isPrototypeTemplate)

{

// We don't force a type transition even when ChangeTypeOnProto() == true, because objects with NullTypeHandlers don't

// have any properties, so there is nothing to invalidate. Types with NullTypeHandlers also aren't cached in typeWithoutProperty

// caches, so there will be no fast property add path that could skip prototype cache invalidation.

DeferredTypeHandlerBase\* protoTypeHandler = DeferredTypeHandler<initializer, DeferredTypeFilter, true, \_inlineSlotCapacity, \_offsetOfInlineSlots>::GetDefaultInstance();

AssertMsg(protoTypeHandler->GetFlags() == (GetFlags() | IsPrototypeFlag), "Why did we change the flags of a DeferredTypeHandler?");

Assert(this->GetIsInlineSlotCapacityLocked() == protoTypeHandler->GetIsInlineSlotCapacityLocked());

protoTypeHandler->SetPropertyTypes(PropertyTypesWritableDataOnly | PropertyTypesWritableDataOnlyDetection, GetPropertyTypes());

SetInstanceTypeHandler(instance, protoTypeHandler);

}

}

}

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

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

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

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

#pragma once

namespace Js

{

template <typename TPropertyIndex>

class DictionaryPropertyDescriptor

{

template <typename T>

friend class DictionaryPropertyDescriptor;

public:

DictionaryPropertyDescriptor(TPropertyIndex dataSlot, bool isInitialized = false, bool isFixed = false, bool usedAsFixed = false) :

Data(dataSlot), Getter(NoSlots), Setter(NoSlots), Attributes(PropertyDynamicTypeDefaults),

PreventFalseReference(true), IsInitialized(isInitialized), IsOnlyOneAccessorInitialized(false), IsFixed(isFixed), UsedAsFixed(usedAsFixed), IsShadowed(false), IsAccessor(false) { }

DictionaryPropertyDescriptor(TPropertyIndex getterSlot, TPropertyIndex setterSlot, bool isInitialized = false, bool isFixed = false, bool usedAsFixed = false) :

Data(NoSlots), Getter(getterSlot), Setter(setterSlot), Attributes(PropertyDynamicTypeDefaults),

PreventFalseReference(true), IsInitialized(isInitialized), IsOnlyOneAccessorInitialized(false), IsFixed(isFixed), UsedAsFixed(usedAsFixed), IsShadowed(false), IsAccessor(true) { }

DictionaryPropertyDescriptor(TPropertyIndex dataSlot, PropertyAttributes attributes, bool isInitialized = false, bool isFixed = false, bool usedAsFixed = false) :

Data(dataSlot), Getter(NoSlots), Setter(NoSlots), Attributes(attributes),

PreventFalseReference(true), IsInitialized(isInitialized), IsOnlyOneAccessorInitialized(false), IsFixed(isFixed), UsedAsFixed(usedAsFixed), IsShadowed(false), IsAccessor(false) { }

DictionaryPropertyDescriptor(TPropertyIndex getterSlot, TPropertyIndex setterSlot, PropertyAttributes attributes, bool isInitialized = false, bool isFixed = false, bool usedAsFixed = false) :

Data(NoSlots), Getter(getterSlot), Setter(setterSlot), Attributes(attributes),

PreventFalseReference(true), IsInitialized(isInitialized), IsOnlyOneAccessorInitialized(false), IsFixed(isFixed), UsedAsFixed(usedAsFixed), IsShadowed(false), IsAccessor(true) { }

// this is for initialization.

DictionaryPropertyDescriptor() : Data(NoSlots), Getter(NoSlots), Setter(NoSlots), Attributes(PropertyDynamicTypeDefaults),

PreventFalseReference(true), IsInitialized(false), IsOnlyOneAccessorInitialized(false), IsFixed(false), UsedAsFixed(false), IsShadowed(false), IsAccessor(false) { }

template <typename TPropertyIndexFrom>

void CopyFrom(DictionaryPropertyDescriptor<TPropertyIndexFrom>& descriptor);

// SimpleDictionaryPropertyDescriptor is allocated by a dictionary along with the PropertyRecord

// so it cannot be allocated as leaf, tag the lower bit to prevent false reference.

bool PreventFalseReference:1;

bool IsInitialized:1;

bool IsOnlyOneAccessorInitialized:1;

bool IsFixed:1;

bool UsedAsFixed:1;

bool IsShadowed : 1;

bool IsAccessor : 1;

PropertyAttributes Attributes;

private:

TPropertyIndex Data;

// CONSIDER: sharing the Data slot with one of these and use the attributes to tell it apart.

TPropertyIndex Getter;

TPropertyIndex Setter;

public:

template <bool allowLetConstGlobal>

TPropertyIndex GetDataPropertyIndex() const;

TPropertyIndex GetGetterPropertyIndex() const;

TPropertyIndex GetSetterPropertyIndex() const;

void ConvertToData();

bool ConvertToGetterSetter(TPropertyIndex& nextSlotIndex);

bool HasNonLetConstGlobal() const

{

return (this->Attributes & PropertyLetConstGlobal) == 0 || this->IsShadowed;

}

void AddShadowedData(TPropertyIndex& nextPropertyIndex, bool addingLetConstGlobal);

private:

static const TPropertyIndex NoSlots = PropertyIndexRanges<TPropertyIndex>::NoSlots;

public:

#if DBG

bool SanityCheckFixedBits()

{

return

((!this->IsFixed && !this->UsedAsFixed) ||

(!(this->Attributes & PropertyDeleted) && (this->Data != NoSlots || this->Getter != NoSlots || this->Setter != NoSlots)));

}

#endif

};

template <typename TPropertyIndex>

template <bool allowLetConstGlobal>

TPropertyIndex DictionaryPropertyDescriptor<TPropertyIndex>::GetDataPropertyIndex() const

{

// If it is let const global, the data slot is the let const property, and if we allow let const global,

// we already return that the Getter/Setter slot may be doubled as the Data Slot

// so only return it if we allow let const

bool const isLetConstGlobal = (this->Attributes & PropertyLetConstGlobal) != 0;

if (isLetConstGlobal)

{

Assert(this->Data != NoSlots); // Should always have slot for LetConstGlobal if specified

if (allowLetConstGlobal)

{

return this->Data;

}

else if (this->IsShadowed && !this->IsAccessor)

{

// if it is a let const global, if the setter slot is empty, then the Getter slot must be

// the shadowed data slot, return that.

return this->Getter;

}

}

else

{

Assert(!this->IsAccessor || this->Data == NoSlots);

return this->Data;

}

return NoSlots;

}

template <typename TPropertyIndex>

TPropertyIndex DictionaryPropertyDescriptor<TPropertyIndex>::GetGetterPropertyIndex() const

{

// Need to check data property index first

Assert(GetDataPropertyIndex<false>() == NoSlots);

return this->Getter;

}

template <typename TPropertyIndex>

TPropertyIndex DictionaryPropertyDescriptor<TPropertyIndex>::GetSetterPropertyIndex() const

{

// Need to check data property index first

Assert(GetDataPropertyIndex<false>() == NoSlots);

return this->Setter;

}

template <typename TPropertyIndex>

void DictionaryPropertyDescriptor<TPropertyIndex>::ConvertToData()

{

if (this->IsAccessor)

{

Assert(this->Getter != NoSlots && this->Setter != NoSlots);

this->IsAccessor = false;

if (this->IsShadowed)

{

Assert(this->Data != NoSlots);

}

else

{

Assert(this->Data == NoSlots);

this->Data = this->Getter;

this->Getter = NoSlots;

}

}

Assert(GetDataPropertyIndex<false>() != NoSlots);

}

template <typename TPropertyIndex>

void DictionaryPropertyDescriptor<TPropertyIndex>::AddShadowedData(TPropertyIndex& nextPropertyIndex, bool addingLetConstGlobal)

{

Assert(!this->IsShadowed);

this->IsShadowed = true;

if (this->IsAccessor)

{

Assert(this->Data == NoSlots);

}

else if (addingLetConstGlobal)

{

this->Getter = this->Data;

this->Data = nextPropertyIndex++;

}

else

{

this->Getter = nextPropertyIndex++;

}

Assert(GetDataPropertyIndex<false>() != NoSlots);

}

template <typename TPropertyIndex>

bool DictionaryPropertyDescriptor<TPropertyIndex>::ConvertToGetterSetter(TPropertyIndex& nextPropertyIndex)

{

// Initial descriptor state and corresponding conversion can be one of the following:

//

// | State | Data | Getter | Setter | Conversion |

// |------------------------------------|---------|----------|----------|---------------------------------------------------------------------------------------------------|

// | Data property | valid | NoSlots? | NoSlots? | Move Data to Getter, set Data to NoSlots, create new slot for Setter if necessary, set IsAccessor |

// | LetConstGlobal | valid | NoSlots? | NoSlots? | Create new slots for Getter and Setter if necessary, set IsAccessor, set IsShadowed |

// | Data property + LetConstGlobal | valid | valid | NoSlots? | Create new slot for Setter if necessary, set IsAccessor |

// | Accessor property | NoSlots | valid | valid | Nothing |

// | Accessor property + LetConstGlobal | valid | valid | valid | Nothing |

// |------------------------------------|---------|----------|----------|---------------------------------------------------------------------------------------------------|

//

// NOTE: Do not create slot for Getter/Setter if they are already valid; possible after previous conversion from Accessor to Data, or deletion of Accessor, etc.

if (this->IsAccessor)

{

// Accessor property

// Accessor property + LetConstGlobal

Assert(this->Getter != NoSlots && this->Setter != NoSlots);

return false;

}

this->IsAccessor = true;

if (this->Attributes & PropertyLetConstGlobal)

{

if (this->IsShadowed)

{

// Data property + LetConstGlobal

Assert(this->Getter != NoSlots);

}

else

{

// LetConstGlobal

this->IsShadowed = true;

}

}

else

{

// Data property

Assert(this->Data != NoSlots);

Assert(this->Getter == NoSlots);

this->Getter = this->Data;

this->Data = NoSlots;

}

bool addedPropertyIndex = false;

if (this->Getter == NoSlots)

{

this->Getter = nextPropertyIndex++;

addedPropertyIndex = true;

}

if (this->Setter == NoSlots)

{

this->Setter = nextPropertyIndex++;

addedPropertyIndex = true;

}

Assert(this->GetGetterPropertyIndex() != NoSlots || this->GetSetterPropertyIndex() != NoSlots);

return addedPropertyIndex;

}

template <typename TPropertyIndex>

template <typename TPropertyIndexFrom>

void DictionaryPropertyDescriptor<TPropertyIndex>::CopyFrom(DictionaryPropertyDescriptor<TPropertyIndexFrom>& descriptor)

{

this->Attributes = descriptor.Attributes;

this->Data = (descriptor.Data == DictionaryPropertyDescriptor<TPropertyIndexFrom>::NoSlots) ? NoSlots : descriptor.Data;

this->Getter = (descriptor.Getter == DictionaryPropertyDescriptor<TPropertyIndexFrom>::NoSlots) ? NoSlots : descriptor.Getter;

this->Setter = (descriptor.Setter == DictionaryPropertyDescriptor<TPropertyIndexFrom>::NoSlots) ? NoSlots : descriptor.Setter;

this->IsInitialized = descriptor.IsInitialized;

this->IsFixed = descriptor.IsFixed;

this->UsedAsFixed = descriptor.UsedAsFixed;

this->IsAccessor = descriptor.IsAccessor;

}

}

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

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

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

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

#include "RuntimeTypePch.h"

namespace Js

{

template <typename T>

DictionaryTypeHandlerBase<T>\* DictionaryTypeHandlerBase<T>::New(Recycler \* recycler, int initialCapacity, uint16 inlineSlotCapacity, uint16 offsetOfInlineSlots)

{

return NewTypeHandler<DictionaryTypeHandlerBase>(recycler, initialCapacity, inlineSlotCapacity, offsetOfInlineSlots);

}

template <typename T>

DictionaryTypeHandlerBase<T>::DictionaryTypeHandlerBase(Recycler\* recycler) :

DynamicTypeHandler(1),

nextPropertyIndex(0),

singletonInstance(nullptr)

{

SetIsInlineSlotCapacityLocked();

propertyMap = RecyclerNew(recycler, PropertyDescriptorMap, recycler, this->GetSlotCapacity());

}

template <typename T>

DictionaryTypeHandlerBase<T>::DictionaryTypeHandlerBase(Recycler\* recycler, int slotCapacity, uint16 inlineSlotCapacity, uint16 offsetOfInlineSlots) :

// Do not RoundUp passed in slotCapacity. This may be called by ConvertTypeHandler for an existing DynamicObject and should use the real existing slotCapacity.

DynamicTypeHandler(slotCapacity, inlineSlotCapacity, offsetOfInlineSlots),

nextPropertyIndex(0),

singletonInstance(nullptr)

{

SetIsInlineSlotCapacityLocked();

Assert(GetSlotCapacity() <= MaxPropertyIndexSize);

propertyMap = RecyclerNew(recycler, PropertyDescriptorMap, recycler, slotCapacity);

}

//

// Takes over a given dictionary typeHandler. Used only by subclass.

//

template <typename T>

DictionaryTypeHandlerBase<T>::DictionaryTypeHandlerBase(DictionaryTypeHandlerBase\* typeHandler) :

DynamicTypeHandler(typeHandler->GetSlotCapacity(), typeHandler->GetInlineSlotCapacity(), typeHandler->GetOffsetOfInlineSlots()),

propertyMap(typeHandler->propertyMap), nextPropertyIndex(typeHandler->nextPropertyIndex),

singletonInstance(typeHandler->singletonInstance)

{

Assert(typeHandler->GetIsInlineSlotCapacityLocked());

CopyPropertyTypes(PropertyTypesWritableDataOnly | PropertyTypesWritableDataOnlyDetection | PropertyTypesInlineSlotCapacityLocked, typeHandler->GetPropertyTypes());

}

template <typename T>

int DictionaryTypeHandlerBase<T>::GetPropertyCount()

{

return propertyMap->Count();

}

template <typename T>

PropertyId DictionaryTypeHandlerBase<T>::GetPropertyId(ScriptContext\* scriptContext, PropertyIndex index)

{

if (index < propertyMap->Count())

{

DictionaryPropertyDescriptor<T> descriptor = propertyMap->GetValueAt(index);

if (!(descriptor.Attributes & PropertyDeleted) && descriptor.HasNonLetConstGlobal())

{

return propertyMap->GetKeyAt(index)->GetPropertyId();

}

}

return Constants::NoProperty;

}

template <typename T>

PropertyId DictionaryTypeHandlerBase<T>::GetPropertyId(ScriptContext\* scriptContext, BigPropertyIndex index)

{

if (index < propertyMap->Count())

{

DictionaryPropertyDescriptor<T> descriptor = propertyMap->GetValueAt(index);

if (!(descriptor.Attributes & PropertyDeleted) && descriptor.HasNonLetConstGlobal())

{

return propertyMap->GetKeyAt(index)->GetPropertyId();

}

}

return Constants::NoProperty;

}

template <typename T>

BOOL DictionaryTypeHandlerBase<T>::FindNextProperty(ScriptContext\* scriptContext, PropertyIndex& index, JavascriptString\*\* propertyStringName,

PropertyId\* propertyId, PropertyAttributes\* attributes, Type\* type, DynamicType \*typeToEnumerate, bool requireEnumerable, bool enumSymbols)

{

Assert(propertyStringName);

Assert(propertyId);

Assert(type);

for(; index < propertyMap->Count(); ++index )

{

DictionaryPropertyDescriptor<T> descriptor = propertyMap->GetValueAt(index);

PropertyAttributes attribs = descriptor.Attributes;

if (!(attribs & PropertyDeleted) && (!requireEnumerable || (attribs & PropertyEnumerable)) &&

(!(attribs & PropertyLetConstGlobal) || descriptor.HasNonLetConstGlobal()))

{

const PropertyRecord\* propertyRecord = propertyMap->GetKeyAt(index);

// Skip this property if it is a symbol and we are not including symbol properties

if (!enumSymbols && propertyRecord->IsSymbol())

{

continue;

}

// Pass back attributes of this property so caller can use them if it needs

if (attributes != nullptr)

{

\*attributes = attribs;

}

\*propertyId = propertyRecord->GetPropertyId();

PropertyString\* propertyString = type->GetScriptContext()->GetPropertyString(\*propertyId);

\*propertyStringName = propertyString;

T dataSlot = descriptor.GetDataPropertyIndex<false>();

if (dataSlot != NoSlots && (attribs & PropertyWritable))

{

uint16 inlineOrAuxSlotIndex;

bool isInlineSlot;

PropertyIndexToInlineOrAuxSlotIndex(dataSlot, &inlineOrAuxSlotIndex, &isInlineSlot);

propertyString->UpdateCache(type, inlineOrAuxSlotIndex, isInlineSlot, descriptor.IsInitialized && !descriptor.IsFixed);

}

else

{

#ifdef DEBUG

PropertyCache const\* cache = propertyString->GetPropertyCache();

Assert(!cache || cache->type != type);

#endif

}

return TRUE;

}

}

return FALSE;

}

template <>

BOOL DictionaryTypeHandlerBase<BigPropertyIndex>::FindNextProperty(ScriptContext\* scriptContext, PropertyIndex& index, JavascriptString\*\* propertyString,

PropertyId\* propertyId, PropertyAttributes\* attributes, Type\* type, DynamicType \*typeToEnumerate, bool requireEnumerable, bool enumSymbols)

{

Assert(false);

Throw::InternalError();

}

template <typename T>

BOOL DictionaryTypeHandlerBase<T>::FindNextProperty(ScriptContext\* scriptContext, BigPropertyIndex& index, JavascriptString\*\* propertyString,

PropertyId\* propertyId, PropertyAttributes\* attributes, Type\* type, DynamicType \*typeToEnumerate, bool requireEnumerable, bool enumSymbols)

{

PropertyIndex local = (PropertyIndex)index;

Assert(index <= Constants::UShortMaxValue || index == Constants::NoBigSlot);

BOOL result = this->FindNextProperty(scriptContext, local, propertyString, propertyId, attributes, type, typeToEnumerate, requireEnumerable, enumSymbols);

index = local;

return result;

}

template <>

BOOL DictionaryTypeHandlerBase<BigPropertyIndex>::FindNextProperty(ScriptContext\* scriptContext, BigPropertyIndex& index, JavascriptString\*\* propertyStringName,

PropertyId\* propertyId, PropertyAttributes\* attributes, Type\* type, DynamicType \*typeToEnumerate, bool requireEnumerable, bool enumSymbols)

{

Assert(propertyStringName);

Assert(propertyId);

Assert(type);

for(; index < propertyMap->Count(); ++index )

{

DictionaryPropertyDescriptor<BigPropertyIndex> descriptor = propertyMap->GetValueAt(index);

PropertyAttributes attribs = descriptor.Attributes;

if (!(attribs & PropertyDeleted) && (!requireEnumerable || (attribs & PropertyEnumerable)) &&

(!(attribs & PropertyLetConstGlobal) || descriptor.HasNonLetConstGlobal()))

{

const PropertyRecord\* propertyRecord = propertyMap->GetKeyAt(index);

// Skip this property if it is a symbol and we are not including symbol properties

if (!enumSymbols && propertyRecord->IsSymbol())

{

continue;

}

if (attributes != nullptr)

{

\*attributes = attribs;

}

\*propertyId = propertyRecord->GetPropertyId();

\*propertyStringName = type->GetScriptContext()->GetPropertyString(\*propertyId);

return TRUE;

}

}

return FALSE;

}

template <typename T>

PropertyIndex DictionaryTypeHandlerBase<T>::GetPropertyIndex(PropertyRecord const\* propertyRecord)

{

return GetPropertyIndex\_Internal<false>(propertyRecord);

}

template <typename T>

PropertyIndex DictionaryTypeHandlerBase<T>::GetRootPropertyIndex(PropertyRecord const\* propertyRecord)

{

return GetPropertyIndex\_Internal<true>(propertyRecord);

}

template <typename T>

bool DictionaryTypeHandlerBase<T>::GetPropertyEquivalenceInfo(PropertyRecord const\* propertyRecord, PropertyEquivalenceInfo& info)

{

DictionaryPropertyDescriptor<T>\* descriptor;

if (this->propertyMap->TryGetReference(propertyRecord, &descriptor) && !(descriptor->Attributes & PropertyDeleted))

{

AssertMsg(descriptor->GetDataPropertyIndex<false>() != Constants::NoSlot, "We don't support equivalent object type spec on accessors.");

AssertMsg(descriptor->GetDataPropertyIndex<false>() <= Constants::PropertyIndexMax, "We don't support equivalent object type spec on big property indexes.");

T propertyIndex = descriptor->GetDataPropertyIndex<false>();

info.slotIndex = propertyIndex <= Constants::PropertyIndexMax ?

AdjustValidSlotIndexForInlineSlots(static\_cast<PropertyIndex>(propertyIndex)) : Constants::NoSlot;

info.isAuxSlot = propertyIndex >= GetInlineSlotCapacity();

info.isWritable = !!(descriptor->Attributes & PropertyWritable);

}

else

{

info.slotIndex = Constants::NoSlot;

info.isAuxSlot = false;

info.isWritable = false;

}

return info.slotIndex != Constants::NoSlot;

}

template <typename T>

bool DictionaryTypeHandlerBase<T>::IsObjTypeSpecEquivalent(const Type\* type, const TypeEquivalenceRecord& record, uint& failedPropertyIndex)

{

uint propertyCount = record.propertyCount;

EquivalentPropertyEntry\* properties = record.properties;

for (uint pi = 0; pi < propertyCount; pi++)

{

const EquivalentPropertyEntry\* refInfo = &properties[pi];

if (!this->IsObjTypeSpecEquivalentImpl<false>(type, refInfo))

{

failedPropertyIndex = pi;

return false;

}

}

return true;

}

template <typename T>

bool DictionaryTypeHandlerBase<T>::IsObjTypeSpecEquivalent(const Type\* type, const EquivalentPropertyEntry \*entry)

{

return this->IsObjTypeSpecEquivalentImpl<true>(type, entry);

}

template <typename T>

template <bool doLock>

bool DictionaryTypeHandlerBase<T>::IsObjTypeSpecEquivalentImpl(const Type\* type, const EquivalentPropertyEntry \*entry)

{

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

T absSlotIndex = Constants::NoSlot;

PropertyIndex relSlotIndex = Constants::NoSlot;

const PropertyRecord\* propertyRecord =

doLock ? scriptContext->GetPropertyNameLocked(entry->propertyId) : scriptContext->GetPropertyName(entry->propertyId);

DictionaryPropertyDescriptor<T>\* descriptor;

if (this->propertyMap->TryGetReference(propertyRecord, &descriptor) && !(descriptor->Attributes & PropertyDeleted))

{

// We don't object type specialize accessors at this point, so if we see an accessor on an object we must have a mismatch.

// When we add support for accessors we will need another bit on EquivalentPropertyEntry indicating whether we expect

// a data or accessor property.

if (descriptor->IsAccessor)

{

return false;

}

absSlotIndex = descriptor->GetDataPropertyIndex<false>();

if (absSlotIndex <= Constants::PropertyIndexMax)

{

relSlotIndex = AdjustValidSlotIndexForInlineSlots(static\_cast<PropertyIndex>(absSlotIndex));

}

}

if (relSlotIndex != Constants::NoSlot)

{

if (relSlotIndex != entry->slotIndex || ((absSlotIndex >= GetInlineSlotCapacity()) != entry->isAuxSlot))

{

return false;

}

if (entry->mustBeWritable && (!(descriptor->Attributes & PropertyWritable) || !descriptor->IsInitialized || descriptor->IsFixed))

{

return false;

}

}

else

{

if (entry->slotIndex != Constants::NoSlot || entry->mustBeWritable)

{

return false;

}

}

return true;

}

template <typename T>

template <bool allowLetConstGlobal>

PropertyIndex DictionaryTypeHandlerBase<T>::GetPropertyIndex\_Internal(PropertyRecord const\* propertyRecord)

{

DictionaryPropertyDescriptor<T>\* descriptor;

if (propertyMap->TryGetReference(propertyRecord, &descriptor) && !(descriptor->Attributes & PropertyDeleted))

{

return descriptor->GetDataPropertyIndex<allowLetConstGlobal>();

}

else

{

return NoSlots;

}

}

template <>

template <bool allowLetConstGlobal>

PropertyIndex DictionaryTypeHandlerBase<BigPropertyIndex>::GetPropertyIndex\_Internal(PropertyRecord const\* propertyRecord)

{

DictionaryPropertyDescriptor<BigPropertyIndex>\* descriptor;

if (propertyMap->TryGetReference(propertyRecord, &descriptor) && !(descriptor->Attributes & PropertyDeleted))

{

BigPropertyIndex dataPropertyIndex = descriptor->GetDataPropertyIndex<allowLetConstGlobal>();

if(dataPropertyIndex < Constants::NoSlot)

{

return (PropertyIndex)dataPropertyIndex;

}

}

return Constants::NoSlot;

}

template <>

PropertyIndex DictionaryTypeHandlerBase<BigPropertyIndex>::GetRootPropertyIndex(PropertyRecord const\* propertyRecord)

{

return Constants::NoSlot;

}

template <typename T>

void DictionaryTypeHandlerBase<T>::Add(

const PropertyRecord\* propertyId,

PropertyAttributes attributes,

ScriptContext \*const scriptContext)

{

return Add(propertyId, attributes, true, false, false, scriptContext);

}

template <typename T>

void DictionaryTypeHandlerBase<T>::Add(

const PropertyRecord\* propertyId,

PropertyAttributes attributes,

bool isInitialized, bool isFixed, bool usedAsFixed,

ScriptContext \*const scriptContext)

{

Assert(this->GetSlotCapacity() <= MaxPropertyIndexSize); // slotCapacity should never exceed MaxPropertyIndexSize

Assert(nextPropertyIndex < this->GetSlotCapacity()); // nextPropertyIndex must be ready

T index = nextPropertyIndex++;

DictionaryPropertyDescriptor<T> descriptor(index, attributes);

Assert((!isFixed && !usedAsFixed) || (!IsInternalPropertyId(propertyId->GetPropertyId()) && this->singletonInstance != nullptr));

descriptor.IsInitialized = isInitialized;

descriptor.IsFixed = isFixed;

descriptor.UsedAsFixed = usedAsFixed;

propertyMap->Add(propertyId, descriptor);

if (!(attributes & PropertyWritable))

{

this->ClearHasOnlyWritableDataProperties();

if(GetFlags() & IsPrototypeFlag)

{

scriptContext->InvalidateStoreFieldCaches(propertyId->GetPropertyId());

scriptContext->GetLibrary()->NoPrototypeChainsAreEnsuredToHaveOnlyWritableDataProperties();

}

}

}

template <typename T>

BOOL DictionaryTypeHandlerBase<T>::HasProperty(DynamicObject\* instance, PropertyId propertyId, bool \*noRedecl)

{

return HasProperty\_Internal<false>(instance, propertyId, noRedecl, nullptr);

}

template <typename T>

BOOL DictionaryTypeHandlerBase<T>::HasRootProperty(DynamicObject\* instance, PropertyId propertyId, bool \*noRedecl, bool \*pDeclaredProperty = nullptr)

{

return HasProperty\_Internal<true>(instance, propertyId, noRedecl, pDeclaredProperty);

}

template <typename T>

template <bool allowLetConstGlobal>

BOOL DictionaryTypeHandlerBase<T>::HasProperty\_Internal(DynamicObject\* instance, PropertyId propertyId, bool \*noRedecl, bool \*pDeclaredProperty)

{

// HasProperty is called with NoProperty in JavascriptDispatch.cpp to for undeferral of the

// deferred type system that DOM objects use. Allow NoProperty for this reason, but only

// here in HasProperty.

if (propertyId == Constants::NoProperty)

{

return false;

}

DictionaryPropertyDescriptor<T>\* descriptor;

Assert(propertyId != Constants::NoProperty);

PropertyRecord const\* propertyRecord = instance->GetScriptContext()->GetPropertyName(propertyId);

if (propertyMap->TryGetReference(propertyRecord, &descriptor))

{

if ((descriptor->Attributes & PropertyDeleted) || (!allowLetConstGlobal && !descriptor->HasNonLetConstGlobal()))

{

return false;

}

if (noRedecl && descriptor->Attributes & PropertyNoRedecl)

{

\*noRedecl = true;

}

if (pDeclaredProperty && descriptor->Attributes & (PropertyNoRedecl | PropertyDeclaredGlobal))

{

\*pDeclaredProperty = true;

}

return true;

}

// Check numeric propertyRecord only if objectArray available

if (instance->HasObjectArray() && propertyRecord->IsNumeric())

{

return DictionaryTypeHandlerBase<T>::HasItem(instance, propertyRecord->GetNumericValue());

}

return false;

}

template <typename T>

BOOL DictionaryTypeHandlerBase<T>::HasProperty(DynamicObject\* instance, JavascriptString\* propertyNameString)

{

AssertMsg(!PropertyRecord::IsPropertyNameNumeric(propertyNameString->GetString(), propertyNameString->GetLength()),

"Numeric property names should have been converted to uint or PropertyRecord\* before calling GetSetter");

JsUtil::CharacterBuffer<WCHAR> propertyName(propertyNameString->GetString(), propertyNameString->GetLength());

DictionaryPropertyDescriptor<T>\* descriptor;

if (propertyMap->TryGetReference(propertyName, &descriptor))

{

if ((descriptor->Attributes & PropertyDeleted) || !descriptor->HasNonLetConstGlobal())

{

return false;

}

return true;

}

return false;

}

template <typename T>

BOOL DictionaryTypeHandlerBase<T>::GetRootProperty(DynamicObject\* instance, Var originalInstance, PropertyId propertyId,

Var\* value, PropertyValueInfo\* info, ScriptContext\* requestContext)

{

AssertMsg(RootObjectBase::Is(instance), "Instance must be a root object!");

return GetProperty\_Internal<true>(instance, originalInstance, propertyId, value, info, requestContext);

}

template <typename T>

BOOL DictionaryTypeHandlerBase<T>::GetProperty(DynamicObject\* instance, Var originalInstance, PropertyId propertyId,

Var\* value, PropertyValueInfo\* info, ScriptContext\* requestContext)

{

return GetProperty\_Internal<false>(instance, originalInstance, propertyId, value, info, requestContext);

}

template <typename T>

template <bool allowLetConstGlobal, typename PropertyType>

BOOL DictionaryTypeHandlerBase<T>::GetPropertyFromDescriptor(DynamicObject\* instance, Var originalInstance,

DictionaryPropertyDescriptor<T>\* descriptor, Var\* value, PropertyValueInfo\* info, PropertyType propertyT, ScriptContext\* requestContext)

{

bool const isLetConstGlobal = (descriptor->Attributes & PropertyLetConstGlobal) != 0;

AssertMsg(!isLetConstGlobal || RootObjectBase::Is(instance), "object must be a global object if letconstglobal is set");

if (allowLetConstGlobal)

{

// GetRootProperty: false if not global

if (!(descriptor->Attributes & PropertyLetConstGlobal) && (descriptor->Attributes & PropertyDeleted))

{

return false;

}

}

else

{

// GetProperty: don't count deleted or global.

if (descriptor->Attributes & (PropertyDeleted | (descriptor->IsShadowed ? 0 : PropertyLetConstGlobal)))

{

return false;

}

}

T dataSlot = descriptor->GetDataPropertyIndex<allowLetConstGlobal>();

if (dataSlot != NoSlots)

{

\*value = instance->GetSlot(dataSlot);

SetPropertyValueInfo(info, instance, dataSlot, descriptor->Attributes);

if (!descriptor->IsInitialized || descriptor->IsFixed)

{

PropertyValueInfo::DisableStoreFieldCache(info);

}

if (descriptor->Attributes & PropertyDeleted)

{

// letconst shadowing a deleted property. don't bother to cache

PropertyValueInfo::SetNoCache(info, instance);

}

}

else if (descriptor->GetGetterPropertyIndex() != NoSlots)

{

// We must update cache before calling a getter, because it can invalidate something. Bug# 593815

SetPropertyValueInfo(info, instance, descriptor->GetGetterPropertyIndex(), descriptor->Attributes);

CacheOperators::CachePropertyReadForGetter(info, originalInstance, propertyT, requestContext);

PropertyValueInfo::SetNoCache(info, instance); // we already cached getter, so we don't have to do it once more

RecyclableObject\* func = RecyclableObject::FromVar(instance->GetSlot(descriptor->GetGetterPropertyIndex()));

\*value = JavascriptOperators::CallGetter(func, originalInstance, requestContext);

return true;

}

else

{

\*value = instance->GetLibrary()->GetUndefined();

return true;

}

return true;

}

template <typename T>

template <bool allowLetConstGlobal>

BOOL DictionaryTypeHandlerBase<T>::GetProperty\_Internal(DynamicObject\* instance, Var originalInstance, PropertyId propertyId,

Var\* value, PropertyValueInfo\* info, ScriptContext\* requestContext)

{

DictionaryPropertyDescriptor<T>\* descriptor;

Assert(propertyId != Constants::NoProperty);

PropertyRecord const\* propertyRecord = instance->GetScriptContext()->GetPropertyName(propertyId);

if (propertyMap->TryGetReference(propertyRecord, &descriptor))

{

return GetPropertyFromDescriptor<allowLetConstGlobal>(instance, originalInstance, descriptor, value, info, propertyId, requestContext);

}

// Check numeric propertyRecord only if objectArray available

if (instance->HasObjectArray() && propertyRecord->IsNumeric())

{

return DictionaryTypeHandlerBase<T>::GetItem(instance, originalInstance, propertyRecord->GetNumericValue(), value, requestContext);

}

return false;

}

template <typename T>

BOOL DictionaryTypeHandlerBase<T>::GetProperty(DynamicObject\* instance, Var originalInstance, JavascriptString\* propertyNameString,

Var\* value, PropertyValueInfo\* info, ScriptContext\* requestContext)

{

AssertMsg(!PropertyRecord::IsPropertyNameNumeric(propertyNameString->GetString(), propertyNameString->GetLength()),

"Numeric property names should have been converted to uint or PropertyRecord\* before calling GetSetter");

JsUtil::CharacterBuffer<WCHAR> propertyName(propertyNameString->GetString(), propertyNameString->GetLength());

DictionaryPropertyDescriptor<T>\* descriptor;

if (propertyMap->TryGetReference(propertyName, &descriptor))

{

return GetPropertyFromDescriptor<false>(instance, originalInstance, descriptor, value, info, propertyName, requestContext);

}

return false;

}

template <typename T>

void DictionaryTypeHandlerBase<T>::SetPropertyValueInfo(PropertyValueInfo\* info, RecyclableObject\* instance, T propIndex, PropertyAttributes attributes, InlineCacheFlags flags)

{

PropertyValueInfo::Set(info, instance, propIndex, attributes, flags);

}

template <>

void DictionaryTypeHandlerBase<BigPropertyIndex>::SetPropertyValueInfo(PropertyValueInfo\* info, RecyclableObject\* instance, BigPropertyIndex propIndex, PropertyAttributes attributes, InlineCacheFlags flags)

{

PropertyValueInfo::SetNoCache(info, instance);

}

template <typename T>

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

{

return GetSetter\_Internal<false>(instance, propertyId, setterValue, info, requestContext);

}

template <typename T>

DescriptorFlags DictionaryTypeHandlerBase<T>::GetRootSetter(DynamicObject\* instance, PropertyId propertyId, Var\* setterValue, PropertyValueInfo\* info, ScriptContext\* requestContext)

{

AssertMsg(RootObjectBase::Is(instance), "Instance must be a root object!");

return GetSetter\_Internal<true>(instance, propertyId, setterValue, info, requestContext);

}

template <typename T>

template <bool allowLetConstGlobal>

DescriptorFlags DictionaryTypeHandlerBase<T>::GetSetter\_Internal(DynamicObject\* instance, PropertyId propertyId, Var\* setterValue, PropertyValueInfo\* info, ScriptContext\* requestContext)

{

DictionaryPropertyDescriptor<T>\* descriptor;

Assert(propertyId != Constants::NoProperty);

PropertyRecord const\* propertyRecord = instance->GetScriptContext()->GetPropertyName(propertyId);

if (propertyMap->TryGetReference(propertyRecord, &descriptor))

{

return GetSetterFromDescriptor<allowLetConstGlobal>(instance, descriptor, setterValue, info);

}

// Check numeric propertyRecord only if objectArray available

if (instance->HasObjectArray() && propertyRecord->IsNumeric())

{

return DictionaryTypeHandlerBase<T>::GetItemSetter(instance, propertyRecord->GetNumericValue(), setterValue, requestContext);

}

return None;

}

template <typename T>

template <bool allowLetConstGlobal>

DescriptorFlags DictionaryTypeHandlerBase<T>::GetSetterFromDescriptor(DynamicObject\* instance, DictionaryPropertyDescriptor<T> \* descriptor, Var\* setterValue, PropertyValueInfo\* info)

{

if (descriptor->Attributes & PropertyDeleted)

{

return None;

}

if (descriptor->GetDataPropertyIndex<allowLetConstGlobal>() != NoSlots)

{

// not a setter but shadows

if (allowLetConstGlobal && (descriptor->Attributes & PropertyLetConstGlobal))

{

return (descriptor->Attributes & PropertyConst) ? (DescriptorFlags)(Const | Data) : WritableData;

}

if (descriptor->Attributes & PropertyWritable)

{

return WritableData;

}

if (descriptor->Attributes & PropertyConst)

{

return (DescriptorFlags)(Const|Data);

}

return Data;

}

else if (descriptor->GetSetterPropertyIndex() != NoSlots)

{

\*setterValue=((DynamicObject\*)instance)->GetSlot(descriptor->GetSetterPropertyIndex());

SetPropertyValueInfo(info, instance, descriptor->GetSetterPropertyIndex(), descriptor->Attributes, InlineCacheSetterFlag);

return Accessor;

}

return None;

}

template <typename T>

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

{

AssertMsg(!PropertyRecord::IsPropertyNameNumeric(propertyNameString->GetString(), propertyNameString->GetLength()),

"Numeric property names should have been converted to uint or PropertyRecord\* before calling GetSetter");

JsUtil::CharacterBuffer<WCHAR> propertyName(propertyNameString->GetString(), propertyNameString->GetLength());

DictionaryPropertyDescriptor<T>\* descriptor;

if (propertyMap->TryGetReference(propertyName, &descriptor))

{

return GetSetterFromDescriptor<false>(instance, descriptor, setterValue, info);

}

return None;

}

template <typename T>

BOOL DictionaryTypeHandlerBase<T>::SetRootProperty(DynamicObject\* instance, PropertyId propertyId, Var value, PropertyOperationFlags flags, PropertyValueInfo\* info)

{

AssertMsg(RootObjectBase::Is(instance), "Instance must be a root object!");

return SetProperty\_Internal<true>(instance, propertyId, value, flags, info);

}

template <typename T>

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

{

return SetProperty\_Internal<false>(instance, propertyId, value, flags, info, true /\* IsInit \*/);

}

template <typename T>

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

{

return SetProperty\_Internal<false>(instance, propertyId, value, flags, info);

}

template <typename T>

template <bool allowLetConstGlobal>

void DictionaryTypeHandlerBase<T>::SetPropertyWithDescriptor(DynamicObject\* instance, PropertyId propertyId, DictionaryPropertyDescriptor<T> \* descriptor,

Var value, PropertyOperationFlags flags, PropertyValueInfo\* info)

{

Assert(instance);

Assert((descriptor->Attributes & PropertyDeleted) == 0 || (allowLetConstGlobal && descriptor->IsShadowed));

// DictionaryTypeHandlers are not supposed to be shared.

Assert(!GetIsOrMayBecomeShared());

DynamicObject\* localSingletonInstance = this->singletonInstance != nullptr ? this->singletonInstance->Get() : nullptr;

Assert(this->singletonInstance == nullptr || localSingletonInstance == instance);

T dataSlot = descriptor->GetDataPropertyIndex<allowLetConstGlobal>();

if (dataSlot != NoSlots)

{

if (allowLetConstGlobal

&& (descriptor->Attributes & PropertyNoRedecl)

&& !(flags & PropertyOperation\_AllowUndecl))

{

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

if (scriptContext->IsUndeclBlockVar(instance->GetSlot(dataSlot)))

{

JavascriptError::ThrowReferenceError(scriptContext, JSERR\_UseBeforeDeclaration);

}

}

if (!descriptor->IsInitialized)

{

if ((flags & PropertyOperation\_PreInit) == 0)

{

descriptor->IsInitialized = true;

if (localSingletonInstance == instance && !IsInternalPropertyId(propertyId) &&

(flags & (PropertyOperation\_NonFixedValue | PropertyOperation\_SpecialValue)) == 0)

{

Assert(value != nullptr);

// We don't want fixed properties on external objects. See DynamicObject::ResetObject for more information.

Assert(!instance->IsExternal());

descriptor->IsFixed = (JavascriptFunction::Is(value) ? ShouldFixMethodProperties() : (ShouldFixDataProperties() && CheckHeuristicsForFixedDataProps(instance, propertyId, value)));

}

}

}

else

{

InvalidateFixedField(instance, propertyId, descriptor);

}

SetSlotUnchecked(instance, dataSlot, value);

// If we just added a fixed method, don't populate the inline cache so that we always take the slow path

// when overwriting this property and correctly invalidate any JIT-ed code that hard-coded this method.

if (descriptor->IsInitialized && !descriptor->IsFixed)

{

SetPropertyValueInfo(info, instance, dataSlot, GetLetConstGlobalPropertyAttributes<allowLetConstGlobal>(descriptor->Attributes));

}

else

{

PropertyValueInfo::SetNoCache(info, instance);

}

}

else if (descriptor->GetSetterPropertyIndex() != NoSlots)

{

RecyclableObject\* func = RecyclableObject::FromVar(instance->GetSlot(descriptor->GetSetterPropertyIndex()));

JavascriptOperators::CallSetter(func, instance, value, NULL);

// Wait for the setter to return before setting up the inline cache info, as the setter may change

// the attributes

T dataSlot = descriptor->GetDataPropertyIndex<false>();

if (dataSlot != NoSlots)

{

SetPropertyValueInfo(info, instance, dataSlot, descriptor->Attributes);

}

else if (descriptor->GetSetterPropertyIndex() != NoSlots)

{

SetPropertyValueInfo(info, instance, descriptor->GetSetterPropertyIndex(), descriptor->Attributes, InlineCacheSetterFlag);

}

}

SetPropertyUpdateSideEffect(instance, propertyId, value, SideEffects\_Any);

}

template <typename T>

template <bool allowLetConstGlobal>

BOOL DictionaryTypeHandlerBase<T>::SetProperty\_Internal(DynamicObject\* instance, PropertyId propertyId, Var value, PropertyOperationFlags flags, PropertyValueInfo\* info, bool isInit)

{

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

DictionaryPropertyDescriptor<T>\* descriptor;

bool throwIfNotExtensible = (flags & (PropertyOperation\_ThrowIfNotExtensible | PropertyOperation\_StrictMode)) != 0;

bool isForce = (flags & PropertyOperation\_Force) != 0;

Assert(propertyId != Constants::NoProperty);

PropertyRecord const\* propertyRecord = scriptContext->GetPropertyName(propertyId);

if (propertyMap->TryGetReference(propertyRecord, &descriptor))

{

Assert(descriptor->SanityCheckFixedBits());

if (descriptor->Attributes & PropertyDeleted)

{

if (!isForce)

{

if (!this->VerifyIsExtensible(scriptContext, throwIfNotExtensible))

{

return false;

}

}

scriptContext->InvalidateProtoCaches(propertyId);

if (descriptor->Attributes & PropertyLetConstGlobal)

{

descriptor->Attributes = PropertyDynamicTypeDefaults | (descriptor->Attributes & (PropertyLetConstGlobal | PropertyNoRedecl));

}

else

{

descriptor->Attributes = PropertyDynamicTypeDefaults;

}

instance->SetHasNoEnumerableProperties(false);

descriptor->ConvertToData();

}

else if (!allowLetConstGlobal && descriptor->HasNonLetConstGlobal() && !(descriptor->Attributes & PropertyWritable))

{

if (!isForce)

{

JavascriptError::ThrowCantAssignIfStrictMode(flags, scriptContext);

}

// Since we separate LdFld and StFld caches there is no point in caching for StFld with non-writable properties, except perhaps

// to prepopulate the type property cache (which we do share between LdFld and StFld), for potential future field loads. This

// would require additional handling in CacheOperators::CachePropertyWrite, such that for !info-IsWritable() we don't populate

// the local cache (that would be illegal), but still populate the type's property cache.

PropertyValueInfo::SetNoCache(info, instance);

return false;

}

else if (isInit && descriptor->IsAccessor)

{

descriptor->ConvertToData();

}

SetPropertyWithDescriptor<allowLetConstGlobal>(instance, propertyId, descriptor, value, flags, info);

return true;

}

// Always check numeric propertyRecord. This may create objectArray.

if (propertyRecord->IsNumeric())

{

// Calls this or subclass implementation

return SetItem(instance, propertyRecord->GetNumericValue(), value, flags);

}

return this->AddProperty(instance, propertyRecord, value, PropertyDynamicTypeDefaults, info, flags, throwIfNotExtensible, SideEffects\_Any);

}

template <typename T>

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

{

// Either the property exists in the dictionary, in which case a PropertyRecord for it exists,

// or we have to add it to the dictionary, in which case we need to get or create a PropertyRecord.

// Thus, just get or create one and call the PropertyId overload of SetProperty.

PropertyRecord const \* propertyRecord;

instance->GetScriptContext()->GetOrAddPropertyRecord(propertyNameString->GetString(), propertyNameString->GetLength(), &propertyRecord);

return DictionaryTypeHandlerBase<T>::SetProperty(instance, propertyRecord->GetPropertyId(), value, flags, info);

}

template <typename T>

BOOL DictionaryTypeHandlerBase<T>::DeleteProperty(DynamicObject\* instance, PropertyId propertyId, PropertyOperationFlags propertyOperationFlags)

{

return DeleteProperty\_Internal<false>(instance, propertyId, propertyOperationFlags);

}

template <typename T>

BOOL DictionaryTypeHandlerBase<T>::DeleteRootProperty(DynamicObject\* instance, PropertyId propertyId, PropertyOperationFlags propertyOperationFlags)

{

AssertMsg(RootObjectBase::Is(instance), "Instance must be a root object!");

return DeleteProperty\_Internal<true>(instance, propertyId, propertyOperationFlags);

}

template <typename T>

template <bool allowLetConstGlobal>

BOOL DictionaryTypeHandlerBase<T>::DeleteProperty\_Internal(DynamicObject\* instance, PropertyId propertyId, PropertyOperationFlags propertyOperationFlags)

{

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

DictionaryPropertyDescriptor<T>\* descriptor;

Assert(propertyId != Constants::NoProperty);

PropertyRecord const\* propertyRecord = scriptContext->GetPropertyName(propertyId);

if (propertyMap->TryGetReference(propertyRecord, &descriptor))

{

Assert(descriptor->SanityCheckFixedBits());

if (descriptor->Attributes & PropertyDeleted)

{

// If PropertyDeleted and PropertyLetConstGlobal are set then we have both

// a deleted global property and let/const variable in this descriptor.

// If allowLetConstGlobal is true then the let/const shadows the property

// and we should return false for a failed delete by going into the else

// if branch below.

if (allowLetConstGlobal && (descriptor->Attributes & PropertyLetConstGlobal))

{

JavascriptError::ThrowCantDeleteIfStrictMode(propertyOperationFlags, scriptContext, propertyRecord->GetBuffer());

return false;

}

return true;

}

else if (!(descriptor->Attributes & PropertyConfigurable) ||

(allowLetConstGlobal && (descriptor->Attributes & PropertyLetConstGlobal)))

{

// Let/const properties do not have attributes and they cannot be deleted

JavascriptError::ThrowCantDeleteIfStrictMode(propertyOperationFlags, scriptContext, scriptContext->GetPropertyName(propertyId)->GetBuffer());

return false;

}

Var undefined = scriptContext->GetLibrary()->GetUndefined();

if (descriptor->HasNonLetConstGlobal())

{

T dataSlot = descriptor->GetDataPropertyIndex<false>();

if (dataSlot != NoSlots)

{

SetSlotUnchecked(instance, dataSlot, undefined);

}

else

{

Assert(descriptor->IsAccessor);

SetSlotUnchecked(instance, descriptor->GetGetterPropertyIndex(), undefined);

SetSlotUnchecked(instance, descriptor->GetSetterPropertyIndex(), undefined);

}

if (this->GetFlags() & IsPrototypeFlag)

{

scriptContext->InvalidateProtoCaches(propertyId);

}

if ((descriptor->Attributes & PropertyLetConstGlobal) == 0)

{

Assert(!descriptor->IsShadowed);

descriptor->Attributes = PropertyDeletedDefaults;

}

else

{

descriptor->Attributes &= ~PropertyDynamicTypeDefaults;

descriptor->Attributes |= PropertyDeletedDefaults;

}

InvalidateFixedField(instance, propertyId, descriptor);

// Change the type so as we can invalidate the cache in fast path jit

instance->ChangeType();

SetPropertyUpdateSideEffect(instance, propertyId, nullptr, SideEffects\_Any);

return true;

}

Assert(descriptor->Attributes & PropertyLetConstGlobal);

return false;

}

// Check numeric propertyRecord only if objectArray available

if (instance->HasObjectArray() && propertyRecord->IsNumeric())

{

return DictionaryTypeHandlerBase<T>::DeleteItem(instance, propertyRecord->GetNumericValue(), propertyOperationFlags);

}

return true;

}

template <typename T>

BOOL DictionaryTypeHandlerBase<T>::IsFixedProperty(const DynamicObject\* instance, PropertyId propertyId)

{

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

DictionaryPropertyDescriptor<T> descriptor;

Assert(propertyId != Constants::NoProperty);

PropertyRecord const\* propertyRecord = scriptContext->GetPropertyName(propertyId);

if (propertyMap->TryGetValue(propertyRecord, &descriptor))

{

return descriptor.IsFixed;

}

else

{

AssertMsg(false, "Asking about a property this type handler doesn't know about?");

return false;

}

}

template <typename T>

BOOL DictionaryTypeHandlerBase<T>::SetItem(DynamicObject\* instance, uint32 index, Var value, PropertyOperationFlags flags)

{

if (!(this->GetFlags() & IsExtensibleFlag) && !instance->HasObjectArray())

{

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

JavascriptError::ThrowCantExtendIfStrictMode(flags, scriptContext);

return false;

}

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

}

template <typename T>

BOOL DictionaryTypeHandlerBase<T>::SetItemWithAttributes(DynamicObject\* instance, uint32 index, Var value, PropertyAttributes attributes)

{

return instance->SetObjectArrayItemWithAttributes(index, value, attributes);

}

template <typename T>

BOOL DictionaryTypeHandlerBase<T>::SetItemAttributes(DynamicObject\* instance, uint32 index, PropertyAttributes attributes)

{

return instance->SetObjectArrayItemAttributes(index, attributes);

}

template <typename T>

BOOL DictionaryTypeHandlerBase<T>::SetItemAccessors(DynamicObject\* instance, uint32 index, Var getter, Var setter)

{

return instance->SetObjectArrayItemAccessors(index, getter, setter);

}

template <typename T>

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

{

if (instance->HasObjectArray())

{

return instance->GetObjectArrayItemSetter(index, setterValue, requestContext);

}

return \_\_super::GetItemSetter(instance, index, setterValue, requestContext);

}

template <typename T>

BOOL DictionaryTypeHandlerBase<T>::IsEnumerable(DynamicObject\* instance, PropertyId propertyId)

{

DictionaryPropertyDescriptor<T>\* descriptor;

Assert(propertyId != Constants::NoProperty);

PropertyRecord const\* propertyRecord = instance->GetScriptContext()->GetPropertyName(propertyId);

if (propertyMap->TryGetReference(propertyRecord, &descriptor))

{

if (!descriptor->HasNonLetConstGlobal())

{

AssertMsg(RootObjectBase::Is(instance), "object must be a global object if letconstglobal is set");

return true;

}

return descriptor->Attributes & PropertyEnumerable;

}

// Check numeric propertyRecord only if objectArray available

if (propertyRecord->IsNumeric())

{

ArrayObject \* objectArray = instance->GetObjectArray();

if (objectArray != nullptr)

{

return objectArray->IsEnumerable(propertyId);

}

}

return true;

}

template <typename T>

BOOL DictionaryTypeHandlerBase<T>::IsWritable(DynamicObject\* instance, PropertyId propertyId)

{

DictionaryPropertyDescriptor<T>\* descriptor;

Assert(propertyId != Constants::NoProperty);

PropertyRecord const\* propertyRecord = instance->GetScriptContext()->GetPropertyName(propertyId);

if (propertyMap->TryGetReference(propertyRecord, &descriptor))

{

if (!descriptor->HasNonLetConstGlobal())

{

AssertMsg(RootObjectBase::Is(instance), "object must be a global object if letconstglobal is set");

return !(descriptor->Attributes & PropertyConst);

}

return descriptor->Attributes & PropertyWritable;

}

// Check numeric propertyRecord only if objectArray available

if (propertyRecord->IsNumeric())

{

ArrayObject \* objectArray = instance->GetObjectArray();

if (objectArray != nullptr)

{

return objectArray->IsWritable(propertyId);

}

}

return true;

}

template <typename T>

BOOL DictionaryTypeHandlerBase<T>::IsConfigurable(DynamicObject\* instance, PropertyId propertyId)

{

DictionaryPropertyDescriptor<T>\* descriptor;

Assert(propertyId != Constants::NoProperty);

PropertyRecord const\* propertyRecord = instance->GetScriptContext()->GetPropertyName(propertyId);

if (propertyMap->TryGetReference(propertyRecord, &descriptor))

{

if (!descriptor->HasNonLetConstGlobal())

{

AssertMsg(RootObjectBase::Is(instance), "object must be a global object if letconstglobal is set");

return true;

}

return descriptor->Attributes & PropertyConfigurable;

}

// Check numeric propertyRecord only if objectArray available

if (propertyRecord->IsNumeric())

{

ArrayObject \* objectArray = instance->GetObjectArray();

if (objectArray != nullptr)

{

return objectArray->IsConfigurable(propertyId);

}

}

return true;

}

template <typename T>

BOOL DictionaryTypeHandlerBase<T>::SetEnumerable(DynamicObject\* instance, PropertyId propertyId, BOOL value)

{

DictionaryPropertyDescriptor<T>\* descriptor;

Assert(propertyId != Constants::NoProperty);

PropertyRecord const\* propertyRecord = instance->GetScriptContext()->GetPropertyName(propertyId);

if (propertyMap->TryGetReference(propertyRecord, &descriptor))

{

if (descriptor->Attributes & PropertyDeleted)

{

return false;

}

if (!descriptor->HasNonLetConstGlobal())

{

AssertMsg(RootObjectBase::Is(instance), "object must be a global object if letconstglobal is set");

return false;

}

if (value)

{

descriptor->Attributes |= PropertyEnumerable;

instance->SetHasNoEnumerableProperties(false);

}

else

{

descriptor->Attributes &= (~PropertyEnumerable);

}

return true;

}

// Check numeric propertyRecord only if objectArray available

if (propertyRecord->IsNumeric())

{

ArrayObject \* objectArray = instance->GetObjectArray();

if (objectArray != nullptr)

{

return objectArray->SetEnumerable(propertyId, value);

}

}

return false;

}

template <typename T>

BOOL DictionaryTypeHandlerBase<T>::SetWritable(DynamicObject\* instance, PropertyId propertyId, BOOL value)

{

DictionaryPropertyDescriptor<T>\* descriptor;

Assert(propertyId != Constants::NoProperty);

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

PropertyRecord const\* propertyRecord = scriptContext->GetPropertyName(propertyId);

if (propertyMap->TryGetReference(propertyRecord, &descriptor))

{

if (descriptor->Attributes & PropertyDeleted)

{

return false;

}

if (!descriptor->HasNonLetConstGlobal())

{

AssertMsg(RootObjectBase::Is(instance), "object must be a global object if letconstglobal is set");

return false;

}

if (value)

{

descriptor->Attributes |= PropertyWritable;

}

else

{

descriptor->Attributes &= (~PropertyWritable);

this->ClearHasOnlyWritableDataProperties();

if(GetFlags() & IsPrototypeFlag)

{

scriptContext->InvalidateStoreFieldCaches(propertyId);

instance->GetLibrary()->NoPrototypeChainsAreEnsuredToHaveOnlyWritableDataProperties();

}

}

instance->ChangeType();

return true;

}

// Check numeric propertyRecord only if objectArray available

if (instance->HasObjectArray() && propertyRecord->IsNumeric())

{

return instance->SetObjectArrayItemWritable(propertyId, value);

}

return false;

}

template <typename T>

BOOL DictionaryTypeHandlerBase<T>::SetConfigurable(DynamicObject\* instance, PropertyId propertyId, BOOL value)

{

DictionaryPropertyDescriptor<T>\* descriptor;

Assert(propertyId != Constants::NoProperty);

PropertyRecord const\* propertyRecord = instance->GetScriptContext()->GetPropertyName(propertyId);

if (propertyMap->TryGetReference(propertyRecord, &descriptor))

{

if (descriptor->Attributes & PropertyDeleted)

{

return false;

}

if (!descriptor->HasNonLetConstGlobal())

{

AssertMsg(RootObjectBase::Is(instance), "object must be a global object if letconstglobal is set");

return false;

}

if (value)

{

descriptor->Attributes |= PropertyConfigurable;

}

else

{

descriptor->Attributes &= (~PropertyConfigurable);

}

return true;

}

// Check numeric propertyRecord only if objectArray available

if (propertyRecord->IsNumeric())

{

ArrayObject \* objectArray = instance->GetObjectArray();

if (objectArray != nullptr)

{

return objectArray->SetConfigurable(propertyId, value);

}

}

return false;

}

template <typename T>

BOOL DictionaryTypeHandlerBase<T>::PreventExtensions(DynamicObject\* instance)

{

this->ClearFlags(IsExtensibleFlag);

ArrayObject \* objectArray = instance->GetObjectArray();

if (objectArray)

{

objectArray->PreventExtensions();

}

return true;

}

template <typename T>

BOOL DictionaryTypeHandlerBase<T>::Seal(DynamicObject\* instance)

{

this->ClearFlags(IsExtensibleFlag);

// Set [[Configurable]] flag of each property to false

DictionaryPropertyDescriptor<T> \*descriptor = nullptr;

for (T index = 0; index < propertyMap->Count(); index++)

{

descriptor = propertyMap->GetReferenceAt(index);

if (descriptor->HasNonLetConstGlobal())

{

descriptor->Attributes &= (~PropertyConfigurable);

}

}

ArrayObject \* objectArray = instance->GetObjectArray();

if (objectArray)

{

objectArray->Seal();

}

return true;

}

template <typename T>

BOOL DictionaryTypeHandlerBase<T>::FreezeImpl(DynamicObject\* instance, bool isConvertedType)

{

this->ClearFlags(IsExtensibleFlag);

// Set [[Writable]] flag of each property to false except for setter\getters

// Set [[Configurable]] flag of each property to false

DictionaryPropertyDescriptor<T> \*descriptor = nullptr;

for (T index = 0; index < propertyMap->Count(); index++)

{

descriptor = propertyMap->GetReferenceAt(index);

if (descriptor->HasNonLetConstGlobal())

{

if (descriptor->GetDataPropertyIndex<false>() != NoSlots)

{

// Only data descriptor has Writable property

descriptor->Attributes &= ~(PropertyWritable | PropertyConfigurable);

}

else

{

descriptor->Attributes &= ~(PropertyConfigurable);

}

}

#if DBG

else

{

AssertMsg(RootObjectBase::Is(instance), "instance needs to be global object when letconst global is set");

}

#endif

}

if (!isConvertedType)

{

// Change of [[Writable]] property requires cache invalidation, hence ChangeType

instance->ChangeType();

}

ArrayObject \* objectArray = instance->GetObjectArray();

if (objectArray)

{

objectArray->Freeze();

}

this->ClearHasOnlyWritableDataProperties();

if(GetFlags() & IsPrototypeFlag)

{

InvalidateStoreFieldCachesForAllProperties(instance->GetScriptContext());

instance->GetLibrary()->NoPrototypeChainsAreEnsuredToHaveOnlyWritableDataProperties();

}

return true;

}

template <typename T>

BOOL DictionaryTypeHandlerBase<T>::IsSealed(DynamicObject\* instance)

{

if (this->GetFlags() & IsExtensibleFlag)

{

return false;

}

DictionaryPropertyDescriptor<T> \*descriptor = nullptr;

for (T index = 0; index < propertyMap->Count(); index++)

{

descriptor = propertyMap->GetReferenceAt(index);

if ((!(descriptor->Attributes & PropertyDeleted) && !(descriptor->Attributes & PropertyLetConstGlobal)))

{

if (descriptor->Attributes & PropertyConfigurable)

{

// [[Configurable]] must be false for all (existing) properties.

// IE9 compatibility: keep IE9 behavior (also check deleted properties)

return false;

}

}

}

ArrayObject \* objectArray = instance->GetObjectArray();

if (objectArray && !objectArray->IsSealed())

{

return false;

}

return true;

}

template <typename T>

BOOL DictionaryTypeHandlerBase<T>::IsFrozen(DynamicObject\* instance)

{

if (this->GetFlags() & IsExtensibleFlag)

{

return false;

}

DictionaryPropertyDescriptor<T> \*descriptor = nullptr;

for (T index = 0; index < propertyMap->Count(); index++)

{

descriptor = propertyMap->GetReferenceAt(index);

if ((!(descriptor->Attributes & PropertyDeleted) && !(descriptor->Attributes & PropertyLetConstGlobal)))

{

if (descriptor->Attributes & PropertyConfigurable)

{

return false;

}

if (descriptor->GetDataPropertyIndex<false>() != NoSlots && (descriptor->Attributes & PropertyWritable))

{

// Only data descriptor has [[Writable]] property

return false;

}

}

}

// Use IsObjectArrayFrozen() to skip "length" [[Writable]] check

ArrayObject \* objectArray = instance->GetObjectArray();

if (objectArray && !objectArray->IsObjectArrayFrozen())

{

return false;

}

return true;

}

template <typename T>

BOOL DictionaryTypeHandlerBase<T>::GetAccessors(DynamicObject\* instance, PropertyId propertyId, Var\* getter, Var\* setter)

{

DictionaryPropertyDescriptor<T>\* descriptor;

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

AssertMsg(nullptr != getter && nullptr != setter, "Getter/Setter must be a valid pointer" );

Assert(propertyId != Constants::NoProperty);

PropertyRecord const\* propertyRecord = scriptContext->GetPropertyName(propertyId);

if (propertyMap->TryGetReference(propertyRecord, &descriptor))

{

if (descriptor->Attributes & PropertyDeleted)

{

return false;

}

if (descriptor->GetDataPropertyIndex<false>() == NoSlots)

{

bool getset = false;

if (descriptor->GetGetterPropertyIndex() != NoSlots)

{

\*getter = instance->GetSlot(descriptor->GetGetterPropertyIndex());

getset = true;

}

if (descriptor->GetSetterPropertyIndex() != NoSlots)

{

\*setter = instance->GetSlot(descriptor->GetSetterPropertyIndex());

getset = true;

}

return getset;

}

}

// Check numeric propertyRecord only if objectArray available

if (propertyRecord->IsNumeric())

{

ArrayObject \* objectArray = instance->GetObjectArray();

if (objectArray != nullptr)

{

return objectArray->GetAccessors(propertyId, getter, setter, scriptContext);

}

}

return false;

}

template <typename T>

BOOL DictionaryTypeHandlerBase<T>::SetAccessors(DynamicObject\* instance, PropertyId propertyId, Var getter, Var setter, PropertyOperationFlags flags)

{

Assert(instance);

JavascriptLibrary\* library = instance->GetLibrary();

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

Assert(this->VerifyIsExtensible(scriptContext, false) || this->HasProperty(instance, propertyId)

|| JavascriptFunction::IsBuiltinProperty(instance, propertyId));

DictionaryPropertyDescriptor<T>\* descriptor;

if (this->GetFlags() & IsPrototypeFlag)

{

scriptContext->InvalidateProtoCaches(propertyId);

}

bool isGetterSet = true;

bool isSetterSet = true;

if (!getter || getter == library->GetDefaultAccessorFunction())

{

isGetterSet = false;

}

if (!setter || setter == library->GetDefaultAccessorFunction())

{

isSetterSet = false;

}

Assert(propertyId != Constants::NoProperty);

PropertyRecord const\* propertyRecord = instance->GetScriptContext()->GetPropertyName(propertyId);

if (propertyMap->TryGetReference(propertyRecord, &descriptor))

{

Assert(descriptor->SanityCheckFixedBits());

if (descriptor->Attributes & PropertyDeleted)

{

if (descriptor->Attributes & PropertyLetConstGlobal)

{

descriptor->Attributes = PropertyDynamicTypeDefaults | (descriptor->Attributes & (PropertyLetConstGlobal | PropertyNoRedecl));

}

else

{

descriptor->Attributes = PropertyDynamicTypeDefaults;

}

}

if (!descriptor->IsAccessor)

{

// New getter/setter, make sure both values are not null and set to the slots

getter = CanonicalizeAccessor(getter, library);

setter = CanonicalizeAccessor(setter, library);

}

// conversion from data-property to accessor property

if (descriptor->ConvertToGetterSetter(nextPropertyIndex))

{

if (this->GetSlotCapacity() <= nextPropertyIndex)

{

if (this->GetSlotCapacity() >= MaxPropertyIndexSize)

{

Throw::OutOfMemory();

}

this->EnsureSlotCapacity(instance);

}

}

// DictionaryTypeHandlers are not supposed to be shared.

Assert(!GetIsOrMayBecomeShared());

DynamicObject\* localSingletonInstance = this->singletonInstance != nullptr ? this->singletonInstance->Get() : nullptr;

Assert(this->singletonInstance == nullptr || localSingletonInstance == instance);

// Although we don't actually have CreateTypeForNewScObject on DictionaryTypeHandler, we could potentially

// transition to a DictionaryTypeHandler with some properties uninitialized.

if (!descriptor->IsInitialized)

{

descriptor->IsInitialized = true;

if (localSingletonInstance == instance && !IsInternalPropertyId(propertyId))

{

// We don't want fixed properties on external objects. See DynamicObject::ResetObject for more information.

Assert(!instance->IsExternal() || (flags & PropertyOperation\_NonFixedValue) != 0);

descriptor->IsFixed = (flags & PropertyOperation\_NonFixedValue) == 0 && ShouldFixAccessorProperties();

}

if (!isGetterSet || !isSetterSet)

{

descriptor->IsOnlyOneAccessorInitialized = true;

}

}

else if (descriptor->IsOnlyOneAccessorInitialized)

{

// Only one of getter/setter was initialized, allow the isFixed to stay if we are defining the other one.

Var oldGetter = GetSlot(instance, descriptor->GetGetterPropertyIndex());

Var oldSetter = GetSlot(instance, descriptor->GetSetterPropertyIndex());

if (((getter == oldGetter || !isGetterSet) && oldSetter == library->GetDefaultAccessorFunction()) ||

((setter == oldSetter || !isSetterSet) && oldGetter == library->GetDefaultAccessorFunction()))

{

descriptor->IsOnlyOneAccessorInitialized = false;

}

else

{

InvalidateFixedField(instance, propertyId, descriptor);

}

}

else

{

InvalidateFixedField(instance, propertyId, descriptor);

}

// don't overwrite an existing accessor with null

if (getter != nullptr)

{

getter = CanonicalizeAccessor(getter, library);

SetSlotUnchecked(instance, descriptor->GetGetterPropertyIndex(), getter);

}

if (setter != nullptr)

{

setter = CanonicalizeAccessor(setter, library);

SetSlotUnchecked(instance, descriptor->GetSetterPropertyIndex(), setter);

}

instance->ChangeType();

this->ClearHasOnlyWritableDataProperties();

if(GetFlags() & IsPrototypeFlag)

{

scriptContext->InvalidateStoreFieldCaches(propertyId);

library->NoPrototypeChainsAreEnsuredToHaveOnlyWritableDataProperties();

}

SetPropertyUpdateSideEffect(instance, propertyId, nullptr, SideEffects\_Any);

// Let's make sure we always have a getter and a setter

Assert(instance->GetSlot(descriptor->GetGetterPropertyIndex()) != nullptr && instance->GetSlot(descriptor->GetSetterPropertyIndex()) != nullptr);

return true;

}

// Always check numeric propertyRecord. This may create objectArray.

if (propertyRecord->IsNumeric())

{

// Calls this or subclass implementation

return SetItemAccessors(instance, propertyRecord->GetNumericValue(), getter, setter);

}

getter = CanonicalizeAccessor(getter, library);

setter = CanonicalizeAccessor(setter, library);

T getterIndex = nextPropertyIndex++;

T setterIndex = nextPropertyIndex++;

DictionaryPropertyDescriptor<T> newDescriptor(getterIndex, setterIndex);

if (this->GetSlotCapacity() <= nextPropertyIndex)

{

if (this->GetSlotCapacity() >= MaxPropertyIndexSize)

{

Throw::OutOfMemory();

}

this->EnsureSlotCapacity(instance);

}

// DictionaryTypeHandlers are not supposed to be shared.

Assert(!GetIsOrMayBecomeShared());

DynamicObject\* localSingletonInstance = this->singletonInstance != nullptr ? this->singletonInstance->Get() : nullptr;

Assert(this->singletonInstance == nullptr || localSingletonInstance == instance);

newDescriptor.IsInitialized = true;

if (localSingletonInstance == instance && !IsInternalPropertyId(propertyId))

{

// We don't want fixed properties on external objects. See DynamicObject::ResetObject for more information.

Assert(!instance->IsExternal() || (flags & PropertyOperation\_NonFixedValue) != 0);

// Even if one (or both?) accessors are the default functions obtained through cannonicalization,

// they are still legitimate functions, so it's ok to mark the whole property as fixed.

newDescriptor.IsFixed = (flags & PropertyOperation\_NonFixedValue) == 0 && ShouldFixAccessorProperties();

if (!isGetterSet || !isSetterSet)

{

newDescriptor.IsOnlyOneAccessorInitialized = true;

}

}

propertyMap->Add(propertyRecord, newDescriptor);

SetSlotUnchecked(instance, newDescriptor.GetGetterPropertyIndex(), getter);

SetSlotUnchecked(instance, newDescriptor.GetSetterPropertyIndex(), setter);

this->ClearHasOnlyWritableDataProperties();

if(GetFlags() & IsPrototypeFlag)

{

scriptContext->InvalidateStoreFieldCaches(propertyId);

library->NoPrototypeChainsAreEnsuredToHaveOnlyWritableDataProperties();

}

SetPropertyUpdateSideEffect(instance, propertyId, nullptr, SideEffects\_Any);

// Let's make sure we always have a getter and a setter

Assert(instance->GetSlot(newDescriptor.GetGetterPropertyIndex()) != nullptr && instance->GetSlot(newDescriptor.GetSetterPropertyIndex()) != nullptr);

return true;

}

// If this type is not extensible and the property being set does not already exist,

// if throwIfNotExtensible is

// \* true, a type error will be thrown

// \* false, FALSE will be returned (unless strict mode is enabled, in which case a type error will be thrown).

// Either way, the property will not be set.

//

// This is used to ensure that we throw in the following scenario, in accordance with

// section 10.2.1.2.2 of the Errata to the ES5 spec:

// Object.preventExtension(this); // make the global object non-extensible

// var x = 4;

//

// throwIfNotExtensible should always be false for non-numeric properties.

template <typename T>

BOOL DictionaryTypeHandlerBase<T>::SetPropertyWithAttributes(DynamicObject\* instance, PropertyId propertyId, Var value,

PropertyAttributes attributes, PropertyValueInfo\* info, PropertyOperationFlags flags, SideEffects possibleSideEffects)

{

DictionaryPropertyDescriptor<T>\* descriptor;

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

bool isForce = (flags & PropertyOperation\_Force) != 0;

bool throwIfNotExtensible = (flags & PropertyOperation\_ThrowIfNotExtensible) != 0;

#ifdef DEBUG

uint32 debugIndex;

Assert(!(throwIfNotExtensible && scriptContext->IsNumericPropertyId(propertyId, &debugIndex)));

#endif

Assert(propertyId != Constants::NoProperty);

PropertyRecord const\* propertyRecord = instance->GetScriptContext()->GetPropertyName(propertyId);

if (propertyMap->TryGetReference(propertyRecord, &descriptor))

{

Assert(descriptor->SanityCheckFixedBits());

if (attributes & descriptor->Attributes & PropertyLetConstGlobal)

{

// Do not need to change the descriptor or its attributes if setting the initial value of a LetConstGlobal

}

else if (descriptor->Attributes & PropertyDeleted && !(attributes & PropertyLetConstGlobal))

{

if (!isForce)

{

if (!this->VerifyIsExtensible(scriptContext, throwIfNotExtensible))

{

return FALSE;

}

}

scriptContext->InvalidateProtoCaches(propertyId);

if (descriptor->Attributes & PropertyLetConstGlobal)

{

descriptor->Attributes = attributes | (descriptor->Attributes & (PropertyLetConstGlobal | PropertyNoRedecl));

}

else

{

descriptor->Attributes = attributes;

}

descriptor->ConvertToData();

}

else if (descriptor->IsShadowed)

{

descriptor->Attributes = attributes | (descriptor->Attributes & (PropertyLetConstGlobal | PropertyNoRedecl));

}

else if ((descriptor->Attributes & PropertyLetConstGlobal) != (attributes & PropertyLetConstGlobal))

{

bool addingLetConstGlobal = (attributes & PropertyLetConstGlobal) != 0;

descriptor->AddShadowedData(nextPropertyIndex, addingLetConstGlobal);

if (addingLetConstGlobal)

{

descriptor->Attributes = descriptor->Attributes | (attributes & PropertyNoRedecl) | PropertyLetConstGlobal;

}

else

{

descriptor->Attributes = attributes | (descriptor->Attributes & PropertyNoRedecl) | PropertyLetConstGlobal;

}

if (this->GetSlotCapacity() <= nextPropertyIndex)

{

if (this->GetSlotCapacity() >= MaxPropertyIndexSize)

{

Throw::OutOfMemory();

}

this->EnsureSlotCapacity(instance);

}

if (addingLetConstGlobal)

{

// If shadowing a global property with a let/const, need to invalidate

// JIT fast path cache since look up could now go to the let/const instead

// of the global property.

//

// Do not need to invalidate when adding a global property that gets shadowed

// by an existing let/const, since all caches will still be correct.

instance->ChangeType();

}

}

else

{

if (descriptor->IsAccessor && !(attributes & PropertyLetConstGlobal))

{

AssertMsg(RootObjectBase::Is(instance) || JavascriptFunction::IsBuiltinProperty(instance, propertyId) ||

// ValidateAndApplyPropertyDescriptor says to preserve Configurable and Enumerable flags

// For InitRootFld, which is equivalent to

// CreateGlobalFunctionBinding called from GlobalDeclarationInstantiation in the spec,

// we can assume that the attributes specified include enumerable and writable. Thus

// we don't need to preserve the original values of these two attributes and therefore

// do not need to change InitRootFld from being a SetPropertyWithAttributes API call to

// something else. All we need to do is convert the descriptor to a data descriptor.

// Built-in Function.prototype properties 'length', 'arguments', and 'caller' are special cases.

(JavascriptOperators::IsClassConstructor(JavascriptOperators::GetProperty(instance, PropertyIds::constructor, scriptContext)) &&

(attributes & PropertyClassMemberDefaults) == PropertyClassMemberDefaults),

// 14.3.9: InitClassMember sets property descriptor to {writable:true, enumerable:false, configurable:true}

"Expect to only come down this path for InitClassMember or InitRootFld (on the global object) overwriting existing accessor property");

if (!(descriptor->Attributes & PropertyConfigurable))

{

if (scriptContext && scriptContext->GetThreadContext()->RecordImplicitException())

{

JavascriptError::ThrowTypeError(scriptContext, JSERR\_DefineProperty\_NotConfigurable, scriptContext->GetThreadContext()->GetPropertyName(propertyId)->GetBuffer());

}

return FALSE;

}

descriptor->ConvertToData();

instance->ChangeType();

}

// Make sure to keep the PropertyLetConstGlobal bit as is while taking the new attributes.

descriptor->Attributes = attributes | (descriptor->Attributes & PropertyLetConstGlobal);

}

if (attributes & PropertyLetConstGlobal)

{

SetPropertyWithDescriptor<true>(instance, propertyId, descriptor, value, flags, info);

}

else

{

SetPropertyWithDescriptor<false>(instance, propertyId, descriptor, value, flags, info);

}

if (descriptor->Attributes & PropertyEnumerable)

{

instance->SetHasNoEnumerableProperties(false);

}

if (!(descriptor->Attributes & PropertyWritable))

{

this->ClearHasOnlyWritableDataProperties();

if(GetFlags() & IsPrototypeFlag)

{

scriptContext->InvalidateStoreFieldCaches(propertyId);

instance->GetLibrary()->NoPrototypeChainsAreEnsuredToHaveOnlyWritableDataProperties();

}

}

SetPropertyUpdateSideEffect(instance, propertyId, value, possibleSideEffects);

return true;

}

// Always check numeric propertyRecord. This may create objectArray.

if (propertyRecord->IsNumeric())

{

// Calls this or subclass implementation

return SetItemWithAttributes(instance, propertyRecord->GetNumericValue(), value, attributes);

}

return this->AddProperty(instance, propertyRecord, value, attributes, info, flags, throwIfNotExtensible, possibleSideEffects);

}

template <typename T>

void DictionaryTypeHandlerBase<T>::EnsureSlotCapacity(DynamicObject \* instance)

{

Assert(this->GetSlotCapacity() < MaxPropertyIndexSize); // Otherwise we can't grow this handler's capacity. We should've evolved to Bigger handler or OOM.

// A Dictionary type is expected to have more properties

// grow exponentially rather linearly to avoid the realloc and moves,

// however use a small exponent to avoid waste

int newSlotCapacity = (nextPropertyIndex + 1);

newSlotCapacity += (newSlotCapacity>>2);

if (newSlotCapacity > MaxPropertyIndexSize)

{

newSlotCapacity = MaxPropertyIndexSize;

}

newSlotCapacity = RoundUpSlotCapacity(newSlotCapacity, GetInlineSlotCapacity());

Assert(newSlotCapacity <= MaxPropertyIndexSize);

instance->EnsureSlots(this->GetSlotCapacity(), newSlotCapacity, instance->GetScriptContext(), this);

this->SetSlotCapacity(newSlotCapacity);

}

template <typename T>

BOOL DictionaryTypeHandlerBase<T>::SetAttributes(DynamicObject\* instance, PropertyId propertyId, PropertyAttributes attributes)

{

DictionaryPropertyDescriptor<T>\* descriptor;

Assert(propertyId != Constants::NoProperty);

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

PropertyRecord const\* propertyRecord = scriptContext->GetPropertyName(propertyId);

if (propertyMap->TryGetReference(propertyRecord, &descriptor))

{

if (descriptor->Attributes & PropertyDeleted)

{

return false;

}

descriptor->Attributes = (descriptor->Attributes & ~PropertyDynamicTypeDefaults) | (attributes & PropertyDynamicTypeDefaults);

if (descriptor->Attributes & PropertyEnumerable)

{

instance->SetHasNoEnumerableProperties(false);

}

if (!(descriptor->Attributes & PropertyWritable))

{

this->ClearHasOnlyWritableDataProperties();

if(GetFlags() & IsPrototypeFlag)

{

scriptContext->InvalidateStoreFieldCaches(propertyId);

instance->GetLibrary()->NoPrototypeChainsAreEnsuredToHaveOnlyWritableDataProperties();

}

}

return true;

}

// Check numeric propertyId only if objectArray available

if (instance->HasObjectArray() && propertyRecord->IsNumeric())

{

return DictionaryTypeHandlerBase<T>::SetItemAttributes(instance, propertyRecord->GetNumericValue(), attributes);

}

return false;

}

template <typename T>

BOOL DictionaryTypeHandlerBase<T>::GetAttributesWithPropertyIndex(DynamicObject \* instance, PropertyId propertyId, BigPropertyIndex index, PropertyAttributes \* attributes)

{

// this might get value that are deleted from the dictionary, but that should be nulled out

DictionaryPropertyDescriptor<T> \* descriptor;

// We can't look it up using the slot index, as one propertyId might have multiple slots, do the propertyId map lookup

PropertyRecord const\* propertyRecord = instance->GetScriptContext()->GetPropertyName(propertyId);

if (!propertyMap->TryGetReference(propertyRecord, &descriptor))

{

return false;

}

// This function is only used by LdRootFld, so the index will allow let const globals

Assert(descriptor->GetDataPropertyIndex<true>() == index);

if (descriptor->Attributes & PropertyDeleted)

{

return false;

}

\*attributes = descriptor->Attributes & PropertyDynamicTypeDefaults;

return true;

}

template <typename T>

Var DictionaryTypeHandlerBase<T>::CanonicalizeAccessor(Var accessor, /\*const\*/ JavascriptLibrary\* library)

{

if (accessor == nullptr || JavascriptOperators::IsUndefinedObject(accessor, library))

{

accessor = library->GetDefaultAccessorFunction();

}

return accessor;

}

template <typename T>

BigDictionaryTypeHandler\* DictionaryTypeHandlerBase<T>::ConvertToBigDictionaryTypeHandler(DynamicObject\* instance)

{

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

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

BigDictionaryTypeHandler\* newTypeHandler = NewBigDictionaryTypeHandler(recycler, GetSlotCapacity(), GetInlineSlotCapacity(), GetOffsetOfInlineSlots());

// We expect the new type handler to start off marked as having only writable data properties.

Assert(newTypeHandler->GetHasOnlyWritableDataProperties());

#ifdef ENABLE\_DEBUG\_CONFIG\_OPTIONS

DynamicType\* oldType = instance->GetDynamicType();

RecyclerWeakReference<DynamicObject>\* oldSingletonInstance = GetSingletonInstance();

TraceFixedFieldsBeforeTypeHandlerChange(L"DictionaryTypeHandler", L"BigDictionaryTypeHandler", instance, this, oldType, oldSingletonInstance);

#endif

CopySingletonInstance(instance, newTypeHandler);

DictionaryPropertyDescriptor<T> descriptor;

DictionaryPropertyDescriptor<BigPropertyIndex> bigDescriptor;

const PropertyRecord\* propertyId;

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

{

descriptor = propertyMap->GetValueAt(i);

propertyId = propertyMap->GetKeyAt(i);

bigDescriptor.CopyFrom(descriptor);

newTypeHandler->propertyMap->Add(propertyId, bigDescriptor);

}

newTypeHandler->nextPropertyIndex = nextPropertyIndex;

ClearSingletonInstance();

AssertMsg((newTypeHandler->GetFlags() & IsPrototypeFlag) == 0, "Why did we create a brand new type handler with a prototype flag set?");

newTypeHandler->SetFlags(IsPrototypeFlag, this->GetFlags());

newTypeHandler->ChangeFlags(IsExtensibleFlag, this->GetFlags());

// Any new type handler we expect to see here should have inline slot capacity locked. If this were to change, we would need

// to update our shrinking logic (see PathTypeHandlerBase::ShrinkSlotAndInlineSlotCapacity).

Assert(newTypeHandler->GetIsInlineSlotCapacityLocked());

newTypeHandler->SetPropertyTypes(PropertyTypesWritableDataOnly | PropertyTypesWritableDataOnlyDetection, this->GetPropertyTypes());

newTypeHandler->SetInstanceTypeHandler(instance);

// Unlike for SimpleDictionaryTypeHandler or PathTypeHandler, the DictionaryTypeHandler copies usedAsFixed indiscriminately above.

// Therefore, we don't care if we changed the type or not, and don't need the assert below.

// We assumed that we don't need to transfer used as fixed bits unless we are a prototype, which is only valid if we also changed the type.

// Assert(instance->GetType() != oldType);

Assert(!newTypeHandler->HasSingletonInstance() || !instance->HasSharedType());

#ifdef ENABLE\_DEBUG\_CONFIG\_OPTIONS

TraceFixedFieldsAfterTypeHandlerChange(instance, this, newTypeHandler, oldType, oldSingletonInstance);

#endif

return newTypeHandler;

}

template <typename T>

BigDictionaryTypeHandler\* DictionaryTypeHandlerBase<T>::NewBigDictionaryTypeHandler(Recycler\* recycler, int slotCapacity, uint16 inlineSlotCapacity, uint16 offsetOfInlineSlots)

{

return RecyclerNew(recycler, BigDictionaryTypeHandler, recycler, slotCapacity, inlineSlotCapacity, offsetOfInlineSlots);

}

template <>

BigDictionaryTypeHandler\* DictionaryTypeHandlerBase<BigPropertyIndex>::ConvertToBigDictionaryTypeHandler(DynamicObject\* instance)

{

Throw::OutOfMemory();

}

template<>

BOOL DictionaryTypeHandlerBase<PropertyIndex>::IsBigDictionaryTypeHandler()

{

return FALSE;

}

template<>

BOOL DictionaryTypeHandlerBase<BigPropertyIndex>::IsBigDictionaryTypeHandler()

{

return TRUE;

}

template <typename T>

BOOL DictionaryTypeHandlerBase<T>::AddProperty(DynamicObject\* instance, const PropertyRecord\* propertyRecord, Var value,

PropertyAttributes attributes, PropertyValueInfo\* info, PropertyOperationFlags flags, bool throwIfNotExtensible, SideEffects possibleSideEffects)

{

AnalysisAssert(instance);

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

bool isForce = (flags & PropertyOperation\_Force) != 0;

#if DBG

DictionaryPropertyDescriptor<T>\* descriptor;

Assert(!propertyMap->TryGetReference(propertyRecord, &descriptor));

Assert(!propertyRecord->IsNumeric());

#endif

if (!isForce)

{

if (!this->VerifyIsExtensible(scriptContext, throwIfNotExtensible))

{

return FALSE;

}

}

if (this->GetSlotCapacity() <= nextPropertyIndex)

{

if (this->GetSlotCapacity() >= MaxPropertyIndexSize ||

(this->GetSlotCapacity() >= CONFIG\_FLAG(BigDictionaryTypeHandlerThreshold) && !this->IsBigDictionaryTypeHandler()))

{

BigDictionaryTypeHandler\* newTypeHandler = ConvertToBigDictionaryTypeHandler(instance);

return newTypeHandler->AddProperty(instance, propertyRecord, value, attributes, info, flags, false, possibleSideEffects);

}

this->EnsureSlotCapacity(instance);

}

T index = nextPropertyIndex++;

DictionaryPropertyDescriptor<T> newDescriptor(index, attributes);

// DictionaryTypeHandlers are not supposed to be shared.

Assert(!GetIsOrMayBecomeShared());

DynamicObject\* localSingletonInstance = this->singletonInstance != nullptr ? this->singletonInstance->Get() : nullptr;

Assert(this->singletonInstance == nullptr || localSingletonInstance == instance);

if ((flags & PropertyOperation\_PreInit) == 0)

{

newDescriptor.IsInitialized = true;

if (localSingletonInstance == instance && !IsInternalPropertyId(propertyRecord->GetPropertyId()) &&

(flags & (PropertyOperation\_NonFixedValue | PropertyOperation\_SpecialValue)) == 0)

{

Assert(value != nullptr);

// We don't want fixed properties on external objects. See DynamicObject::ResetObject for more information.

Assert(!instance->IsExternal());

newDescriptor.IsFixed = (JavascriptFunction::Is(value) ? ShouldFixMethodProperties() : (ShouldFixDataProperties() & CheckHeuristicsForFixedDataProps(instance, propertyRecord, value)));

}

}

propertyMap->Add(propertyRecord, newDescriptor);

if (attributes & PropertyEnumerable)

{

instance->SetHasNoEnumerableProperties(false);

}

if (!(attributes & PropertyWritable))

{

this->ClearHasOnlyWritableDataProperties();

if(GetFlags() & IsPrototypeFlag)

{

instance->GetScriptContext()->InvalidateStoreFieldCaches(propertyRecord->GetPropertyId());

instance->GetLibrary()->NoPrototypeChainsAreEnsuredToHaveOnlyWritableDataProperties();

}

}

SetSlotUnchecked(instance, index, value);

// If we just added a fixed method, don't populate the inline cache so that we always take the

// slow path when overwriting this property and correctly invalidate any JIT-ed code that hard-coded

// this method.

if (newDescriptor.IsFixed)

{

PropertyValueInfo::SetNoCache(info, instance);

}

else

{

SetPropertyValueInfo(info, instance, index, attributes);

}

if (!IsInternalPropertyId(propertyRecord->GetPropertyId()) && ((this->GetFlags() & IsPrototypeFlag)

|| JavascriptOperators::HasProxyOrPrototypeInlineCacheProperty(instance, propertyRecord->GetPropertyId())))

{

// We don't evolve dictionary types when adding a field, so we need to invalidate prototype caches.

// We only have to do this though if the current type is used as a prototype, or the current property

// is found on the prototype chain.

scriptContext->InvalidateProtoCaches(propertyRecord->GetPropertyId());

}

SetPropertyUpdateSideEffect(instance, propertyRecord->GetPropertyId(), value, possibleSideEffects);

return true;

}

//

// Converts (upgrades) this dictionary type handler to an ES5 array type handler. The new handler takes

// over all members of this handler including the property map.

//

template <typename T>

ES5ArrayTypeHandlerBase<T>\* DictionaryTypeHandlerBase<T>::ConvertToES5ArrayType(DynamicObject \*instance)

{

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

ES5ArrayTypeHandlerBase<T>\* newTypeHandler = RecyclerNew(recycler, ES5ArrayTypeHandlerBase<T>, recycler, this);

// Don't need to transfer the singleton instance, because the new handler takes over this handler.

AssertMsg((newTypeHandler->GetFlags() & IsPrototypeFlag) == 0, "Why did we create a brand new type handler with a prototype flag set?");

newTypeHandler->SetFlags(IsPrototypeFlag, this->GetFlags());

// Property types were copied in the constructor.

//newTypeHandler->SetPropertyTypes(PropertyTypesWritableDataOnly | PropertyTypesWritableDataOnlyDetection | PropertyTypesInlineSlotCapacityLocked, this->GetPropertyTypes());

newTypeHandler->SetInstanceTypeHandler(instance);

return newTypeHandler;

}

template <typename T>

void DictionaryTypeHandlerBase<T>::SetAllPropertiesToUndefined(DynamicObject\* instance, bool invalidateFixedFields)

{

// The Var for window is reused across navigation. we shouldn't preserve the IsExtensibleFlag when we don't keep

// the expandoes. Reset the IsExtensibleFlag in cleanup scenario should be good enough

// to cover all the preventExtension/Freeze/Seal scenarios.

// Note that we don't change the flag for keepProperties scenario: the flags should be preserved and that's consistent

// with other browsers.

ChangeFlags(IsExtensibleFlag | IsSealedOnceFlag | IsFrozenOnceFlag, IsExtensibleFlag);

// Note: This method is currently only called from ResetObject, which in turn only applies to external objects.

// Before using for other purposes, make sure the assumptions made here make sense in the new context. In particular,

// the invalidateFixedFields == false is only correct if a) the object is known not to have any, or b) the type of the

// object has changed and/or property guards have already been invalidated through some other means.

int propertyCount = this->propertyMap->Count();

if (invalidateFixedFields)

{

for (int propertyIndex = 0; propertyIndex < propertyCount; propertyIndex++)

{

const PropertyRecord\* propertyRecord = this->propertyMap->GetKeyAt(propertyIndex);

DictionaryPropertyDescriptor<T>\* descriptor = this->propertyMap->GetReferenceAt(propertyIndex);

InvalidateFixedField(instance, propertyRecord->GetPropertyId(), descriptor);

}

}

Js::RecyclableObject\* undefined = instance->GetLibrary()->GetUndefined();

Js::JavascriptFunction\* defaultAccessor = instance->GetLibrary()->GetDefaultAccessorFunction();

for (int propertyIndex = 0; propertyIndex < propertyCount; propertyIndex++)

{

DictionaryPropertyDescriptor<T>\* descriptor = this->propertyMap->GetReferenceAt(propertyIndex);

T dataPropertyIndex = descriptor->GetDataPropertyIndex<false>();

if (dataPropertyIndex != NoSlots)

{

SetSlotUnchecked(instance, dataPropertyIndex, undefined);

}

else

{

SetSlotUnchecked(instance, descriptor->GetGetterPropertyIndex(), defaultAccessor);

SetSlotUnchecked(instance, descriptor->GetSetterPropertyIndex(), defaultAccessor);

}

}

}

template <typename T>

void DictionaryTypeHandlerBase<T>::MarshalAllPropertiesToScriptContext(DynamicObject\* instance, ScriptContext\* targetScriptContext, bool invalidateFixedFields)

{

// Note: This method is currently only called from ResetObject, which in turn only applies to external objects.

// Before using for other purposes, make sure the assumptions made here make sense in the new context. In particular,

// the invalidateFixedFields == false is only correct if a) the object is known not to have any, or b) the type of the

// object has changed and/or property guards have already been invalidated through some other means.

if (invalidateFixedFields)

{

int propertyCount = this->propertyMap->Count();

for (int propertyIndex = 0; propertyIndex < propertyCount; propertyIndex++)

{

const PropertyRecord\* propertyRecord = this->propertyMap->GetKeyAt(propertyIndex);

DictionaryPropertyDescriptor<T>\* descriptor = this->propertyMap->GetReferenceAt(propertyIndex);

InvalidateFixedField(instance, propertyRecord->GetPropertyId(), descriptor);

}

}

int slotCount = this->nextPropertyIndex;

for (int slotIndex = 0; slotIndex < slotCount; slotIndex++)

{

SetSlotUnchecked(instance, slotIndex, CrossSite::MarshalVar(targetScriptContext, GetSlot(instance, slotIndex)));

}

}

template <typename T>

DynamicTypeHandler\* DictionaryTypeHandlerBase<T>::ConvertToTypeWithItemAttributes(DynamicObject\* instance)

{

return JavascriptArray::Is(instance) ? ConvertToES5ArrayType(instance) : this;

}

template <typename T>

void DictionaryTypeHandlerBase<T>::SetIsPrototype(DynamicObject\* instance)

{

// Don't return if IsPrototypeFlag is set, because we may still need to do a type transition and

// set fixed bits. If this handler were to be shared, this instance may not be a prototype yet.

// We might need to convert to a non-shared type handler and/or change type.

if (!ChangeTypeOnProto() && !(GetIsOrMayBecomeShared() && IsolatePrototypes()))

{

SetFlags(IsPrototypeFlag);

return;

}

Assert(!GetIsShared() || this->singletonInstance == nullptr);

Assert(this->singletonInstance == nullptr || this->singletonInstance->Get() == instance);

// Review (jedmiad): Why isn't this getting inlined?

const auto setFixedFlags = [instance](const PropertyRecord\* propertyRecord, DictionaryPropertyDescriptor<T>\* const descriptor, bool hasNewType)

{

if (IsInternalPropertyId(propertyRecord->GetPropertyId()))

{

return;

}

if (!(descriptor->Attributes & PropertyDeleted))

{

// See PathTypeHandlerBase::ConvertToSimpleDictionaryType for rules governing fixed field bits during type

// handler transitions. In addition, we know that the current instance is not yet a prototype.

Assert(descriptor->SanityCheckFixedBits());

if (descriptor->IsInitialized)

{

// Since DictionaryTypeHandlers are never shared, we can set fixed fields and clear used as fixed as long

// as we have changed the type. Otherwise populated load field caches would still be valid and would need

// to be explicitly invalidated if the property value changes.

if (hasNewType)

{

T dataSlot = descriptor->GetDataPropertyIndex<false>();

if (dataSlot != NoSlots)

{

Var value = instance->GetSlot(dataSlot);

// Because DictionaryTypeHandlers are never shared we should always have a property value if the handler

// says it's initialized.

Assert(value != nullptr);

descriptor->IsFixed = (JavascriptFunction::Is(value) ? ShouldFixMethodProperties() : (ShouldFixDataProperties() && CheckHeuristicsForFixedDataProps(instance, propertyRecord, value)));

}

else if (descriptor->IsAccessor)

{

Assert(descriptor->GetGetterPropertyIndex() != NoSlots && descriptor->GetSetterPropertyIndex() != NoSlots);

descriptor->IsFixed = ShouldFixAccessorProperties();

}

// Since we have a new type we can clear all used as fixed bits. That's because any instance field loads

// will have been invalidated by the type transition, and there are no proto fields loads from this object

// because it is just now becoming a proto.

descriptor->UsedAsFixed = false;

}

}

else

{

Assert(!descriptor->IsFixed && !descriptor->UsedAsFixed);

}

Assert(descriptor->SanityCheckFixedBits());

}

};

// DictionaryTypeHandlers are never shared. If we allow sharing, we will have to handle this case

// just like SimpleDictionaryTypeHandler.

Assert(!GetIsOrMayBecomeShared());

#ifdef ENABLE\_DEBUG\_CONFIG\_OPTIONS

DynamicType\* oldType = instance->GetDynamicType();

RecyclerWeakReference<DynamicObject>\* oldSingletonInstance = GetSingletonInstance();

TraceFixedFieldsBeforeSetIsProto(instance, this, oldType, oldSingletonInstance);

#endif

bool hasNewType = false;

if (ChangeTypeOnProto())

{

// Forcing a type transition allows us to fix all fields (even those that were previously marked as non-fixed).

instance->ChangeType();

Assert(!instance->HasSharedType());

hasNewType = true;

}

// Currently there is no way to become the prototype if you are a stack instance

Assert(!ThreadContext::IsOnStack(instance));

if (AreSingletonInstancesNeeded() && this->singletonInstance == nullptr)

{

this->singletonInstance = instance->CreateWeakReferenceToSelf();

}

// We don't want fixed properties on external objects. See DynamicObject::ResetObject for more information.

if (!instance->IsExternal())

{

// The propertyMap dictionary is guaranteed to have contiguous entries because we never remove entries from it.

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

{

const PropertyRecord\* propertyRecord = propertyMap->GetKeyAt(i);

DictionaryPropertyDescriptor<T>\* const descriptor = propertyMap->GetReferenceAt(i);

setFixedFlags(propertyRecord, descriptor, hasNewType);

}

}

SetFlags(IsPrototypeFlag);

#ifdef ENABLE\_DEBUG\_CONFIG\_OPTIONS

TraceFixedFieldsAfterSetIsProto(instance, this, this, oldType, oldSingletonInstance);

#endif

}

template <typename T>

bool DictionaryTypeHandlerBase<T>::HasSingletonInstance() const

{

return this->singletonInstance != nullptr;

}

template <typename T>

bool DictionaryTypeHandlerBase<T>::TryUseFixedProperty(PropertyRecord const \* propertyRecord, Var \* pProperty, FixedPropertyKind propertyType, ScriptContext \* requestContext)

{

bool result = TryGetFixedProperty<false, true>(propertyRecord, pProperty, propertyType, requestContext);

TraceUseFixedProperty(propertyRecord, pProperty, result, L"DictionaryTypeHandler", requestContext);

return result;

}

template <typename T>

bool DictionaryTypeHandlerBase<T>::TryUseFixedAccessor(PropertyRecord const \* propertyRecord, Var \* pAccessor, FixedPropertyKind propertyType, bool getter, ScriptContext \* requestContext)

{

bool result = TryGetFixedAccessor<false, true>(propertyRecord, pAccessor, propertyType, getter, requestContext);

TraceUseFixedProperty(propertyRecord, pAccessor, result, L"DictionaryTypeHandler", requestContext);

return result;

}

#if DBG

template <typename T>

bool DictionaryTypeHandlerBase<T>::CanStorePropertyValueDirectly(const DynamicObject\* instance, PropertyId propertyId, bool allowLetConst)

{

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

DictionaryPropertyDescriptor<T> descriptor;

// We pass Constants::NoProperty for ActivationObjects for functions with same named formals.

if (propertyId == Constants::NoProperty)

{

return true;

}

PropertyRecord const\* propertyRecord = scriptContext->GetPropertyName(propertyId);

if (propertyMap->TryGetValue(propertyRecord, &descriptor))

{

if (allowLetConst && (descriptor.Attributes & PropertyLetConstGlobal))

{

return true;

}

else

{

return descriptor.IsInitialized && !descriptor.IsFixed;

}

}

else

{

AssertMsg(false, "Asking about a property this type handler doesn't know about?");

return false;

}

}

template <typename T>

bool DictionaryTypeHandlerBase<T>::CheckFixedProperty(PropertyRecord const \* propertyRecord, Var \* pProperty, ScriptContext \* requestContext)

{

return TryGetFixedProperty<true, false>(propertyRecord, pProperty, (Js::FixedPropertyKind)(Js::FixedPropertyKind::FixedMethodProperty | Js::FixedPropertyKind::FixedDataProperty), requestContext);

}

template <typename T>

bool DictionaryTypeHandlerBase<T>::HasAnyFixedProperties() const

{

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

{

DictionaryPropertyDescriptor<T> descriptor = propertyMap->GetValueAt(i);

if (descriptor.IsFixed)

{

return true;

}

}

return false;

}

#endif

template <typename T>

template <bool allowNonExistent, bool markAsUsed>

bool DictionaryTypeHandlerBase<T>::TryGetFixedProperty(PropertyRecord const \* propertyRecord, Var \* pProperty, FixedPropertyKind propertyType, ScriptContext \* requestContext)

{

// Note: This function is not thread-safe and cannot be called from the JIT thread. That's why we collect and

// cache any fixed function instances during work item creation on the main thread.

DynamicObject\* localSingletonInstance = this->singletonInstance != nullptr ? this->singletonInstance->Get() : nullptr;

if (localSingletonInstance != nullptr && localSingletonInstance->GetScriptContext() == requestContext)

{

DictionaryPropertyDescriptor<T>\* descriptor;

if (propertyMap->TryGetReference(propertyRecord, &descriptor))

{

if (descriptor->Attributes & PropertyDeleted || !descriptor->IsFixed)

{

return false;

}

T dataSlot = descriptor->GetDataPropertyIndex<false>();

if (dataSlot != NoSlots)

{

Assert(!IsInternalPropertyId(propertyRecord->GetPropertyId()));

Var value = localSingletonInstance->GetSlot(dataSlot);

if (value && ((IsFixedMethodProperty(propertyType) && JavascriptFunction::Is(value)) || IsFixedDataProperty(propertyType)))

{

\*pProperty = value;

if (markAsUsed)

{

descriptor->UsedAsFixed = true;

}

return true;

}

}

}

else

{

AssertMsg(allowNonExistent, "Trying to get a fixed function instance for a non-existent property?");

}

}

return false;

}

template <typename T>

template <bool allowNonExistent, bool markAsUsed>

bool DictionaryTypeHandlerBase<T>::TryGetFixedAccessor(PropertyRecord const \* propertyRecord, Var \* pAccessor, FixedPropertyKind propertyType, bool getter, ScriptContext \* requestContext)

{

// Note: This function is not thread-safe and cannot be called from the JIT thread. That's why we collect and

// cache any fixed function instances during work item creation on the main thread.

DynamicObject\* localSingletonInstance = this->singletonInstance != nullptr ? this->singletonInstance->Get() : nullptr;

if (localSingletonInstance != nullptr && localSingletonInstance->GetScriptContext() == requestContext)

{

DictionaryPropertyDescriptor<T>\* descriptor;

if (propertyMap->TryGetReference(propertyRecord, &descriptor))

{

if (descriptor->Attributes & PropertyDeleted || !descriptor->IsAccessor || !descriptor->IsFixed)

{

return false;

}

T accessorSlot = getter ? descriptor->GetGetterPropertyIndex() : descriptor->GetSetterPropertyIndex();

if (accessorSlot != NoSlots)

{

Assert(!IsInternalPropertyId(propertyRecord->GetPropertyId()));

Var value = localSingletonInstance->GetSlot(accessorSlot);

if (value && IsFixedAccessorProperty(propertyType) && JavascriptFunction::Is(value))

{

\*pAccessor = value;

if (markAsUsed)

{

descriptor->UsedAsFixed = true;

}

return true;

}

}

}

else

{

AssertMsg(allowNonExistent, "Trying to get a fixed function instance for a non-existent property?");

}

}

return false;

}

template <typename T>

void DictionaryTypeHandlerBase<T>::CopySingletonInstance(DynamicObject\* instance, DynamicTypeHandler\* typeHandler)

{

if (this->singletonInstance != nullptr)

{

Assert(AreSingletonInstancesNeeded());

Assert(this->singletonInstance->Get() == instance);

typeHandler->SetSingletonInstanceUnchecked(this->singletonInstance);

}

}

template <typename T>

void DictionaryTypeHandlerBase<T>::InvalidateFixedField(DynamicObject\* instance, PropertyId propertyId, DictionaryPropertyDescriptor<T>\* descriptor)

{

// DictionaryTypeHandlers are never shared, but if they were we would need to invalidate even if

// there wasn't a singleton instance. See SimpleDictionaryTypeHandler::InvalidateFixedFields.

Assert(!GetIsOrMayBecomeShared());

if (this->singletonInstance != nullptr)

{

Assert(this->singletonInstance->Get() == instance);

// Even if we wrote a new value into this property (overwriting a previously fixed one), we don't

// consider the new one fixed. This also means that it's ok to populate the inline caches for

// this property from now on.

descriptor->IsFixed = false;

if (descriptor->UsedAsFixed)

{

// Invalidate any JIT-ed code that hard coded this method. No need to invalidate

// any store field inline caches, because they have never been populated.

#if ENABLE\_NATIVE\_CODEGEN

instance->GetScriptContext()->GetThreadContext()->InvalidatePropertyGuards(propertyId);

#endif

descriptor->UsedAsFixed = false;

}

}

}

#if DBG

template <typename T>

bool DictionaryTypeHandlerBase<T>::IsLetConstGlobal(DynamicObject\* instance, PropertyId propertyId)

{

DictionaryPropertyDescriptor<T>\* descriptor;

PropertyRecord const\* propertyRecord = instance->GetScriptContext()->GetPropertyName(propertyId);

if (propertyMap->TryGetReference(propertyRecord, &descriptor) && (descriptor->Attributes & PropertyLetConstGlobal))

{

return true;

}

return false;

}

#endif

template <typename T>

bool DictionaryTypeHandlerBase<T>::NextLetConstGlobal(int& index, RootObjectBase\* instance, const PropertyRecord\*\* propertyRecord, Var\* value, bool\* isConst)

{

for (; index < propertyMap->Count(); index++)

{

DictionaryPropertyDescriptor<T> descriptor = propertyMap->GetValueAt(index);

if (descriptor.Attributes & PropertyLetConstGlobal)

{

\*propertyRecord = propertyMap->GetKeyAt(index);

\*value = instance->GetSlot(descriptor.GetDataPropertyIndex<true>());

\*isConst = (descriptor.Attributes & PropertyConst) != 0;

index += 1;

return true;

}

}

return false;

}

#ifdef ENABLE\_DEBUG\_CONFIG\_OPTIONS

template <typename T>

void DictionaryTypeHandlerBase<T>::DumpFixedFields() const {

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

{

DictionaryPropertyDescriptor<T> descriptor = propertyMap->GetValueAt(i);

const PropertyRecord\* propertyRecord = propertyMap->GetKeyAt(i);

Output::Print(L" %s %d%d%d,", propertyRecord->GetBuffer(),

descriptor.IsInitialized ? 1 : 0, descriptor.IsFixed ? 1 : 0, descriptor.UsedAsFixed ? 1 : 0);

}

}

template <typename T>

void DictionaryTypeHandlerBase<T>::TraceFixedFieldsBeforeTypeHandlerChange(

const wchar\_t\* oldTypeHandlerName, const wchar\_t\* newTypeHandlerName,

DynamicObject\* instance, DynamicTypeHandler\* oldTypeHandler,

DynamicType\* oldType, RecyclerWeakReference<DynamicObject>\* oldSingletonInstanceBefore)

{

if (PHASE\_VERBOSE\_TRACE1(FixMethodPropsPhase))

{

Output::Print(L"FixedFields: converting 0x%p from %s to %s:\n", instance, oldTypeHandlerName, newTypeHandlerName);

Output::Print(L" before: type = 0x%p, type handler = 0x%p, old singleton = 0x%p(0x%p)\n",

oldType, oldTypeHandler, oldSingletonInstanceBefore, oldSingletonInstanceBefore != nullptr ? oldSingletonInstanceBefore->Get() : nullptr);

Output::Print(L" fixed fields:");

oldTypeHandler->DumpFixedFields();

Output::Print(L"\n");

}

if (PHASE\_VERBOSE\_TESTTRACE1(FixMethodPropsPhase))

{

Output::Print(L"FixedFields: converting instance from %s to %s:\n", oldTypeHandlerName, newTypeHandlerName);

Output::Print(L" old singleton before %s null \n", oldSingletonInstanceBefore == nullptr ? L"==" : L"!=");

Output::Print(L" fixed fields before:");

oldTypeHandler->DumpFixedFields();

Output::Print(L"\n");

}

}

template <typename T>

void DictionaryTypeHandlerBase<T>::TraceFixedFieldsAfterTypeHandlerChange(

DynamicObject\* instance, DynamicTypeHandler\* oldTypeHandler, DynamicTypeHandler\* newTypeHandler,

DynamicType\* oldType, RecyclerWeakReference<DynamicObject>\* oldSingletonInstanceBefore)

{

if (PHASE\_VERBOSE\_TRACE1(FixMethodPropsPhase))

{

RecyclerWeakReference<DynamicObject>\* oldSingletonInstanceAfter = oldTypeHandler->GetSingletonInstance();

RecyclerWeakReference<DynamicObject>\* newSingletonInstanceAfter = newTypeHandler->GetSingletonInstance();

Output::Print(L" after: type = 0x%p, type handler = 0x%p, old singleton = 0x%p(0x%p), new singleton = 0x%p(0x%p)\n",

instance->GetType(), newTypeHandler,

oldSingletonInstanceAfter, oldSingletonInstanceAfter != nullptr ? oldSingletonInstanceAfter->Get() : nullptr,

newSingletonInstanceAfter, newSingletonInstanceAfter != nullptr ? newSingletonInstanceAfter->Get() : nullptr);

Output::Print(L" fixed fields after:");

newTypeHandler->DumpFixedFields();

Output::Print(L"\n");

Output::Flush();

}

if (PHASE\_VERBOSE\_TESTTRACE1(FixMethodPropsPhase))

{

Output::Print(L" type %s, typeHandler %s, old singleton after %s null (%s), new singleton after %s null\n",

oldTypeHandler != newTypeHandler ? L"changed" : L"unchanged",

oldType != instance->GetType() ? L"changed" : L"unchanged",

oldSingletonInstanceBefore == nullptr ? L"==" : L"!=",

oldSingletonInstanceBefore != oldTypeHandler->GetSingletonInstance() ? L"changed" : L"unchanged",

newTypeHandler->GetSingletonInstance() == nullptr ? L"==" : L"!=");

Output::Print(L" fixed fields after:");

newTypeHandler->DumpFixedFields();

Output::Print(L"\n");

Output::Flush();

}

}

template <typename T>

void DictionaryTypeHandlerBase<T>::TraceFixedFieldsBeforeSetIsProto(

DynamicObject\* instance, DynamicTypeHandler\* oldTypeHandler, DynamicType\* oldType, RecyclerWeakReference<DynamicObject>\* oldSingletonInstanceBefore)

{

if (PHASE\_VERBOSE\_TRACE1(FixMethodPropsPhase))

{

Output::Print(L"FixedFields: PathTypeHandler::SetIsPrototype(0x%p):\n", instance);

Output::Print(L" before: type = 0x%p, old singleton = 0x%p(0x%p)\n",

oldType, oldSingletonInstanceBefore, oldSingletonInstanceBefore != nullptr ? oldSingletonInstanceBefore->Get() : nullptr);

Output::Print(L" fixed fields:");

oldTypeHandler->DumpFixedFields();

Output::Print(L"\n");

}

if (PHASE\_VERBOSE\_TESTTRACE1(FixMethodPropsPhase))

{

Output::Print(L"FixedFields: PathTypeHandler::SetIsPrototype():\n");

Output::Print(L" old singleton before %s null \n", oldSingletonInstanceBefore == nullptr ? L"==" : L"!=");

Output::Print(L" fixed fields before:");

oldTypeHandler->DumpFixedFields();

Output::Print(L"\n");

}

}

template <typename T>

void DictionaryTypeHandlerBase<T>::TraceFixedFieldsAfterSetIsProto(

DynamicObject\* instance, DynamicTypeHandler\* oldTypeHandler, DynamicTypeHandler\* newTypeHandler,

DynamicType\* oldType, RecyclerWeakReference<DynamicObject>\* oldSingletonInstanceBefore)

{

if (PHASE\_VERBOSE\_TRACE1(FixMethodPropsPhase))

{

RecyclerWeakReference<DynamicObject>\* oldSingletonInstanceAfter = oldTypeHandler->GetSingletonInstance();

RecyclerWeakReference<DynamicObject>\* newSingletonInstanceAfter = newTypeHandler->GetSingletonInstance();

Output::Print(L" after: type = 0x%p, type handler = 0x%p, old singleton = 0x%p(0x%p), new singleton = 0x%p(0x%p)\n",

instance->GetType(), newTypeHandler,

oldSingletonInstanceAfter, oldSingletonInstanceAfter != nullptr ? oldSingletonInstanceAfter->Get() : nullptr,

newSingletonInstanceAfter, newSingletonInstanceAfter != nullptr ? newSingletonInstanceAfter->Get() : nullptr);

Output::Print(L" fixed fields:");

newTypeHandler->DumpFixedFields();

Output::Print(L"\n");

Output::Flush();

}

if (PHASE\_VERBOSE\_TESTTRACE1(FixMethodPropsPhase))

{

Output::Print(L" type %s, old singleton after %s null (%s)\n",

oldType != instance->GetType() ? L"changed" : L"unchanged",

oldSingletonInstanceBefore == nullptr ? L"==" : L"!=",

oldSingletonInstanceBefore != oldTypeHandler->GetSingletonInstance() ? L"changed" : L"unchanged");

Output::Print(L" fixed fields after:");

newTypeHandler->DumpFixedFields();

Output::Print(L"\n");

Output::Flush();

}

}

#endif

template class DictionaryTypeHandlerBase<PropertyIndex>;

template class DictionaryTypeHandlerBase<BigPropertyIndex>;

template <bool allowLetConstGlobal>

PropertyAttributes GetLetConstGlobalPropertyAttributes(PropertyAttributes attributes)

{

return (allowLetConstGlobal && (attributes & PropertyLetConstGlobal) != 0) ? (attributes | PropertyWritable) : attributes;

}

}

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

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

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

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

#pragma once

namespace Js

{

template <typename T>

class DictionaryTypeHandlerBase: public DynamicTypeHandler

{

private:

friend class NullTypeHandlerBase;

friend class DeferredTypeHandlerBase;

template <DeferredTypeInitializer initializer, typename DeferredTypeFilter, bool isPrototypeTemplate, uint16 inlineSlotCapacity, uint16 offsetOfInlineSlots>

friend class DeferredTypeHandler;

friend class PathTypeHandlerBase;

template<size\_t size>

friend class SimpleTypeHandler;

friend class DynamicObject;

friend class DynamicTypeHandler;

template <typename TPropertyIndex, typename TMapKey, bool IsNotExtensibleSupported> friend class SimpleDictionaryTypeHandlerBase;

template <typename T> friend class DictionaryTypeHandlerBase;

// Explicit non leaf allocator as the key is non-leaf

typedef JsUtil::BaseDictionary<const PropertyRecord\*, DictionaryPropertyDescriptor<T>, RecyclerNonLeafAllocator, DictionarySizePolicy<PowerOf2Policy, 1>, PropertyRecordStringHashComparer>

PropertyDescriptorMap;

typedef PropertyDescriptorMap PropertyDescriptorMapType; // alias used by diagnostics

private:

PropertyDescriptorMap\* propertyMap;

T nextPropertyIndex;

RecyclerWeakReference<DynamicObject>\* singletonInstance;

typedef PropertyIndexRanges<T> PropertyIndexRangesType;

static const T MaxPropertyIndexSize = PropertyIndexRangesType::MaxValue;

static const T NoSlots = PropertyIndexRangesType::NoSlots;

public:

DEFINE\_GETCPPNAME();

protected:

DictionaryTypeHandlerBase(Recycler\* recycler);

DictionaryTypeHandlerBase(Recycler\* recycler, int slotCapacity, uint16 inlineSlotCapacity, uint16 offsetOfInlineSlots);

DictionaryTypeHandlerBase(DictionaryTypeHandlerBase\* typeHandler);

DEFINE\_VTABLE\_CTOR\_NO\_REGISTER(DictionaryTypeHandlerBase, DynamicTypeHandler);

// Create a new type handler for a future DynamicObject. This is for public usage. "initialCapacity" indicates desired slotCapacity, subject to alignment round up.

template <typename SubClassType>

static SubClassType\* NewTypeHandler(Recycler \* recycler, int initialCapacity, uint16 inlineSlotCapacity, uint16 offsetOfInlineSlots)

{

PropertyIndexRangesType::VerifySlotCapacity(initialCapacity);

return RecyclerNew(recycler, SubClassType, recycler, initialCapacity, inlineSlotCapacity, offsetOfInlineSlots);

}

public:

typedef T PropertyIndexType;

// Create a new type handler for a future DynamicObject. This is for public usage. "initialCapacity" indicates desired slotCapacity, subject to alignment round up.

static DictionaryTypeHandlerBase\* New(Recycler \* recycler, int initialCapacity, uint16 inlineSlotCapacity, uint16 offsetOfInlineSlots);

BOOL IsBigDictionaryTypeHandler();

virtual BOOL IsLockable() const override { return false; }

virtual BOOL IsSharable() const override { return false; }

virtual void DoShareTypeHandler(ScriptContext\* scriptContext) override { AssertMsg(false, "DictionaryTypeHandlers cannot be shared."); };

virtual int GetPropertyCount() override;

virtual PropertyId GetPropertyId(ScriptContext\* scriptContext, PropertyIndex index) override;

virtual PropertyId GetPropertyId(ScriptContext\* scriptContext, BigPropertyIndex index) override;

virtual BOOL FindNextProperty(ScriptContext\* scriptContext, PropertyIndex& index, JavascriptString\*\* propertyString,

PropertyId\* propertyId, PropertyAttributes\* attributes, Type\* type, DynamicType \*typeToEnumerate, bool requireEnumerable, bool enumSymbols = false) override;

virtual BOOL FindNextProperty(ScriptContext\* scriptContext, BigPropertyIndex& index, JavascriptString\*\* propertyString,

PropertyId\* propertyId, PropertyAttributes\* attributes, Type\* type, DynamicType \*typeToEnumerate, bool requireEnumerable, bool enumSymbols = false) override;

virtual PropertyIndex GetPropertyIndex(PropertyRecord const\* propertyRecord) override;

virtual bool GetPropertyEquivalenceInfo(PropertyRecord const\* propertyRecord, PropertyEquivalenceInfo& info) override;

virtual bool IsObjTypeSpecEquivalent(const Type\* type, const TypeEquivalenceRecord& record, uint& failedPropertyIndex) override;

virtual bool IsObjTypeSpecEquivalent(const Type\* type, const EquivalentPropertyEntry\* entry) override;

virtual BOOL HasProperty(DynamicObject\* instance, PropertyId propertyId, bool \*noRedecl = nullptr) override;

virtual BOOL HasProperty(DynamicObject\* instance, JavascriptString\* propertyNameString) override;

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

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

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

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

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

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

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

virtual BOOL DeleteProperty(DynamicObject\* instance, PropertyId propertyId, PropertyOperationFlags flags) override;

virtual PropertyIndex GetRootPropertyIndex(PropertyRecord const\* propertyRecord) override;

virtual BOOL HasRootProperty(DynamicObject\* instance, PropertyId propertyId, bool \*noRedecl, bool \*pDeclaredProperty = nullptr) override;

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

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

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

virtual BOOL DeleteRootProperty(DynamicObject\* instance, PropertyId propertyId, PropertyOperationFlags flags) override;

virtual BOOL IsDictionaryTypeHandler() const {return TRUE;}

#if DBG

virtual bool IsLetConstGlobal(DynamicObject\* instance, PropertyId propertyId) override;

#endif

virtual bool NextLetConstGlobal(int& index, RootObjectBase\* instance, const PropertyRecord\*\* propertyRecord, Var\* value, bool\* isConst) override;

virtual BOOL IsFixedProperty(const DynamicObject\* instance, PropertyId propertyId) override;

virtual BOOL IsEnumerable(DynamicObject\* instance, PropertyId propertyId) override;

virtual BOOL IsWritable(DynamicObject\* instance, PropertyId propertyId) override;

virtual BOOL IsConfigurable(DynamicObject\* instance, PropertyId propertyId) override;

virtual BOOL SetEnumerable(DynamicObject\* instance, PropertyId propertyId, BOOL value) override;

virtual BOOL SetWritable(DynamicObject\* instance, PropertyId propertyId, BOOL value) override;

virtual BOOL SetConfigurable(DynamicObject\* instance, PropertyId propertyId, BOOL value) override;

virtual BOOL SetItem(DynamicObject\* instance, uint32 index, Var value, PropertyOperationFlags flags) override;

virtual BOOL SetItemWithAttributes(DynamicObject\* instance, uint32 index, Var value, PropertyAttributes attributes) override;

virtual BOOL SetItemAttributes(DynamicObject\* instance, uint32 index, PropertyAttributes attributes) override;

virtual BOOL SetItemAccessors(DynamicObject\* instance, uint32 index, Var getter, Var setter) override;

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

virtual BOOL SetAccessors(DynamicObject\* instance, PropertyId propertyId, Var getter, Var setter, PropertyOperationFlags flags = PropertyOperation\_None) override sealed;

virtual BOOL GetAccessors(DynamicObject\* instance, PropertyId propertyId, Var\* getter, Var\* setter) override;

virtual BOOL PreventExtensions(DynamicObject \*instance) override;

virtual BOOL Seal(DynamicObject\* instance) override;

virtual BOOL IsSealed(DynamicObject\* instance) override;

virtual BOOL IsFrozen(DynamicObject\* instance) override;

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

virtual BOOL SetAttributes(DynamicObject\* instance, PropertyId propertyId, PropertyAttributes attributes) override;

virtual BOOL GetAttributesWithPropertyIndex(DynamicObject \* instance, PropertyId propertyId, BigPropertyIndex index, PropertyAttributes \* attributes) override;

virtual void SetAllPropertiesToUndefined(DynamicObject\* instance, bool invalidateFixedFields) override;

virtual void MarshalAllPropertiesToScriptContext(DynamicObject\* instance, ScriptContext\* targetScriptContext, bool invalidateFixedFields) override;

virtual DynamicTypeHandler\* ConvertToTypeWithItemAttributes(DynamicObject\* instance) override;

virtual void SetIsPrototype(DynamicObject\* instance) override;

#if DBG

virtual bool SupportsPrototypeInstances() const { return true; }

#endif

virtual bool HasSingletonInstance() const override sealed;

virtual bool TryUseFixedProperty(PropertyRecord const \* propertyRecord, Var \* pProperty, FixedPropertyKind propertyType, ScriptContext \* requestContext) override;

virtual bool TryUseFixedAccessor(PropertyRecord const \* propertyRecord, Var \* pAccessor, FixedPropertyKind propertyType, bool getter, ScriptContext \* requestContext) override;

#if DBG

virtual bool CanStorePropertyValueDirectly(const DynamicObject\* instance, PropertyId propertyId, bool allowLetConst) override;

virtual bool CheckFixedProperty(PropertyRecord const \* propertyRecord, Var \* pProperty, ScriptContext \* requestContext) override;

virtual bool HasAnyFixedProperties() const override;

#endif

#ifdef ENABLE\_DEBUG\_CONFIG\_OPTIONS

virtual void DumpFixedFields() const override;

static void TraceFixedFieldsBeforeTypeHandlerChange(

const wchar\_t\* oldTypeHandlerName, const wchar\_t\* newTypeHandlerName,

DynamicObject\* instance, DynamicTypeHandler\* oldTypeHandler, DynamicType\* oldType, RecyclerWeakReference<DynamicObject>\* oldSingletonInstanceBefore);

static void TraceFixedFieldsAfterTypeHandlerChange(

DynamicObject\* instance, DynamicTypeHandler\* oldTypeHandler, DynamicTypeHandler\* newTypeHandler,

DynamicType\* oldType, RecyclerWeakReference<DynamicObject>\* oldSingletonInstanceBefore);

static void TraceFixedFieldsBeforeSetIsProto(

DynamicObject\* instance, DynamicTypeHandler\* oldTypeHandler, DynamicType\* oldType, RecyclerWeakReference<DynamicObject>\* oldSingletonInstanceBefore);

static void TraceFixedFieldsAfterSetIsProto(

DynamicObject\* instance, DynamicTypeHandler\* oldTypeHandler, DynamicTypeHandler\* newTypeHandler,

DynamicType\* oldType, RecyclerWeakReference<DynamicObject>\* oldSingletonInstanceBefore);

#endif

private:

template<bool doLock>

bool IsObjTypeSpecEquivalentImpl(const Type\* type, const EquivalentPropertyEntry \*entry);

template <bool allowNonExistent, bool markAsUsed>

bool TryGetFixedProperty(PropertyRecord const \* propertyRecord, Var \* pProperty, FixedPropertyKind propertyType, ScriptContext \* requestContext);

template <bool allowNonExistent, bool markAsUsed>

bool TryGetFixedAccessor(PropertyRecord const \* propertyRecord, Var \* pAccessor, FixedPropertyKind propertyType, bool getter, ScriptContext \* requestContext);

public:

virtual RecyclerWeakReference<DynamicObject>\* GetSingletonInstance() const sealed { Assert(HasSingletonInstanceOnlyIfNeeded()); return this->singletonInstance; }

virtual void SetSingletonInstanceUnchecked(RecyclerWeakReference<DynamicObject>\* instance) override

{

Assert(!GetIsShared());

Assert(this->singletonInstance == nullptr);

this->singletonInstance = instance;

}

virtual void ClearSingletonInstance() override sealed

{

Assert(HasSingletonInstanceOnlyIfNeeded());

this->singletonInstance = nullptr;

}

#if DBG

bool HasSingletonInstanceOnlyIfNeeded() const

{

return AreSingletonInstancesNeeded() || this->singletonInstance == nullptr;

}

#endif

private:

void CopySingletonInstance(DynamicObject\* instance, DynamicTypeHandler\* typeHandler);

void InvalidateFixedField(DynamicObject\* instance, PropertyId propertyId, DictionaryPropertyDescriptor<T>\* descriptor);

private:

void SetNumDeletedProperties(const byte n) {}

void Add(const PropertyRecord\* propertyId, PropertyAttributes attributes, ScriptContext \*const scriptContext);

void Add(const PropertyRecord\* propertyId, PropertyAttributes attributes, bool isInitialized, bool isFixed, bool usedAsFixed, ScriptContext \*const scriptContext);

void EnsureSlotCapacity(DynamicObject \* instance);

BOOL AddProperty(DynamicObject\* instance, const PropertyRecord\* propertyRecord, Var value, PropertyAttributes attributes, PropertyValueInfo\* info, PropertyOperationFlags flags, bool throwIfNotExtensible, SideEffects possibleSideEffects);

ES5ArrayTypeHandlerBase<T>\* ConvertToES5ArrayType(DynamicObject \*instance);

BigDictionaryTypeHandler\* ConvertToBigDictionaryTypeHandler(DynamicObject\* instance);

void SetPropertyValueInfo(PropertyValueInfo\* info, RecyclableObject\* instance, T propIndex, PropertyAttributes attributes, InlineCacheFlags flags = InlineCacheNoFlags);

template<bool allowLetConstGlobal>

\_\_inline BOOL HasProperty\_Internal(DynamicObject\* instance, PropertyId propertyId, bool \*noRedecl, bool \*pDeclaredProperty);

template<bool allowLetConstGlobal>

\_\_inline PropertyIndex GetPropertyIndex\_Internal(PropertyRecord const\* propertyRecord);

template<bool allowLetConstGlobal>

\_\_inline BOOL GetProperty\_Internal(DynamicObject\* instance, Var originalInstance, PropertyId propertyId, Var\* value, PropertyValueInfo\* info, ScriptContext\* requestContext);

template<bool allowLetConstGlobal>

\_\_inline BOOL SetProperty\_Internal(DynamicObject\* instance, PropertyId propertyId, Var value, PropertyOperationFlags flags, PropertyValueInfo\* info, bool isInit = false);

template<bool allowLetConstGlobal>

\_\_inline DescriptorFlags GetSetter\_Internal(DynamicObject\* instance, PropertyId propertyId, Var\* setterValue, PropertyValueInfo\* info, ScriptContext\* requestContext);

template<bool allowLetConstGlobal>

\_\_inline BOOL DeleteProperty\_Internal(DynamicObject\* instance, PropertyId propertyId, PropertyOperationFlags flags);

template<bool allowLetConstGlobal, typename PropertyType>

\_\_inline BOOL GetPropertyFromDescriptor(DynamicObject\* instance, Var originalInstance, DictionaryPropertyDescriptor<T>\* descriptor,

Var\* value, PropertyValueInfo\* info, PropertyType propertyT, ScriptContext\* requestContext);

template<bool allowLetConstGlobal>

\_\_inline DescriptorFlags GetSetterFromDescriptor(DynamicObject\* instance, DictionaryPropertyDescriptor<T> \* descriptor, Var\* setterValue, PropertyValueInfo\* info);

template <bool allowLetConstGlobal>

\_\_inline void SetPropertyWithDescriptor(DynamicObject\* instance, PropertyId propertyId, DictionaryPropertyDescriptor<T> \* descriptor,

Var value, PropertyOperationFlags flags, PropertyValueInfo\* info);

protected:

virtual BOOL FreezeImpl(DynamicObject\* instance, bool isConvertedType) override;

virtual BigDictionaryTypeHandler\* NewBigDictionaryTypeHandler(Recycler\* recycler, int slotCapacity, uint16 inlineSlotCapacity, uint16 offsetOfInlineSlots);

static Var CanonicalizeAccessor(Var accessor, /\*const\*/ JavascriptLibrary\* library);

};

template <bool allowLetConstGlobal>

\_\_inline PropertyAttributes GetLetConstGlobalPropertyAttributes(PropertyAttributes attributes);

}

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

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

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

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

#include "RuntimeTypePch.h"

namespace Js

{

DEFINE\_RECYCLER\_TRACKER\_PERF\_COUNTER(DynamicObject);

DEFINE\_RECYCLER\_TRACKER\_WEAKREF\_PERF\_COUNTER(DynamicObject);

DynamicObject::DynamicObject(DynamicType \* type, const bool initSlots) :

RecyclableObject(type),

auxSlots(nullptr),

objectArray(nullptr)

{

Assert(!UsesObjectArrayOrFlagsAsFlags());

if(initSlots)

{

InitSlots(this);

}

else

{

Assert(type->GetTypeHandler()->GetInlineSlotCapacity() == type->GetTypeHandler()->GetSlotCapacity());

}

}

DynamicObject::DynamicObject(DynamicType \* type, ScriptContext \* scriptContext) :

#if DBG || defined(PROFILE\_TYPES)

RecyclableObject(type, scriptContext),

#else

RecyclableObject(type),

#endif

auxSlots(nullptr),

objectArray(nullptr)

{

Assert(!UsesObjectArrayOrFlagsAsFlags());

InitSlots(this, scriptContext);

}

DynamicObject::DynamicObject(DynamicObject \* instance) :

RecyclableObject(instance->type),

auxSlots(instance->auxSlots),

objectArray(instance->objectArray) // copying the array should copy the array flags and array call site index as well

{

DynamicTypeHandler \* typeHandler = this->GetTypeHandler();

// TODO: stack allocate aux Slots

Assert(typeHandler->IsObjectHeaderInlinedTypeHandler() || !ThreadContext::IsOnStack(this->auxSlots));

int propertyCount = typeHandler->GetPropertyCount();

int inlineSlotCapacity = GetTypeHandler()->GetInlineSlotCapacity();

int inlineSlotCount = min(inlineSlotCapacity, propertyCount);

Var \* srcSlots = reinterpret\_cast<Var\*>(reinterpret\_cast<size\_t>(instance) + typeHandler->GetOffsetOfInlineSlots());

Var \* dstSlots = reinterpret\_cast<Var\*>(reinterpret\_cast<size\_t>(this) + typeHandler->GetOffsetOfInlineSlots());

#if !FLOATVAR

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

#endif

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

{

#if !FLOATVAR

// Currently we only support temp numbers assigned to stack objects

dstSlots[i] = JavascriptNumber::BoxStackNumber(srcSlots[i], scriptContext);

#else

dstSlots[i] = srcSlots[i];

#endif

}

if (propertyCount > inlineSlotCapacity)

{

uint auxSlotCount = propertyCount - inlineSlotCapacity;

for (uint i = 0; i < auxSlotCount; i++)

{

#if !FLOATVAR

// Currently we only support temp numbers assigned to stack objects

auxSlots[i] = JavascriptNumber::BoxStackNumber(instance->auxSlots[i], scriptContext);

#else

auxSlots[i] = instance->auxSlots[i];

#endif

}

}

}

DynamicObject \* DynamicObject::New(Recycler \* recycler, DynamicType \* type)

{

return NewObject<DynamicObject>(recycler, type);

}

bool DynamicObject::Is(Var aValue)

{

return RecyclableObject::Is(aValue) && (RecyclableObject::FromVar(aValue)->GetTypeId() == TypeIds\_Object);

}

DynamicObject\* DynamicObject::FromVar(Var aValue)

{

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

AssertMsg(obj->DbgIsDynamicObject(), "Ensure instance is actually a DynamicObject");

Assert(DynamicType::Is(obj->GetTypeId()));

return static\_cast<DynamicObject\*>(obj);

}

ArrayObject\* DynamicObject::EnsureObjectArray()

{

if (!HasObjectArray())

{

ScriptContext\* scriptContext = GetScriptContext();

ArrayObject\* objArray = scriptContext->GetLibrary()->CreateArray(0, SparseArraySegmentBase::SMALL\_CHUNK\_SIZE);

SetObjectArray(objArray);

}

Assert(HasObjectArray());

return GetObjectArrayOrFlagsAsArray();

}

void DynamicObject::SetObjectArray(ArrayObject\* objArray)

{

Assert(!IsAnyArray(this));

DeoptimizeObjectHeaderInlining();

this->objectArray = objArray;

if (objArray)

{

if (!this->IsExtensible()) // sync objectArray isExtensible

{

objArray->PreventExtensions();

}

// sync objectArray is prototype

if ((this->GetTypeHandler()->GetFlags() & DynamicTypeHandler::IsPrototypeFlag) != 0)

{

objArray->SetIsPrototype();

}

}

}

bool DynamicObject::HasNonEmptyObjectArray() const

{

return HasObjectArray() && GetObjectArrayOrFlagsAsArray()->GetLength() > 0;

}

// Check if a typeId is of any array type (JavascriptArray or ES5Array).

bool DynamicObject::IsAnyArrayTypeId(TypeId typeId)

{

return JavascriptArray::Is(typeId) || typeId == TypeIds\_ES5Array;

}

// Check if a Var is either a JavascriptArray\* or ES5Array\*.

bool DynamicObject::IsAnyArray(const Var aValue)

{

return IsAnyArrayTypeId(JavascriptOperators::GetTypeId(aValue));

}

BOOL DynamicObject::HasObjectArrayItem(uint32 index)

{

return HasObjectArray() && GetObjectArrayOrFlagsAsArray()->HasItem(index);

}

BOOL DynamicObject::DeleteObjectArrayItem(uint32 index, PropertyOperationFlags flags)

{

if (HasObjectArray())

{

return GetObjectArrayOrFlagsAsArray()->DeleteItem(index, flags);

}

return true;

}

BOOL DynamicObject::GetObjectArrayItem(Var originalInstance, uint32 index, Var\* value, ScriptContext\* requestContext)

{

return HasObjectArray() && GetObjectArrayOrFlagsAsArray()->GetItem(originalInstance, index, value, requestContext);

}

DescriptorFlags DynamicObject::GetObjectArrayItemSetter(uint32 index, Var\* setterValue, ScriptContext\* requestContext)

{

return HasObjectArray() ? GetObjectArrayOrFlagsAsArray()->GetItemSetter(index, setterValue, requestContext) : None;

}

BOOL DynamicObject::SetObjectArrayItem(uint32 index, Var value, PropertyOperationFlags flags)

{

const auto result = EnsureObjectArray()->SetItem(index, value, flags);

// We don't track non-enumerable items in object arrays. Any object with an object array reports having

// enumerable properties. See comment in DynamicObject::GetHasNoEnumerableProperties.

//SetHasNoEnumerableProperties(false);

return result;

}

BOOL DynamicObject::SetObjectArrayItemWithAttributes(uint32 index, Var value, PropertyAttributes attributes)

{

const auto result = EnsureObjectArray()->SetItemWithAttributes(index, value, attributes);

// We don't track non-enumerable items in object arrays. Any object with an object array reports having

// enumerable properties. See comment in DynamicObject::GetHasNoEnumerableProperties.

//if (attributes & PropertyEnumerable)

//{

// SetHasNoEnumerableProperties(false);

//}

if (!(attributes & PropertyWritable) && result)

{

InvalidateHasOnlyWritableDataPropertiesInPrototypeChainCacheIfPrototype();

}

return result;

}

BOOL DynamicObject::SetObjectArrayItemAttributes(uint32 index, PropertyAttributes attributes)

{

const auto result = HasObjectArray() && GetObjectArrayOrFlagsAsArray()->SetItemAttributes(index, attributes);

// We don't track non-enumerable items in object arrays. Any object with an object array reports having

// enumerable properties. See comment in DynamicObject::GetHasNoEnumerableProperties.

//if (attributes & PropertyEnumerable)

//{

// SetHasNoEnumerableProperties(false);

//}

if (!(attributes & PropertyWritable) && result)

{

InvalidateHasOnlyWritableDataPropertiesInPrototypeChainCacheIfPrototype();

}

return result;

}

BOOL DynamicObject::SetObjectArrayItemWritable(PropertyId propertyId, BOOL writable)

{

const auto result = HasObjectArray() && GetObjectArrayOrFlagsAsArray()->SetWritable(propertyId, writable);

if (!writable && result)

{

InvalidateHasOnlyWritableDataPropertiesInPrototypeChainCacheIfPrototype();

}

return result;

}

BOOL DynamicObject::SetObjectArrayItemAccessors(uint32 index, Var getter, Var setter)

{

const auto result = EnsureObjectArray()->SetItemAccessors(index, getter, setter);

if (result)

{

InvalidateHasOnlyWritableDataPropertiesInPrototypeChainCacheIfPrototype();

}

return result;

}

void DynamicObject::InvalidateHasOnlyWritableDataPropertiesInPrototypeChainCacheIfPrototype()

{

if (GetTypeHandler()->GetFlags() & DynamicTypeHandler::IsPrototypeFlag)

{

// No need to invalidate store field caches for non-writable properties here. We're dealing

// with numeric properties only, and we never cache these in add property inline caches.

// If this object is used as a prototype, the has-only-writable-data-properties-in-prototype-chain cache needs to be

// invalidated here since the type handler of 'objectArray' is not marked as being used as a prototype

GetType()->GetLibrary()->NoPrototypeChainsAreEnsuredToHaveOnlyWritableDataProperties();

}

}

bool DynamicObject::HasLockedType() const

{

return this->GetDynamicType()->GetIsLocked();

}

bool DynamicObject::HasSharedType() const

{

return this->GetDynamicType()->GetIsShared();

}

bool DynamicObject::HasSharedTypeHandler() const

{

return this->GetTypeHandler()->GetIsShared();

}

void DynamicObject::ReplaceType(DynamicType \* type)

{

Assert(!type->isLocked || type->GetTypeHandler()->GetIsLocked());

Assert(!type->isShared || type->GetTypeHandler()->GetIsShared());

// For now, i have added only Aux Slot -> so new inlineSlotCapacity should be 2.

AssertMsg(DynamicObject::IsTypeHandlerCompatibleForObjectHeaderInlining(this->GetTypeHandler(), type->GetTypeHandler()),

"Object is ObjectHeaderInlined and should have compatible TypeHandlers for proper transition");

this->type = type;

}

DWORD DynamicObject::GetOffsetOfAuxSlots()

{

return offsetof(DynamicObject, auxSlots);

}

DWORD DynamicObject::GetOffsetOfObjectArray()

{

return offsetof(DynamicObject, objectArray);

}

DWORD DynamicObject::GetOffsetOfType()

{

return offsetof(DynamicObject, type);

}

void DynamicObject::EnsureSlots(int oldCount, int newCount, ScriptContext \* scriptContext, DynamicTypeHandler \* newTypeHandler)

{

this->GetTypeHandler()->EnsureSlots(this, oldCount, newCount, scriptContext, newTypeHandler);

}

void DynamicObject::EnsureSlots(int newCount, ScriptContext \* scriptContext)

{

EnsureSlots(GetTypeHandler()->GetSlotCapacity(), newCount, scriptContext);

}

Var DynamicObject::GetSlot(int index)

{

return this->GetTypeHandler()->GetSlot(this, index);

}

Var DynamicObject::GetInlineSlot(int index)

{

return this->GetTypeHandler()->GetInlineSlot(this, index);

}

Var DynamicObject::GetAuxSlot(int index)

{

return this->GetTypeHandler()->GetAuxSlot(this, index);

}

#if DBG

void DynamicObject::SetSlot(PropertyId propertyId, bool allowLetConst, int index, Var value)

{

this->GetTypeHandler()->SetSlot(this, propertyId, allowLetConst, index, value);

}

void DynamicObject::SetInlineSlot(PropertyId propertyId, bool allowLetConst, int index, Var value)

{

this->GetTypeHandler()->SetInlineSlot(this, propertyId, allowLetConst, index, value);

}

void DynamicObject::SetAuxSlot(PropertyId propertyId, bool allowLetConst, int index, Var value)

{

this->GetTypeHandler()->SetAuxSlot(this, propertyId, allowLetConst, index, value);

}

#else

void DynamicObject::SetSlot(int index, Var value)

{

this->GetTypeHandler()->SetSlot(this, index, value);

}

void DynamicObject::SetInlineSlot(int index, Var value)

{

this->GetTypeHandler()->SetInlineSlot(this, index, value);

}

void DynamicObject::SetAuxSlot(int index, Var value)

{

this->GetTypeHandler()->SetAuxSlot(this, index, value);

}

#endif

bool

DynamicObject::GetIsExtensible() const

{

return this->GetTypeHandler()->GetFlags() & DynamicTypeHandler::IsExtensibleFlag;

}

Var

DynamicObject::GetNextProperty(PropertyIndex& index, DynamicType \*typeToEnumerate, bool requireEnumerable, bool enumSymbols)

{

JavascriptString\* propertyString= nullptr;

PropertyId propertyId = Constants::NoProperty;

if (!this->GetTypeHandler()->FindNextProperty(this->GetScriptContext(), index, &propertyString, &propertyId, nullptr, this->GetType(), typeToEnumerate, requireEnumerable, enumSymbols))

{

return nullptr;

}

Assert(propertyString);

Assert(propertyId);

if (VirtualTableInfo<PropertyString>::HasVirtualTable(propertyString))

{

PropertyCache const\* cache = ((PropertyString\*)propertyString)->GetPropertyCache();

if (cache && cache->type == this->GetType())

{

if (cache->isInlineSlot)

{

return this->GetInlineSlot(cache->dataSlotIndex);

}

else

{

return this->GetAuxSlot(cache->dataSlotIndex);

}

}

}

Var value = nullptr;

BOOL result;

if (propertyId != Constants::NoProperty)

{

result = this->GetProperty(this, propertyId, &value, NULL, this->GetScriptContext());

}

else

{

result = this->GetProperty(this, propertyString, &value, NULL, this->GetScriptContext());

}

Assert(result);

return value;

}

Var

DynamicObject::GetNextProperty(BigPropertyIndex& index, DynamicType \*typeToEnumerate, bool requireEnumerable, bool enumSymbols)

{

JavascriptString\* propertyString = nullptr;

PropertyId propertyId = Constants::NoProperty;

if (!this->GetTypeHandler()->FindNextProperty(this->GetScriptContext(), index, &propertyString, &propertyId, nullptr, this->GetType(), typeToEnumerate, requireEnumerable, enumSymbols))

{

return nullptr;

}

Assert(propertyString);

Assert(propertyId);

Var value = nullptr;

BOOL result;

if (propertyId != Constants::NoProperty)

{

result = this->GetProperty(this, propertyId, &value, NULL, this->GetScriptContext());

}

else

{

result = this->GetProperty(this, propertyString, &value, NULL, this->GetScriptContext());

}

Assert(result);

return value;

}

BOOL

DynamicObject::FindNextProperty(PropertyIndex& index, JavascriptString\*\* propertyString, PropertyId\* propertyId, PropertyAttributes\* attributes, DynamicType \*typeToEnumerate, bool requireEnumerable, bool enumSymbols) const

{

if (index == Constants::NoSlot)

{

return FALSE;

}

return this->GetTypeHandler()->FindNextProperty(this->GetScriptContext(), index, propertyString, propertyId, attributes, this->GetType(), typeToEnumerate, requireEnumerable, enumSymbols);

}

BOOL

DynamicObject::FindNextProperty(BigPropertyIndex& index, JavascriptString\*\* propertyString, PropertyId\* propertyId, PropertyAttributes\* attributes, DynamicType \*typeToEnumerate, bool requireEnumerable, bool enumSymbols) const

{

if (index == Constants::NoBigSlot)

{

return FALSE;

}

return this->GetTypeHandler()->FindNextProperty(this->GetScriptContext(), index, propertyString, propertyId, attributes, this->GetType(), typeToEnumerate, requireEnumerable, enumSymbols);

}

BOOL

DynamicObject::HasDeferredTypeHandler() const

{

return this->GetTypeHandler()->IsDeferredTypeHandler();

}

DynamicTypeHandler \*

DynamicObject::GetTypeHandler() const

{

return this->GetDynamicType()->GetTypeHandler();

}

uint16 DynamicObject::GetOffsetOfInlineSlots() const

{

return this->GetDynamicType()->GetTypeHandler()->GetOffsetOfInlineSlots();

}

void

DynamicObject::SetTypeHandler(DynamicTypeHandler \* typeHandler, bool hasChanged)

{

if (hasChanged && this->HasLockedType())

{

this->ChangeType();

}

this->GetDynamicType()->typeHandler = typeHandler;

}

DynamicType\* DynamicObject::DuplicateType()

{

return RecyclerNew(GetRecycler(), DynamicType, this->GetDynamicType());

}

/\*

\* DynamicObject::IsTypeHandlerCompatibleForObjectHeaderInlining

\* - Checks if the TypeHandlers are compatible for transition from oldTypeHandler to newTypeHandler

\*/

bool DynamicObject::IsTypeHandlerCompatibleForObjectHeaderInlining(DynamicTypeHandler \* oldTypeHandler, DynamicTypeHandler \* newTypeHandler)

{

Assert(oldTypeHandler);

Assert(newTypeHandler);

return

oldTypeHandler->GetInlineSlotCapacity() == newTypeHandler->GetInlineSlotCapacity() ||

(

oldTypeHandler->IsObjectHeaderInlinedTypeHandler() &&

newTypeHandler->GetInlineSlotCapacity() ==

oldTypeHandler->GetInlineSlotCapacity() - DynamicTypeHandler::GetObjectHeaderInlinableSlotCapacity()

);

}

bool DynamicObject::IsObjectHeaderInlinedTypeHandlerUnchecked() const

{

return this->GetTypeHandler()->IsObjectHeaderInlinedTypeHandlerUnchecked();

}

bool DynamicObject::IsObjectHeaderInlinedTypeHandler() const

{

return this->GetTypeHandler()->IsObjectHeaderInlinedTypeHandler();

}

bool DynamicObject::DeoptimizeObjectHeaderInlining()

{

if(!IsObjectHeaderInlinedTypeHandler())

{

return false;

}

if (PHASE\_TRACE1(Js::ObjectHeaderInliningPhase))

{

Output::Print(L"ObjectHeaderInlining: De-optimizing the object.\n");

Output::Flush();

}

PathTypeHandlerBase \*const oldTypeHandler = PathTypeHandlerBase::FromTypeHandler(GetTypeHandler());

SimplePathTypeHandler \*const newTypeHandler = oldTypeHandler->DeoptimizeObjectHeaderInlining(GetLibrary());

const PropertyIndex newInlineSlotCapacity = newTypeHandler->GetInlineSlotCapacity();

DynamicTypeHandler::AdjustSlots(

this,

newInlineSlotCapacity,

newTypeHandler->GetSlotCapacity() - newInlineSlotCapacity);

DynamicType \*const newType = DuplicateType();

newType->typeHandler = newTypeHandler;

newType->ShareType();

type = newType;

return true;

}

void DynamicObject::ChangeType()

{

Assert(!GetDynamicType()->GetIsShared() || GetTypeHandler()->GetIsShared());

this->type = this->DuplicateType();

}

void DynamicObject::ChangeTypeIf(const Type\* oldType)

{

if (this->type == oldType)

{

ChangeType();

}

}

DynamicObjectFlags DynamicObject::GetArrayFlags() const

{

Assert(IsAnyArray(const\_cast<DynamicObject \*>(this)));

Assert(UsesObjectArrayOrFlagsAsFlags()); // an array object never has another internal array

return arrayFlags & DynamicObjectFlags::AllArrayFlags;

}

DynamicObjectFlags DynamicObject::GetArrayFlags\_Unchecked() const // do not use except in extreme circumstances

{

return arrayFlags & DynamicObjectFlags::AllArrayFlags;

}

void DynamicObject::InitArrayFlags(const DynamicObjectFlags flags)

{

Assert(IsAnyArray(this));

Assert(this->objectArray == nullptr);

Assert((flags & DynamicObjectFlags::ObjectArrayFlagsTag) == DynamicObjectFlags::ObjectArrayFlagsTag);

Assert((flags & ~DynamicObjectFlags::AllFlags) == DynamicObjectFlags::None);

this->arrayFlags = flags;

}

void DynamicObject::SetArrayFlags(const DynamicObjectFlags flags)

{

Assert(IsAnyArray(this));

Assert(UsesObjectArrayOrFlagsAsFlags()); // an array object never has another internal array

// Make sure we don't attempt to set any flags outside of the range of array flags.

Assert((arrayFlags & ~DynamicObjectFlags::AllArrayFlags) == DynamicObjectFlags::ObjectArrayFlagsTag);

Assert((flags & ~DynamicObjectFlags::AllArrayFlags) == DynamicObjectFlags::None);

arrayFlags = flags | DynamicObjectFlags::ObjectArrayFlagsTag;

}

ProfileId DynamicObject::GetArrayCallSiteIndex() const

{

Assert(IsAnyArray(const\_cast<DynamicObject \*>(this)));

return arrayCallSiteIndex;

}

void DynamicObject::SetArrayCallSiteIndex(ProfileId profileId)

{

Assert(IsAnyArray(this));

arrayCallSiteIndex = profileId;

}

void DynamicObject::SetIsPrototype()

{

DynamicTypeHandler\* currentTypeHandler = this->GetTypeHandler();

Js::DynamicType\* oldType = this->GetDynamicType();

#if DBG

bool wasShared = currentTypeHandler->GetIsShared();

bool wasPrototype = (currentTypeHandler->GetFlags() & DynamicTypeHandler::IsPrototypeFlag) != 0;

Assert(!DynamicTypeHandler::IsolatePrototypes() || !currentTypeHandler->RespectsIsolatePrototypes() || !currentTypeHandler->GetIsOrMayBecomeShared() || !wasPrototype);

#endif

// If this handler is not shared and it already has a prototype flag then we must have taken the required

// type transition (if any) earlier when the singleton object first became a prototype.

if ((currentTypeHandler->GetFlags() & (DynamicTypeHandler::IsSharedFlag | DynamicTypeHandler::IsPrototypeFlag)) == DynamicTypeHandler::IsPrototypeFlag)

{

Assert(this->GetObjectArray() == nullptr || (this->GetObjectArray()->GetTypeHandler()->GetFlags() & DynamicTypeHandler::IsPrototypeFlag) != 0);

return;

}

currentTypeHandler->SetIsPrototype(this);

// Get type handler again, in case it got changed by SetIsPrototype.

currentTypeHandler = this->GetTypeHandler();

// Set the object array as an prototype as well, so if it is an ES5 array, we will disable the array set element fast path

ArrayObject \* objectArray = this->GetObjectArray();

if (objectArray)

{

objectArray->SetIsPrototype();

}

#if DBG

Assert(currentTypeHandler->SupportsPrototypeInstances());

Assert(!DynamicTypeHandler::IsolatePrototypes() || !currentTypeHandler->RespectsIsolatePrototypes() || !currentTypeHandler->GetIsOrMayBecomeShared());

Assert((wasPrototype && !wasShared) || !DynamicTypeHandler::ChangeTypeOnProto() || !currentTypeHandler->RespectsChangeTypeOnProto() || this->GetDynamicType() != oldType);

#endif

// If we haven't changed type we must explicitly invalidate store field inline caches to avoid properties

// getting added to this prototype object on the fast path without proper invalidation.

if (this->GetDynamicType() == oldType)

{

currentTypeHandler->InvalidateStoreFieldCachesForAllProperties(this->GetScriptContext());

}

}

bool

DynamicObject::LockType()

{

return this->GetDynamicType()->LockType();

}

bool

DynamicObject::ShareType()

{

return this->GetDynamicType()->ShareType();

}

void

DynamicObject::ResetObject(DynamicType\* newType, BOOL keepProperties)

{

Assert(newType != NULL);

Assert(!keepProperties || (!newType->GetTypeHandler()->IsDeferredTypeHandler() && newType->GetTypeHandler()->GetPropertyCount() == 0));

// This is what's going on here. The newType comes from the (potentially) new script context, but the object is

// described by the old type handler, so we want to keep that type handler. We set the new type on the object, but

// then re-set the type handler of that type back to the old type handler. In the process, we may actually change

// the type of the object again (if the new type was locked) via DuplicateType; the newer type will then also be

// from the new script context.

DynamicType \* oldType = this->GetDynamicType();

DynamicTypeHandler\* oldTypeHandler = oldType->GetTypeHandler();

// Consider: Because we've disabled fixed properties on DOM objects, we don't need to rely on a type change here to

// invalidate fixed properties. Under some circumstances (with F12 tools enabled) an object which

// is already in the new context can be reset and newType == oldType. If we reeanable fixed properties on DOM objects

// we'll have to investigate and address this issue.

// Assert(newType != oldType);

// We only expect DOM objects to ever be reset and we explicitly disable fixed properties on DOM objects.

Assert(!oldTypeHandler->HasAnyFixedProperties());

this->type = newType;

if (!IsAnyArray(this))

{

this->objectArray = nullptr;

}

oldTypeHandler->ResetTypeHandler(this);

Assert(this->GetScriptContext() == newType->GetScriptContext());

if (this->GetTypeHandler()->IsDeferredTypeHandler())

{

return;

}

if (!keepProperties)

{

this->GetTypeHandler()->SetAllPropertiesToUndefined(this, false);

}

// Marshalling cannot handle non-Var values, so extract

// the two internal property values that could appear on a CEO, clear them to null which

// marshalling does handle, and then restore them after marshalling. Neither property's

// data needs marshalling because:

// 1. StackTrace's data does not contain references to JavaScript objects that would need marshalling.

// 2. Values in the WeakMapKeyMap can only be accessed by the WeakMap object that put them there. If

// that WeakMap is marshalled it will take care of any necessary marshalling of the value by virtue

// of being wrapped in CrossSite<>.

Var stackTraceValue = nullptr;

if (this->GetInternalProperty(this, InternalPropertyIds::StackTrace, &stackTraceValue, nullptr, nullptr))

{

this->SetInternalProperty(InternalPropertyIds::StackTrace, nullptr, PropertyOperation\_None, nullptr);

}

Var weakMapKeyMapValue = nullptr;

if (this->GetInternalProperty(this, InternalPropertyIds::WeakMapKeyMap, &weakMapKeyMapValue, nullptr, nullptr))

{

this->SetInternalProperty(InternalPropertyIds::WeakMapKeyMap, nullptr, PropertyOperation\_Force, nullptr);

}

Var mutationBpValue = nullptr;

if (this->GetInternalProperty(this, InternalPropertyIds::MutationBp, &mutationBpValue, nullptr, nullptr))

{

this->SetInternalProperty(InternalPropertyIds::MutationBp, nullptr, PropertyOperation\_Force, nullptr);

}

if (keepProperties)

{

this->GetTypeHandler()->MarshalAllPropertiesToScriptContext(this, this->GetScriptContext(), false);

if (stackTraceValue)

{

this->SetInternalProperty(InternalPropertyIds::StackTrace, stackTraceValue, PropertyOperation\_None, nullptr);

}

if (weakMapKeyMapValue)

{

this->SetInternalProperty(InternalPropertyIds::WeakMapKeyMap, weakMapKeyMapValue, PropertyOperation\_Force, nullptr);

}

if (mutationBpValue)

{

this->SetInternalProperty(InternalPropertyIds::MutationBp, mutationBpValue, PropertyOperation\_Force, nullptr);

}

}

}

bool

DynamicObject::GetHasNoEnumerableProperties()

{

if (!this->GetTypeHandler()->EnsureObjectReady(this))

{

return false;

}

if (!this->GetDynamicType()->GetHasNoEnumerableProperties())

{

return false;

}

if (HasObjectArray() || (JavascriptArray::Is(this) && JavascriptArray::FromVar(this)->GetLength() != 0))

{

return false;

}

return true;

}

bool

DynamicObject::SetHasNoEnumerableProperties(bool value)

{

return this->GetDynamicType()->SetHasNoEnumerableProperties(value);

}

BigPropertyIndex

DynamicObject::GetPropertyIndexFromInlineSlotIndex(uint inlineSlotIndex)

{

return this->GetTypeHandler()->GetPropertyIndexFromInlineSlotIndex(inlineSlotIndex);

}

BigPropertyIndex

DynamicObject::GetPropertyIndexFromAuxSlotIndex(uint auxIndex)

{

return this->GetTypeHandler()->GetPropertyIndexFromAuxSlotIndex(auxIndex);

}

BOOL

DynamicObject::GetAttributesWithPropertyIndex(PropertyId propertyId, BigPropertyIndex index, PropertyAttributes \* attributes)

{

return this->GetTypeHandler()->GetAttributesWithPropertyIndex(this, propertyId, index, attributes);

}

RecyclerWeakReference<DynamicObject>\* DynamicObject::CreateWeakReferenceToSelf()

{

Assert(!ThreadContext::IsOnStack(this));

return GetRecycler()->CreateWeakReferenceHandle(this);

}

DynamicObject \*

DynamicObject::BoxStackInstance(DynamicObject \* instance)

{

Assert(ThreadContext::IsOnStack(instance));

// On the stack, the we reserved a pointer before the object as to store the boxed value

DynamicObject \*\* boxedInstanceRef = ((DynamicObject \*\*)instance) - 1;

DynamicObject \* boxedInstance = \*boxedInstanceRef;

if (boxedInstance)

{

return boxedInstance;

}

size\_t inlineSlotsSize = instance->GetTypeHandler()->GetInlineSlotsSize();

if (inlineSlotsSize)

{

boxedInstance = RecyclerNewPlusZ(instance->GetRecycler(), inlineSlotsSize, DynamicObject, instance);

}

else

{

boxedInstance = RecyclerNew(instance->GetRecycler(), DynamicObject, instance);

}

\*boxedInstanceRef = boxedInstance;

return boxedInstance;

}

#ifdef RECYCLER\_STRESS

void DynamicObject::Finalize(bool isShutdown)

{

// If -RecyclerTrackStress is enabled, DynamicObject will be allocated as Track (and thus Finalize too).

// Just ignore this.

if (Js::Configuration::Global.flags.RecyclerTrackStress)

{

return;

}

RecyclableObject::Finalize(isShutdown);

}

void DynamicObject::Dispose(bool isShutdown)

{

// If -RecyclerTrackStress is enabled, DynamicObject will be allocated as Track (and thus Finalize too).

// Just ignore this.

if (Js::Configuration::Global.flags.RecyclerTrackStress)

{

return;

}

RecyclableObject::Dispose(isShutdown);

}

void DynamicObject::Mark(Recycler \*recycler)

{

// If -RecyclerTrackStress is enabled, DynamicObject will be allocated as Track (and thus Finalize too).

// Process the mark now.

if (Js::Configuration::Global.flags.RecyclerTrackStress)

{

size\_t inlineSlotsSize = this->GetDynamicType()->GetTypeHandler()->GetInlineSlotsSize();

size\_t objectSize = sizeof(DynamicObject) + inlineSlotsSize;

void \*\* obj = (void \*\*)this;

void \*\* objEnd = obj + (objectSize / sizeof(void \*));

do

{

recycler->TryMarkNonInterior(\*obj, nullptr);

obj++;

} while (obj != objEnd);

return;

}

RecyclableObject::Mark(recycler);

}

#endif

} // namespace Js

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

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

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

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

#pragma once

class ScriptSite;

namespace Js

{

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

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

friend class Js::CrossSiteObject<T>; \

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

{ \

Assert(this->GetScriptContext() != scriptContext); \

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

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

}

#else

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

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

#endif

#if DBG

#define SetSlotArguments(propertyId, slotIndex, value) propertyId, false, slotIndex, value

#define SetSlotArgumentsRoot(propertyId, allowLetConst, slotIndex, value) propertyId, allowLetConst, slotIndex, value

#else

#define SetSlotArguments(propertyId, slotIndex, value) slotIndex, value

#define SetSlotArgumentsRoot(propertyId, allowLetConst, slotIndex, value) slotIndex, value

#endif

enum class DynamicObjectFlags : uint16

{

None = 0u,

ObjectArrayFlagsTag = 1u << 0, // Tag bit used to indicate the objectArrayOrFlags field is used as flags as opposed to object array pointer.

HasSegmentMap = 1u << 1,

HasNoMissingValues = 1u << 2, // The head segment of a JavascriptArray has no missing values.

InitialArrayValue = ObjectArrayFlagsTag | HasNoMissingValues,

AllArrayFlags = HasNoMissingValues | HasSegmentMap,

AllFlags = ObjectArrayFlagsTag | HasNoMissingValues | HasSegmentMap

};

ENUM\_CLASS\_HELPERS(DynamicObjectFlags, uint16);

class DynamicObject : public RecyclableObject

{

friend class CrossSite;

friend class DynamicTypeHandler;

template <typename T> friend class DynamicObjectEnumeratorBase;

template <typename T, bool enumNonEnumerable, bool enumSymbols, bool snapShotSementics> friend class DynamicObjectEnumerator;

friend class RecyclableObject;

friend struct InlineCache;

friend class ForInObjectEnumerator; // for cache enumerator

friend class JavascriptOperators; // for ReplaceType

friend class PathTypeHandlerBase; // for ReplaceType

friend class JavascriptLibrary; // for ReplaceType

friend class ScriptFunction; // for ReplaceType;

friend class JSON::JSONParser; //for ReplaceType

private:

Var\* auxSlots;

// The objectArrayOrFlags field can store one of two things:

// a) a pointer to the object array holding numeric properties of this object, or

// b) a bitfield of flags.

// Because object arrays are not commonly used, the storage space can be reused to carry information that

// can improve performance for typical objects. To indicate the bitfield usage we set the least significant bit to 1.

// Object array pointer always trumps the flags, such that when the first numeric property is added to an

// object, its flags will be wiped out. Hence flags can only be used as a form of cache to improve performance.

// For functional correctness, some other fallback mechanism must exist to convey the information contained in flags.

// This fields always starts off initialized to null. Currently, only JavascriptArray overrides it to store flags, the

// bits it uses are DynamicObjectFlags::AllArrayFlags.

union

{

ArrayObject \* objectArray; // Only if !IsAnyArray

struct // Only if IsAnyArray

{

DynamicObjectFlags arrayFlags;

ProfileId arrayCallSiteIndex;

};

};

CompileAssert(sizeof(ProfileId) == 2);

CompileAssert(static\_cast<intptr\_t>(DynamicObjectFlags::ObjectArrayFlagsTag) != 0);

void InitSlots(DynamicObject \* instance, ScriptContext \* scriptContext);

void SetTypeHandler(DynamicTypeHandler \* typeHandler, bool hasChanged);

void ReplaceType(DynamicType \* type);

protected:

DEFINE\_VTABLE\_CTOR(DynamicObject, RecyclableObject);

DEFINE\_MARSHAL\_OBJECT\_TO\_SCRIPT\_CONTEXT(DynamicObject);

DynamicObject(DynamicType \* type, const bool initSlots = true);

DynamicObject(DynamicType \* type, ScriptContext \* scriptContext);

// For boxing stack instance

DynamicObject(DynamicObject \* instance);

DynamicTypeHandler \* GetTypeHandler() const;

uint16 GetOffsetOfInlineSlots() const;

template <class T>

static T\* NewObject(Recycler \* recycler, DynamicType \* type);

public:

static DynamicObject \* New(Recycler \* recycler, DynamicType \* type);

static bool Is(Var aValue);

static DynamicObject\* FromVar(Var value);

void EnsureSlots(int oldCount, int newCount, ScriptContext \* scriptContext, DynamicTypeHandler \* newTypeHandler = nullptr);

void EnsureSlots(int newCount, ScriptContext \*scriptContext);

Var GetSlot(int index);

Var GetInlineSlot(int index);

Var GetAuxSlot(int index);

#if DBG

void SetSlot(PropertyId propertyId, bool allowLetConst, int index, Var value);

void SetInlineSlot(PropertyId propertyId, bool allowLetConst, int index, Var value);

void SetAuxSlot(PropertyId propertyId, bool allowLetConst, int index, Var value);

#else

void SetSlot(int index, Var value);

void SetInlineSlot(int index, Var value);

void SetAuxSlot(int index, Var value);

#endif

private:

bool IsObjectHeaderInlinedTypeHandlerUnchecked() const;

public:

bool IsObjectHeaderInlinedTypeHandler() const;

bool DeoptimizeObjectHeaderInlining();

public:

bool HasNonEmptyObjectArray() const;

DynamicType \* GetDynamicType() const { return (DynamicType \*)this->GetType(); }

// Check if a typeId is of any array type (JavascriptArray or ES5Array).

static bool IsAnyArrayTypeId(TypeId typeId);

// Check if a Var is either a JavascriptArray\* or ES5Array\*.

static bool IsAnyArray(const Var aValue);

bool UsesObjectArrayOrFlagsAsFlags() const

{

return !!(arrayFlags & DynamicObjectFlags::ObjectArrayFlagsTag);

}

ArrayObject\* GetObjectArray() const

{

return HasObjectArray() ? GetObjectArrayOrFlagsAsArray() : nullptr;

}

bool HasObjectArray() const

{

// Only JavascriptArray uses the objectArrayOrFlags as flags.

Assert(DynamicObject::IsAnyArray((Var)this) || !UsesObjectArrayOrFlagsAsFlags() || IsObjectHeaderInlinedTypeHandler());

return ((objectArray != nullptr) && !UsesObjectArrayOrFlagsAsFlags() && !IsObjectHeaderInlinedTypeHandler());

}

ArrayObject\* GetObjectArrayUnchecked() const

{

return HasObjectArrayUnchecked() ? GetObjectArrayOrFlagsAsArray() : nullptr;

}

bool HasObjectArrayUnchecked() const

{

return ((objectArray != nullptr) && !UsesObjectArrayOrFlagsAsFlags() && !IsObjectHeaderInlinedTypeHandlerUnchecked());

}

BOOL HasObjectArrayItem(uint32 index);

BOOL DeleteObjectArrayItem(uint32 index, PropertyOperationFlags flags);

BOOL GetObjectArrayItem(Var originalInstance, uint32 index, Var\* value, ScriptContext\* requestContext);

DescriptorFlags GetObjectArrayItemSetter(uint32 index, Var\* setterValue, ScriptContext\* requestContext);

BOOL SetObjectArrayItem(uint32 index, Var value, PropertyOperationFlags flags);

BOOL SetObjectArrayItemWithAttributes(uint32 index, Var value, PropertyAttributes attributes);

BOOL SetObjectArrayItemAttributes(uint32 index, PropertyAttributes attributes);

BOOL SetObjectArrayItemWritable(PropertyId propertyId, BOOL writable);

BOOL SetObjectArrayItemAccessors(uint32 index, Var getter, Var setter);

void InvalidateHasOnlyWritableDataPropertiesInPrototypeChainCacheIfPrototype();

void ResetObject(DynamicType\* type, BOOL keepProperties);

virtual void SetIsPrototype();

bool HasLockedType() const;

bool HasSharedType() const;

bool HasSharedTypeHandler() const;

bool LockType();

bool ShareType();

bool GetIsExtensible() const;

bool GetHasNoEnumerableProperties();

bool SetHasNoEnumerableProperties(bool value);

virtual bool HasReadOnlyPropertiesInvisibleToTypeHandler() { return false; }

void InitSlots(DynamicObject\* instance);

virtual int GetPropertyCount() override;

virtual PropertyId GetPropertyId(PropertyIndex index) override;

virtual PropertyId GetPropertyId(BigPropertyIndex index) override;

PropertyIndex GetPropertyIndex(PropertyId propertyId) sealed;

virtual BOOL HasProperty(PropertyId propertyId) override;

virtual BOOL HasOwnProperty(PropertyId propertyId) override;

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

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

virtual BOOL GetInternalProperty(Var instance, PropertyId internalPropertyId, Var\* value, PropertyValueInfo\* info, ScriptContext\* requestContext) override;

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

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

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

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

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

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

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

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

virtual BOOL DeleteProperty(PropertyId propertyId, PropertyOperationFlags flags) override;

virtual BOOL IsFixedProperty(PropertyId propertyId) override;

virtual BOOL HasItem(uint32 index) override;

virtual BOOL HasOwnItem(uint32 index) override;

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

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

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

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

virtual BOOL DeleteItem(uint32 index, PropertyOperationFlags flags) override;

virtual BOOL ToPrimitive(JavascriptHint hint, Var\* result, ScriptContext \* requestContext) override;

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

virtual BOOL SetAccessors(PropertyId propertyId, Var getter, Var setter, PropertyOperationFlags flags = PropertyOperation\_None) override;

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

virtual BOOL IsWritable(PropertyId propertyId) override;

virtual BOOL IsConfigurable(PropertyId propertyId) override;

virtual BOOL IsEnumerable(PropertyId propertyId) override;

virtual BOOL SetEnumerable(PropertyId propertyId, BOOL value) override;

virtual BOOL SetWritable(PropertyId propertyId, BOOL value) override;

virtual BOOL SetConfigurable(PropertyId propertyId, BOOL value) override;

virtual BOOL SetAttributes(PropertyId propertyId, PropertyAttributes attributes) override;

virtual BOOL IsExtensible() override { return GetIsExtensible(); };

virtual BOOL PreventExtensions() override;

virtual BOOL Seal() override;

virtual BOOL Freeze() override;

virtual BOOL IsSealed() override;

virtual BOOL IsFrozen() override;

virtual BOOL GetDiagValueString(StringBuilder<ArenaAllocator>\* stringBuilder, ScriptContext\* requestContext) override;

virtual BOOL GetDiagTypeString(StringBuilder<ArenaAllocator>\* stringBuilder, ScriptContext\* requestContext) override;

virtual Var GetTypeOfString(ScriptContext \* requestContext) override;

#if DBG

virtual bool CanStorePropertyValueDirectly(PropertyId propertyId, bool allowLetConst) override;

#endif

virtual void RemoveFromPrototype(ScriptContext \* requestContext) override;

virtual void AddToPrototype(ScriptContext \* requestContext) override;

virtual void SetPrototype(RecyclableObject\* newPrototype) override;

virtual BOOL IsCrossSiteObject() const { return FALSE; }

virtual DynamicType\* DuplicateType();

static bool IsTypeHandlerCompatibleForObjectHeaderInlining(DynamicTypeHandler \* oldTypeHandler, DynamicTypeHandler \* newTypeHandler);

void ChangeType();

void ChangeTypeIf(const Type\* oldType);

Var GetNextProperty(PropertyIndex& index, DynamicType \*typeToEnumerate, bool requireEnumerable, bool enumSymbols = false);

Var GetNextProperty(BigPropertyIndex& index, DynamicType \*typeToEnumerate, bool requireEnumerable, bool enumSymbols = false);

BOOL FindNextProperty(PropertyIndex& index, JavascriptString\*\* propertyString, PropertyId\* propertyId, PropertyAttributes\* attributes, DynamicType \*typeToEnumerate, bool requireEnumerable, bool enumSymbols = false) const;

BOOL FindNextProperty(BigPropertyIndex& index, JavascriptString\*\* propertyString, PropertyId\* propertyId, PropertyAttributes\* attributes, DynamicType \*typeToEnumerate, bool requireEnumerable, bool enumSymbols = false) const;

virtual BOOL HasDeferredTypeHandler() const sealed;

static DWORD GetOffsetOfAuxSlots();

static DWORD GetOffsetOfObjectArray();

static DWORD GetOffsetOfType();

Js::BigPropertyIndex GetPropertyIndexFromInlineSlotIndex(uint inlineSlotIndex);

Js::BigPropertyIndex GetPropertyIndexFromAuxSlotIndex(uint auxIndex);

BOOL GetAttributesWithPropertyIndex(PropertyId propertyId, BigPropertyIndex index, PropertyAttributes \* attributes);

RecyclerWeakReference<DynamicObject>\* CreateWeakReferenceToSelf();

void SetObjectArray(ArrayObject\* objectArray);

protected:

// These are only call for arrays

void InitArrayFlags(DynamicObjectFlags flags);

DynamicObjectFlags GetArrayFlags() const;

DynamicObjectFlags GetArrayFlags\_Unchecked() const; // do not use except in extreme circumstances

void SetArrayFlags(const DynamicObjectFlags flags);

ProfileId GetArrayCallSiteIndex() const;

void SetArrayCallSiteIndex(ProfileId profileId);

static DynamicObject \* BoxStackInstance(DynamicObject \* instance);

private:

ArrayObject\* EnsureObjectArray();

ArrayObject\* GetObjectArrayOrFlagsAsArray() const { return objectArray; }

template <PropertyId propertyId>

BOOL ToPrimitiveImpl(Var\* result, ScriptContext \* requestContext);

BOOL CallToPrimitiveFunction(Var toPrimitiveFunction, PropertyId propertyId, Var\* result, ScriptContext \* requestContext);

#if DBG

public:

virtual bool DbgIsDynamicObject() const override { return true; }

#endif

#ifdef RECYCLER\_STRESS

public:

virtual void Finalize(bool isShutdown) override;

virtual void Dispose(bool isShutdown) override;

virtual void Mark(Recycler \*recycler) override;

#endif

};

} // namespace Js

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

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

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

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

#pragma once

namespace Js

{

#ifdef RECYCLER\_STRESS

// Only enable RecyclerTrackStress on DynamicObject

template <class T> bool IsRecyclerTrackStressType() { return false; }

template <> inline bool IsRecyclerTrackStressType<DynamicObject>() { return true; }

#endif

template <class T>

\_\_inline T \* DynamicObject::NewObject(Recycler \* recycler, DynamicType \* type)

{

size\_t inlineSlotsSize = type->GetTypeHandler()->GetInlineSlotsSize();

if (inlineSlotsSize)

{

#ifdef RECYCLER\_STRESS

if (Js::Configuration::Global.flags.RecyclerTrackStress && IsRecyclerTrackStressType<T>())

{

return RecyclerNewTrackedLeafPlusZ(recycler, inlineSlotsSize, T, type);

}

#endif

return RecyclerNewPlusZ(recycler, inlineSlotsSize, T, type);

}

else

{

#ifdef RECYCLER\_STRESS

if (Js::Configuration::Global.flags.RecyclerTrackStress && IsRecyclerTrackStressType<T>())

{

return RecyclerNewTrackedLeaf(recycler, T, type);

}

#endif

return RecyclerNew(recycler, T, type);

}

}

}

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

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

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

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

#include "RuntimeTypePch.h"

namespace Js

{

template <typename T, bool enumNonEnumerable, bool enumSymbols, bool snapShotSementics>

JavascriptEnumerator\* DynamicObjectEnumerator<T, enumNonEnumerable, enumSymbols, snapShotSementics>::New(ScriptContext\* scriptContext, DynamicObject\* object)

{

DynamicObjectEnumerator\* enumerator = RecyclerNew(scriptContext->GetRecycler(), DynamicObjectEnumerator, scriptContext);

enumerator->Initialize(object);

return enumerator;

}

template <typename T, bool enumNonEnumerable, bool enumSymbols, bool snapShotSementics>

DynamicType \*DynamicObjectEnumerator<T, enumNonEnumerable, enumSymbols, snapShotSementics>::GetTypeToEnumerate() const

{

return

snapShotSementics &&

initialType->GetIsLocked() &&

CONFIG\_FLAG(TypeSnapshotEnumeration)

? initialType

: object->GetDynamicType();

}

template <typename T, bool enumNonEnumerable, bool enumSymbols, bool snapShotSementics>

Var DynamicObjectEnumerator<T, enumNonEnumerable, enumSymbols, snapShotSementics>::GetCurrentIndex()

{

if (arrayEnumerator)

{

return arrayEnumerator->GetCurrentIndex();

}

JavascriptString\* propertyString = nullptr;

PropertyId propertyId = Constants::NoProperty;

if (!object->FindNextProperty(objectIndex, &propertyString, &propertyId, nullptr, GetTypeToEnumerate(), !enumNonEnumerable, enumSymbols))

{

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

}

Assert(propertyId == Constants::NoProperty || !Js::IsInternalPropertyId(propertyId));

return propertyString;

}

template <typename T, bool enumNonEnumerable, bool enumSymbols, bool snapShotSementics>

Var DynamicObjectEnumerator<T, enumNonEnumerable, enumSymbols, snapShotSementics>::GetCurrentValue()

{

if (arrayEnumerator)

{

return arrayEnumerator->GetCurrentValue();

}

return object->GetNextProperty(objectIndex, GetTypeToEnumerate(), !enumNonEnumerable, enumSymbols);

}

template <typename T, bool enumNonEnumerable, bool enumSymbols, bool snapShotSementics>

BOOL DynamicObjectEnumerator<T, enumNonEnumerable, enumSymbols, snapShotSementics>::MoveNext(PropertyAttributes\* attributes)

{

PropertyId propId;

return GetCurrentAndMoveNext(propId, attributes) != NULL;

}

template <typename T, bool enumNonEnumerable, bool enumSymbols, bool snapShotSementics>

bool DynamicObjectEnumerator<T, enumNonEnumerable, enumSymbols, snapShotSementics>::GetCurrentPropertyId(PropertyId \*pPropertyId)

{

if (arrayEnumerator)

{

return arrayEnumerator->GetCurrentPropertyId(pPropertyId);

}

Js::PropertyId propertyId = object->GetPropertyId((T) objectIndex);

if ((enumNonEnumerable || (propertyId != Constants::NoProperty && object->IsEnumerable(propertyId))))

{

\*pPropertyId = propertyId;

return true;

}

else

{

return false;

}

}

template <typename T, bool enumNonEnumerable, bool enumSymbols, bool snapShotSementics>

uint32 DynamicObjectEnumerator<T, enumNonEnumerable, enumSymbols, snapShotSementics>::GetCurrentItemIndex()

{

if (arrayEnumerator)

{

return arrayEnumerator->GetCurrentItemIndex();

}

else

{

return JavascriptArray::InvalidIndex;

}

}

template <typename T, bool enumNonEnumerable, bool enumSymbols, bool snapShotSementics>

void DynamicObjectEnumerator<T, enumNonEnumerable, enumSymbols, snapShotSementics>::Reset()

{

ResetHelper();

}

// Initialize (or reuse) this enumerator for a given object.

template <typename T, bool enumNonEnumerable, bool enumSymbols, bool snapShotSementics>

void DynamicObjectEnumerator<T, enumNonEnumerable, enumSymbols, snapShotSementics>::Initialize(DynamicObject\* object)

{

this->object = object;

ResetHelper();

}

template <typename T, bool enumNonEnumerable, bool enumSymbols, bool snapShotSementics>

Var DynamicObjectEnumerator<T, enumNonEnumerable, enumSymbols, snapShotSementics>::GetCurrentAndMoveNext(PropertyId& propertyId, PropertyAttributes\* attributes)

{

if (arrayEnumerator)

{

Var currentIndex = arrayEnumerator->GetCurrentAndMoveNext(propertyId, attributes);

if(currentIndex != NULL)

{

return currentIndex;

}

arrayEnumerator = NULL;

}

JavascriptString\* propertyString;

do

{

objectIndex++;

propertyString = nullptr;

if (!object->FindNextProperty(objectIndex, &propertyString, &propertyId, attributes, GetTypeToEnumerate(), !enumNonEnumerable, enumSymbols))

{

// No more properties

objectIndex--;

break;

}

}

while (Js::IsInternalPropertyId(propertyId));

return propertyString;

}

template <typename T, bool enumNonEnumerable, bool enumSymbols, bool snapShotSementics>

void DynamicObjectEnumerator<T, enumNonEnumerable, enumSymbols, snapShotSementics>::ResetHelper()

{

if (object->HasObjectArray())

{

// Pass "object" as originalInstance to objectArray enumerator

BOOL result = object->GetObjectArrayOrFlagsAsArray()->GetEnumerator(object, enumNonEnumerable, (Var\*)&arrayEnumerator, GetScriptContext(), snapShotSementics, enumSymbols);

Assert(result);

}

else

{

arrayEnumerator = nullptr;

}

initialType = object->GetDynamicType();

objectIndex = (T)-1; // This is Constants::NoSlot or Constants::NoBigSlot

}

template class DynamicObjectEnumerator<PropertyIndex, /\*enumNonEnumerable\*/true, /\*enumSymbols\*/true, /\*snapShotSementics\*/false>;

template class DynamicObjectEnumerator<BigPropertyIndex, /\*enumNonEnumerable\*/true, /\*enumSymbols\*/true, /\*snapShotSementics\*/false>;

template class DynamicObjectEnumerator<PropertyIndex, /\*enumNonEnumerable\*/false, /\*enumSymbols\*/true, /\*snapShotSementics\*/false>;

template class DynamicObjectEnumerator<BigPropertyIndex, /\*enumNonEnumerable\*/false, /\*enumSymbols\*/true, /\*snapShotSementics\*/false>;

template class DynamicObjectEnumerator<PropertyIndex, /\*enumNonEnumerable\*/false, /\*enumSymbols\*/true, /\*snapShotSementics\*/true>;

template class DynamicObjectEnumerator<BigPropertyIndex, /\*enumNonEnumerable\*/false, /\*enumSymbols\*/true, /\*snapShotSementics\*/true>;

template class DynamicObjectEnumerator<PropertyIndex, /\*enumNonEnumerable\*/true, /\*enumSymbols\*/false, /\*snapShotSementics\*/false>;

template class DynamicObjectEnumerator<BigPropertyIndex, /\*enumNonEnumerable\*/true, /\*enumSymbols\*/false, /\*snapShotSementics\*/false>;

template class DynamicObjectEnumerator<PropertyIndex, /\*enumNonEnumerable\*/false, /\*enumSymbols\*/false, /\*snapShotSementics\*/false>;

template class DynamicObjectEnumerator<BigPropertyIndex, /\*enumNonEnumerable\*/false, /\*enumSymbols\*/false, /\*snapShotSementics\*/false>;

template class DynamicObjectEnumerator<PropertyIndex, /\*enumNonEnumerable\*/false, /\*enumSymbols\*/false, /\*snapShotSementics\*/true>;

template class DynamicObjectEnumerator<BigPropertyIndex, /\*enumNonEnumerable\*/false, /\*enumSymbols\*/false, /\*snapShotSementics\*/true>;

template class DynamicObjectEnumerator<PropertyIndex, /\*enumNonEnumerable\*/true, /\*enumSymbols\*/false, /\*snapShotSementics\*/true>;

template class DynamicObjectEnumerator<BigPropertyIndex, /\*enumNonEnumerable\*/true, /\*enumSymbols\*/false, /\*snapShotSementics\*/true>;

template class DynamicObjectEnumerator<PropertyIndex, /\*enumNonEnumerable\*/true, /\*enumSymbols\*/true, /\*snapShotSementics\*/true>;

template class DynamicObjectEnumerator<BigPropertyIndex, /\*enumNonEnumerable\*/true, /\*enumSymbols\*/true, /\*snapShotSementics\*/true>;

}

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

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

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

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

#pragma once

namespace Js

{

template <typename T, bool enumNonEnumerable, bool enumSymbols, bool snapShotSementics>

class DynamicObjectEnumerator : public JavascriptEnumerator

{

protected:

DynamicObject\* object;

JavascriptEnumerator\* arrayEnumerator;

DynamicType \*initialType;

T objectIndex;

DynamicObjectEnumerator(ScriptContext\* scriptContext)

: JavascriptEnumerator(scriptContext)

{

}

DEFINE\_VTABLE\_CTOR(DynamicObjectEnumerator, JavascriptEnumerator);

DEFINE\_MARSHAL\_ENUMERATOR\_TO\_SCRIPT\_CONTEXT(DynamicObjectEnumerator);

void ResetHelper();

void Initialize(DynamicObject\* object);

DynamicObjectEnumerator() { /\* Do nothing, needed by the vtable ctor for ForInObjectEnumeratorWrapper \*/ }

public:

static JavascriptEnumerator\* New(ScriptContext\* scriptContext, DynamicObject\* object);

protected:

DynamicType \*GetTypeToEnumerate() const;

public:

virtual Var GetCurrentIndex() override;

virtual Var GetCurrentValue() override;

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

virtual void Reset() override;

virtual bool GetCurrentPropertyId(PropertyId \*propertyId) override;

virtual uint32 GetCurrentItemIndex() override;

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

static uint32 GetOffsetOfObject() { return offsetof(DynamicObjectEnumerator, object); }

static uint32 GetOffsetOfArrayEnumerator() { return offsetof(DynamicObjectEnumerator, arrayEnumerator); }

static uint32 GetOffsetOfObjectIndex() { return offsetof(DynamicObjectEnumerator, objectIndex); }

};

} // namespace Js

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

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

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

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

#include "RuntimeTypePch.h"

namespace Js

{

template <typename T, bool enumNonEnumerable, bool enumSymbols>

JavascriptEnumerator\* DynamicObjectSnapshotEnumerator<T, enumNonEnumerable, enumSymbols>::New(ScriptContext\* scriptContext, DynamicObject\* object)

{

DynamicObjectSnapshotEnumerator\* enumerator = RecyclerNew(scriptContext->GetRecycler(), DynamicObjectSnapshotEnumerator, scriptContext);

enumerator->Initialize(object);

return enumerator;

}

template <typename T, bool enumNonEnumerable, bool enumSymbols>

Var DynamicObjectSnapshotEnumerator<T, enumNonEnumerable, enumSymbols>::GetCurrentAndMoveNextFromArray(PropertyId& propertyId, PropertyAttributes\* attributes)

{

if (arrayEnumerator)

{

Var currentIndex = arrayEnumerator->GetCurrentAndMoveNext(propertyId, attributes);

if(currentIndex != nullptr)

{

return currentIndex;

}

arrayEnumerator = nullptr;

}

return nullptr;

}

template <typename T, bool enumNonEnumerable, bool enumSymbols>

JavascriptString \* DynamicObjectSnapshotEnumerator<T, enumNonEnumerable, enumSymbols>::GetCurrentAndMoveNextFromObject(T& index, PropertyId& propertyId, PropertyAttributes\* attributes)

{

JavascriptString\* propertyString = nullptr;

auto newIndex = objectIndex;

do

{

newIndex++;

if (!object->FindNextProperty(newIndex, &propertyString, &propertyId, attributes, GetTypeToEnumerate(), !enumNonEnumerable, /\*enumSymbols\*/enumSymbols) || newIndex >= initialPropertyCount)

{

newIndex--;

propertyString = nullptr;

break;

}

}

while (Js::IsInternalPropertyId(propertyId));

index = newIndex;

return propertyString;

}

template <typename T, bool enumNonEnumerable, bool enumSymbols>

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

{

Var currentIndex = GetCurrentAndMoveNextFromArray(propertyId, attributes);

return (currentIndex != nullptr)? currentIndex :

this->GetCurrentAndMoveNextFromObject(objectIndex, propertyId, attributes);

}

template <typename T, bool enumNonEnumerable, bool enumSymbols>

void DynamicObjectSnapshotEnumerator<T, enumNonEnumerable, enumSymbols>::Reset()

{

\_\_super::Reset();

initialPropertyCount = object->GetPropertyCount();

}

template <typename T, bool enumNonEnumerable, bool enumSymbols>

void DynamicObjectSnapshotEnumerator<T, enumNonEnumerable, enumSymbols>::Initialize(DynamicObject\* object)

{

\_\_super::Initialize(object);

initialPropertyCount = object->GetPropertyCount();

}

template class DynamicObjectSnapshotEnumerator<PropertyIndex, true, true>;

template class DynamicObjectSnapshotEnumerator<PropertyIndex, true, false>;

template class DynamicObjectSnapshotEnumerator<PropertyIndex, false, true>;

template class DynamicObjectSnapshotEnumerator<PropertyIndex, false, false>;

template class DynamicObjectSnapshotEnumerator<BigPropertyIndex, true, true>;

template class DynamicObjectSnapshotEnumerator<BigPropertyIndex, true, false>;

template class DynamicObjectSnapshotEnumerator<BigPropertyIndex, false, true>;

template class DynamicObjectSnapshotEnumerator<BigPropertyIndex, false, false>;

}

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

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

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

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

#pragma once

namespace Js

{

template <typename T, bool enumNonEnumerable, bool enumSymbols>

class DynamicObjectSnapshotEnumerator : public DynamicObjectEnumerator<T, enumNonEnumerable, enumSymbols, /\*snapShotSementics\*/true>

{

protected:

int initialPropertyCount;

DynamicObjectSnapshotEnumerator(ScriptContext\* scriptContext)

: DynamicObjectEnumerator(scriptContext)

{

}

DEFINE\_VTABLE\_CTOR(DynamicObjectSnapshotEnumerator, DynamicObjectEnumerator);

DEFINE\_MARSHAL\_ENUMERATOR\_TO\_SCRIPT\_CONTEXT(DynamicObjectSnapshotEnumerator);

Var GetCurrentAndMoveNextFromArray(PropertyId& propertyId, PropertyAttributes\* attributes);

JavascriptString \* GetCurrentAndMoveNextFromObject(T& index, PropertyId& propertyId, PropertyAttributes\* attributes);

DynamicObjectSnapshotEnumerator() { /\* Do nothing, needed by the vtable ctor for ForInObjectEnumeratorWrapper \*/ }

void Initialize(DynamicObject\* object);

public:

static JavascriptEnumerator\* New(ScriptContext\* scriptContext, DynamicObject\* object);

virtual void Reset() override;

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

};

} // namespace Js

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

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

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

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

#include "RuntimeTypePch.h"

namespace Js

{

template <typename T, bool enumNonEnumerable, bool enumSymbols>

JavascriptEnumerator\* DynamicObjectSnapshotEnumeratorWPCache<T, enumNonEnumerable, enumSymbols>::New(ScriptContext\* scriptContext, DynamicObject\* object)

{

DynamicObjectSnapshotEnumeratorWPCache\* enumerator = RecyclerNew(scriptContext->GetRecycler(), DynamicObjectSnapshotEnumeratorWPCache, scriptContext);

enumerator->Initialize(object, false);

return enumerator;

}

template <typename T, bool enumNonEnumerable, bool enumSymbols>

void DynamicObjectSnapshotEnumeratorWPCache<T, enumNonEnumerable, enumSymbols>::Initialize(DynamicObject\* object, bool allowUnlockedType/\*= false\*/)

{

\_\_super::Initialize(object);

enumeratedCount = 0;

if (!allowUnlockedType)

{

Assert(initialType->GetIsLocked());

}

else if (initialType->GetIsLocked())

{

VirtualTableInfo<DynamicObjectSnapshotEnumeratorWPCache>::SetVirtualTable(this); // Fix vtable which could have been downgraded

}

else

{

VirtualTableInfo<DynamicObjectSnapshotEnumerator>::SetVirtualTable(this); // Downgrade to normal snapshot enumerator

return;

}

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

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

CachedData \* data = (CachedData \*)threadContext->GetDynamicObjectEnumeratorCache(initialType);

if (data == nullptr || data->enumNonEnumerable != enumNonEnumerable || data->enumSymbols != enumSymbols)

{

data = RecyclerNewStructPlus(scriptContext->GetRecycler(),

initialPropertyCount \* sizeof(PropertyString \*) + initialPropertyCount \* sizeof(T) + initialPropertyCount \* sizeof(PropertyAttributes), CachedData);

data->cachedCount = 0;

data->strings = (PropertyString \*\*)(data + 1);

data->indexes = (T \*)(data->strings + initialPropertyCount);

data->attributes = (PropertyAttributes\*)(data->indexes + initialPropertyCount);

data->completed = false;

data->enumNonEnumerable = enumNonEnumerable;

data->enumSymbols = enumSymbols;

threadContext->AddDynamicObjectEnumeratorCache(initialType, data);

}

this->cachedData = data;

}

template <typename T, bool enumNonEnumerable, bool enumSymbols>

JavascriptString \*

DynamicObjectSnapshotEnumeratorWPCache<T, enumNonEnumerable, enumSymbols>::GetCurrentAndMoveNextFromObjectWPCache(T& index, PropertyId& propertyId, PropertyAttributes\* attributes)

{

if (initialType != object->GetDynamicType())

{

if (this->IsCrossSiteEnumerator())

{

// downgrade back to the normal snapshot enumerator

VirtualTableInfo<CrossSiteEnumerator<DynamicObjectSnapshotEnumerator>>::SetVirtualTable(this);

}

else

{

// downgrade back to the normal snapshot enumerator

VirtualTableInfo<DynamicObjectSnapshotEnumerator>::SetVirtualTable(this);

}

return this->GetCurrentAndMoveNextFromObject(objectIndex, propertyId, attributes);

}

Assert(enumeratedCount <= cachedData->cachedCount);

JavascriptString\* propertyStringName;

PropertyAttributes propertyAttributes = PropertyNone;

if (enumeratedCount < cachedData->cachedCount)

{

PropertyString \* propertyString = cachedData->strings[enumeratedCount];

propertyStringName = propertyString;

propertyId = propertyString->GetPropertyRecord()->GetPropertyId();

#if DBG

PropertyId tempPropertyId;

/\* JavascriptString \* tempPropertyString = \*/ this->GetCurrentAndMoveNextFromObject(objectIndex, tempPropertyId, attributes);

Assert(tempPropertyId == propertyId);

Assert(objectIndex == cachedData->indexes[enumeratedCount]);

#endif

objectIndex = cachedData->indexes[enumeratedCount];

propertyAttributes = cachedData->attributes[enumeratedCount];

enumeratedCount++;

}

else if (!cachedData->completed)

{

propertyStringName = this->GetCurrentAndMoveNextFromObject(objectIndex, propertyId, &propertyAttributes);

if (propertyStringName && VirtualTableInfo<PropertyString>::HasVirtualTable(propertyStringName))

{

Assert(enumeratedCount < initialPropertyCount);

cachedData->strings[enumeratedCount] = (PropertyString\*)propertyStringName;

cachedData->indexes[enumeratedCount] = objectIndex;

cachedData->attributes[enumeratedCount] = propertyAttributes;

cachedData->cachedCount = ++enumeratedCount;

}

else

{

cachedData->completed = true;

}

}

else

{

#if DBG

PropertyId tempPropertyId;

Assert(this->GetCurrentAndMoveNextFromObject(objectIndex, tempPropertyId, attributes) == nullptr);

#endif

propertyStringName = nullptr;

}

if (attributes != nullptr)

{

\*attributes = propertyAttributes;

}

return propertyStringName;

}

template <typename T, bool enumNonEnumerable, bool enumSymbols>

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

{

Var currentIndex = GetCurrentAndMoveNextFromArray(propertyId, attributes);

if (currentIndex == nullptr)

{

currentIndex = this->GetCurrentAndMoveNextFromObjectWPCache(objectIndex, propertyId, attributes);

}

return currentIndex;

}

template <typename T, bool enumNonEnumerable, bool enumSymbols>

void DynamicObjectSnapshotEnumeratorWPCache<T, enumNonEnumerable, enumSymbols>::Reset()

{

// If we are reusing the enumerator the object type should be the same

Assert(object->GetDynamicType() == initialType);

Assert(initialPropertyCount == object->GetPropertyCount());

\_\_super::Reset();

this->enumeratedCount = 0;

}

template class DynamicObjectSnapshotEnumeratorWPCache<PropertyIndex, true, true>;

template class DynamicObjectSnapshotEnumeratorWPCache<PropertyIndex, true, false>;

template class DynamicObjectSnapshotEnumeratorWPCache<PropertyIndex, false, true>;

template class DynamicObjectSnapshotEnumeratorWPCache<PropertyIndex, false, false>;

template class DynamicObjectSnapshotEnumeratorWPCache<BigPropertyIndex, true, true>;

template class DynamicObjectSnapshotEnumeratorWPCache<BigPropertyIndex, true, false>;

template class DynamicObjectSnapshotEnumeratorWPCache<BigPropertyIndex, false, true>;

template class DynamicObjectSnapshotEnumeratorWPCache<BigPropertyIndex, false, false>;

};

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

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

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

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

#pragma once

namespace Js

{

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* DynamicObjectSnapshotEnumeratorWPCache

\* This variant of the snapshot enumerator is only used by shared types.

\* Shared type's enumerator order doesn't change, so we can cache the enumeration

\* order in a map on the thread context and reuse the same order the next time

\* the same type is enumerated, thus speeding up enumeration by eliminating

\* virtual calls, internal property checks, and property string lookup

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

template <typename T, bool enumNonEnumerable, bool enumSymbols>

class DynamicObjectSnapshotEnumeratorWPCache : public DynamicObjectSnapshotEnumerator<T, enumNonEnumerable, enumSymbols>

{

protected:

DEFINE\_VTABLE\_CTOR(DynamicObjectSnapshotEnumeratorWPCache, DynamicObjectSnapshotEnumerator);

DEFINE\_MARSHAL\_ENUMERATOR\_TO\_SCRIPT\_CONTEXT(DynamicObjectSnapshotEnumeratorWPCache);

private:

DynamicObjectSnapshotEnumeratorWPCache(ScriptContext\* scriptContext)

: DynamicObjectSnapshotEnumerator(scriptContext)

{

}

JavascriptString \* GetCurrentAndMoveNextFromObjectWPCache(T& index, PropertyId& propertyId, PropertyAttributes\* attributes);

struct CachedData

{

PropertyString \*\* strings;

T \* indexes;

PropertyAttributes \* attributes;

int cachedCount;

bool completed;

bool enumNonEnumerable;

bool enumSymbols;

} \* cachedData;

int enumeratedCount; // Use int type to make fast path cmp easier. Note this works for both small and big PropertyIndex.

friend class ForInObjectEnumerator;

DynamicObjectSnapshotEnumeratorWPCache() { /\* Do nothing, needed by the vtable ctor for ForInObjectEnumeratorWrapper \*/ }

void Initialize(DynamicObject\* object, bool allowUnlockedType = false);

public:

static JavascriptEnumerator\* New(ScriptContext\* scriptContext, DynamicObject\* object);

virtual void Reset() override;

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

static uint32 GetOffsetOfInitialType() { return offsetof(DynamicObjectSnapshotEnumeratorWPCache, initialType); }

static uint32 GetOffsetOfEnumeratedCount() { return offsetof(DynamicObjectSnapshotEnumeratorWPCache, enumeratedCount); }

static uint32 GetOffsetOfCachedData() { return offsetof(DynamicObjectSnapshotEnumeratorWPCache, cachedData); }

static uint32 GetOffsetOfCachedDataStrings() { return offsetof(CachedData, strings); }

static uint32 GetOffsetOfCachedDataIndexes() { return offsetof(CachedData, indexes); }

static uint32 GetOffsetOfCachedDataCachedCount() { return offsetof(CachedData, cachedCount); }

static uint32 GetOffsetOfCachedDataPropertyAttributes() { return offsetof(CachedData, attributes); }

};

}

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

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

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

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

#include "RuntimeTypePch.h"

namespace Js

{

DEFINE\_RECYCLER\_TRACKER\_PERF\_COUNTER(DynamicType);

DEFINE\_RECYCLER\_TRACKER\_WEAKREF\_PERF\_COUNTER(DynamicType);

DynamicType::DynamicType(DynamicType \* type, DynamicTypeHandler \*typeHandler, bool isLocked, bool isShared)

: Type(type), typeHandler(typeHandler), isLocked(isLocked), isShared(isShared)

{

Assert(!this->isLocked || this->typeHandler->GetIsLocked());

Assert(!this->isShared || this->typeHandler->GetIsShared());

}

DynamicType::DynamicType(ScriptContext\* scriptContext, TypeId typeId, RecyclableObject\* prototype, JavascriptMethod entryPoint, DynamicTypeHandler \* typeHandler, bool isLocked, bool isShared)

: Type(scriptContext, typeId, prototype, entryPoint) , typeHandler(typeHandler), isLocked(isLocked), isShared(isShared), hasNoEnumerableProperties(false)

{

Assert(typeHandler != nullptr);

Assert(!this->isLocked || this->typeHandler->GetIsLocked());

Assert(!this->isShared || this->typeHandler->GetIsShared());

}

DynamicType \*

DynamicType::New(ScriptContext\* scriptContext, TypeId typeId, RecyclableObject\* prototype, JavascriptMethod entryPoint, DynamicTypeHandler \* typeHandler, bool isLocked, bool isShared)

{

return RecyclerNew(scriptContext->GetRecycler(), DynamicType, scriptContext, typeId, prototype, entryPoint, typeHandler, isLocked, isShared);

}

bool

DynamicType::Is(TypeId typeId)

{

return !StaticType::Is(typeId);

}

bool

DynamicType::SetHasNoEnumerableProperties(bool value)

{

if (!value)

{

this->hasNoEnumerableProperties = value;

return false;

}

#if DEBUG

PropertyIndex propertyIndex = (PropertyIndex)-1;

JavascriptString\* propertyString = nullptr;

PropertyId propertyId = Constants::NoProperty;

Assert(!this->GetTypeHandler()->FindNextProperty(this->GetScriptContext(), propertyIndex, &propertyString, &propertyId, nullptr, this, this, true));

#endif

this->hasNoEnumerableProperties = true;

return true;

}

void DynamicType::PrepareForTypeSnapshotEnumeration()

{

if(!GetIsLocked() && CONFIG\_FLAG(TypeSnapshotEnumeration))

{

// Lock the type and handler, enabling us to enumerate properties of the type snapshotted

// at the beginning of enumeration, despite property changes made by script during enumeration.

LockType(); // Note: this only works for type handlers that support locking.

}

}

void DynamicObject::InitSlots(DynamicObject\* instance)

{

InitSlots(instance, GetScriptContext());

}

void DynamicObject::InitSlots(DynamicObject \* instance, ScriptContext \* scriptContext)

{

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

int slotCapacity = GetTypeHandler()->GetSlotCapacity();

int inlineSlotCapacity = GetTypeHandler()->GetInlineSlotCapacity();

if (slotCapacity > inlineSlotCapacity)

{

instance->auxSlots = RecyclerNewArrayZ(recycler, Var, slotCapacity - inlineSlotCapacity);

}

}

int DynamicObject::GetPropertyCount()

{

if (!this->GetTypeHandler()->EnsureObjectReady(this))

{

return 0;

}

return GetTypeHandler()->GetPropertyCount();

}

PropertyId DynamicObject::GetPropertyId(PropertyIndex index)

{

return GetTypeHandler()->GetPropertyId(this->GetScriptContext(), index);

}

PropertyId DynamicObject::GetPropertyId(BigPropertyIndex index)

{

return GetTypeHandler()->GetPropertyId(this->GetScriptContext(), index);

}

PropertyIndex DynamicObject::GetPropertyIndex(PropertyId propertyId)

{

Assert(!Js::IsInternalPropertyId(propertyId));

Assert(propertyId != Constants::NoProperty);

return GetTypeHandler()->GetPropertyIndex(this->GetScriptContext()->GetPropertyName(propertyId));

}

BOOL DynamicObject::HasProperty(PropertyId propertyId)

{

// HasProperty can be invoked with propertyId = NoProperty in some cases, namely cross-thread and DOM

// This is done to force creation of a type handler in case the type handler is deferred

Assert(!Js::IsInternalPropertyId(propertyId) || propertyId == Js::Constants::NoProperty);

return GetTypeHandler()->HasProperty(this, propertyId);

}

BOOL DynamicObject::HasOwnProperty(PropertyId propertyId)

{

Assert(!Js::IsInternalPropertyId(propertyId));

return HasProperty(propertyId);

}

BOOL DynamicObject::GetProperty(Var originalInstance, PropertyId propertyId, Var\* value, PropertyValueInfo\* info, ScriptContext\* requestContext)

{

Assert(!Js::IsInternalPropertyId(propertyId));

return GetTypeHandler()->GetProperty(this, originalInstance, propertyId, value, info, requestContext);

}

BOOL DynamicObject::GetProperty(Var originalInstance, JavascriptString\* propertyNameString, Var\* value, PropertyValueInfo\* info, ScriptContext\* requestContext)

{

AssertMsg(!PropertyRecord::IsPropertyNameNumeric(propertyNameString->GetString(), propertyNameString->GetLength()),

"Numeric property names should have been converted to uint or PropertyRecord\* before calling GetProperty");

return GetTypeHandler()->GetProperty(this, originalInstance, propertyNameString, value, info, requestContext);

}

BOOL DynamicObject::GetInternalProperty(Var originalInstance, PropertyId propertyId, Var\* value, PropertyValueInfo\* info, ScriptContext\* requestContext)

{

Assert(Js::IsInternalPropertyId(propertyId));

return GetTypeHandler()->GetProperty(this, originalInstance, propertyId, value, nullptr, nullptr);

}

BOOL DynamicObject::GetPropertyReference(Var originalInstance, PropertyId propertyId, Var\* value, PropertyValueInfo\* info, ScriptContext\* requestContext)

{

Assert(!Js::IsInternalPropertyId(propertyId));

return GetTypeHandler()->GetProperty(this, originalInstance, propertyId, value, info, requestContext);

}

BOOL DynamicObject::SetProperty(PropertyId propertyId, Var value, PropertyOperationFlags flags, PropertyValueInfo\* info)

{

Assert(!Js::IsInternalPropertyId(propertyId));

return GetTypeHandler()->SetProperty(this, propertyId, value, flags, info);

}

BOOL DynamicObject::SetProperty(JavascriptString\* propertyNameString, Var value, PropertyOperationFlags flags, PropertyValueInfo\* info)

{

AssertMsg(!PropertyRecord::IsPropertyNameNumeric(propertyNameString->GetString(), propertyNameString->GetLength()),

"Numeric property names should have been converted to uint or PropertyRecord\* before calling SetProperty");

return GetTypeHandler()->SetProperty(this, propertyNameString, value, flags, info);

}

BOOL DynamicObject::SetInternalProperty(PropertyId propertyId, Var value, PropertyOperationFlags flags, PropertyValueInfo\* info)

{

Assert(Js::IsInternalPropertyId(propertyId));

return GetTypeHandler()->SetProperty(this, propertyId, value, flags, nullptr);

}

DescriptorFlags DynamicObject::GetSetter(PropertyId propertyId, Var\* setterValue, PropertyValueInfo\* info, ScriptContext\* requestContext)

{

Assert(!Js::IsInternalPropertyId(propertyId));

return GetTypeHandler()->GetSetter(this, propertyId, setterValue, info, requestContext);

}

DescriptorFlags DynamicObject::GetSetter(JavascriptString\* propertyNameString, Var\* setterValue, PropertyValueInfo\* info, ScriptContext\* requestContext)

{

AssertMsg(!PropertyRecord::IsPropertyNameNumeric(propertyNameString->GetString(), propertyNameString->GetLength()),

"Numeric property names should have been converted to uint or PropertyRecord\* before calling GetSetter");

return GetTypeHandler()->GetSetter(this, propertyNameString, setterValue, info, requestContext);

}

BOOL DynamicObject::InitProperty(PropertyId propertyId, Var value, PropertyOperationFlags flags, PropertyValueInfo\* info)

{

Assert(!Js::IsInternalPropertyId(propertyId));

return GetTypeHandler()->InitProperty(this, propertyId, value, flags, info);

}

BOOL DynamicObject::DeleteProperty(PropertyId propertyId, PropertyOperationFlags flags)

{

Assert(!Js::IsInternalPropertyId(propertyId));

return GetTypeHandler()->DeleteProperty(this, propertyId, flags);

}

BOOL DynamicObject::IsFixedProperty(PropertyId propertyId)

{

Assert(!Js::IsInternalPropertyId(propertyId));

return GetTypeHandler()->IsFixedProperty(this, propertyId);

}

BOOL DynamicObject::HasItem(uint32 index)

{

return GetTypeHandler()->HasItem(this, index);

}

BOOL DynamicObject::HasOwnItem(uint32 index)

{

return HasItem(index);

}

BOOL DynamicObject::GetItem(Var originalInstance, uint32 index, Var\* value, ScriptContext \* requestContext)

{

return GetTypeHandler()->GetItem(this, originalInstance, index, value, requestContext);

}

BOOL DynamicObject::GetItemReference(Var originalInstance, uint32 index, Var\* value, ScriptContext \* requestContext)

{

return GetTypeHandler()->GetItem(this, originalInstance, index, value, requestContext);

}

DescriptorFlags DynamicObject::GetItemSetter(uint32 index, Var\* setterValue, ScriptContext\* requestContext)

{

return GetTypeHandler()->GetItemSetter(this, index, setterValue, requestContext);

}

BOOL DynamicObject::SetItem(uint32 index, Var value, PropertyOperationFlags flags)

{

return GetTypeHandler()->SetItem(this, index, value, flags);

}

BOOL DynamicObject::DeleteItem(uint32 index, PropertyOperationFlags flags)

{

return GetTypeHandler()->DeleteItem(this, index, flags);

}

BOOL DynamicObject::ToPrimitive(JavascriptHint hint, Var\* result, ScriptContext \* requestContext)

{

if(hint == JavascriptHint::HintString)

{

return ToPrimitiveImpl<PropertyIds::toString>(result, requestContext)

|| ToPrimitiveImpl<PropertyIds::valueOf>(result, requestContext);

}

else

{

Assert(hint == JavascriptHint::None || hint == JavascriptHint::HintNumber);

return ToPrimitiveImpl<PropertyIds::valueOf>(result, requestContext)

|| ToPrimitiveImpl<PropertyIds::toString>(result, requestContext);

}

}

template <PropertyId propertyId>

BOOL DynamicObject::ToPrimitiveImpl(Var\* result, ScriptContext \* requestContext)

{

CompileAssert(propertyId == PropertyIds::valueOf || propertyId == PropertyIds::toString);

InlineCache \* inlineCache = propertyId == PropertyIds::valueOf ? requestContext->GetValueOfInlineCache() : requestContext->GetToStringInlineCache();

// Use per script context inline cache for valueOf and toString

Var aValue = JavascriptOperators::PatchGetValueUsingSpecifiedInlineCache(inlineCache, this, this, propertyId, requestContext);

// Fast path to the default valueOf/toString implementation

if (propertyId == PropertyIds::valueOf)

{

if (aValue == requestContext->GetLibrary()->GetObjectValueOfFunction())

{

Assert(JavascriptConversion::IsCallable(aValue));

// The default Object.prototype.valueOf will in turn just call ToObject().

// The result is always an object if it is not undefined or null (which "this" is not)

return false;

}

}

else

{

if (aValue == requestContext->GetLibrary()->GetObjectToStringFunction())

{

Assert(JavascriptConversion::IsCallable(aValue));

// These typeIds should never be here (they override ToPrimitive or they don't derive to DynamicObject::ToPrimitive)

// Otherwise, they may case implicit call in ToStringHelper

Assert(this->GetTypeId() != TypeIds\_HostDispatch

&& this->GetTypeId() != TypeIds\_HostObject);

\*result = JavascriptObject::ToStringHelper(this, requestContext);

return true;

}

}

return CallToPrimitiveFunction(aValue, propertyId, result, requestContext);

}

BOOL DynamicObject::CallToPrimitiveFunction(Var toPrimitiveFunction, PropertyId propertyId, Var\* result, ScriptContext \* requestContext)

{

if (JavascriptConversion::IsCallable(toPrimitiveFunction))

{

RecyclableObject\* toStringFunction = RecyclableObject::FromVar(toPrimitiveFunction);

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

Var aResult = threadContext->ExecuteImplicitCall(toStringFunction, ImplicitCall\_ToPrimitive, [=]() -> Js::Var

{

// Stack object should have a pre-op bail on implicit call. We shouldn't see them here.

Assert(!ThreadContext::IsOnStack(this) || threadContext->HasNoSideEffect(toStringFunction));

return toStringFunction->GetEntryPoint()(toStringFunction, CallInfo(CallFlags\_Value, 1), this);

});

if (!aResult)

{

// There was an implicit call and implicit calls are disabled. This would typically cause a bailout.

Assert(threadContext->IsDisableImplicitCall());

\*result = requestContext->GetLibrary()->GetNull();

return true;

}

if (JavascriptOperators::GetTypeId(aResult) <= TypeIds\_LastToPrimitiveType)

{

\*result = aResult;

return true;

}

}

return false;

}

BOOL DynamicObject::GetEnumerator(BOOL enumNonEnumerable, Var\* enumerator, ScriptContext \* requestContext, bool preferSnapshotSemantics, bool enumSymbols)

{

if (!this->GetTypeHandler()->EnsureObjectReady(this))

{

\*enumerator = nullptr;

return FALSE;

}

// Create the appropriate enumerator object.

if (preferSnapshotSemantics)

{

if (this->GetTypeHandler()->GetPropertyCount() == 0 && !this->HasObjectArray())

{

\*enumerator = requestContext->GetLibrary()->GetNullEnumerator();

}

else

{

GetDynamicType()->PrepareForTypeSnapshotEnumeration();

if (this->GetDynamicType()->GetIsLocked())

{

if (enumSymbols)

{

if (enumNonEnumerable)

{

\*enumerator = DynamicObjectSnapshotEnumeratorWPCache<BigPropertyIndex, /\*enumNonEnumerable\*/true, /\*enumSymbols\*/true>::New(requestContext, this);

}

else

{

\*enumerator = DynamicObjectSnapshotEnumeratorWPCache<BigPropertyIndex, /\*enumNonEnumerable\*/false, /\*enumSymbols\*/true>::New(requestContext, this);

}

}

else if (enumNonEnumerable)

{

\*enumerator = DynamicObjectSnapshotEnumeratorWPCache<BigPropertyIndex, /\*enumNonEnumerable\*/true, /\*enumSymbols\*/false>::New(requestContext, this);

}

else

{

\*enumerator = DynamicObjectSnapshotEnumeratorWPCache<BigPropertyIndex, /\*enumNonEnumerable\*/false, /\*enumSymbols\*/false>::New(requestContext, this);

}

}

else

{

if (enumSymbols)

{

if (enumNonEnumerable)

{

\*enumerator = DynamicObjectSnapshotEnumerator<BigPropertyIndex, /\*enumNonEnumerable\*/true, /\*enumSymbols\*/true>::New(requestContext, this);

}

else

{

\*enumerator = DynamicObjectSnapshotEnumerator<BigPropertyIndex, /\*enumNonEnumerable\*/false, /\*enumSymbols\*/true>::New(requestContext, this);

}

}

else if (enumNonEnumerable)

{

\*enumerator = DynamicObjectSnapshotEnumerator<BigPropertyIndex, /\*enumNonEnumerable\*/true, /\*enumSymbols\*/false>::New(requestContext, this);

}

else

{

\*enumerator = DynamicObjectSnapshotEnumerator<BigPropertyIndex, /\*enumNonEnumerable\*/false, /\*enumSymbols\*/false>::New(requestContext, this);

}

}

}

}

else if (enumSymbols)

{

if (enumNonEnumerable)

{

\*enumerator = DynamicObjectEnumerator<BigPropertyIndex, /\*enumNonEnumerable\*/true, /\*enumSymbols\*/true, /\*snapShotSementics\*/false>::New(requestContext, this);

}

else

{

\*enumerator = DynamicObjectEnumerator<BigPropertyIndex, /\*enumNonEnumerable\*/false, /\*enumSymbols\*/true, /\*snapShotSementics\*/false>::New(requestContext, this);

}

}

else if (enumNonEnumerable)

{

\*enumerator = DynamicObjectEnumerator<BigPropertyIndex, /\*enumNonEnumerable\*/true, /\*enumSymbols\*/false, /\*snapShotSementics\*/false>::New(requestContext, this);

}

else

{

\*enumerator = DynamicObjectEnumerator<BigPropertyIndex, /\*enumNonEnumerable\*/false, /\*enumSymbols\*/false, /\*snapShotSementics\*/false>::New(requestContext, this);

}

return true;

}

BOOL DynamicObject::SetAccessors(PropertyId propertyId, Var getter, Var setter, PropertyOperationFlags flags)

{

return GetTypeHandler()->SetAccessors(this, propertyId, getter, setter, flags);

}

BOOL DynamicObject::GetAccessors(PropertyId propertyId, Var \*getter, Var \*setter, ScriptContext \* requestContext)

{

return GetTypeHandler()->GetAccessors(this, propertyId, getter, setter);

}

BOOL DynamicObject::PreventExtensions()

{

return GetTypeHandler()->PreventExtensions(this);

}

BOOL DynamicObject::Seal()

{

return GetTypeHandler()->Seal(this);

}

BOOL DynamicObject::Freeze()

{

Type\* oldType = this->GetType();

BOOL ret = GetTypeHandler()->Freeze(this);

// We just made all properties on this object non-writable.

// Make sure the type is evolved so that the property string caches

// are no longer hit.

if (this->GetType() == oldType)

{

this->ChangeType();

}

return ret;

}

BOOL DynamicObject::IsSealed()

{

return GetTypeHandler()->IsSealed(this);

}

BOOL DynamicObject::IsFrozen()

{

return GetTypeHandler()->IsFrozen(this);

}

BOOL DynamicObject::IsWritable(PropertyId propertyId)

{

return GetTypeHandler()->IsWritable(this, propertyId);

}

BOOL DynamicObject::IsConfigurable(PropertyId propertyId)

{

return GetTypeHandler()->IsConfigurable(this, propertyId);

}

BOOL DynamicObject::IsEnumerable(PropertyId propertyId)

{

return GetTypeHandler()->IsEnumerable(this, propertyId);

}

BOOL DynamicObject::SetEnumerable(PropertyId propertyId, BOOL value)

{

return GetTypeHandler()->SetEnumerable(this, propertyId, value);

}

BOOL DynamicObject::SetWritable(PropertyId propertyId, BOOL value)

{

return GetTypeHandler()->SetWritable(this, propertyId, value);

}

BOOL DynamicObject::SetConfigurable(PropertyId propertyId, BOOL value)

{

return GetTypeHandler()->SetConfigurable(this, propertyId, value);

}

BOOL DynamicObject::SetAttributes(PropertyId propertyId, PropertyAttributes attributes)

{

return GetTypeHandler()->SetAttributes(this, propertyId, attributes);

}

BOOL DynamicObject::GetDiagValueString(StringBuilder<ArenaAllocator>\* stringBuilder, ScriptContext\* requestContext)

{

stringBuilder->AppendCppLiteral(L"{...}");

return TRUE;

}

BOOL DynamicObject::GetDiagTypeString(StringBuilder<ArenaAllocator>\* stringBuilder, ScriptContext\* requestContext)

{

stringBuilder->AppendCppLiteral(L"Object");

return TRUE;

}

Var DynamicObject::GetTypeOfString(ScriptContext \* requestContext)

{

return requestContext->GetLibrary()->GetObjectTypeDisplayString();

}

// If this object is not extensible and the property being set does not already exist,

// if throwIfNotExtensible is

// \* true, a type error will be thrown

// \* false, FALSE will be returned (unless strict mode is enabled, in which case a type error will be thrown).

// Either way, the property will not be set.

//

// throwIfNotExtensible should always be false for non-numeric properties.

BOOL DynamicObject::SetPropertyWithAttributes(PropertyId propertyId, Var value, PropertyAttributes attributes, PropertyValueInfo\* info, PropertyOperationFlags flags, SideEffects possibleSideEffects)

{

return GetTypeHandler()->SetPropertyWithAttributes(this, propertyId, value, attributes, info, flags, possibleSideEffects);

}

#if DBG

bool DynamicObject::CanStorePropertyValueDirectly(PropertyId propertyId, bool allowLetConst)

{

return GetTypeHandler()->CanStorePropertyValueDirectly(this, propertyId, allowLetConst);

}

#endif

void DynamicObject::RemoveFromPrototype(ScriptContext \* requestContext)

{

GetTypeHandler()->RemoveFromPrototype(this, requestContext);

}

void DynamicObject::AddToPrototype(ScriptContext \* requestContext)

{

GetTypeHandler()->AddToPrototype(this, requestContext);

}

void DynamicObject::SetPrototype(RecyclableObject\* newPrototype)

{

// Mark newPrototype it is being set as prototype

newPrototype->SetIsPrototype();

GetTypeHandler()->SetPrototype(this, newPrototype);

}

}

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

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

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

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

#pragma once

namespace Js

{

class DynamicType : public Type

{

#if DBG

friend class JavascriptFunction;

#endif

friend class DynamicObject;

friend class DynamicTypeHandler;

friend class CrossSite;

friend class TypePath;

friend class PathTypeHandlerBase;

friend class SimplePathTypeHandler;

friend class PathTypeHandler;

friend class ES5ArrayType;

friend class JavascriptOperators;

template <typename TPropertyIndex, typename TMapKey, bool IsNotExtensibleSupported>

friend class SimpleDictionaryTypeHandlerBase;

private:

DynamicTypeHandler \* typeHandler;

bool isLocked;

bool isShared;

bool hasNoEnumerableProperties;

protected:

DynamicType(DynamicType \* type) : Type(type), typeHandler(type->typeHandler), isLocked(false), isShared(false) {}

DynamicType(DynamicType \* type, DynamicTypeHandler \*typeHandler, bool isLocked, bool isShared);

DynamicType(ScriptContext\* scriptContext, TypeId typeId, RecyclableObject\* prototype, JavascriptMethod entryPoint, DynamicTypeHandler \* typeHandler, bool isLocked, bool isShared);

public:

DynamicTypeHandler \* GetTypeHandler() const { return typeHandler; }

void SetPrototype(RecyclableObject\* newPrototype) { this->prototype = newPrototype; }

bool GetIsLocked() const { return this->isLocked; }

bool GetIsShared() const { return this->isShared; }

void SetEntryPoint(JavascriptMethod method) { entryPoint = method; }

BOOL AllPropertiesAreEnumerable() { return typeHandler->AllPropertiesAreEnumerable(); }

bool LockType()

{

if (GetIsLocked())

{

Assert(this->GetTypeHandler()->IsLockable());

return true;

}

if (this->GetTypeHandler()->IsLockable())

{

this->GetTypeHandler()->LockTypeHandler();

this->isLocked = true;

return true;

}

return false;

}

bool ShareType()

{

if (this->GetIsShared())

{

Assert(this->GetTypeHandler()->IsSharable());

return true;

}

if (this->GetTypeHandler()->IsSharable())

{

LockType();

this->GetTypeHandler()->ShareTypeHandler(this->GetScriptContext());

this->isShared = true;

return true;

}

return false;

}

bool GetHasNoEnumerableProperties() const { return hasNoEnumerableProperties; }

bool SetHasNoEnumerableProperties(bool value);

void PrepareForTypeSnapshotEnumeration();

static bool Is(TypeId typeId);

static DynamicType \* New(ScriptContext\* scriptContext, TypeId typeId, RecyclableObject\* prototype, JavascriptMethod entryPoint, DynamicTypeHandler \* typeHandler, bool isLocked = false, bool isShared = false);

static uint32 GetOffsetOfTypeHandler() { return offsetof(DynamicType, typeHandler); }

static uint32 GetOffsetOfIsShared() { return offsetof(DynamicType, isShared); }

private:

void SetIsLocked() { Assert(this->GetTypeHandler()->GetIsLocked()); this->isLocked = true; }

void SetIsShared() { Assert(this->GetIsLocked() && this->GetTypeHandler()->GetIsShared()); this->isShared = true; }

void SetIsLockedAndShared() { SetIsLocked(); SetIsShared(); }

};

};

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

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

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

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

enum TypeId

{

TypeIds\_Undefined = 0,

TypeIds\_Null = 1,

TypeIds\_UndefinedOrNull = TypeIds\_Null,

TypeIds\_Boolean = 2,

// backend typeof() == "number" is true for typeIds

// between TypeIds\_FirstNumberType <= typeId <= TypeIds\_LastNumberType

TypeIds\_Integer = 3,

TypeIds\_FirstNumberType = TypeIds\_Integer,

TypeIds\_Number = 4,

TypeIds\_Int64Number = 5,

TypeIds\_UInt64Number = 6,

TypeIds\_LastNumberType = TypeIds\_UInt64Number,

TypeIds\_String = 7,

TypeIds\_Symbol = 8,

TypeIds\_LastToPrimitiveType = TypeIds\_Symbol,

TypeIds\_Enumerator = 9,

TypeIds\_VariantDate = 10,

// SIMD types

TypeIds\_SIMDFloat32x4 = 11,

TypeIds\_SIMDFloat64x2 = 12,

TypeIds\_SIMDInt32x4 = 13,

TypeIds\_SIMDInt8x16 = 14,

TypeIds\_LastJavascriptPrimitiveType = TypeIds\_SIMDInt8x16,

TypeIds\_HostDispatch = 15,

TypeIds\_WithScopeObject = 16,

TypeIds\_UndeclBlockVar = 17,

TypeIds\_LastStaticType = TypeIds\_UndeclBlockVar,

TypeIds\_Proxy = 18,

TypeIds\_Function = 19,

//

// The backend expects only objects whose typeof() === "object" to have a

// TypeId >= TypeIds\_Object. Only 'null' is a special case because it

// has a static type.

//

TypeIds\_Object = 20,

TypeIds\_Array = 21,

TypeIds\_ArrayFirst = TypeIds\_Array,

TypeIds\_NativeIntArray = 22,

#if ENABLE\_COPYONACCESS\_ARRAY

TypeIds\_CopyOnAccessNativeIntArray = 23,

#endif

TypeIds\_NativeFloatArray = 24,

TypeIds\_ArrayLast = TypeIds\_NativeFloatArray,

TypeIds\_Date = 25,

TypeIds\_RegEx = 26,

TypeIds\_Error = 27,

TypeIds\_BooleanObject = 28,

TypeIds\_NumberObject = 29,

TypeIds\_StringObject = 30,

TypeIds\_Arguments = 31,

TypeIds\_ES5Array = 32,

TypeIds\_ArrayBuffer = 33,

TypeIds\_Int8Array = 34,

TypeIds\_TypedArrayMin = TypeIds\_Int8Array,

TypeIds\_TypedArraySCAMin = TypeIds\_Int8Array, // Min SCA supported TypedArray TypeId

TypeIds\_Uint8Array = 35,

TypeIds\_Uint8ClampedArray = 36,

TypeIds\_Int16Array = 37,

TypeIds\_Uint16Array = 38,

TypeIds\_Int32Array = 39,

TypeIds\_Uint32Array = 40,

TypeIds\_Float32Array = 41,

TypeIds\_Float64Array = 42,

TypeIds\_TypedArraySCAMax = TypeIds\_Float64Array, // Max SCA supported TypedArray TypeId

TypeIds\_Int64Array = 43,

TypeIds\_Uint64Array = 44,

TypeIds\_CharArray = 45,

TypeIds\_BoolArray = 46,

TypeIds\_TypedArrayMax = TypeIds\_BoolArray,

TypeIds\_EngineInterfaceObject = 47,

TypeIds\_DataView = 48,

TypeIds\_WinRTDate = 49,

TypeIds\_Map = 50,

TypeIds\_Set = 51,

TypeIds\_WeakMap = 52,

TypeIds\_WeakSet = 53,

TypeIds\_SymbolObject = 54,

TypeIds\_ArrayIterator = 55,

TypeIds\_MapIterator = 56,

TypeIds\_SetIterator = 57,

TypeIds\_StringIterator = 58,

TypeIds\_JavascriptEnumeratorIterator = 59,

TypeIds\_Generator = 60,

TypeIds\_Promise = 61,

TypeIds\_LastBuiltinDynamicObject = TypeIds\_Promise,

TypeIds\_GlobalObject = 62,

TypeIds\_ModuleRoot = 63,

TypeIds\_LastTrueJavascriptObjectType = TypeIds\_ModuleRoot,

TypeIds\_HostObject = 64,

TypeIds\_ActivationObject = 65,

TypeIds\_SpreadArgument = 66,

TypeIds\_Limit //add a new TypeId before TypeIds\_Limit or before TypeIds\_LastTrueJavascriptObjectType

};

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

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

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

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

#include "RuntimeTypePch.h"

#include "Library\ES5Array.h"

namespace Js

{

IndexPropertyDescriptorMap::IndexPropertyDescriptorMap(Recycler\* recycler)

: recycler(recycler), indexList(NULL), lastIndexAt(-1)

{

indexPropertyMap = RecyclerNew(recycler, InnerMap, recycler);

}

void IndexPropertyDescriptorMap::Add(uint32 key, const IndexPropertyDescriptor& value)

{

if (indexPropertyMap->Count() >= (INT\_MAX / 2))

{

Js::Throw::OutOfMemory(); // Would possibly overflow our dictionary

}

indexList = NULL; // Discard indexList on change

indexPropertyMap->Add(key, value);

}

//

// Build sorted index list if not found.

//

void IndexPropertyDescriptorMap::EnsureIndexList()

{

if (!indexList)

{

int length = Count();

indexList = RecyclerNewArrayLeaf(recycler, uint32, length);

lastIndexAt = -1; // Reset lastAccessorIndexAt

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

{

indexList[i] = GetKeyAt(i);

}

::qsort(indexList, length, sizeof(uint32), &CompareIndex);

}

}

//

// Try get the last index in this map if it contains any valid index.

//

bool IndexPropertyDescriptorMap::TryGetLastIndex(uint32\* lastIndex)

{

if (Count() == 0)

{

return false;

}

EnsureIndexList();

// Search the index list backwards for the last index

for (int i = Count() - 1; i >= 0; i--)

{

uint32 key = indexList[i];

IndexPropertyDescriptor\* descriptor;

bool b = TryGetReference(key, &descriptor);

Assert(b && descriptor);

if (!(descriptor->Attributes & PropertyDeleted))

{

\*lastIndex = key;

return true;

}

}

return false;

}

//

// Get the next index in the map, similar to JavascriptArray::GetNextIndex().

//

BOOL IndexPropertyDescriptorMap::IsValidDescriptorToken(void \* descriptorValidationToken) const

{

return indexList != nullptr && descriptorValidationToken == indexList;

}

uint32 IndexPropertyDescriptorMap::GetNextDescriptor(uint32 key, IndexPropertyDescriptor\*\* ppDescriptor, void \*\* pDescriptorValidationToken)

{

\*pDescriptorValidationToken = nullptr;

\*ppDescriptor = nullptr;

if (Count() == 0)

{

return JavascriptArray::InvalidIndex;

}

EnsureIndexList();

// Find the first item index > key

int low = 0;

if (key != JavascriptArray::InvalidIndex)

{

Assert(lastIndexAt < Count()); // lastIndexAt must be either -1 or in range [0, Count)

if (lastIndexAt >= 0 && indexList[lastIndexAt] == key)

{

low = lastIndexAt + 1;

}

else

{

int high = Count() - 1;

while (low < high)

{

int mid = (low + high) / 2;

if (indexList[mid] <= key)

{

low = mid + 1;

}

else

{

high = mid;

}

}

if (low < Count() && indexList[low] <= key)

{

++low;

}

}

}

// Search for the next valid index

for (; low < Count(); low++)

{

uint32 index = indexList[low];

IndexPropertyDescriptor\* descriptor;

bool b = TryGetReference(index, &descriptor);

Assert(b && descriptor);

if (!(descriptor->Attributes & PropertyDeleted))

{

lastIndexAt = low; // Save last index location

\*pDescriptorValidationToken = indexList; // use the index list to keep track of where the descriptor has been changed.

\*ppDescriptor = descriptor;

return index;

}

}

return JavascriptArray::InvalidIndex;

}

//

// Try to delete the range [firstKey, length) from right to left, stop if running into an element whose

// [[CanDelete]] is false. Return the index where [index, ...) are all deleted.

//

uint32 IndexPropertyDescriptorMap::DeleteDownTo(uint32 firstKey)

{

EnsureIndexList();

// Iterate the index list backwards to delete from right to left

for (int i = Count() - 1; i >= 0; i--)

{

uint32 key = indexList[i];

if (key < firstKey)

{

break; // We are done, [firstKey, ...) have already been deleted

}

IndexPropertyDescriptor\* descriptor;

bool b = TryGetReference(key, &descriptor);

Assert(b && descriptor);

if (descriptor->Attributes & PropertyDeleted)

{

continue; // Skip empty entry

}

if (descriptor->Attributes & PropertyConfigurable)

{

descriptor->Getter = NULL;

descriptor->Setter = NULL;

descriptor->Attributes = PropertyDeleted | PropertyWritable | PropertyConfigurable;

}

else

{

// Cannot delete key, and [key + 1, ...) are all deleted

return key + 1;

}

}

return firstKey;

}

template <class T>

ES5ArrayTypeHandlerBase<T>\* ES5ArrayTypeHandlerBase<T>::New(Recycler \* recycler, int initialCapacity, uint16 inlineSlotCapacity, uint16 offsetOfInlineSlots)

{

return NewTypeHandler<ES5ArrayTypeHandlerBase<T>>(recycler, initialCapacity, inlineSlotCapacity, offsetOfInlineSlots);

}

template <class T>

ES5ArrayTypeHandlerBase<T>::ES5ArrayTypeHandlerBase(Recycler\* recycler)

: DictionaryTypeHandlerBase<T>(recycler), dataItemAttributes(PropertyDynamicTypeDefaults), lengthWritable(true)

{

indexPropertyMap = RecyclerNew(recycler, IndexPropertyDescriptorMap, recycler);

}

template <class T>

ES5ArrayTypeHandlerBase<T>::ES5ArrayTypeHandlerBase(Recycler\* recycler, int slotCapacity, uint16 inlineSlotCapacity, uint16 offsetOfInlineSlots)

: DictionaryTypeHandlerBase<T>(recycler, slotCapacity, inlineSlotCapacity, offsetOfInlineSlots), dataItemAttributes(PropertyDynamicTypeDefaults), lengthWritable(true)

{

indexPropertyMap = RecyclerNew(recycler, IndexPropertyDescriptorMap, recycler);

}

template <class T>

ES5ArrayTypeHandlerBase<T>::ES5ArrayTypeHandlerBase(Recycler\* recycler, DictionaryTypeHandlerBase<T>\* typeHandler)

: DictionaryTypeHandlerBase<T>(typeHandler), dataItemAttributes(PropertyDynamicTypeDefaults), lengthWritable(true)

{

indexPropertyMap = RecyclerNew(recycler, IndexPropertyDescriptorMap, recycler);

}

template <class T>

void ES5ArrayTypeHandlerBase<T>::SetIsPrototype(DynamicObject \* instance)

{

\_\_super::SetIsPrototype(instance);

// We have ES5 array has array/object prototype, we can't use array fast path for set

// as index could be readonly or be getter/setter in the prototype

// TODO: we may be able to separate out the array fast path and the object array fast path

// here.

instance->GetScriptContext()->optimizationOverrides.DisableArraySetElementFastPath();

}

template <class T>

void ES5ArrayTypeHandlerBase<T>::SetInstanceTypeHandler(DynamicObject\* instance, bool hasChanged)

{

Assert(JavascriptArray::Is(instance));

if (this->GetFlags() & DynamicTypeHandler::IsPrototypeFlag)

{

// We have ES5 array has array/object prototype, we can't use array fast path for set

// as index could be readonly or be getter/setter in the prototype

// TODO: we may be able to separate out the array fast path and the object array fast path

// here.

instance->GetScriptContext()->optimizationOverrides.DisableArraySetElementFastPath();

}

#if ENABLE\_COPYONACCESS\_ARRAY

JavascriptLibrary::CheckAndConvertCopyOnAccessNativeIntArray<Var>(instance);

#endif

JavascriptArray \* arrayInstance = JavascriptArray::EnsureNonNativeArray(JavascriptArray::FromVar(instance));

#if DBG

bool doneConversion = false;

Js::Type\* oldType = arrayInstance->GetType();

#endif

bool isCrossSiteObject = false;

\_\_try

{

if (!CrossSite::IsCrossSiteObjectTyped(arrayInstance))

{

// Convert instance to an ES5Array

Assert(VirtualTableInfo<JavascriptArray>::HasVirtualTable(arrayInstance));

VirtualTableInfo<ES5Array>::SetVirtualTable(arrayInstance);

}

else

{

// If instance was a cross-site JavascriptArray, convert to a cross-site ES5Array

Assert(VirtualTableInfo<CrossSiteObject<JavascriptArray>>::HasVirtualTable(arrayInstance));

VirtualTableInfo<CrossSiteObject<ES5Array>>::SetVirtualTable(arrayInstance);

isCrossSiteObject = true;

}

arrayInstance->ChangeType(); // force change TypeId

\_\_super::SetInstanceTypeHandler(arrayInstance, false); // after forcing the type change, we don't need to changeType again.

#if DBG

doneConversion = true;

#endif

}

\_\_finally

{

if (AbnormalTermination())

{

Assert(!doneConversion);

// change vtbl shouldn't OOM. revert back the vtable.

if (isCrossSiteObject)

{

Assert(VirtualTableInfo<CrossSiteObject<ES5Array>>::HasVirtualTable(arrayInstance));

VirtualTableInfo<CrossSiteObject<JavascriptArray>>::SetVirtualTable(arrayInstance);

}

else

{

Assert(VirtualTableInfo<ES5Array>::HasVirtualTable(arrayInstance));

VirtualTableInfo<JavascriptArray>::SetVirtualTable(arrayInstance);

}

// The only allocation is in ChangeType, which won't have changed the type yet.

Assert(arrayInstance->GetType() == oldType);

}

}

}

template <class T>

BOOL ES5ArrayTypeHandlerBase<T>::HasDataItem(ES5Array\* arr, uint32 index)

{

Var value;

return arr->DirectGetItemAt(index, &value);

}

//

// Check if the array contains any data item not in attribute map (so that we know there are items

// using shared data item attributes)

//

template <class T>

bool ES5ArrayTypeHandlerBase<T>::HasAnyDataItemNotInMap(ES5Array\* arr)

{

JavascriptArray::ArrayElementEnumerator e(arr);

while (e.MoveNext<Var>())

{

if (!indexPropertyMap->ContainsKey(e.GetIndex()))

{

return true;

}

}

return false;

}

template <class T>

PropertyAttributes ES5ArrayTypeHandlerBase<T>::GetDataItemAttributes() const

{

return dataItemAttributes;

}

template <class T>

void ES5ArrayTypeHandlerBase<T>::SetDataItemSealed()

{

dataItemAttributes &= ~(PropertyConfigurable);

}

template <class T>

void ES5ArrayTypeHandlerBase<T>::SetDataItemFrozen()

{

dataItemAttributes &= ~(PropertyWritable | PropertyConfigurable);

this->ClearHasOnlyWritableDataProperties();

}

template <class T>

BOOL ES5ArrayTypeHandlerBase<T>::CantAssign(PropertyOperationFlags flags, ScriptContext\* scriptContext)

{

JavascriptError::ThrowCantAssignIfStrictMode(flags, scriptContext);

return false;

}

template <class T>

BOOL ES5ArrayTypeHandlerBase<T>::CantExtend(PropertyOperationFlags flags, ScriptContext\* scriptContext)

{

JavascriptError::ThrowCantExtendIfStrictMode(flags, scriptContext);

return false;

}

template <class T>

BOOL ES5ArrayTypeHandlerBase<T>::HasItem(ES5Array\* arr, uint32 index)

{

// We have the item if we have its descriptor.

IndexPropertyDescriptor\* descriptor;

if (indexPropertyMap->TryGetReference(index, &descriptor))

{

return !(descriptor->Attributes & PropertyDeleted);

}

// Otherwise check if we have such a data item.

return HasDataItem(arr, index);

}

template <class T>

BOOL ES5ArrayTypeHandlerBase<T>::SetItem(ES5Array\* arr, DynamicObject\* instance, uint32 index, Var value, PropertyOperationFlags flags)

{

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

// Reject if we need to grow non-writable length

if (!CanSetItemAt(arr, index))

{

return false;

}

IndexPropertyDescriptor\* descriptor;

if (indexPropertyMap->TryGetReference(index, &descriptor))

{

if (descriptor->Attributes & PropertyDeleted)

{

if (!(this->GetFlags() & IsExtensibleFlag))

{

return CantExtend(flags, scriptContext);

}

// No need to change hasNoEnumerableProperties. See comment in ES5ArrayTypeHandlerBase<T>::SetItemWithAttributes.

Assert(!arr->GetHasNoEnumerableProperties());

Assert(!descriptor->Getter && !descriptor->Setter);

descriptor->Attributes = PropertyDynamicTypeDefaults;

arr->DirectSetItemAt(index, value);

return true;

}

if (!(descriptor->Attributes & PropertyWritable))

{

return CantAssign(flags, scriptContext);

}

if (HasDataItem(arr, index))

{

arr->DirectSetItemAt(index, value);

}

else if (descriptor->Setter)

{

RecyclableObject\* func = RecyclableObject::FromVar(descriptor->Setter);

// TODO : request context

JavascriptOperators::CallSetter(func, instance, value, NULL);

}

return true;

}

//

// Not found in attribute map. Extend or update data item.

//

if (!(this->GetFlags() & IsExtensibleFlag))

{

if (!HasDataItem(arr, index))

{

return CantExtend(flags, scriptContext);

}

else if (!(GetDataItemAttributes() & PropertyWritable))

{

return CantAssign(flags, scriptContext);

}

}

// No need to change hasNoEnumerableProperties. See comment in ES5ArrayTypeHandlerBase<T>::SetItemWithAttributes.

Assert(!arr->GetHasNoEnumerableProperties());

arr->DirectSetItemAt(index, value); // sharing data item attributes

return true;

}

template <class T>

BOOL ES5ArrayTypeHandlerBase<T>::SetItemWithAttributes(ES5Array\* arr, DynamicObject\* instance, uint32 index, Var value, PropertyAttributes attributes)

{

// Reject if we need to grow non-writable length

if (!CanSetItemAt(arr, index))

{

return false;

}

// We don't track non-enumerable items in object array. Objects with an object array

// report having enumerable properties. See DynamicObject::GetHasNoEnumerableProperties.

// Array objects (which don't have an object array, and could report their hasNoEnumerableProperties

// directly) take an explicit type transition before switching to ES5ArrayTypeHandler, so their

// hasNoEnumerableProperties flag gets cleared.

Assert(!arr->GetHasNoEnumerableProperties());

if (!(attributes & PropertyWritable))

{

this->ClearHasOnlyWritableDataProperties();

if(GetFlags() & IsPrototypeFlag)

{

instance->GetLibrary()->NoPrototypeChainsAreEnsuredToHaveOnlyWritableDataProperties();

}

}

IndexPropertyDescriptor\* descriptor;

if (indexPropertyMap->TryGetReference(index, &descriptor))

{

if (descriptor->Attributes & PropertyDeleted)

{

Assert(!descriptor->Getter && !descriptor->Setter);

descriptor->Attributes = attributes;

arr->DirectSetItemAt(index, value);

return true;

}

descriptor->Attributes = attributes;

if (HasDataItem(arr, index))

{

arr->DirectSetItemAt(index, value);

}

else if (descriptor->Setter)

{

RecyclableObject\* func = RecyclableObject::FromVar(descriptor->Setter);

// TODO : request context

JavascriptOperators::CallSetter(func, instance, value, NULL);

}

}

else

{

// See comment for the same assert above.

Assert(!arr->GetHasNoEnumerableProperties());

// Not found in attribute map

arr->DirectSetItemAt(index, value);

indexPropertyMap->Add(index, IndexPropertyDescriptor(attributes));

}

return true;

}

template <class T>

BOOL ES5ArrayTypeHandlerBase<T>::SetItemAttributes(ES5Array\* arr, DynamicObject\* instance, uint32 index, PropertyAttributes attributes)

{

IndexPropertyDescriptor\* descriptor;

if (indexPropertyMap->TryGetReference(index, &descriptor))

{

if (descriptor->Attributes & PropertyDeleted)

{

return false;

}

// No need to change hasNoEnumerableProperties. See comment in ES5ArrayTypeHandlerBase<T>::SetItemWithAttributes.

Assert(!arr->GetHasNoEnumerableProperties());

descriptor->Attributes = (descriptor->Attributes & ~PropertyDynamicTypeDefaults) | (attributes & PropertyDynamicTypeDefaults);

if (!(descriptor->Attributes & PropertyWritable))

{

this->ClearHasOnlyWritableDataProperties();

if(GetFlags() & IsPrototypeFlag)

{

instance->GetLibrary()->NoPrototypeChainsAreEnsuredToHaveOnlyWritableDataProperties();

}

}

return true;

}

else if (HasDataItem(arr, index))

{

// No need to change hasNoEnumerableProperties. See comment in ES5ArrayTypeHandlerBase<T>::SetItemWithAttributes.

Assert(!arr->GetHasNoEnumerableProperties());

indexPropertyMap->Add(index, IndexPropertyDescriptor(attributes & PropertyDynamicTypeDefaults));

if (!(attributes & PropertyWritable))

{

this->ClearHasOnlyWritableDataProperties();

if(GetFlags() & IsPrototypeFlag)

{

instance->GetLibrary()->NoPrototypeChainsAreEnsuredToHaveOnlyWritableDataProperties();

}

}

return true;

}

return false;

}

template <class T>

BOOL ES5ArrayTypeHandlerBase<T>::SetItemAccessors(ES5Array\* arr, DynamicObject\* instance, uint32 index, Var getter, Var setter)

{

// Reject if we need to grow non-writable length

if (!CanSetItemAt(arr, index))

{

return false;

}

JavascriptLibrary\* lib = instance->GetLibrary();

if (getter)

{

getter = CanonicalizeAccessor(getter, lib);

}

if (setter)

{

setter = CanonicalizeAccessor(setter, lib);

}

// conversion from data-property to accessor property

arr->DirectDeleteItemAt<Var>(index);

IndexPropertyDescriptor\* descriptor;

if (indexPropertyMap->TryGetReference(index, &descriptor))

{

if (descriptor->Attributes & PropertyDeleted)

{

descriptor->Attributes = PropertyDynamicTypeDefaults;

}

if (getter)

{

descriptor->Getter = getter;

}

if (setter)

{

descriptor->Setter = setter;

}

}

else

{

indexPropertyMap->Add(index, IndexPropertyDescriptor(getter, setter));

}

if (arr->GetLength() <= index)

{

uint32 newLength = index;

UInt32Math::Inc(newLength);

arr->SetLength(newLength);

}

this->ClearHasOnlyWritableDataProperties();

if(GetFlags() & IsPrototypeFlag)

{

instance->GetLibrary()->NoPrototypeChainsAreEnsuredToHaveOnlyWritableDataProperties();

}

return true;

}

template <class T>

BOOL ES5ArrayTypeHandlerBase<T>::GetItemAccessors(ES5Array\* arr, DynamicObject\* instance, uint32 index, Var\* getter, Var\* setter)

{

IndexPropertyDescriptor\* descriptor;

if (indexPropertyMap->TryGetReference(index, &descriptor))

{

if (descriptor->Attributes & PropertyDeleted)

{

return false;

}

if (!HasDataItem(arr, index)) // if not shadowed by data item

{

\*getter = descriptor->Getter;

\*setter = descriptor->Setter;

return descriptor->Getter || descriptor->Setter;

}

}

return false;

}

// Check if this array can set item at the given index.

template <class T>

bool ES5ArrayTypeHandlerBase<T>::CanSetItemAt(ES5Array\* arr, uint32 index) const

{

return IsLengthWritable() || index < arr->GetLength();

}

template <class T>

BOOL ES5ArrayTypeHandlerBase<T>::DeleteItem(ES5Array\* arr, DynamicObject\* instance, uint32 index, PropertyOperationFlags propertyOperationFlags)

{

IndexPropertyDescriptor\* descriptor;

if (indexPropertyMap->TryGetReference(index, &descriptor))

{

if (descriptor->Attributes & PropertyDeleted)

{

return true;

}

else if (!(descriptor->Attributes & PropertyConfigurable))

{

JavascriptError::ThrowCantDeleteIfStrictMode(propertyOperationFlags, instance->GetScriptContext(), TaggedInt::ToString(index, instance->GetScriptContext())->GetString());

return false;

}

arr->DirectDeleteItemAt<Var>(index);

descriptor->Getter = NULL;

descriptor->Setter = NULL;

descriptor->Attributes = PropertyDeleted | PropertyWritable | PropertyConfigurable;

return true;

}

// Not in attribute map

if (!(GetDataItemAttributes() & PropertyConfigurable))

{

return !HasDataItem(arr, index); // CantDelete

}

return arr->DirectDeleteItemAt<Var>(index);

}

template <class T>

BOOL ES5ArrayTypeHandlerBase<T>::GetItem(ES5Array\* arr, DynamicObject\* instance, Var originalInstance, uint32 index, Var\* value, ScriptContext\* requestContext)

{

if (arr->DirectGetItemAt<Var>(index, value))

{

return true;

}

IndexPropertyDescriptor\* descriptor;

if (indexPropertyMap->TryGetReference(index, &descriptor))

{

if (descriptor->Attributes & PropertyDeleted)

{

return false;

}

if (descriptor->Getter)

{

RecyclableObject\* func = RecyclableObject::FromVar(descriptor->Getter);

\*value = Js::JavascriptOperators::CallGetter(func, originalInstance, requestContext);

}

else

{

\*value = instance->GetLibrary()->GetUndefined();

}

return true;

}

return false;

}

template <class T>

DescriptorFlags ES5ArrayTypeHandlerBase<T>::GetItemSetter(ES5Array\* arr, DynamicObject\* instance, uint32 index, Var\* setterValue, ScriptContext\* requestContext)

{

IndexPropertyDescriptor\* descriptor;

if (indexPropertyMap->TryGetReference(index, &descriptor))

{

if (descriptor->Attributes & PropertyDeleted)

{

return None;

}

if (HasDataItem(ES5Array::FromVar(instance), index))

{

// not a setter but shadows

return (descriptor->Attributes & PropertyWritable) ? WritableData : Data;

}

else if (descriptor->Setter)

{

\*setterValue = descriptor->Setter;

return Accessor;

}

}

else if (HasDataItem(ES5Array::FromVar(instance), index))

{

return (GetDataItemAttributes() & PropertyWritable) ? WritableData : Data;

}

return None;

}

template <class T>

BOOL ES5ArrayTypeHandlerBase<T>::HasProperty(DynamicObject\* instance, PropertyId propertyId, bool \*noRedecl)

{

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

uint32 index;

if (noRedecl != nullptr)

{

\*noRedecl = false;

}

if (scriptContext->IsNumericPropertyId(propertyId, &index))

{

// Call my version of HasItem

return ES5ArrayTypeHandlerBase<T>::HasItem(instance, index);

}

return \_\_super::HasProperty(instance, propertyId, noRedecl);

}

template <class T>

BOOL ES5ArrayTypeHandlerBase<T>::HasProperty(DynamicObject\* instance, JavascriptString\* propertyNameString)

{

AssertMsg(!PropertyRecord::IsPropertyNameNumeric(propertyNameString->GetString(), propertyNameString->GetLength()),

"Numeric property names should have been converted to uint or PropertyRecord\*");

return \_\_super::HasProperty(instance, propertyNameString);

}

template <class T>

BOOL ES5ArrayTypeHandlerBase<T>::GetProperty(DynamicObject\* instance, Var originalInstance, PropertyId propertyId, Var\* value, PropertyValueInfo\* info, ScriptContext\* requestContext)

{

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

uint32 index;

if (scriptContext->IsNumericPropertyId(propertyId, &index))

{

return GetItem(ES5Array::FromVar(instance), instance, index, value, requestContext);

}

return \_\_super::GetProperty(instance, originalInstance, propertyId, value, info, requestContext);

}

template <class T>

BOOL ES5ArrayTypeHandlerBase<T>::GetProperty(DynamicObject\* instance, Var originalInstance, JavascriptString\* propertyNameString, Var\* value, PropertyValueInfo\* info, ScriptContext\* requestContext)

{

AssertMsg(!PropertyRecord::IsPropertyNameNumeric(propertyNameString->GetString(), propertyNameString->GetLength()),

"Numeric property names should have been converted to uint or PropertyRecord\*");

return \_\_super::GetProperty(instance, originalInstance, propertyNameString, value, info, requestContext);

}

template <class T>

DescriptorFlags ES5ArrayTypeHandlerBase<T>::GetSetter(DynamicObject\* instance, PropertyId propertyId, Var\* setterValue, PropertyValueInfo\* info, ScriptContext\* requestContext)

{

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

uint32 index;

if (scriptContext->IsNumericPropertyId(propertyId, &index))

{

PropertyValueInfo::SetNoCache(info, instance);

return ES5ArrayTypeHandlerBase<T>::GetItemSetter(instance, index, setterValue, requestContext);

}

return \_\_super::GetSetter(instance, propertyId, setterValue, info, requestContext);

}

template <class T>

DescriptorFlags ES5ArrayTypeHandlerBase<T>::GetSetter(DynamicObject\* instance, JavascriptString\* propertyNameString, Var\* setterValue, PropertyValueInfo\* info, ScriptContext\* requestContext)

{

AssertMsg(!PropertyRecord::IsPropertyNameNumeric(propertyNameString->GetString(), propertyNameString->GetLength()),

"Numeric property names should have been converted to uint or PropertyRecord\*");

return \_\_super::GetSetter(instance, propertyNameString, setterValue, info, requestContext);

}

template <class T>

BOOL ES5ArrayTypeHandlerBase<T>::DeleteProperty(DynamicObject\* instance, PropertyId propertyId, PropertyOperationFlags flags)

{

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

uint32 index;

if (scriptContext->IsNumericPropertyId(propertyId, &index))

{

return DeleteItem(ES5Array::FromVar(instance), instance, index, flags);

}

return \_\_super::DeleteProperty(instance, propertyId, flags);

}

template <class T>

BOOL ES5ArrayTypeHandlerBase<T>::HasItem(DynamicObject\* instance, uint32 index)

{

return HasItem(ES5Array::FromVar(instance), index);

}

template <class T>

BOOL ES5ArrayTypeHandlerBase<T>::SetItem(DynamicObject\* instance, uint32 index, Var value, PropertyOperationFlags flags)

{

return SetItem(ES5Array::FromVar(instance), instance, index, value, flags);

}

template <class T>

BOOL ES5ArrayTypeHandlerBase<T>::SetItemWithAttributes(DynamicObject\* instance, uint32 index, Var value, PropertyAttributes attributes)

{

return SetItemWithAttributes(ES5Array::FromVar(instance), instance, index, value, attributes);

}

template <class T>

BOOL ES5ArrayTypeHandlerBase<T>::SetItemAttributes(DynamicObject\* instance, uint32 index, PropertyAttributes attributes)

{

return SetItemAttributes(ES5Array::FromVar(instance), instance, index, attributes);

}

template <class T>

BOOL ES5ArrayTypeHandlerBase<T>::SetItemAccessors(DynamicObject\* instance, uint32 index, Var getter, Var setter)

{

return SetItemAccessors(ES5Array::FromVar(instance), instance, index, getter, setter);

}

template <class T>

BOOL ES5ArrayTypeHandlerBase<T>::DeleteItem(DynamicObject\* instance, uint32 index, PropertyOperationFlags flags)

{

return DeleteItem(ES5Array::FromVar(instance), instance, index, flags);

}

template <class T>

BOOL ES5ArrayTypeHandlerBase<T>::GetItem(DynamicObject\* instance, Var originalInstance, uint32 index, Var\* value, ScriptContext\* requestContext)

{

return GetItem(ES5Array::FromVar(instance), instance, originalInstance, index, value, requestContext);

}

template <class T>

DescriptorFlags ES5ArrayTypeHandlerBase<T>::GetItemSetter(DynamicObject\* instance, uint32 index, Var\* setterValue, ScriptContext\* requestContext)

{

return GetItemSetter(ES5Array::FromVar(instance), instance, index, setterValue, requestContext);

}

template <class T>

bool ES5ArrayTypeHandlerBase<T>::IsLengthWritable() const

{

return lengthWritable;

}

template <class T>

void ES5ArrayTypeHandlerBase<T>::SetLengthWritable(bool writable)

{

lengthWritable = writable;

if (!writable)

{

ClearHasOnlyWritableDataProperties();

}

}

//

// Try to delete the range [firstKey, length) from right to left, stop if running into an element whose

// [[CanDelete]] is false. Return the index where [index, ...) can all be deleted.

//

// Note that this helper method finds the max range to delete but may or may not delete the data items.

// The caller needs to call JavascriptArray::SetLength to trim the data items.

//

template <class T>

uint32 ES5ArrayTypeHandlerBase<T>::DeleteDownTo(ES5Array\* arr, uint32 first, PropertyOperationFlags propertyOperationFlags)

{

Assert(first < arr->GetLength()); // Only called when newLen < oldLen

// If the number of elements to be deleted is small, iterate on it.

uint32 count = arr->GetLength() - first;

if (count < 5)

{

uint32 oldLen = arr->GetLength();

while (first < oldLen)

{

if (!arr->DeleteItem(oldLen - 1, propertyOperationFlags))

{

break;

}

--oldLen;

}

return oldLen;

}

// If data items are [[CanDelete]], check attribute map only.

if (GetDataItemAttributes() & PropertyConfigurable)

{

return indexPropertyMap->DeleteDownTo(first);

}

else

{

// The array isSealed. No existing item can be deleted. Look for the max index.

uint32 lastIndex;

if (indexPropertyMap->TryGetLastIndex(&lastIndex) && lastIndex >= first)

{

first = lastIndex + 1;

}

if (TryGetLastDataItemIndex(arr, first, &lastIndex))

{

first = lastIndex + 1;

}

return first;

}

}

//

// Try get the last data item index in the range of [first, length).

//

template <class T>

bool ES5ArrayTypeHandlerBase<T>::TryGetLastDataItemIndex(ES5Array\* arr, uint32 first, uint32\* lastIndex)

{

uint32 index = JavascriptArray::InvalidIndex;

JavascriptArray::ArrayElementEnumerator e(arr, first);

while (e.MoveNext<Var>())

{

index = e.GetIndex();

}

if (index != JavascriptArray::InvalidIndex)

{

\*lastIndex = index;

return true;

}

return false;

}

template <class T>

void ES5ArrayTypeHandlerBase<T>::SetLength(ES5Array\* arr, uint32 newLen, PropertyOperationFlags propertyOperationFlags)

{

Assert(IsLengthWritable()); // Should have already checked

if (newLen < arr->GetLength())

{

newLen = DeleteDownTo(arr, newLen, propertyOperationFlags); // Result newLen might be different

}

// Trim data items and set length

arr->SetLength(newLen);

//

// Strict mode TODO: In strict mode we may need to throw if we cannot delete to

// requested newLen (ES5 15.4.5.1 3.l.III.4).

//

}

template <class T>

BOOL ES5ArrayTypeHandlerBase<T>::IsAttributeSet(uint32 index, PropertyAttributes attr)

{

IndexPropertyDescriptor\* descriptor;

if (indexPropertyMap->TryGetReference(index, &descriptor))

{

if (!(descriptor->Attributes & PropertyDeleted))

{

return descriptor->Attributes & attr;

}

}

else

{

return GetDataItemAttributes() & attr;

}

return true;

}

template <class T>

BOOL ES5ArrayTypeHandlerBase<T>::IsAttributeSet(DynamicObject\* instance, PropertyId propertyId, PropertyAttributes attr, BOOL& isNumericPropertyId)

{

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

uint32 index;

isNumericPropertyId = scriptContext->IsNumericPropertyId(propertyId, &index);

if (isNumericPropertyId)

{

return IsAttributeSet(index, attr);

}

return true;

}

template <class T>

BOOL ES5ArrayTypeHandlerBase<T>::UpdateAttribute(DynamicObject\* instance, PropertyId propertyId, PropertyAttributes attr, BOOL value, BOOL& isNumericPropertyId)

{

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

uint32 index;

isNumericPropertyId = scriptContext->IsNumericPropertyId(propertyId, &index);

if (isNumericPropertyId)

{

IndexPropertyDescriptor\* descriptor;

if (indexPropertyMap->TryGetReference(index, &descriptor))

{

if (descriptor->Attributes & PropertyDeleted)

{

return false;

}

if (value)

{

descriptor->Attributes |= attr;

}

else

{

descriptor->Attributes &= (~attr);

if (!(descriptor->Attributes & PropertyWritable))

{

this->ClearHasOnlyWritableDataProperties();

if(GetFlags() & IsPrototypeFlag)

{

instance->GetLibrary()->NoPrototypeChainsAreEnsuredToHaveOnlyWritableDataProperties();

}

}

}

}

else

{

if (!HasDataItem(ES5Array::FromVar(instance), index))

{

return false;

}

PropertyAttributes newAttr = GetDataItemAttributes();

if (value)

{

newAttr |= attr;

}

else

{

newAttr &= (~attr);

}

if (newAttr != GetDataItemAttributes())

{

indexPropertyMap->Add(index, IndexPropertyDescriptor(newAttr));

if (!(newAttr & PropertyWritable))

{

this->ClearHasOnlyWritableDataProperties();

if(GetFlags() & IsPrototypeFlag)

{

instance->GetLibrary()->NoPrototypeChainsAreEnsuredToHaveOnlyWritableDataProperties();

}

}

}

}

return true;

}

return false;

}

template <class T>

BOOL ES5ArrayTypeHandlerBase<T>::IsItemEnumerable(ES5Array\* arr, uint32 index)

{

return IsAttributeSet(index, PropertyEnumerable);

}

template <class T>

BOOL ES5ArrayTypeHandlerBase<T>::IsEnumerable(DynamicObject\* instance, PropertyId propertyId)

{

BOOL isNumericPropertyId;

return IsAttributeSet(instance, propertyId, PropertyEnumerable, isNumericPropertyId)

&& (isNumericPropertyId || \_\_super::IsEnumerable(instance, propertyId));

}

template <class T>

BOOL ES5ArrayTypeHandlerBase<T>::IsWritable(DynamicObject\* instance, PropertyId propertyId)

{

BOOL isNumericPropertyId;

return IsAttributeSet(instance, propertyId, PropertyWritable, isNumericPropertyId)

&& (isNumericPropertyId || \_\_super::IsWritable(instance, propertyId));

}

template <class T>

BOOL ES5ArrayTypeHandlerBase<T>::IsConfigurable(DynamicObject\* instance, PropertyId propertyId)

{

BOOL isNumericPropertyId;

return IsAttributeSet(instance, propertyId, PropertyConfigurable, isNumericPropertyId)

&& (isNumericPropertyId || \_\_super::IsConfigurable(instance, propertyId));

}

template <class T>

BOOL ES5ArrayTypeHandlerBase<T>::SetEnumerable(DynamicObject\* instance, PropertyId propertyId, BOOL value)

{

if (propertyId == PropertyIds::length)

{

Assert(!value); // Can only set enumerable to false

return true;

}

BOOL isNumericPropertyId;

return UpdateAttribute(instance, propertyId, PropertyEnumerable, value, isNumericPropertyId)

|| (!isNumericPropertyId && \_\_super::SetEnumerable(instance, propertyId, value));

}

template <class T>

BOOL ES5ArrayTypeHandlerBase<T>::SetWritable(DynamicObject\* instance, PropertyId propertyId, BOOL value)

{

if (propertyId == PropertyIds::length)

{

SetLengthWritable(value ? true : false);

if(!value && GetFlags() & IsPrototypeFlag)

{

instance->GetLibrary()->NoPrototypeChainsAreEnsuredToHaveOnlyWritableDataProperties();

}

return true;

}

BOOL isNumericPropertyId;

return UpdateAttribute(instance, propertyId, PropertyWritable, value, isNumericPropertyId)

|| (!isNumericPropertyId && \_\_super::SetWritable(instance, propertyId, value));

}

template <class T>

BOOL ES5ArrayTypeHandlerBase<T>::SetConfigurable(DynamicObject\* instance, PropertyId propertyId, BOOL value)

{

if (propertyId == PropertyIds::length)

{

Assert(!value); // Can only set configurable to false

return true;

}

BOOL isNumericPropertyId;

return UpdateAttribute(instance, propertyId, PropertyConfigurable, value, isNumericPropertyId)

|| (!isNumericPropertyId && \_\_super::SetConfigurable(instance, propertyId, value));

}

template <class T>

BOOL ES5ArrayTypeHandlerBase<T>::GetAccessors(DynamicObject\* instance, PropertyId propertyId, Var\* getter, Var\* setter)

{

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

uint32 index;

if (scriptContext->IsNumericPropertyId(propertyId, &index))

{

return GetItemAccessors(ES5Array::FromVar(instance), instance, index, getter, setter);

}

return \_\_super::GetAccessors(instance, propertyId, getter, setter);

}

template <class T>

BOOL ES5ArrayTypeHandlerBase<T>::Seal(DynamicObject\* instance)

{

IndexPropertyDescriptor\* descriptor = NULL;

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

{

descriptor = indexPropertyMap->GetReferenceAt(i);

descriptor->Attributes &= (~PropertyConfigurable);

}

this->SetDataItemSealed(); // set shared data item attributes sealed

return \_\_super::Seal(instance);

}

template <class T>

BOOL ES5ArrayTypeHandlerBase<T>::FreezeImpl(DynamicObject\* instance, bool isConvertedType)

{

ES5Array\* arr = ES5Array::FromVar(instance);

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

{

uint32 index = indexPropertyMap->GetKeyAt(i);

IndexPropertyDescriptor\* descriptor = indexPropertyMap->GetReferenceAt(i);

if (HasDataItem(arr, index))

{

//Only data descriptor has Writable property

descriptor->Attributes &= ~(PropertyWritable | PropertyConfigurable);

}

else

{

descriptor->Attributes &= ~(PropertyConfigurable);

}

}

this->SetDataItemFrozen(); // set shared data item attributes frozen

SetLengthWritable(false); // Freeze "length" as well

return \_\_super::FreezeImpl(instance, isConvertedType);

}

template <class T>

BigDictionaryTypeHandler\* ES5ArrayTypeHandlerBase<T>::NewBigDictionaryTypeHandler(Recycler\* recycler, int slotCapacity, uint16 inlineSlotCapacity, uint16 offsetOfInlineSlots)

{

return RecyclerNew(recycler, BigES5ArrayTypeHandler, recycler, slotCapacity, inlineSlotCapacity, offsetOfInlineSlots, this);

}

template <class T>

BOOL ES5ArrayTypeHandlerBase<T>::IsSealed(DynamicObject\* instance)

{

if (!\_\_super::IsSealed(instance))

{

return false;

}

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

{

IndexPropertyDescriptor\* descriptor = indexPropertyMap->GetReferenceAt(i);

if (descriptor->Attributes & PropertyDeleted)

{

continue; // Skip deleted

}

if (descriptor->Attributes & PropertyConfigurable)

{

//[[Configurable]] must be false for all properties.

return false;

}

}

// Check data item not in map

if (this->GetDataItemAttributes() & PropertyConfigurable)

{

if (HasAnyDataItemNotInMap(ES5Array::FromVar(instance)))

{

return false;

}

}

return true;

}

//

// When arr is objectArray of an object, we should skip "length" while testing isFrozen. "length" is an

// own property of arr, but not of the containing object.

//

template <class T>

BOOL ES5ArrayTypeHandlerBase<T>::IsObjectArrayFrozen(ES5Array\* arr)

{

if (!\_\_super::IsFrozen(arr))

{

return false;

}

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

{

uint32 index = indexPropertyMap->GetKeyAt(i);

IndexPropertyDescriptor\* descriptor = indexPropertyMap->GetReferenceAt(i);

if (descriptor->Attributes & PropertyDeleted)

{

continue; // Skip deleted

}

if (descriptor->Attributes & PropertyConfigurable)

{

return false;

}

if ((descriptor->Attributes & PropertyWritable) && HasDataItem(arr, index))

{

//Only data descriptor has Writable property

return false;

}

}

// Check data item not in map

if (this->GetDataItemAttributes() & (PropertyWritable | PropertyConfigurable))

{

if (HasAnyDataItemNotInMap(arr))

{

return false;

}

}

return true;

}

template <class T>

BOOL ES5ArrayTypeHandlerBase<T>::IsFrozen(DynamicObject\* instance)

{

// We need to check "length" frozen for standalone ES5Array

if (IsLengthWritable())

{

return false;

}

return IsObjectArrayFrozen(ES5Array::FromVar(instance));

}

template <class T>

BOOL ES5ArrayTypeHandlerBase<T>::SetAttributes(DynamicObject\* instance, PropertyId propertyId, PropertyAttributes attributes)

{

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

uint32 index;

if (scriptContext->IsNumericPropertyId(propertyId, &index))

{

return SetItemAttributes(ES5Array::FromVar(instance), instance, index, attributes);

}

return \_\_super::SetAttributes(instance, propertyId, attributes);

}

template <class T>

BOOL ES5ArrayTypeHandlerBase<T>::IsValidDescriptorToken(void \* descriptorValidationToken) const

{

return indexPropertyMap->IsValidDescriptorToken(descriptorValidationToken);

}

template <class T>

uint32 ES5ArrayTypeHandlerBase<T>::GetNextDescriptor(uint32 key, IndexPropertyDescriptor\*\* descriptor, void \*\* descriptorValidationToken)

{

return indexPropertyMap->GetNextDescriptor(key, descriptor, descriptorValidationToken);

}

template <class T>

BOOL ES5ArrayTypeHandlerBase<T>::GetDescriptor(uint32 index, Js::IndexPropertyDescriptor \*\*ppDescriptor) {

return indexPropertyMap->TryGetReference(index, ppDescriptor);

}

template class ES5ArrayTypeHandlerBase<PropertyIndex>;

template class ES5ArrayTypeHandlerBase<BigPropertyIndex>;

}

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

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

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

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

#pragma once

namespace Js

{

class IndexPropertyDescriptor

{

public:

PropertyAttributes Attributes;

Var Getter;

Var Setter;

IndexPropertyDescriptor(PropertyAttributes attributes = PropertyDynamicTypeDefaults,

Var getter = NULL, Var setter = NULL)

: Attributes(attributes), Getter(getter), Setter(setter)

{

}

IndexPropertyDescriptor(Var getter, Var setter)

: Attributes(PropertyDynamicTypeDefaults), Getter(getter), Setter(setter)

{

}

};

//

// IndexPropertyDescriptorMap uses a dictionary for quick index attribute look up. When visiting

// in order is needed, it creates an ordered index list on the fly.

//

class IndexPropertyDescriptorMap

{

private:

// Note: IndexPropertyDescriptor contains references. We need to allocate entries as non-leaf node.

typedef JsUtil::BaseDictionary<uint32, IndexPropertyDescriptor, ForceNonLeafAllocator<Recycler>::AllocatorType, PowerOf2SizePolicy>

InnerMap;

Recycler\* recycler;

InnerMap\* indexPropertyMap; // The internal real index property map

uint32\* indexList; // The index list that's created on demand

int lastIndexAt; // Last used index list entry

private:

void EnsureIndexList();

public:

IndexPropertyDescriptorMap(Recycler\* recycler);

void Add(uint32 key, const IndexPropertyDescriptor& descriptor);

bool TryGetLastIndex(uint32\* lastIndex);

BOOL IsValidDescriptorToken(void \* descriptorValidationToken) const;

uint32 GetNextDescriptor(uint32 key, IndexPropertyDescriptor\*\* descriptor, void \*\* descriptorValidationToken);

uint32 DeleteDownTo(uint32 firstKey);

int Count() const

{

return indexPropertyMap->Count();

}

uint32 GetKeyAt(int i) const

{

return indexPropertyMap->GetKeyAt(i);

}

IndexPropertyDescriptor\* GetReferenceAt(int i) const

{

return indexPropertyMap->GetReferenceAt(i);

}

bool ContainsKey(uint32 key) const

{

return indexPropertyMap->ContainsKey(key);

}

bool TryGetReference(uint32 key, IndexPropertyDescriptor\*\* value) const

{

return indexPropertyMap->TryGetReference(key, value);

}

private:

static int \_\_cdecl CompareIndex(const void\* left, const void\* right)

{

return \*static\_cast<const uint32\*>(left) - \*static\_cast<const uint32\*>(right);

}

};

//

// Private type handler used by ES5Array

//

template <class T>

class ES5ArrayTypeHandlerBase sealed: public DictionaryTypeHandlerBase<T>

{

friend class NullTypeHandlerBase;

friend class DeferredTypeHandlerBase;

template <DeferredTypeInitializer initializer, typename DeferredTypeFilter, bool isPrototypeTemplate, uint16 inlineSlotCapacity, uint16 offsetOfInlineSlots>

friend class DeferredTypeHandler;

friend class PathTypeHandlerBase;

template<size\_t size>

friend class SimpleTypeHandler;

template <typename TPropertyIndex, typename TMapKey, bool IsNotExtensibleSupported> friend class SimpleDictionaryTypeHandlerBase;

template <typename T> friend class DictionaryTypeHandlerBase;

template <typename T> friend class ES5ArrayTypeHandlerBase;

private:

IndexPropertyDescriptorMap\* indexPropertyMap;

PropertyAttributes dataItemAttributes; // attributes for data item not in map

bool lengthWritable;

public:

DEFINE\_GETCPPNAME();

private:

ES5ArrayTypeHandlerBase(Recycler\* recycler);

ES5ArrayTypeHandlerBase(Recycler\* recycler, int slotCapacity, uint16 inlineSlotCapacity, uint16 offsetOfInlineSlots);

ES5ArrayTypeHandlerBase(Recycler\* recycler, DictionaryTypeHandlerBase<T>\* typeHandler);

DEFINE\_VTABLE\_CTOR\_NO\_REGISTER(ES5ArrayTypeHandlerBase, DictionaryTypeHandlerBase<T>);

// This constructor is used to grow small ES5ArrayTypeHandler into BigES5ArrayTypeHandler. We simply take over all own fields here

// as the Small/Big difference only exists in base DictionaryTypeHandler. Base class is responsible to handle non-index properties.

template <class SmallIndexType>

ES5ArrayTypeHandlerBase(Recycler\* recycler, int slotCapacity, uint16 inlineSlotCapacity, uint16 offsetOfInlineSlots, ES5ArrayTypeHandlerBase<SmallIndexType>\* typeHandler)

: DictionaryTypeHandlerBase<T>(recycler, slotCapacity, inlineSlotCapacity, offsetOfInlineSlots),

indexPropertyMap(typeHandler->indexPropertyMap),

dataItemAttributes(typeHandler->dataItemAttributes),

lengthWritable(typeHandler->lengthWritable)

{

}

void SetInstanceTypeHandler(DynamicObject \* instance, bool hasChanged = true);

BOOL HasDataItem(ES5Array\* arr, uint32 index);

bool HasAnyDataItemNotInMap(ES5Array\* arr);

PropertyAttributes GetDataItemAttributes() const;

void SetDataItemSealed();

void SetDataItemFrozen();

static BOOL CantAssign(PropertyOperationFlags flags, ScriptContext\* scriptContext);

static BOOL CantExtend(PropertyOperationFlags flags, ScriptContext\* scriptContext);

BOOL IsAttributeSet(uint32 index, PropertyAttributes attr);

BOOL IsAttributeSet(DynamicObject\* instance, PropertyId propertyId, PropertyAttributes attr, BOOL& isNumericPropertyId);

BOOL UpdateAttribute(DynamicObject\* instance, PropertyId propertyId, PropertyAttributes attr, BOOL value, BOOL& isNumericPropertyId);

uint32 DeleteDownTo(ES5Array\* arr, uint32 first, PropertyOperationFlags propertyOperationFlags);

bool TryGetLastDataItemIndex(ES5Array\* arr, uint32 first, uint32\* lastIndex);

bool CanSetItemAt(ES5Array\* arr, uint32 index) const;

public:

// Create a new type handler for a future DynamicObject. This is for public usage. "initialCapacity" indicates desired slotCapacity, subject to alignment round up.

static ES5ArrayTypeHandlerBase\* New(Recycler \* recycler, int initialCapacity, uint16 inlineSlotCapacity, uint16 offsetOfInlineSlots);

void SetLengthWritable(bool writable);

BOOL HasItem(ES5Array\* arr, uint32 index);

BOOL SetItem(ES5Array\* arr, DynamicObject\* instance, uint32 index, Var value, PropertyOperationFlags flags);

BOOL SetItemWithAttributes(ES5Array\* arr, DynamicObject\* instance, uint32 index, Var value, PropertyAttributes attributes);

BOOL SetItemAttributes(ES5Array\* arr, DynamicObject\* instance, uint32 index, PropertyAttributes attributes);

BOOL SetItemAccessors(ES5Array\* arr, DynamicObject\* instance, uint32 index, Var getter, Var setter);

BOOL DeleteItem(ES5Array\* arr, DynamicObject\* instance, uint32 index, PropertyOperationFlags propertyOperationFlags);

BOOL GetItem(ES5Array\* arr, DynamicObject\* instance, Var originalInstance, uint32 index, Var\* value, ScriptContext\* requestContext);

DescriptorFlags GetItemSetter(ES5Array\* arr, DynamicObject\* instance, uint32 index, Var\* setterValue, ScriptContext\* requestContext);

BOOL GetItemAccessors(ES5Array\* arr, DynamicObject\* instance, uint32 index, Var\* getter, Var\* setter);

public:

virtual BOOL HasProperty(DynamicObject\* instance, PropertyId propertyId, bool \*noRedecl = nullptr) override;

virtual BOOL HasProperty(DynamicObject\* instance, JavascriptString\* propertyNameString) override;

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

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

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

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

virtual BOOL DeleteProperty(DynamicObject\* instance, PropertyId propertyId, PropertyOperationFlags flags) override;

virtual BOOL HasItem(DynamicObject\* instance, uint32 index) override;

virtual BOOL SetItem(DynamicObject\* instance, uint32 index, Var value, PropertyOperationFlags flags) override;

virtual BOOL SetItemWithAttributes(DynamicObject\* instance, uint32 index, Var value, PropertyAttributes attributes) override;

virtual BOOL SetItemAttributes(DynamicObject\* instance, uint32 index, PropertyAttributes attributes) override;

virtual BOOL SetItemAccessors(DynamicObject\* instance, uint32 index, Var getter, Var setter) override;

virtual BOOL DeleteItem(DynamicObject\* instance, uint32 index, PropertyOperationFlags flags) override;

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

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

virtual BOOL IsEnumerable(DynamicObject\* instance, PropertyId propertyId) override;

virtual BOOL IsWritable(DynamicObject\* instance, PropertyId propertyId) override;

virtual BOOL IsConfigurable(DynamicObject\* instance, PropertyId propertyId) override;

virtual BOOL SetEnumerable(DynamicObject\* instance, PropertyId propertyId, BOOL value) override;

virtual BOOL SetWritable(DynamicObject\* instance, PropertyId propertyId, BOOL value) override;

virtual BOOL SetConfigurable(DynamicObject\* instance, PropertyId propertyId, BOOL value) override;

virtual BOOL GetAccessors(DynamicObject\* instance, PropertyId propertyId, Var\* getter, Var\* setter) override;

virtual BOOL Seal(DynamicObject\* instance) override;

virtual BOOL IsSealed(DynamicObject\* instance) override;

virtual BOOL IsFrozen(DynamicObject\* instance) override;

virtual BOOL SetAttributes(DynamicObject\* instance, PropertyId propertyId, PropertyAttributes attributes) override;

virtual bool IsLengthWritable() const override;

virtual void SetLength(ES5Array\* arr, uint32 newLen, PropertyOperationFlags propertyOperationFlags) override;

virtual BOOL IsObjectArrayFrozen(ES5Array\* arr) override;

virtual BOOL IsItemEnumerable(ES5Array\* arr, uint32 index) override;

virtual BOOL IsValidDescriptorToken(void \* descriptorValidationToken) const override;

virtual uint32 GetNextDescriptor(uint32 key, IndexPropertyDescriptor\*\* descriptor, void \*\* descriptorValidationToken) override;

virtual BOOL GetDescriptor(uint32 index, Js::IndexPropertyDescriptor \*\*ppDescriptor) override;

virtual void SetIsPrototype(DynamicObject\* instance) override;

private:

virtual BOOL FreezeImpl(DynamicObject\* instance, bool isConvertedType) override;

virtual BigDictionaryTypeHandler\* NewBigDictionaryTypeHandler(Recycler\* recycler, int slotCapacity, uint16 inlineSlotCapacity, uint16 offsetOfInlineSlots) override;

};

}

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

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

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

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

#include "RuntimeTypePch.h"

namespace Js

{

JavascriptEnumerator::JavascriptEnumerator(ScriptContext\* scriptContext) : RecyclableObject(scriptContext->GetLibrary()->GetEnumeratorType())

{

Assert(scriptContext != NULL);

}

bool JavascriptEnumerator::Is(Var aValue)

{

return JavascriptOperators::GetTypeId(aValue) == TypeIds\_Enumerator;

}

JavascriptEnumerator\* JavascriptEnumerator::FromVar(Var aValue)

{

AssertMsg(Is(aValue), "Ensure var is actually a 'JavascriptEnumerator'");

return static\_cast<JavascriptEnumerator \*>(RecyclableObject::FromVar(aValue));

}

}

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

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

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

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

namespace Js {

class JavascriptEnumerator : public RecyclableObject

{

friend class CrossSite;

friend class ExternalObject;

protected:

DEFINE\_VTABLE\_CTOR\_ABSTRACT(JavascriptEnumerator, RecyclableObject);

virtual void MarshalToScriptContext(Js::ScriptContext \* scriptContext) = 0;

JavascriptEnumerator() { /\* Do nothing, needed by the vtable ctor for ForInObjectEnumeratorWrapper \*/ }

public:

JavascriptEnumerator(ScriptContext\* scriptContext);

//

// Returns item index for all nonnamed Enumerators

//optional override

//

virtual uint32 GetCurrentItemIndex() { return Constants::InvalidSourceIndex; }

//

// Returns the current index

//

virtual Var GetCurrentIndex() = 0;

//

// Returns the current value

//

virtual Var GetCurrentValue() = 0;

//

// Moves to next element

//

virtual BOOL MoveNext(PropertyAttributes\* attributes = nullptr) = 0;

//

// Sets the enumerator to its initial position

//

virtual void Reset() = 0;

//

// Moves to the next element and gets the current value.

// PropertyId: Sets the propertyId of the current value.

// In some cases, i.e. arrays, propertyId is not returned successfully.

// Returns: NULL if there are no more elements.

//

// Note: in the future we might want to enumerate specialPropertyIds

// If that code is added in this base class use JavaScriptRegExpEnumerator.h/cpp

// as a reference and then remove it. If you have already made the edits before

// seeing this comment please just consolidate the changes.

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

{

propertyId = Constants::NoProperty;

if (MoveNext(attributes))

{

Var currentIndex = GetCurrentIndex();

return currentIndex;

}

return NULL;

}

virtual Var GetCurrentBothAndMoveNext(PropertyId& propertyId, Var\* currentValueRef)

{

propertyId = Constants::NoProperty;

if (MoveNext())

{

Var currentIndex = GetCurrentIndex();

\*currentValueRef = GetCurrentValue();

return currentIndex;

}

return NULL;

}

virtual bool GetCurrentPropertyId(PropertyId \*propertyId)

{

\*propertyId = Constants::NoProperty;

return false;

};

virtual BOOL IsCrossSiteEnumerator()

{

return false;

}

static bool Is(Var aValue);

static JavascriptEnumerator\* FromVar(Var varValue);

};

}

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

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

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

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

#include "RuntimeTypePch.h"

namespace Js

{

void MissingPropertyTypeHandler::SetUndefinedPropertySlot(DynamicObject\* instance)

{

Var \* slots = reinterpret\_cast<Var\*>(reinterpret\_cast<size\_t>(instance) + sizeof(DynamicObject));

slots[0] = instance->GetLibrary()->GetUndefined();

}

MissingPropertyTypeHandler::MissingPropertyTypeHandler() :

DynamicTypeHandler(1, 1, (uint16)sizeof(DynamicObject)) {}

PropertyId MissingPropertyTypeHandler::GetPropertyId(ScriptContext\* scriptContext, PropertyIndex index)

{

return Constants::NoProperty;

}

PropertyId MissingPropertyTypeHandler::GetPropertyId(ScriptContext\* scriptContext, BigPropertyIndex index)

{

return Constants::NoProperty;

}

BOOL MissingPropertyTypeHandler::FindNextProperty(ScriptContext\* scriptContext, PropertyIndex& index, JavascriptString\*\* propertyStringName,

PropertyId\* propertyId, PropertyAttributes\* attributes, Type\* type, DynamicType \*typeToEnumerate, bool requireEnumerable, bool enumSymbols)

{

return FALSE;

}

PropertyIndex MissingPropertyTypeHandler::GetPropertyIndex(PropertyRecord const\* propertyRecord)

{

return 0;

}

bool MissingPropertyTypeHandler::GetPropertyEquivalenceInfo(PropertyRecord const\* propertyRecord, PropertyEquivalenceInfo& info)

{

info.slotIndex = Constants::NoSlot;

info.isWritable = false;

return false;

}

bool MissingPropertyTypeHandler::IsObjTypeSpecEquivalent(const Type\* type, const TypeEquivalenceRecord& record, uint& failedPropertyIndex)

{

failedPropertyIndex = 0;

return false;

}

bool MissingPropertyTypeHandler::IsObjTypeSpecEquivalent(const Type\* type, const EquivalentPropertyEntry \*entry)

{

return false;

}

BOOL MissingPropertyTypeHandler::HasProperty(DynamicObject\* instance, PropertyId propertyId, \_\_out\_opt bool \*noRedecl)

{

if (noRedecl != nullptr)

{

\*noRedecl = false;

}

return false;

}

BOOL MissingPropertyTypeHandler::HasProperty(DynamicObject\* instance, JavascriptString\* propertyNameString)

{

return false;

}

BOOL MissingPropertyTypeHandler::GetProperty(DynamicObject\* instance, Var originalInstance, PropertyId propertyId, Var\* value, PropertyValueInfo\* info, ScriptContext\* requestContext)

{

return false;

}

BOOL MissingPropertyTypeHandler::GetProperty(DynamicObject\* instance, Var originalInstance, JavascriptString\* propertyNameString, Var\* value, PropertyValueInfo\* info, ScriptContext\* requestContext)

{

return false;

}

BOOL MissingPropertyTypeHandler::SetProperty(DynamicObject\* instance, PropertyId propertyId, Var value, PropertyOperationFlags flags, PropertyValueInfo\* info)

{

Throw::FatalInternalError();

}

BOOL MissingPropertyTypeHandler::SetProperty(DynamicObject\* instance, JavascriptString\* propertyNameString, Var value, PropertyOperationFlags flags, PropertyValueInfo\* info)

{

Throw::FatalInternalError();

}

DescriptorFlags MissingPropertyTypeHandler::GetSetter(DynamicObject\* instance, PropertyId propertyId, Var\* setterValue, PropertyValueInfo\* info, ScriptContext\* requestContext)

{

PropertyValueInfo::SetNoCache(info, instance);

return None;

}

DescriptorFlags MissingPropertyTypeHandler::GetSetter(DynamicObject\* instance, JavascriptString\* propertyNameString, Var\* setterValue, PropertyValueInfo\* info, ScriptContext\* requestContext)

{

PropertyValueInfo::SetNoCache(info, instance);

return None;

}

BOOL MissingPropertyTypeHandler::DeleteProperty(DynamicObject\* instance, PropertyId propertyId, PropertyOperationFlags propertyOperationFlags)

{

Throw::FatalInternalError();

}

BOOL MissingPropertyTypeHandler::IsEnumerable(DynamicObject\* instance, PropertyId propertyId)

{

Throw::FatalInternalError();

}

BOOL MissingPropertyTypeHandler::IsWritable(DynamicObject\* instance, PropertyId propertyId)

{

Throw::FatalInternalError();

}

BOOL MissingPropertyTypeHandler::IsConfigurable(DynamicObject\* instance, PropertyId propertyId)

{

Throw::FatalInternalError();

}

BOOL MissingPropertyTypeHandler::SetEnumerable(DynamicObject\* instance, PropertyId propertyId, BOOL value)

{

Throw::FatalInternalError();

}

BOOL MissingPropertyTypeHandler::SetWritable(DynamicObject\* instance, PropertyId propertyId, BOOL value)

{

Throw::FatalInternalError();

}

BOOL MissingPropertyTypeHandler::SetConfigurable(DynamicObject\* instance, PropertyId propertyId, BOOL value)

{

Throw::FatalInternalError();

}

//

// Set an attribute bit. Return true if change is made.

//

BOOL MissingPropertyTypeHandler::SetAttribute(DynamicObject\* instance, int index, PropertyAttributes attribute)

{

Throw::FatalInternalError();

}

//

// Clear an attribute bit. Return true if change is made.

//

BOOL MissingPropertyTypeHandler::ClearAttribute(DynamicObject\* instance, int index, PropertyAttributes attribute)

{

Throw::FatalInternalError();

}

BOOL MissingPropertyTypeHandler::SetAccessors(DynamicObject\* instance, PropertyId propertyId, Var getter, Var setter, PropertyOperationFlags flags)

{

Throw::FatalInternalError();

}

BOOL MissingPropertyTypeHandler::PreventExtensions(DynamicObject\* instance)

{

Throw::FatalInternalError();

}

BOOL MissingPropertyTypeHandler::Seal(DynamicObject\* instance)

{

Throw::FatalInternalError();

}

BOOL MissingPropertyTypeHandler::FreezeImpl(DynamicObject\* instance, bool isConvertedType)

{

Throw::FatalInternalError();

}

BOOL MissingPropertyTypeHandler::SetPropertyWithAttributes(DynamicObject\* instance, PropertyId propertyId, Var value, PropertyAttributes attributes, PropertyValueInfo\* info, PropertyOperationFlags flags, SideEffects possibleSideEffects)

{

Throw::FatalInternalError();

}

BOOL MissingPropertyTypeHandler::SetAttributes(DynamicObject\* instance, PropertyId propertyId, PropertyAttributes attributes)

{

Throw::FatalInternalError();

}

BOOL MissingPropertyTypeHandler::GetAttributesWithPropertyIndex(DynamicObject \* instance, PropertyId propertyId, BigPropertyIndex index, PropertyAttributes \* attributes)

{

Throw::FatalInternalError();

}

BOOL MissingPropertyTypeHandler::AddProperty(DynamicObject\* instance, PropertyId propertyId, Var value, PropertyAttributes attributes, PropertyValueInfo\* info, PropertyOperationFlags flags, SideEffects possibleSideEffects)

{

Throw::FatalInternalError();

}

void MissingPropertyTypeHandler::SetAllPropertiesToUndefined(DynamicObject\* instance, bool invalidateFixedFields)

{

Throw::FatalInternalError();

}

void MissingPropertyTypeHandler::MarshalAllPropertiesToScriptContext(DynamicObject\* instance, ScriptContext\* targetScriptContext, bool invalidateFixedFields)

{

Throw::FatalInternalError();

}

DynamicTypeHandler\* MissingPropertyTypeHandler::ConvertToTypeWithItemAttributes(DynamicObject\* instance)

{

Throw::FatalInternalError();

}

void MissingPropertyTypeHandler::SetIsPrototype(DynamicObject\* instance)

{

Throw::FatalInternalError();

}

#if DBG

bool MissingPropertyTypeHandler::CanStorePropertyValueDirectly(const DynamicObject\* instance, PropertyId propertyId, bool allowLetConst)

{

Throw::FatalInternalError();

}

#endif

}

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

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

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

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

#pragma once

namespace Js

{

class MissingPropertyTypeHandler : public DynamicTypeHandler

{

public:

DEFINE\_GETCPPNAME();

MissingPropertyTypeHandler();

static void SetUndefinedPropertySlot(DynamicObject\* instance);

protected:

DEFINE\_VTABLE\_CTOR\_NO\_REGISTER(MissingPropertyTypeHandler, DynamicTypeHandler);

virtual BOOL IsLockable() const override { return true; }

virtual BOOL IsSharable() const override { return true; }

virtual int GetPropertyCount() override { return 0; }

virtual PropertyId GetPropertyId(ScriptContext\* scriptContext, PropertyIndex index) override;

virtual PropertyId GetPropertyId(ScriptContext\* scriptContext, BigPropertyIndex index) override;

virtual BOOL FindNextProperty(ScriptContext\* scriptContext, PropertyIndex& index, JavascriptString\*\* propertyString,

PropertyId\* propertyId, PropertyAttributes\* attributes, Type\* type, DynamicType \*typeToEnumerate, bool requireEnumerable, bool enumSymbols = false) override;

virtual PropertyIndex GetPropertyIndex(PropertyRecord const\* propertyRecord) override;

virtual bool GetPropertyEquivalenceInfo(PropertyRecord const\* propertyRecord, PropertyEquivalenceInfo& info) override;

virtual bool IsObjTypeSpecEquivalent(const Type\* type, const TypeEquivalenceRecord& record, uint& failedPropertyIndex) override;

virtual bool IsObjTypeSpecEquivalent(const Type\* type, const EquivalentPropertyEntry\* entry) override;

virtual BOOL HasProperty(DynamicObject\* instance, PropertyId propertyId, \_\_out\_opt bool \*noRedecl = nullptr) override;

virtual BOOL HasProperty(DynamicObject\* instance, JavascriptString\* propertyNameString) override;

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

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

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

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

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

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

virtual BOOL DeleteProperty(DynamicObject\* instance, PropertyId propertyId, PropertyOperationFlags flags) override;

virtual BOOL IsEnumerable(DynamicObject\* instance, PropertyId propertyId) override;

virtual BOOL IsWritable(DynamicObject\* instance, PropertyId propertyId) override;

virtual BOOL IsConfigurable(DynamicObject\* instance, PropertyId propertyId) override;

virtual BOOL SetEnumerable(DynamicObject\* instance, PropertyId propertyId, BOOL value) override;

virtual BOOL SetWritable(DynamicObject\* instance, PropertyId propertyId, BOOL value) override;

virtual BOOL SetConfigurable(DynamicObject\* instance, PropertyId propertyId, BOOL value) override;

virtual BOOL SetAccessors(DynamicObject\* instance, PropertyId propertyId, Var getter, Var setter, PropertyOperationFlags flags = PropertyOperation\_None) override;

virtual BOOL PreventExtensions(DynamicObject \*instance) override;

virtual BOOL Seal(DynamicObject\* instance) override;

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

virtual BOOL SetAttributes(DynamicObject\* instance, PropertyId propertyId, PropertyAttributes attributes) override;

virtual BOOL GetAttributesWithPropertyIndex(DynamicObject \* instance, PropertyId propertyId, BigPropertyIndex index, PropertyAttributes \* attributes) override;

virtual void SetAllPropertiesToUndefined(DynamicObject\* instance, bool invalidateFixedFields) override;

virtual void MarshalAllPropertiesToScriptContext(DynamicObject\* instance, ScriptContext\* targetScriptContext, bool invalidateFixedFields) override;

virtual DynamicTypeHandler\* ConvertToTypeWithItemAttributes(DynamicObject\* instance) override;

virtual void SetIsPrototype(DynamicObject\* instance) override;

#if DBG

virtual bool SupportsPrototypeInstances() const { Assert(false); return false; }

virtual bool CanStorePropertyValueDirectly(const DynamicObject\* instance, PropertyId propertyId, bool allowLetConst) override;

#endif

private:

template <typename T>

T\* ConvertToTypeHandler(DynamicObject\* instance);

DictionaryTypeHandler\* ConvertToDictionaryType(DynamicObject\* instance);

SimpleDictionaryTypeHandler\* ConvertToSimpleDictionaryType(DynamicObject\* instance);

ES5ArrayTypeHandler\* ConvertToES5ArrayType(DynamicObject\* instance);

template<size\_t size>

SimpleTypeHandler<size>\* ConvertToNonSharedSimpleType(DynamicObject \* instance);

BOOL GetDescriptor(PropertyId propertyId, int \* index);

BOOL SetAttribute(DynamicObject\* instance, int index, PropertyAttributes attribute);

BOOL ClearAttribute(DynamicObject\* instance, int index, PropertyAttributes attribute);

BOOL AddProperty(DynamicObject\* instance, PropertyId propertyId, Var value, PropertyAttributes attributes, PropertyValueInfo\* info, PropertyOperationFlags flags, SideEffects possibleSideEffects);

virtual BOOL FreezeImpl(DynamicObject\* instance, bool isConvertedType) override;

};

}

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

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

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

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

#include "RuntimeTypePch.h"

#include "Types\NullTypeHandler.h"

#include "Types\SimpleTypeHandler.h"

namespace Js

{

int NullTypeHandlerBase::GetPropertyCount()

{

return 0;

}

PropertyId NullTypeHandlerBase::GetPropertyId(ScriptContext\* scriptContext, PropertyIndex index)

{

return Constants::NoProperty;

}

PropertyId NullTypeHandlerBase::GetPropertyId(ScriptContext\* scriptContext, BigPropertyIndex index)

{

return Constants::NoProperty;

}

BOOL NullTypeHandlerBase::FindNextProperty(ScriptContext\* scriptContext, PropertyIndex& index, JavascriptString\*\* propertyString, PropertyId\* propertyId,

PropertyAttributes\* attributes, Type\* type, DynamicType \*typeToEnumerate, bool requireEnumerable, bool enumSymbols)

{

Assert(propertyString);

Assert(propertyId);

Assert(type);

return FALSE;

}

PropertyIndex NullTypeHandlerBase::GetPropertyIndex(PropertyRecord const\* propertyRecord)

{

return Constants::NoSlot;

}

bool NullTypeHandlerBase::GetPropertyEquivalenceInfo(PropertyRecord const\* propertyRecord, PropertyEquivalenceInfo& info)

{

info.slotIndex = Constants::NoSlot;

info.isAuxSlot = false;

info.isWritable = false;

return false;

}

bool NullTypeHandlerBase::IsObjTypeSpecEquivalent(const Type\* type, const TypeEquivalenceRecord& record, uint& failedPropertyIndex)

{

uint propertyCount = record.propertyCount;

EquivalentPropertyEntry\* properties = record.properties;

for (uint pi = 0; pi < propertyCount; pi++)

{

const EquivalentPropertyEntry\* refInfo = &properties[pi];

if (!this->NullTypeHandlerBase::IsObjTypeSpecEquivalent(type, refInfo))

{

return false;

}

}

return true;

}

bool NullTypeHandlerBase::IsObjTypeSpecEquivalent(const Type\* type, const EquivalentPropertyEntry \*entry)

{

return entry->slotIndex == Constants::NoSlot && !entry->mustBeWritable;

}

BOOL NullTypeHandlerBase::HasProperty(DynamicObject\* instance, PropertyId propertyId, \_\_out\_opt bool \*noRedecl)

{

// Check numeric propertyId only if objectArray is available

uint32 indexVal;

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

if (noRedecl != nullptr)

{

\*noRedecl = false;

}

if (instance->HasObjectArray() && scriptContext->IsNumericPropertyId(propertyId, &indexVal))

{

return DynamicTypeHandler::HasItem(instance, indexVal);

}

return false;

}

BOOL NullTypeHandlerBase::HasProperty(DynamicObject\* instance, JavascriptString\* propertyNameString)

{

PropertyRecord const\* propertyRecord;

instance->GetScriptContext()->GetOrAddPropertyRecord(propertyNameString->GetString(), propertyNameString->GetLength(), &propertyRecord);

return NullTypeHandlerBase::HasProperty(instance, propertyRecord->GetPropertyId());

}

BOOL NullTypeHandlerBase::GetProperty(DynamicObject\* instance, Var originalInstance, PropertyId propertyId, Var\* value, PropertyValueInfo\* info, ScriptContext\* requestContext)

{

// Check numeric propertyId only if objectArray is available

uint32 indexVal;

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

if (instance->HasObjectArray() && scriptContext->IsNumericPropertyId(propertyId, &indexVal))

{

return DynamicTypeHandler::GetItem(instance, originalInstance, indexVal, value, requestContext);

}

return false;

}

BOOL NullTypeHandlerBase::GetProperty(DynamicObject\* instance, Var originalInstance, JavascriptString\* propertyNameString, Var\* value, PropertyValueInfo\* info, ScriptContext\* requestContext)

{

PropertyRecord const\* propertyRecord;

instance->GetScriptContext()->GetOrAddPropertyRecord(propertyNameString->GetString(), propertyNameString->GetLength(), &propertyRecord);

return NullTypeHandlerBase::GetProperty(instance, originalInstance, propertyRecord->GetPropertyId(), value, info, requestContext);

}

BOOL NullTypeHandlerBase::SetProperty(DynamicObject\* instance, PropertyId propertyId, Var value, PropertyOperationFlags flags, PropertyValueInfo\* info)

{

return ConvertToSimpleType(instance)->SetProperty(instance, propertyId, value, flags, info);

}

BOOL NullTypeHandlerBase::SetProperty(DynamicObject\* instance, JavascriptString\* propertyNameString, Var value, PropertyOperationFlags flags, PropertyValueInfo\* info)

{

return ConvertToSimpleType(instance)->SetProperty(instance, propertyNameString, value, flags, info);

}

BOOL NullTypeHandlerBase::AddProperty(DynamicObject\* instance, PropertyId propertyId, Var value, PropertyAttributes attributes, PropertyValueInfo\* info, PropertyOperationFlags flags, SideEffects possibleSideEffects)

{

if (this->isPrototype && (ChangeTypeOnProto() || (GetIsShared() && IsolatePrototypes())))

{

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

return ConvertToSimpleDictionaryType(instance)->AddProperty(instance, scriptContext->GetPropertyName(propertyId), value, attributes, info, flags, possibleSideEffects);

}

else

{

return ConvertToSimpleType(instance)->AddProperty(instance, propertyId, value, attributes, info, flags, possibleSideEffects);

}

}

BOOL NullTypeHandlerBase::DeleteProperty(DynamicObject\* instance, PropertyId propertyId, PropertyOperationFlags flags)

{

// Check numeric propertyId only if objectArray is available

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

uint32 indexVal;

if (instance->HasObjectArray() && scriptContext->IsNumericPropertyId(propertyId, &indexVal))

{

return DynamicTypeHandler::DeleteItem(instance, indexVal, flags);

}

return true;

}

BOOL NullTypeHandlerBase::IsEnumerable(DynamicObject\* instance, PropertyId propertyId)

{

return true;

}

BOOL NullTypeHandlerBase::IsWritable(DynamicObject\* instance, PropertyId propertyId)

{

return true;

}

BOOL NullTypeHandlerBase::IsConfigurable(DynamicObject\* instance, PropertyId propertyId)

{

return true;

}

BOOL NullTypeHandlerBase::SetEnumerable(DynamicObject\* instance, PropertyId propertyId, BOOL value)

{

return false;

}

BOOL NullTypeHandlerBase::SetWritable(DynamicObject\* instance, PropertyId propertyId, BOOL value)

{

return false;

}

BOOL NullTypeHandlerBase::SetConfigurable(DynamicObject\* instance, PropertyId propertyId, BOOL value)

{

return false;

}

BOOL NullTypeHandlerBase::SetAccessors(DynamicObject\* instance, PropertyId propertyId, Var getter, Var setter, PropertyOperationFlags flags)

{

return ConvertToDictionaryType(instance)->SetAccessors(instance, propertyId, getter, setter, flags);

}

BOOL NullTypeHandlerBase::PreventExtensions(DynamicObject\* instance)

{

return ConvertToDictionaryType(instance)->PreventExtensions(instance);

}

BOOL NullTypeHandlerBase::Seal(DynamicObject\* instance)

{

return ConvertToDictionaryType(instance)->Seal(instance);

}

BOOL NullTypeHandlerBase::FreezeImpl(DynamicObject\* instance, bool isConvertedType)

{

return ConvertToDictionaryType(instance)->Freeze(instance, true);

}

template <typename T>

T\* NullTypeHandlerBase::ConvertToTypeHandler(DynamicObject\* instance)

{

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

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

T \* newTypeHandler = RecyclerNew(recycler, T, recycler);

Assert((newTypeHandler->GetFlags() & IsPrototypeFlag) == 0);

// EnsureSlots before updating the type handler and instance, as EnsureSlots allocates and may throw.

instance->EnsureSlots(0, newTypeHandler->GetSlotCapacity(), scriptContext, newTypeHandler);

Assert(((this->GetFlags() & IsPrototypeFlag) != 0) == this->isPrototype);

newTypeHandler->SetFlags(IsPrototypeFlag, this->GetFlags());

newTypeHandler->SetPropertyTypes(PropertyTypesWritableDataOnly | PropertyTypesWritableDataOnlyDetection | PropertyTypesInlineSlotCapacityLocked, this->GetPropertyTypes());

if (instance->HasReadOnlyPropertiesInvisibleToTypeHandler())

{

newTypeHandler->ClearHasOnlyWritableDataProperties();

}

newTypeHandler->SetInstanceTypeHandler(instance);

return newTypeHandler;

}

SimpleTypeHandler<1>\* NullTypeHandlerBase::ConvertToSimpleType(DynamicObject\* instance)

{

SimpleTypeHandler<1>\* newTypeHandler = ConvertToTypeHandler<SimpleTypeHandler<1>>(instance);

#ifdef PROFILE\_TYPES

instance->GetScriptContext()->convertNullToSimpleCount++;

#endif

return newTypeHandler;

}

SimpleDictionaryTypeHandler \* NullTypeHandlerBase::ConvertToSimpleDictionaryType(DynamicObject \* instance)

{

SimpleDictionaryTypeHandler\* newTypeHandler = ConvertToTypeHandler<SimpleDictionaryTypeHandler>(instance);

#ifdef PROFILE\_TYPES

instance->GetScriptContext()->convertNullToSimpleDictionaryCount++;

#endif

return newTypeHandler;

}

DictionaryTypeHandler \* NullTypeHandlerBase::ConvertToDictionaryType(DynamicObject \* instance)

{

DictionaryTypeHandler\* newTypeHandler = ConvertToTypeHandler<DictionaryTypeHandler>(instance);

#ifdef PROFILE\_TYPES

instance->GetScriptContext()->convertNullToDictionaryCount++;

#endif

return newTypeHandler;

}

ES5ArrayTypeHandler\* NullTypeHandlerBase::ConvertToES5ArrayType(DynamicObject \* instance)

{

ES5ArrayTypeHandler\* newTypeHandler = ConvertToTypeHandler<ES5ArrayTypeHandler>(instance);

#ifdef PROFILE\_TYPES

instance->GetScriptContext()->convertNullToDictionaryCount++;

#endif

return newTypeHandler;

}

BOOL NullTypeHandlerBase::SetPropertyWithAttributes(DynamicObject\* instance, PropertyId propertyId, Var value, PropertyAttributes attributes, PropertyValueInfo\* info, PropertyOperationFlags flags, SideEffects possibleSideEffects)

{

// Always check numeric propertyId. May create objectArray.

uint32 indexVal;

if (instance->GetScriptContext()->IsNumericPropertyId(propertyId, &indexVal))

{

return NullTypeHandlerBase::SetItemWithAttributes(instance, indexVal, value, attributes);

}

return this->AddProperty(instance, propertyId, value, attributes, info, flags, possibleSideEffects);

}

BOOL NullTypeHandlerBase::SetAttributes(DynamicObject\* instance, PropertyId propertyId, PropertyAttributes attributes)

{

return false;

}

BOOL NullTypeHandlerBase::GetAttributesWithPropertyIndex(DynamicObject \* instance, PropertyId propertyId, BigPropertyIndex index, PropertyAttributes \* attributes)

{

return false;

}

DynamicTypeHandler\* NullTypeHandlerBase::ConvertToTypeWithItemAttributes(DynamicObject\* instance)

{

return JavascriptArray::Is(instance) ?

ConvertToES5ArrayType(instance) : ConvertToDictionaryType(instance);

}

void NullTypeHandlerBase::SetIsPrototype(DynamicObject\* instance)

{

if (!this->isPrototype)

{

// We don't force a type transition even when ChangeTypeOnProto() == true, because objects with NullTypeHandlers don't

// have any properties, so there is nothing to invalidate. Types with NullTypeHandlers also aren't cached in typeWithoutProperty

// caches, so there will be no fast property add path that could skip prototype cache invalidation.

NullTypeHandler<true>\* protoTypeHandler = NullTypeHandler<true>::GetDefaultInstance();

AssertMsg(protoTypeHandler->GetFlags() == (GetFlags() | IsPrototypeFlag), "Why did we change the flags of a NullTypeHandler?");

Assert(this->GetIsInlineSlotCapacityLocked() == protoTypeHandler->GetIsInlineSlotCapacityLocked());

protoTypeHandler->SetPropertyTypes(PropertyTypesWritableDataOnly | PropertyTypesWritableDataOnlyDetection, GetPropertyTypes());

SetInstanceTypeHandler(instance, protoTypeHandler);

}

}

template<bool IsPrototypeTemplate>

NullTypeHandler<IsPrototypeTemplate> NullTypeHandler<IsPrototypeTemplate>::defaultInstance;

template class NullTypeHandler<false>;

template class NullTypeHandler<true>;

}

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

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

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

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

#pragma once

namespace Js

{

class NullTypeHandlerBase : public DynamicTypeHandler

{

protected:

NullTypeHandlerBase(bool isPrototype) :

DynamicTypeHandler(0, 0, 0, DefaultFlags | IsLockedFlag | MayBecomeSharedFlag | IsSharedFlag | (isPrototype ? IsPrototypeFlag : 0)),

isPrototype(isPrototype) {}

DEFINE\_VTABLE\_CTOR\_NO\_REGISTER(NullTypeHandlerBase, DynamicTypeHandler);

private:

bool isPrototype;

public:

DEFINE\_GETCPPNAME();

public:

virtual BOOL IsLockable() const override { return true; }

virtual BOOL IsSharable() const override { return true; }

virtual int GetPropertyCount() override;

virtual PropertyId GetPropertyId(ScriptContext\* scriptContext, PropertyIndex index) override;

virtual PropertyId GetPropertyId(ScriptContext\* scriptContext, BigPropertyIndex index) override;

virtual BOOL FindNextProperty(ScriptContext\* scriptContext, PropertyIndex& index, JavascriptString\*\* propertyString,

PropertyId\* propertyId, PropertyAttributes\* attributes, Type\* type, DynamicType \*typeToEnumerate, bool requireEnumerable, bool enumSymbols = false) override;

virtual PropertyIndex GetPropertyIndex(PropertyRecord const\* propertyRecord) override;

virtual bool GetPropertyEquivalenceInfo(PropertyRecord const\* propertyRecord, PropertyEquivalenceInfo& info) override;

virtual bool IsObjTypeSpecEquivalent(const Type\* type, const TypeEquivalenceRecord& record, uint& failedPropertyIndex) override;

virtual bool IsObjTypeSpecEquivalent(const Type\* type, const EquivalentPropertyEntry\* entry) override;

virtual BOOL HasProperty(DynamicObject\* instance, PropertyId propertyId, \_\_out\_opt bool \*noRedecl = nullptr) override;

virtual BOOL HasProperty(DynamicObject\* instance, JavascriptString\* propertyNameString) override;

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

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

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

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

virtual BOOL DeleteProperty(DynamicObject\* instance, PropertyId propertyId, PropertyOperationFlags flags) override;

virtual BOOL IsEnumerable(DynamicObject\* instance, PropertyId propertyId) override;

virtual BOOL IsWritable(DynamicObject\* instance, PropertyId propertyId) override;

virtual BOOL IsConfigurable(DynamicObject\* instance, PropertyId propertyId) override;

virtual BOOL SetEnumerable(DynamicObject\* instance, PropertyId propertyId, BOOL value) override;

virtual BOOL SetWritable(DynamicObject\* instance, PropertyId propertyId, BOOL value) override;

virtual BOOL SetConfigurable(DynamicObject\* instance, PropertyId propertyId, BOOL value) override;

virtual BOOL SetAccessors(DynamicObject\* instance, PropertyId propertyId, Var getter, Var setter, PropertyOperationFlags flags = PropertyOperation\_None) override;

virtual BOOL PreventExtensions(DynamicObject \*instance) override;

virtual BOOL Seal(DynamicObject\* instance) override;

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

virtual BOOL SetAttributes(DynamicObject\* instance, PropertyId propertyId, PropertyAttributes attributes) override;

virtual BOOL GetAttributesWithPropertyIndex(DynamicObject \* instance, PropertyId propertyId, BigPropertyIndex index, PropertyAttributes \* attributes) override;

virtual void SetAllPropertiesToUndefined(DynamicObject\* instance, bool invalidateFixedFields) override {};

virtual void MarshalAllPropertiesToScriptContext(DynamicObject\* instance, ScriptContext\* targetScriptContext, bool invalidateFixedFields) override {};

virtual DynamicTypeHandler\* ConvertToTypeWithItemAttributes(DynamicObject\* instance) override;

virtual void SetIsPrototype(DynamicObject\* instance) override;

#if DBG

virtual bool SupportsPrototypeInstances() const override { return this->isPrototype; }

virtual bool RespectsIsolatePrototypes() const { return false; }

virtual bool RespectsChangeTypeOnProto() const { return false; }

#endif

private:

template <typename T>

T\* ConvertToTypeHandler(DynamicObject\* instance);

SimpleTypeHandler<1>\* ConvertToSimpleType(DynamicObject\* instance);

SimpleDictionaryTypeHandler \* ConvertToSimpleDictionaryType(DynamicObject\* instance);

DictionaryTypeHandler \* ConvertToDictionaryType(DynamicObject\* instance);

ES5ArrayTypeHandler \* ConvertToES5ArrayType(DynamicObject\* instance);

BOOL AddProperty(DynamicObject\* instance, PropertyId propertyId, Var value, PropertyAttributes attributes, PropertyValueInfo\* info, PropertyOperationFlags flags, SideEffects possibleSideEffects);

virtual BOOL FreezeImpl(DynamicObject\* instance, bool isConvertedType) override;

};

template <bool IsPrototypeTemplate>

class NullTypeHandler : public NullTypeHandlerBase

{

public:

DEFINE\_GETCPPNAME();

private:

NullTypeHandler() : NullTypeHandlerBase(IsPrototypeTemplate) {}

DEFINE\_VTABLE\_CTOR\_NO\_REGISTER(NullTypeHandler, NullTypeHandlerBase);

static NullTypeHandler defaultInstance;

public:

static NullTypeHandler \* GetDefaultInstance() { return &defaultInstance; }

};

}

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

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

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

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

#include "RuntimeTypePch.h"

namespace Js

{

PathTypeHandlerBase::PathTypeHandlerBase(TypePath\* typePath, uint16 pathLength, const PropertyIndex slotCapacity, uint16 inlineSlotCapacity, uint16 offsetOfInlineSlots, bool isLocked, bool isShared, DynamicType\* predecessorType) :

DynamicTypeHandler(slotCapacity, inlineSlotCapacity, offsetOfInlineSlots, DefaultFlags | (isLocked ? IsLockedFlag : 0) | (isShared ? (MayBecomeSharedFlag | IsSharedFlag) : 0)),

typePath(typePath),

predecessorType(predecessorType)

{

Assert(pathLength <= slotCapacity);

Assert(inlineSlotCapacity <= slotCapacity);

SetUnusedBytesValue(pathLength);

isNotPathTypeHandlerOrHasUserDefinedCtor = predecessorType == nullptr ? false : predecessorType->GetTypeHandler()->GetIsNotPathTypeHandlerOrHasUserDefinedCtor();

}

int PathTypeHandlerBase::GetPropertyCount()

{

return GetPathLength();

}

PropertyId PathTypeHandlerBase::GetPropertyId(ScriptContext\* scriptContext, PropertyIndex index)

{

if (index < GetPathLength())

{

return typePath->GetPropertyId(index)->GetPropertyId();

}

else

{

return Constants::NoProperty;

}

}

PropertyId PathTypeHandlerBase::GetPropertyId(ScriptContext\* scriptContext, BigPropertyIndex index)

{

if (index < GetPathLength())

{

return typePath->GetPropertyId(index)->GetPropertyId();

}

else

{

return Constants::NoProperty;

}

}

BOOL PathTypeHandlerBase::FindNextProperty(ScriptContext\* scriptContext, PropertyIndex& index, JavascriptString\*\* propertyStringName, PropertyId\* propertyId,

PropertyAttributes\* attributes, Type\* type, DynamicType \*typeToEnumerate, bool requireEnumerable, bool enumSymbols)

{

Assert(propertyStringName);

Assert(propertyId);

Assert(type);

for (; index < GetPathLength(); ++index)

{

const PropertyRecord\* propertyRecord = typePath->GetPropertyId(index);

// Skip this property if it is a symbol and we are not including symbol properties

if (!enumSymbols && propertyRecord->IsSymbol())

{

continue;

}

if (attributes)

{

\*attributes = PropertyEnumerable;

}

\*propertyId = propertyRecord->GetPropertyId();

PropertyString\* propertyString = type->GetScriptContext()->GetPropertyString(\*propertyId);

\*propertyStringName = propertyString;

uint16 inlineOrAuxSlotIndex;

bool isInlineSlot;

PropertyIndexToInlineOrAuxSlotIndex(index, &inlineOrAuxSlotIndex, &isInlineSlot);

propertyString->UpdateCache(type, inlineOrAuxSlotIndex, isInlineSlot,

!FixPropsOnPathTypes() || (this->GetPathLength() < this->typePath->GetMaxInitializedLength() && !this->typePath->GetIsFixedFieldAt(index, this->GetPathLength())));

return TRUE;

}

return FALSE;

}

PropertyIndex PathTypeHandlerBase::GetPropertyIndex(const PropertyRecord\* propertyRecord)

{

return typePath->LookupInline(propertyRecord->GetPropertyId(), GetPathLength());

}

PropertyIndex PathTypeHandlerBase::GetPropertyIndex(PropertyId propertyId)

{

return typePath->LookupInline(propertyId, GetPathLength());

}

bool PathTypeHandlerBase::GetPropertyEquivalenceInfo(PropertyRecord const\* propertyRecord, PropertyEquivalenceInfo& info)

{

Js::PropertyIndex absSlotIndex = typePath->LookupInline(propertyRecord->GetPropertyId(), GetPathLength());

info.slotIndex = AdjustSlotIndexForInlineSlots(absSlotIndex);

info.isAuxSlot = absSlotIndex >= this->inlineSlotCapacity;

info.isWritable = info.slotIndex != Constants::NoSlot;

return info.slotIndex != Constants::NoSlot;

}

bool PathTypeHandlerBase::IsObjTypeSpecEquivalent(const Type\* type, const TypeEquivalenceRecord& record, uint& failedPropertyIndex)

{

uint propertyCount = record.propertyCount;

Js::EquivalentPropertyEntry\* properties = record.properties;

for (uint pi = 0; pi < propertyCount; pi++)

{

const EquivalentPropertyEntry\* entry = &properties[pi];

if (!this->PathTypeHandlerBase::IsObjTypeSpecEquivalent(type, entry))

{

failedPropertyIndex = pi;

return false;

}

}

return true;

}

bool PathTypeHandlerBase::IsObjTypeSpecEquivalent(const Type\* type, const EquivalentPropertyEntry \*entry)

{

Js::PropertyIndex absSlotIndex = typePath->LookupInline(entry->propertyId, GetPathLength());

if (absSlotIndex != Constants::NoSlot)

{

Js::PropertyIndex relSlotIndex = AdjustValidSlotIndexForInlineSlots(absSlotIndex);

if (relSlotIndex != entry->slotIndex || ((absSlotIndex >= GetInlineSlotCapacity()) != entry->isAuxSlot))

{

return false;

}

int maxInitializedLength = this->typePath->GetMaxInitializedLength();

if (entry->mustBeWritable && FixPropsOnPathTypes() && (absSlotIndex >= maxInitializedLength || this->typePath->GetIsFixedFieldAt(absSlotIndex, this->GetPathLength())))

{

return false;

}

}

else

{

if (entry->slotIndex != Constants::NoSlot || entry->mustBeWritable)

{

return false;

}

}

return true;

}

BOOL PathTypeHandlerBase::HasProperty(DynamicObject\* instance, PropertyId propertyId, \_\_out\_opt bool \*noRedecl)

{

uint32 indexVal;

if (noRedecl != nullptr)

{

\*noRedecl = false;

}

if (PathTypeHandlerBase::GetPropertyIndex(propertyId) != Constants::NoSlot)

{

return true;

}

// Check numeric propertyId only if objectArray is available

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

if (instance->HasObjectArray() && scriptContext->IsNumericPropertyId(propertyId, &indexVal))

{

return PathTypeHandlerBase::HasItem(instance, indexVal);

}

return false;

}

BOOL PathTypeHandlerBase::HasProperty(DynamicObject\* instance, JavascriptString\* propertyNameString)

{

// Consider: Implement actual string hash lookup

PropertyRecord const\* propertyRecord;

instance->GetScriptContext()->GetOrAddPropertyRecord(propertyNameString->GetString(), propertyNameString->GetLength(), &propertyRecord);

return PathTypeHandlerBase::HasProperty(instance, propertyRecord->GetPropertyId());

}

BOOL PathTypeHandlerBase::GetProperty(DynamicObject\* instance, Var originalInstance, PropertyId propertyId, Var\* value, PropertyValueInfo\* info, ScriptContext\* requestContext)

{

PropertyIndex index = typePath->LookupInline(propertyId, GetPathLength());

if (index != Constants::NoSlot)

{

\*value = instance->GetSlot(index);

PropertyValueInfo::Set(info, instance, index);

if (FixPropsOnPathTypes() && (index >= this->typePath->GetMaxInitializedLength() || this->typePath->GetIsFixedFieldAt(index, GetPathLength())))

{

PropertyValueInfo::DisableStoreFieldCache(info);

}

return true;

}

// Check numeric propertyId only if objectArray available

uint32 indexVal;

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

if (instance->HasObjectArray() && scriptContext->IsNumericPropertyId(propertyId, &indexVal))

{

return PathTypeHandlerBase::GetItem(instance, originalInstance, indexVal, value, requestContext);

}

return false;

}

BOOL PathTypeHandlerBase::GetProperty(DynamicObject\* instance, Var originalInstance, JavascriptString\* propertyNameString, Var\* value, PropertyValueInfo\* info, ScriptContext\* requestContext)

{

// Consider: Implement actual string hash lookup

Assert(requestContext);

PropertyRecord const\* propertyRecord;

wchar\_t const \* propertyName = propertyNameString->GetString();

charcount\_t const propertyNameLength = propertyNameString->GetLength();

if (instance->HasObjectArray())

{

requestContext->GetOrAddPropertyRecord(propertyName, propertyNameLength, &propertyRecord);

}

else

{

requestContext->FindPropertyRecord(propertyName, propertyNameLength, &propertyRecord);

if (propertyRecord == nullptr)

{

return false;

}

}

return PathTypeHandlerBase::GetProperty(instance, originalInstance, propertyRecord->GetPropertyId(), value, info, requestContext);

}

BOOL PathTypeHandlerBase::SetProperty(DynamicObject\* instance, PropertyId propertyId, Var value, PropertyOperationFlags flags, PropertyValueInfo\* info)

{

return SetPropertyInternal(instance, propertyId, value, info, flags, SideEffects\_Any);

}

BOOL PathTypeHandlerBase::SetProperty(DynamicObject\* instance, JavascriptString\* propertyNameString, Var value, PropertyOperationFlags flags, PropertyValueInfo\* info)

{

// Consider: Implement actual string hash lookup

PropertyRecord const\* propertyRecord;

instance->GetScriptContext()->GetOrAddPropertyRecord(propertyNameString->GetString(), propertyNameString->GetLength(), &propertyRecord);

return PathTypeHandlerBase::SetProperty(instance, propertyRecord->GetPropertyId(), value, flags, info);

}

BOOL PathTypeHandlerBase::SetPropertyInternal(DynamicObject\* instance, PropertyId propertyId, Var value, PropertyValueInfo\* info, PropertyOperationFlags flags, SideEffects possibleSideEffects)

{

// Path type handler doesn't support pre-initialization (PropertyOperation\_PreInit). Pre-initialized properties

// will get marked as fixed when pre-initialized and then as non-fixed when their actual values are set.

Assert(value != nullptr || IsInternalPropertyId(propertyId));

PropertyIndex index = PathTypeHandlerBase::GetPropertyIndex(propertyId);

if (index != Constants::NoSlot)

{

// If type is shared then the handler must be shared as well. This is a weaker invariant than in AddPropertyInternal,

// because the type coming in here may be the result of DynamicObject::ChangeType(). In that case the handler may have

// already been shared, but the newly created type isn't - and likely never will be - shared (is typically unreachable).

// In CacheOperators::CachePropertyWrite we ensure that we never cache property adds for types that aren't shared.

Assert(!instance->GetDynamicType()->GetIsShared() || GetIsShared());

Assert(instance->GetDynamicType()->GetIsShared() == GetIsShared());

bool populateInlineCache = GetIsShared() ||

ProcessFixedFieldChange(instance, propertyId, index, value, (flags & PropertyOperation\_NonFixedValue) != 0);

SetSlotUnchecked(instance, index, value);

if (populateInlineCache)

{

Assert((instance->GetDynamicType()->GetIsShared()) || (FixPropsOnPathTypes() && instance->GetDynamicType()->GetTypeHandler()->GetIsOrMayBecomeShared()));

// Can't assert the following. With NewScObject we can jump to the type handler at the tip (where the singleton is),

// even though we haven't yet initialized the properties all the way to the tip, and we don't want to kill

// the singleton in that case yet. It's basically a transient inconsistent state, but we have to live with it.

// The user's code will never see the object in this state.

//Assert(!instance->GetTypeHandler()->HasSingletonInstance());

PropertyValueInfo::Set(info, instance, index);

}

else

{

PropertyValueInfo::SetNoCache(info, instance);

}

SetPropertyUpdateSideEffect(instance, propertyId, value, possibleSideEffects);

return true;

}

// Always check numeric propertyId. This may create an objectArray.

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

uint32 indexVal;

if (scriptContext->IsNumericPropertyId(propertyId, &indexVal))

{

return PathTypeHandlerBase::SetItem(instance, indexVal, value, PropertyOperation\_None);

}

return PathTypeHandlerBase::AddPropertyInternal(instance, propertyId, value, info, flags, possibleSideEffects);

}

BOOL PathTypeHandlerBase::DeleteProperty(DynamicObject\* instance, PropertyId propertyId, PropertyOperationFlags flags)

{

#ifdef PROFILE\_TYPES

instance->GetScriptContext()->convertPathToDictionaryCount2++;

#endif

// Check numeric propertyId only if objectArray available

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

uint32 indexVal;

if (instance->HasObjectArray() && scriptContext->IsNumericPropertyId(propertyId, &indexVal))

{

return PathTypeHandlerBase::DeleteItem(instance, indexVal, flags);

}

return ConvertToSimpleDictionaryType(instance, GetPathLength())->DeleteProperty(instance, propertyId, flags);

}

BOOL PathTypeHandlerBase::IsFixedProperty(const DynamicObject\* instance, PropertyId propertyId)

{

if (!FixPropsOnPathTypes())

{

return false;

}

PropertyIndex index = PathTypeHandlerBase::GetPropertyIndex(propertyId);

Assert(index != Constants::NoSlot);

return this->typePath->GetIsFixedFieldAt(index, GetPathLength());

}

BOOL PathTypeHandlerBase::IsEnumerable(DynamicObject\* instance, PropertyId propertyId)

{

return true;

}

BOOL PathTypeHandlerBase::IsWritable(DynamicObject\* instance, PropertyId propertyId)

{

return true;

}

BOOL PathTypeHandlerBase::IsConfigurable(DynamicObject\* instance, PropertyId propertyId)

{

return true;

}

BOOL PathTypeHandlerBase::SetEnumerable(DynamicObject\* instance, PropertyId propertyId, BOOL value)

{

#ifdef PROFILE\_TYPES

instance->GetScriptContext()->convertPathToDictionaryCount3++;

#endif

return value || ConvertToSimpleDictionaryType(instance, GetPathLength())->SetEnumerable(instance, propertyId, value);

}

BOOL PathTypeHandlerBase::SetWritable(DynamicObject\* instance, PropertyId propertyId, BOOL value)

{

#ifdef PROFILE\_TYPES

instance->GetScriptContext()->convertPathToDictionaryCount3++;

#endif

return value || ConvertToSimpleDictionaryType(instance, GetPathLength())->SetWritable(instance, propertyId, value);

}

BOOL PathTypeHandlerBase::SetConfigurable(DynamicObject\* instance, PropertyId propertyId, BOOL value)

{

#ifdef PROFILE\_TYPES

instance->GetScriptContext()->convertPathToDictionaryCount3++;

#endif

return value || ConvertToSimpleDictionaryType(instance, GetPathLength())->SetConfigurable(instance, propertyId, value);

}

BOOL PathTypeHandlerBase::SetAccessors(DynamicObject\* instance, PropertyId propertyId, Var getter, Var setter, PropertyOperationFlags flags)

{

#ifdef PROFILE\_TYPES

instance->GetScriptContext()->convertPathToDictionaryCount4++;

#endif

return ConvertToDictionaryType(instance)->SetAccessors(instance, propertyId, getter, setter, flags);

}

BOOL PathTypeHandlerBase::PreventExtensions(DynamicObject\* instance)

{

#ifdef PROFILE\_TYPES

instance->GetScriptContext()->convertPathToDictionaryCount4++;

#endif

BOOL tempResult = this->ConvertToSharedNonExtensibleTypeIfNeededAndCallOperation(instance, InternalPropertyRecords::NonExtensibleType,

[&](SimpleDictionaryTypeHandlerWithNontExtensibleSupport\* newTypeHandler)

{

return newTypeHandler->PreventExtensionsInternal(instance);

});

Assert(tempResult);

if (tempResult)

{

// Call preventExtensions on the objectArray -- which will create different type for array type handler.

ArrayObject \* objectArray = instance->GetObjectArray();

if (objectArray)

{

objectArray->PreventExtensions();

}

}

return tempResult;

}

BOOL PathTypeHandlerBase::Seal(DynamicObject\* instance)

{

#ifdef PROFILE\_TYPES

instance->GetScriptContext()->convertPathToDictionaryCount4++;

#endif

// For seal we need an array with non-default attributes, which is ES5Array,

// and in current design ES5Array goes side-by-side with DictionaryTypeHandler.

// Note that 2 instances can have same PathTypehandler but still different objectArray items, e.g. {x:0, 0:0} and {x:0, 1:0}.

// Technically we could change SimpleDictionaryTypehandler to override \*Item\* methods,

// similar to DictionaryTypeHandler, but objects with numeric properties are currently seen as low priority,

// so just don't share the type.

if (instance->HasObjectArray())

{

return this->ConvertToDictionaryType(instance)->Seal(instance);

}

else

{

return this->ConvertToSharedNonExtensibleTypeIfNeededAndCallOperation(instance, InternalPropertyRecords::SealedType,

[&](SimpleDictionaryTypeHandlerWithNontExtensibleSupport\* newTypeHandler)

{

return newTypeHandler->SealInternal(instance);

});

}

}

BOOL PathTypeHandlerBase::FreezeImpl(DynamicObject\* instance, bool isConvertedType)

{

#ifdef PROFILE\_TYPES

instance->GetScriptContext()->convertPathToDictionaryCount4++;

#endif

// See the comment inside Seal WRT HasObjectArray branch.

if (instance->HasObjectArray())

{

return this->ConvertToDictionaryType(instance)->Freeze(instance, isConvertedType);

}

else

{

return this->ConvertToSharedNonExtensibleTypeIfNeededAndCallOperation(instance, InternalPropertyRecords::FrozenType,

[&](SimpleDictionaryTypeHandlerWithNontExtensibleSupport\* newTypeHandler)

{

return newTypeHandler->FreezeInternal(instance, true); // true: we don't want to change type in FreezeInternal.

});

}

}

// Checks whether conversion to shared type is needed and performs it, then calls actual operation on the shared type.

// Template method used for PreventExtensions, Seal, Freeze.

// Parameters:

// - instance: object instance to operate on.

// - operationInternalPropertyRecord: the internal property record for preventExtensions/seal/freeze.

// - FType: functor/lambda to perform actual forced operation (such as PreventExtensionsInternal) on the shared type.

template<typename FType>

BOOL PathTypeHandlerBase::ConvertToSharedNonExtensibleTypeIfNeededAndCallOperation(DynamicObject\* instance, const PropertyRecord\* operationInternalPropertyRecord, FType operation)

{

AssertMsg(operationInternalPropertyRecord == InternalPropertyRecords::NonExtensibleType ||

operationInternalPropertyRecord == InternalPropertyRecords::SealedType ||

operationInternalPropertyRecord == InternalPropertyRecords::FrozenType,

"Wrong/unsupported value of operationInternalPropertyRecord.");

RecyclerWeakReference<DynamicType>\* newTypeWeakRef = nullptr;

DynamicType \* oldType = instance->GetDynamicType();

// See if we already have shared type for this type and convert to it, otherwise create a new one.

if (!GetSuccessor(operationInternalPropertyRecord, &newTypeWeakRef) || newTypeWeakRef->Get() == nullptr)

{

// Convert to new shared type with shared simple dictionary type handler and call operation on it.

SimpleDictionaryTypeHandlerWithNontExtensibleSupport\* newTypeHandler = ConvertToSimpleDictionaryType

<SimpleDictionaryTypeHandlerWithNontExtensibleSupport>(instance, this->GetPathLength(), true);

Assert(newTypeHandler->GetMayBecomeShared() && !newTypeHandler->GetIsShared());

DynamicType\* newType = instance->GetDynamicType();

newType->LockType();

Assert(!newType->GetIsShared());

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

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

SetSuccessor(oldType, operationInternalPropertyRecord, recycler->CreateWeakReferenceHandle<DynamicType>(newType), scriptContext);

return operation(newTypeHandler);

}

else

{

DynamicType\* newType = newTypeWeakRef->Get();

DynamicTypeHandler\* newTypeHandler = newType->GetTypeHandler();

// Consider: Consider doing something special for frozen objects, whose values cannot

// change and so we could retain them as fixed, even when the type becomes shared.

newType->ShareType();

// Consider: If we isolate prototypes, we should never get here with the prototype flag set.

// There should be nothing to transfer.

// Assert(!IsolatePrototypes() || (this->GetFlags() & IsPrototypeFlag) == 0);

newTypeHandler->SetFlags(IsPrototypeFlag, this->GetFlags());

Assert(!newTypeHandler->HasSingletonInstance());

if(instance->IsObjectHeaderInlinedTypeHandler())

{

const PropertyIndex newInlineSlotCapacity = newTypeHandler->GetInlineSlotCapacity();

AdjustSlots(instance, newInlineSlotCapacity, newTypeHandler->GetSlotCapacity() - newInlineSlotCapacity);

}

ReplaceInstanceType(instance, newType);

}

return TRUE;

}

DynamicType\* PathTypeHandlerBase::PromoteType(DynamicObject\* instance, const PropertyRecord\* propertyRecord, PropertyIndex\* propertyIndex)

{

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

DynamicType\* currentType = instance->GetDynamicType();

DynamicType\* nextType = this->PromoteType<false>(currentType, propertyRecord, false, scriptContext, instance, propertyIndex);

PathTypeHandlerBase\* nextPath = (PathTypeHandlerBase\*) nextType->GetTypeHandler();

instance->EnsureSlots(this->GetSlotCapacity(), nextPath->GetSlotCapacity(), scriptContext, nextType->GetTypeHandler());

ReplaceInstanceType(instance, nextType);

return nextType;

}

template <typename T>

T\* PathTypeHandlerBase::ConvertToTypeHandler(DynamicObject\* instance)

{

Assert(instance);

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

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

PathTypeHandlerBase \* oldTypeHandler;

// Ideally 'this' and oldTypeHandler->GetTypeHandler() should be same

// But we can have calls from external DOM objects, which requests us to replace the type of the

// object with a new type. And in such cases, this API gets called with oldTypeHandler and the

// new type (obtained from the External DOM object)

// We use the duplicated typeHandler, if we deOptimized the object successfully, else we retain the earlier

// behavior of using 'this' pointer.

if (instance->DeoptimizeObjectHeaderInlining())

{

oldTypeHandler = reinterpret\_cast<PathTypeHandlerBase \*>(instance->GetTypeHandler());

}

else

{

oldTypeHandler = this;

}

Assert(oldTypeHandler);

T\* newTypeHandler = RecyclerNew(recycler, T, recycler, oldTypeHandler->GetSlotCapacity(), oldTypeHandler->GetInlineSlotCapacity(), oldTypeHandler->GetOffsetOfInlineSlots());

// We expect the new type handler to start off marked as having only writable data properties.

Assert(newTypeHandler->GetHasOnlyWritableDataProperties());

#ifdef ENABLE\_DEBUG\_CONFIG\_OPTIONS

DynamicType\* oldType = instance->GetDynamicType();

RecyclerWeakReference<DynamicObject>\* oldSingletonInstance = oldTypeHandler->GetSingletonInstance();

oldTypeHandler->TraceFixedFieldsBeforeTypeHandlerChange(L"converting", L"PathTypeHandler", L"DictionaryTypeHandler", instance, oldTypeHandler, oldType, oldSingletonInstance);

#endif

bool const canBeSingletonInstance = DynamicTypeHandler::CanBeSingletonInstance(instance);

// If this type had been installed on a stack instance it shouldn't have a singleton Instance

Assert(canBeSingletonInstance || !oldTypeHandler->HasSingletonInstance());

// This instance may not be the singleton instance for this handler. There may be a singleton at the tip

// and this instance may be getting initialized via an object literal and one of the properties may

// be an accessor. In this case we will convert to a DictionaryTypeHandler and it's correct to

// transfer this instance, even tough different from the singleton. Ironically, this instance

// may even appear to be at the tip along with the other singleton, because the set of properties (by

// name, not value) may be identical.

// Consider: Consider threading PropertyOperation\_Init through InitProperty and SetAccessors,

// to be sure that we don't assert only in this narrow case.

// Assert(this->typePath->GetSingletonInstance() == instance);

Assert(oldTypeHandler->HasSingletonInstanceOnlyIfNeeded());

// Don't install stack instance as singleton instance

if (canBeSingletonInstance)

{

if (DynamicTypeHandler::AreSingletonInstancesNeeded())

{

RecyclerWeakReference<DynamicObject>\* curSingletonInstance = oldTypeHandler->typePath->GetSingletonInstance();

if (curSingletonInstance != nullptr && curSingletonInstance->Get() == instance)

{

newTypeHandler->SetSingletonInstance(curSingletonInstance);

}

else

{

newTypeHandler->SetSingletonInstance(instance->CreateWeakReferenceToSelf());

}

}

}

bool transferFixed = canBeSingletonInstance;

// If we are a prototype or may become a prototype we must transfer used as fixed bits. See point 4 in ConvertToSimpleDictionaryType.

Assert(!DynamicTypeHandler::IsolatePrototypes() || ((oldTypeHandler->GetFlags() & IsPrototypeFlag) == 0));

bool transferUsedAsFixed = ((oldTypeHandler->GetFlags() & IsPrototypeFlag) != 0 || (oldTypeHandler->GetIsOrMayBecomeShared() && !DynamicTypeHandler::IsolatePrototypes()));

for (PropertyIndex i = 0; i < oldTypeHandler->GetPathLength(); i++)

{

// Consider: As noted in point 2 in ConvertToSimpleDictionaryType, when converting to non-shared handler we could be more

// aggressive and mark every field as fixed, because we will always take a type transition. We have to remember to respect

// the switches as to which kinds of properties we should fix, and for that we need the values from the instance. Even if

// the type handler says the property is initialized, the current instance may not have a value for it. Check for value != null.

if (PathTypeHandlerBase::FixPropsOnPathTypes())

{

TypePath \* typePath = oldTypeHandler->typePath;

newTypeHandler->Add(typePath->GetPropertyId(i), PropertyDynamicTypeDefaults,

i < typePath->GetMaxInitializedLength(),

transferFixed && typePath->GetIsFixedFieldAt(i, oldTypeHandler->GetPathLength()),

transferUsedAsFixed && typePath->GetIsUsedFixedFieldAt(i, oldTypeHandler->GetPathLength()),

scriptContext);

}

else

{

newTypeHandler->Add(oldTypeHandler->typePath->GetPropertyId(i), PropertyDynamicTypeDefaults, true, false, false, scriptContext);

}

}

#ifdef SUPPORT\_FIXED\_FIELDS\_ON\_PATH\_TYPES

if (PathTypeHandlerBase::FixPropsOnPathTypes())

{

Assert(oldTypeHandler->HasSingletonInstanceOnlyIfNeeded());

oldTypeHandler->typePath->ClearSingletonInstanceIfSame(instance);

}

#endif

// PathTypeHandlers are always shared, so if we're isolating prototypes, a PathTypeHandler should

// never have the prototype flag set.

Assert(!DynamicTypeHandler::IsolatePrototypes() || ((oldTypeHandler->GetFlags() & IsPrototypeFlag) == 0));

AssertMsg(!newTypeHandler->GetIsPrototype(), "Why did we create a brand new type handler with a prototype flag set?");

newTypeHandler->SetFlags(IsPrototypeFlag, oldTypeHandler->GetFlags());

// Any new type handler we expect to see here should have inline slot capacity locked. If this were to change, we would need

// to update our shrinking logic (see ShrinkSlotAndInlineSlotCapacity).

Assert(newTypeHandler->GetIsInlineSlotCapacityLocked());

newTypeHandler->SetPropertyTypes(PropertyTypesWritableDataOnly | PropertyTypesWritableDataOnlyDetection, oldTypeHandler->GetPropertyTypes());

newTypeHandler->SetInstanceTypeHandler(instance);

Assert(!newTypeHandler->HasSingletonInstance() || !instance->HasSharedType());

#ifdef ENABLE\_DEBUG\_CONFIG\_OPTIONS

PathTypeHandlerBase::TraceFixedFieldsAfterTypeHandlerChange(instance, oldTypeHandler, newTypeHandler, oldType, instance->GetDynamicType(), oldSingletonInstance);

#endif

return newTypeHandler;

}

DictionaryTypeHandler\* PathTypeHandlerBase::ConvertToDictionaryType(DynamicObject\* instance)

{

return ConvertToTypeHandler<DictionaryTypeHandler>(instance);

}

ES5ArrayTypeHandler\* PathTypeHandlerBase::ConvertToES5ArrayType(DynamicObject\* instance)

{

return ConvertToTypeHandler<ES5ArrayTypeHandler>(instance);

}

template <typename T>

T\* PathTypeHandlerBase::ConvertToSimpleDictionaryType(DynamicObject\* instance, int propertyCapacity, bool mayBecomeShared)

{

Assert(instance);

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

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

// Ideally 'this' and oldTypeHandler->GetTypeHandler() should be same

// But we can have calls from external DOM objects, which requests us to replace the type of the

// object with a new type. And in such cases, this API gets called with oldTypeHandler and the

// new type (obtained from the External DOM object)

// We use the duplicated typeHandler, if we deOptimized the object successfully, else we retain the earlier

// behavior of using 'this' pointer.

PathTypeHandlerBase \* oldTypeHandler = nullptr;

if (instance->DeoptimizeObjectHeaderInlining())

{

Assert(instance->GetTypeHandler()->IsPathTypeHandler());

oldTypeHandler = reinterpret\_cast<PathTypeHandlerBase \*>(instance->GetTypeHandler());

}

else

{

oldTypeHandler = this;

}

Assert(oldTypeHandler);

DynamicType\* oldType = instance->GetDynamicType();

T\* newTypeHandler = RecyclerNew(recycler, T, recycler, oldTypeHandler->GetSlotCapacity(), propertyCapacity, oldTypeHandler->GetInlineSlotCapacity(), oldTypeHandler->GetOffsetOfInlineSlots());

// We expect the new type handler to start off marked as having only writable data properties.

Assert(newTypeHandler->GetHasOnlyWritableDataProperties());

// Care must be taken to correctly set up fixed field bits whenever a type's handler is changed. Exactly what needs to

// be done depends on whether the current handler is shared, whether the new handler is shared, whether the current

// handler has the prototype flag set, and even whether we take a type transition as part of the process.

//

// 1. Can we set fixed bits on new handler for the fields that are marked as fixed on current handler?

//

// Yes, if the new type handler is not shared. If the handler is not shared, we know that only this instance will

// ever use it. Otherwise, a different instance could transition to the same type handler, but have different values

// for fields marked as fixed.

//

// 2. Can we set fixed bits on new handler even for the fields that are not marked as fixed on current handler?

//

// Yes, if the new type handler is not shared and we take a type transition during conversion. The first condition

// is required for the same reason as in point 1 above. The type transition is needed to ensure that any store

// field fast paths for this instance get invalidated. If they didn't, then the newly fixed field could get

// overwritten on the fast path without triggering necessary invalidation.

//

// Note that it's desirable to mark additional fields as fixed (particularly when the instance becomes a prototype)

// to counteract the effect of false type sharing, which may unnecessarily turn off some fixed field bits.

//

// 3. Do we need to clear any fixed field bits on the old or new type handler?

//

// Yes, we must clear fixed fields bits for properties that aren't also used as fixed, but only if both type handlers

// are shared and we don't isolate prototypes. This is rather tricky and results from us pre-creating certain handlers

// even before any instances actually have values for all represented properties. We must avoid the situation, in which

// one instance switched to a new type handler with some fixed field not yet used as fixed, and later the second

// instance follows the same handler evolution with the same field used as fixed. Upon switching to the new handler

// the second instance would "forget" that the field was used as fixed and fail to invalidate when overwritten.

//

// Example: Instance A with TH1 has a fixed method FOO, which has not been used as fixed yet. Then instance B gets

// pre-created and lands on TH1 (and so far assumes FOO is fixed). As B's pre-creation continues, it moves to TH2, but

// thus far FOO has not been used as fixed. Now instance A becomes a prototype, and its method FOO is used in a hard-coded

// JIT sequence, thus marking it as used as fixed. Instance A then transitions to TH2 and we lose track of FOO being used

// as fixed. If FOO is then overwritten on A, the hard-coded JIT sequence does not get invalidated and continues to call

// the old method FOO.

//

// 4. Can we avoid setting used as fixed bits on new handler for fields marked as used as fixed on current handler?

//

// Yes, if the current type handler doesn't have the prototype flag and current handler is not shared or new handler

// is not shared or we isolate prototypes, and we take a type transition as part of the conversion.

//

// Type transition ensures that any field loads from the instance are invalidated (including

// any that may have hard-coded the fixed field's value). Hence, if the fixed field on this instance were to be later

// overwritten it will not cause any functional issues. On the other hand, field loads from prototype are not affected

// by the prototype object's type change. Therefore, if this instance is a prototype we must carry the used as fixed

// bits forward to ensure that if we overwrite any fixed field we explicitly trigger invalidation.

//

// Review: Actually, the comment below is overly conservative. If the second instance that became a prototype

// followed the same type evolution path, it would have to have invalidated all fixed fields, so there should be no need

// to transfer used as fixed bits, unless the current instance is already a prototype.

// In addition, if current handler is shared and the new handler is shared, a different instance with the current handler

// may later become a prototype (if we don't isolate prototypes) and follow the same conversion to the new handler, even

// if the current instance is not a prototype. Hence, the new type handler must retain the used as fixed bits, so that

// proper invalidation can be triggered later, if overwritten.

//

// Note that this may lead to the new type handler with some fields not marked as fixed, but marked as used as fixed.

//

// Note also that if we isolate prototypes, we guarantee that no prototype instance will share a type handler with any

// other instance. Hence, the problem sequence above could not take place.

//

// 5. Do we need to invalidate JIT-ed code for any fields marked as used as fixed on current handler?

//

// No. With the rules above any necessary invalidation will be triggered when the value actually gets overwritten.

//

#ifdef ENABLE\_DEBUG\_CONFIG\_OPTIONS

RecyclerWeakReference<DynamicObject>\* oldSingletonInstance = oldTypeHandler->GetSingletonInstance();

oldTypeHandler->TraceFixedFieldsBeforeTypeHandlerChange(L"converting", L"PathTypeHandler", L"SimpleDictionaryTypeHandler", instance, oldTypeHandler, oldType, oldSingletonInstance);

#endif

bool const canBeSingletonInstance = DynamicTypeHandler::CanBeSingletonInstance(instance);

// If this type had been installed on a stack instance it shouldn't have a singleton Instance

Assert(canBeSingletonInstance || !oldTypeHandler->HasSingletonInstance());

// Consider: It looks like we're delaying sharing of these type handlers until the second instance arrives, so we could

// set the singleton here and zap it later.

if (!mayBecomeShared && canBeSingletonInstance)

{

Assert(oldTypeHandler->HasSingletonInstanceOnlyIfNeeded());

if (DynamicTypeHandler::AreSingletonInstancesNeeded())

{

RecyclerWeakReference<DynamicObject>\* curSingletonInstance = oldTypeHandler->typePath->GetSingletonInstance();

if (curSingletonInstance != nullptr && curSingletonInstance->Get() == instance)

{

newTypeHandler->SetSingletonInstance(curSingletonInstance);

}

else

{

newTypeHandler->SetSingletonInstance(instance->CreateWeakReferenceToSelf());

}

}

}

// It would be nice to transfer fixed fields if the new type handler may become fixed later (but isn't yet). This would allow

// singleton instances to retain fixed fields. It would require that when we do actually share the target type (when the second

// instance arrives), we clear (and invalidate, if necessary) any fixed fields. This may be a reasonable trade-off.

bool transferIsFixed = !mayBecomeShared && canBeSingletonInstance;

// If we are a prototype or may become a prototype we must transfer used as fixed bits. See point 4 above.

Assert(!DynamicTypeHandler::IsolatePrototypes() || ((oldTypeHandler->GetFlags() & IsPrototypeFlag) == 0));

// For the global object we don't emit a type check before a hard-coded use of a fixed field. Therefore a type transition isn't sufficient to

// invalidate any used fixed fields, and we must continue tracking them on the new type handler. The global object should never have a path

// type handler.

Assert(instance->GetTypeId() != TypeIds\_GlobalObject);

// If the type isn't locked, we may not change the type of the instance, and we must also track the used fixed fields on the new handler.

bool transferUsedAsFixed = !instance->GetDynamicType()->GetIsLocked() || ((oldTypeHandler->GetFlags() & IsPrototypeFlag) != 0 || (oldTypeHandler->GetIsOrMayBecomeShared() && !DynamicTypeHandler::IsolatePrototypes()));

// Consider: As noted in point 2 above, when converting to non-shared SimpleDictionaryTypeHandler we could be more aggressive

// and mark every field as fixed, because we will always take a type transition. We have to remember to respect the switches as

// to which kinds of properties we should fix, and for that we need the values from the instance. Even if the type handler

// says the property is initialized, the current instance may not have a value for it. Check for value != null.

for (PropertyIndex i = 0; i < oldTypeHandler->GetPathLength(); i++)

{

if (PathTypeHandlerBase::FixPropsOnPathTypes())

{

Js::TypePath \* typePath = oldTypeHandler->typePath;

newTypeHandler->Add(typePath->GetPropertyId(i), PropertyDynamicTypeDefaults,

i < typePath->GetMaxInitializedLength(),

transferIsFixed && typePath->GetIsFixedFieldAt(i, GetPathLength()),

transferUsedAsFixed && typePath->GetIsUsedFixedFieldAt(i, GetPathLength()),

scriptContext);

}

else

{

newTypeHandler->Add(oldTypeHandler->typePath->GetPropertyId(i), PropertyDynamicTypeDefaults, true, false, false, scriptContext);

}

// No need to clear fixed fields not used as fixed, because we never convert during pre-creation of type handlers and we always

// add properties in order they appear on the type path. Hence, any existing fixed fields will be turned off by any other

// instance following this type path. See point 3 above.

}

#ifdef SUPPORT\_FIXED\_FIELDS\_ON\_PATH\_TYPES

// Clear the singleton from this handler regardless of mayBecomeShared, because this instance no longer uses this handler.

if (PathTypeHandlerBase::FixPropsOnPathTypes())

{

Assert(oldTypeHandler->HasSingletonInstanceOnlyIfNeeded());

oldTypeHandler->typePath->ClearSingletonInstanceIfSame(instance);

}

#endif

if (mayBecomeShared)

{

newTypeHandler->SetFlags(IsLockedFlag | MayBecomeSharedFlag);

}

Assert(!DynamicTypeHandler::IsolatePrototypes() || !oldTypeHandler->GetIsOrMayBecomeShared() || ((oldTypeHandler->GetFlags() & IsPrototypeFlag) == 0));

AssertMsg((newTypeHandler->GetFlags() & IsPrototypeFlag) == 0, "Why did we create a brand new type handler with a prototype flag set?");

newTypeHandler->SetFlags(IsPrototypeFlag, oldTypeHandler->GetFlags());

// Any new type handler we expect to see here should have inline slot capacity locked. If this were to change, we would need

// to update our shrinking logic (see ShrinkSlotAndInlineSlotCapacity).

Assert(newTypeHandler->GetIsInlineSlotCapacityLocked());

newTypeHandler->SetPropertyTypes(PropertyTypesWritableDataOnly | PropertyTypesWritableDataOnlyDetection, oldTypeHandler->GetPropertyTypes());

newTypeHandler->SetInstanceTypeHandler(instance);

Assert(!newTypeHandler->HasSingletonInstance() || !instance->HasSharedType());

// We assumed that we don't need to transfer used as fixed bits unless we are a prototype, which is only valid if we also changed the type.

Assert(transferUsedAsFixed || (instance->GetType() != oldType && oldType->GetTypeId() != TypeIds\_GlobalObject));

#ifdef ENABLE\_DEBUG\_CONFIG\_OPTIONS

PathTypeHandlerBase::TraceFixedFieldsAfterTypeHandlerChange(instance, oldTypeHandler, newTypeHandler, oldType, instance->GetDynamicType(), oldSingletonInstance);

#endif

#ifdef PROFILE\_TYPES

scriptContext->convertPathToSimpleDictionaryCount++;

#endif

return newTypeHandler;

}

BOOL PathTypeHandlerBase::SetPropertyWithAttributes(DynamicObject\* instance, PropertyId propertyId, Var value, PropertyAttributes attributes, PropertyValueInfo\* info, PropertyOperationFlags flags, SideEffects possibleSideEffects)

{

if (attributes == PropertyDynamicTypeDefaults)

{

return PathTypeHandlerBase::SetPropertyInternal(instance, propertyId, value, info, flags, possibleSideEffects);

}

else

{

return ConvertToSimpleDictionaryType(instance, GetPathLength() + 1)->SetPropertyWithAttributes(instance, propertyId, value, attributes, info, flags, possibleSideEffects);

}

}

BOOL PathTypeHandlerBase::SetAttributes(DynamicObject\* instance, PropertyId propertyId, PropertyAttributes attributes)

{

if ( (attributes & PropertyDynamicTypeDefaults) != PropertyDynamicTypeDefaults)

{

#ifdef PROFILE\_TYPES

instance->GetScriptContext()->convertPathToDictionaryCount3++;

#endif

return ConvertToSimpleDictionaryType(instance, GetPathLength())->SetAttributes(instance, propertyId, attributes);

}

return true;

}

BOOL PathTypeHandlerBase::GetAttributesWithPropertyIndex(DynamicObject \* instance, PropertyId propertyId, BigPropertyIndex index, PropertyAttributes \* attributes)

{

if (index < this->GetPathLength())

{

Assert(this->GetPropertyId(instance->GetScriptContext(), index) == propertyId);

\*attributes = PropertyDynamicTypeDefaults;

return true;

}

return false;

}

bool PathTypeHandlerBase::UsePathTypeHandlerForObjectLiteral(

const PropertyIdArray \*const propIds,

ScriptContext \*const scriptContext,

bool \*const check\_\_proto\_\_Ref)

{

Assert(propIds);

Assert(scriptContext);

// Always check \_\_proto\_\_ entry, now that object literals always honor \_\_proto\_\_

const bool check\_\_proto\_\_ = propIds->has\_\_proto\_\_ && scriptContext->GetConfig()->Is\_\_proto\_\_Enabled();

if(check\_\_proto\_\_Ref)

{

\*check\_\_proto\_\_Ref = check\_\_proto\_\_;

}

return !check\_\_proto\_\_ && propIds->count < TypePath::MaxPathTypeHandlerLength && !propIds->hadDuplicates;

}

DynamicType\* PathTypeHandlerBase::CreateTypeForNewScObject(ScriptContext\* scriptContext, DynamicType\* type, const Js::PropertyIdArray \*propIds, bool shareType)

{

uint count = propIds->count;

bool check\_\_proto\_\_;

if (UsePathTypeHandlerForObjectLiteral(propIds, scriptContext, &check\_\_proto\_\_))

{

#ifdef PROFILE\_OBJECT\_LITERALS

scriptContext->objectLiteralCount[count]++;

#endif

for (uint i = 0; i < count; i++)

{

PathTypeHandlerBase \*pathHandler = (PathTypeHandlerBase \*)type->typeHandler;

Js::PropertyId propertyId = propIds->elements[i];

PropertyIndex propertyIndex = pathHandler->GetPropertyIndex(propertyId);

if (propertyIndex != Constants::NoSlot)

{

continue;

}

#ifdef PROFILE\_OBJECT\_LITERALS

{

RecyclerWeakReference<DynamicType>\* nextTypeWeakRef;

if (!pathHandler->GetSuccessor(scriptContext->GetPropertyName(propertyId), &nextTypeWeakRef) || nextTypeWeakRef->Get() == nullptr)

{

scriptContext->objectLiteralPathCount++;

}

}

#endif

type = pathHandler->PromoteType<true>(type, scriptContext->GetPropertyName(propertyId), shareType, scriptContext, nullptr, &propertyIndex);

}

}

else if (count <= static\_cast<uint>(SimpleDictionaryTypeHandler::MaxPropertyIndexSize))

{

type = SimpleDictionaryTypeHandler::CreateTypeForNewScObject(scriptContext, type, propIds, shareType, check\_\_proto\_\_);

}

else if (count <= static\_cast<uint>(BigSimpleDictionaryTypeHandler::MaxPropertyIndexSize))

{

type = BigSimpleDictionaryTypeHandler::CreateTypeForNewScObject(scriptContext, type, propIds, shareType, check\_\_proto\_\_);

}

else

{

Throw::OutOfMemory();

}

return type;

}

DynamicType \*

PathTypeHandlerBase::CreateNewScopeObject(ScriptContext \*scriptContext, DynamicType \*type, const PropertyIdArray \*propIds, PropertyAttributes extraAttributes, uint extraAttributesSlotCount)

{

uint count = propIds->count;

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

SimpleDictionaryTypeHandler\* typeHandler = SimpleDictionaryTypeHandler::New(recycler, count, 0, 0, true, true);

for (uint i = 0; i < count; i++)

{

PropertyId propertyId = propIds->elements[i];

const PropertyRecord\* propertyRecord = propertyId == Constants::NoProperty ? NULL : scriptContext->GetPropertyName(propertyId);

// This will add the property as initialized and non-fixed. That's fine because we will populate the property values on the

// scope object right after this (see JavascriptOperators::OP\_InitCachedScope). We will not treat these properties as fixed.

PropertyAttributes attributes = PropertyWritable | PropertyEnumerable;

if (i < extraAttributesSlotCount)

{

attributes |= extraAttributes;

}

typeHandler->Add(propertyRecord, attributes, scriptContext);

}

AssertMsg((typeHandler->GetFlags() & IsPrototypeFlag) == 0, "Why does a newly created type handler have a IsPrototypeFlag set?");

#ifdef PROFILE\_OBJECT\_LITERALS

scriptContext->objectLiteralSimpleDictionaryCount++;

#endif

type = RecyclerNew(recycler, DynamicType, type, typeHandler, /\* isLocked = \*/ true, /\* isShared = \*/ true);

return type;

}

template <bool isObjectLiteral>

DynamicType\* PathTypeHandlerBase::PromoteType(DynamicType\* predecessorType, const PropertyRecord\* propertyRecord, bool shareType, ScriptContext\* scriptContext, DynamicObject\* instance, PropertyIndex\* propertyIndex)

{

Assert(propertyIndex != nullptr);

Assert(isObjectLiteral || instance != nullptr);

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

PropertyIndex index;

DynamicType \* nextType;

RecyclerWeakReference<DynamicType>\* nextTypeWeakRef = nullptr;

PathTypeHandlerBase \* nextPath;

if (!GetSuccessor(propertyRecord, &nextTypeWeakRef) || nextTypeWeakRef->Get() == nullptr)

{

#ifdef ENABLE\_DEBUG\_CONFIG\_OPTIONS

DynamicType\* oldType = predecessorType;

RecyclerWeakReference<DynamicObject>\* oldSingletonInstance = GetSingletonInstance();

bool branching = typePath->GetPathLength() > GetPathLength();

TraceFixedFieldsBeforeTypeHandlerChange(branching ? L"branching" : L"advancing", L"PathTypeHandler", L"PathTypeHandler", instance, this, oldType, oldSingletonInstance);

#endif

TypePath \* newTypePath = typePath;

if (typePath->GetPathLength() > GetPathLength())

{

// We need to branch the type path.

newTypePath = typePath->Branch(recycler, GetPathLength(), GetIsOrMayBecomeShared() && !IsolatePrototypes());

#ifdef PROFILE\_TYPES

scriptContext->branchCount++;

#endif

#ifdef PROFILE\_OBJECT\_LITERALS

if (isObjectLiteral)

{

scriptContext->objectLiteralBranchCount++;

}

#endif

}

else if (typePath->GetPathLength() == typePath->GetPathSize())

{

// We need to grow the type path.

newTypePath = typePath->Grow(recycler);

// Update all the predecessor types that use this TypePath to the new TypePath.

// This will allow the old TypePath to be collected, and will ensure that the

// fixed field info is correct for those types.

PathTypeHandlerBase \* typeHandlerToUpdate = this;

TypePath \* oldTypePath = typePath;

while (true)

{

typeHandlerToUpdate->typePath = newTypePath;

DynamicType \* nextType = typeHandlerToUpdate->GetPredecessorType();

if (nextType == nullptr)

{

break;

}

Assert(nextType->GetTypeHandler()->IsPathTypeHandler());

typeHandlerToUpdate = (PathTypeHandlerBase \*)nextType->GetTypeHandler();

if (typeHandlerToUpdate->typePath != oldTypePath)

{

break;

}

}

}

index = (PropertyIndex)newTypePath->AddInternal(propertyRecord);

const PropertyIndex newPropertyCount = GetPathLength() + 1;

const PropertyIndex newSlotCapacity = max(newPropertyCount, static\_cast<PropertyIndex>(GetSlotCapacity()));

PropertyIndex newInlineSlotCapacity = GetInlineSlotCapacity();

uint16 newOffsetOfInlineSlots = GetOffsetOfInlineSlots();

if(IsObjectHeaderInlinedTypeHandler() && newSlotCapacity > GetSlotCapacity())

{

newInlineSlotCapacity -= GetObjectHeaderInlinableSlotCapacity();

newOffsetOfInlineSlots = sizeof(DynamicObject);

}

bool markTypeAsShared = !FixPropsOnPathTypes() || shareType;

nextPath = SimplePathTypeHandler::New(scriptContext, newTypePath, newPropertyCount, newSlotCapacity, newInlineSlotCapacity, newOffsetOfInlineSlots, true, markTypeAsShared, predecessorType);

if (!markTypeAsShared) nextPath->SetMayBecomeShared();

Assert(nextPath->GetHasOnlyWritableDataProperties());

nextPath->CopyPropertyTypes(PropertyTypesWritableDataOnly | PropertyTypesWritableDataOnlyDetection, GetPropertyTypes());

nextPath->SetPropertyTypes(PropertyTypesInlineSlotCapacityLocked, GetPropertyTypes());

if (shareType)

{

nextPath->AddBlankFieldAt(propertyRecord->GetPropertyId(), index, scriptContext);

}

#ifdef PROFILE\_TYPES

scriptContext->maxPathLength = max(GetPathLength() + 1, scriptContext->maxPathLength);

#endif

if (isObjectLiteral)

{

// The new type isn't shared yet. We will make it shared when the second instance attains it.

nextType = RecyclerNew(recycler, DynamicType, predecessorType, nextPath, /\* isLocked = \*/ true, /\* isShared = \*/ markTypeAsShared);

}

else

{

// The new type isn't shared yet. We will make it shared when the second instance attains it.

nextType = instance->DuplicateType();

nextType->typeHandler = nextPath;

markTypeAsShared ? nextType->SetIsLockedAndShared() : nextType->SetIsLocked();

}

SetSuccessor(predecessorType, propertyRecord, recycler->CreateWeakReferenceHandle<DynamicType>(nextType), scriptContext);

// We just extended the current type path to a new tip or created a brand new type path. We should

// be at the tip of the path and there should be no instances there yet.

Assert(nextPath->GetPathLength() == newTypePath->GetPathLength());

Assert(!FixPropsOnPathTypes() || shareType || nextPath->GetPathLength() > newTypePath->GetMaxInitializedLength());

#ifdef PROFILE\_TYPES

scriptContext->promoteCount++;

#endif

#ifdef PROFILE\_OBJECT\_LITERALS

if (isObjectLiteral)

{

scriptContext->objectLiteralPromoteCount++;

}

#endif

#ifdef ENABLE\_DEBUG\_CONFIG\_OPTIONS

TraceFixedFieldsAfterTypeHandlerChange(instance, this, nextPath, oldType, nextType, oldSingletonInstance);

#endif

}

else

{

#ifdef PROFILE\_TYPES

scriptContext->cacheCount++;

#endif

// Now that the second (or subsequent) instance reached this type, make sure that it's shared.

nextType = nextTypeWeakRef->Get();

nextPath = (PathTypeHandlerBase \*)nextType->GetTypeHandler();

Assert(nextPath->GetIsInlineSlotCapacityLocked() == this->GetIsInlineSlotCapacityLocked());

index = nextPath->GetPropertyIndex(propertyRecord);

Assert((FixPropsOnPathTypes() && nextPath->GetMayBecomeShared()) || (nextPath->GetIsShared() && nextType->GetIsShared()));

if (FixPropsOnPathTypes() && !nextType->GetIsShared())

{

if (!nextPath->GetIsShared())

{

nextPath->AddBlankFieldAt(propertyRecord->GetPropertyId(), index, scriptContext);

nextPath->DoShareTypeHandlerInternal<false>(scriptContext);

}

nextType->ShareType();

}

}

Assert(!IsolatePrototypes() || !GetIsOrMayBecomeShared() || !GetIsPrototype());

nextPath->SetFlags(IsPrototypeFlag, this->GetFlags());

Assert(this->GetHasOnlyWritableDataProperties() == nextPath->GetHasOnlyWritableDataProperties());

Assert(this->GetIsInlineSlotCapacityLocked() == nextPath->GetIsInlineSlotCapacityLocked());

nextPath->SetPropertyTypes(PropertyTypesWritableDataOnlyDetection, this->GetPropertyTypes());

(\*propertyIndex) = index;

return nextType;

}

void

PathTypeHandlerBase::ResetTypeHandler(DynamicObject \* instance)

{

// The type path is allocated in the type allocator associated with the script context.

// So we can't reuse it in other context. Just convert the type to a simple dictionary type

this->ConvertToSimpleDictionaryType(instance, GetPathLength());

}

void PathTypeHandlerBase::SetAllPropertiesToUndefined(DynamicObject\* instance, bool invalidateFixedFields)

{

// Note: This method is currently only called from ResetObject, which in turn only applies to external objects.

// Before using for other purposes, make sure the assumptions made here make sense in the new context. In particular,

// the invalidateFixedFields == false is only correct if a) the object is known not to have any, or b) the type of the

// object has changed and/or property guards have already been invalidated through some other means.

int propertyCount = GetPathLength();

if (invalidateFixedFields)

{

Js::ScriptContext\* scriptContext = instance->GetScriptContext();

for (PropertyIndex propertyIndex = 0; propertyIndex < propertyCount; propertyIndex++)

{

PropertyId propertyId = this->typePath->GetPropertyIdUnchecked(propertyIndex)->GetPropertyId();

InvalidateFixedFieldAt(propertyId, propertyIndex, scriptContext);

}

}

Js::RecyclableObject\* undefined = instance->GetLibrary()->GetUndefined();

for (PropertyIndex propertyIndex = 0; propertyIndex < propertyCount; propertyIndex++)

{

SetSlotUnchecked(instance, propertyIndex, undefined);

}

}

void PathTypeHandlerBase::MarshalAllPropertiesToScriptContext(DynamicObject\* instance, ScriptContext\* targetScriptContext, bool invalidateFixedFields)

{

// Note: This method is currently only called from ResetObject, which in turn only applies to external objects.

// Before using for other purposes, make sure the assumptions made here make sense in the new context. In particular,

// the invalidateFixedFields == false is only correct if a) the object is known not to have any, or b) the type of the

// object has changed and/or property guards have already been invalidated through some other means.

int propertyCount = GetPathLength();

if (invalidateFixedFields)

{

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

for (PropertyIndex propertyIndex = 0; propertyIndex < propertyCount; propertyIndex++)

{

PropertyId propertyId = this->typePath->GetPropertyIdUnchecked(propertyIndex)->GetPropertyId();

InvalidateFixedFieldAt(propertyId, propertyIndex, scriptContext);

}

}

for (int slotIndex = 0; slotIndex < propertyCount; slotIndex++)

{

SetSlotUnchecked(instance, slotIndex, CrossSite::MarshalVar(targetScriptContext, GetSlot(instance, slotIndex)));

}

}

BOOL PathTypeHandlerBase::AddProperty(DynamicObject \* instance, PropertyId propertyId, Js::Var value, PropertyAttributes attributes, PropertyValueInfo\* info, PropertyOperationFlags flags, SideEffects possibleSideEffects)

{

if (attributes != PropertyDynamicTypeDefaults)

{

Assert(propertyId != Constants::NoProperty);

PropertyRecord const\* propertyRecord = instance->GetScriptContext()->GetPropertyName(propertyId);

return ConvertToSimpleDictionaryType(instance, GetPathLength() + 1)->AddProperty(instance, propertyRecord, value, attributes, info, flags, possibleSideEffects);

}

return AddPropertyInternal(instance, propertyId, value, info, flags, possibleSideEffects);

}

BOOL PathTypeHandlerBase::AddPropertyInternal(DynamicObject \* instance, PropertyId propertyId, Js::Var value, PropertyValueInfo\* info, PropertyOperationFlags flags, SideEffects possibleSideEffects)

{

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

#if DBG

uint32 indexVal;

Assert(GetPropertyIndex(propertyId) == Constants::NoSlot);

Assert(!scriptContext->IsNumericPropertyId(propertyId, &indexVal));

#endif

Assert(propertyId != Constants::NoProperty);

PropertyRecord const\* propertyRecord = instance->GetScriptContext()->GetPropertyName(propertyId);

if (GetPathLength() >= TypePath::MaxPathTypeHandlerLength)

{

#ifdef PROFILE\_TYPES

scriptContext->convertPathToDictionaryCount1++;

#endif

return ConvertToSimpleDictionaryType(instance, GetPathLength() + 1)->AddProperty(instance, propertyRecord, value, PropertyDynamicTypeDefaults, info, PropertyOperation\_None, possibleSideEffects);

}

PropertyIndex index;

DynamicType\* newType = PromoteType(instance, propertyRecord, &index);

Assert(instance->GetTypeHandler()->IsPathTypeHandler());

PathTypeHandlerBase\* newTypeHandler = (PathTypeHandlerBase\*)newType->GetTypeHandler();

if (propertyId == PropertyIds::constructor)

{

newTypeHandler->isNotPathTypeHandlerOrHasUserDefinedCtor = true;

}

Assert(newType->GetIsShared() == newTypeHandler->GetIsShared());

// Don't populate inline cache if this handler isn't yet shared. If we did, a new instance could

// reach this handler without us noticing and we could fail to release the old singleton instance, which may later

// become collectible (not referenced by anything other than this handler), thus we would leak the old singleton instance.

bool populateInlineCache = newTypeHandler->GetIsShared() ||

ProcessFixedFieldChange(instance, propertyId, index, value, (flags & PropertyOperation\_NonFixedValue) != 0, propertyRecord);

SetSlotUnchecked(instance, index, value);

if (populateInlineCache)

{

Assert((instance->GetDynamicType()->GetIsShared()) || (FixPropsOnPathTypes() && instance->GetDynamicType()->GetTypeHandler()->GetIsOrMayBecomeShared()));

// Can't assert this. With NewScObject we can jump to the type handler at the tip (where the singleton is),

// even though we haven't yet initialized the properties all the way to the tip, and we don't want to kill

// the singleton in that case yet. It's basically a transient inconsistent state, but we have to live with it.

// Assert(!instance->GetTypeHandler()->HasSingletonInstance());

PropertyValueInfo::Set(info, instance, index);

}

else

{

PropertyValueInfo::SetNoCache(info, instance);

}

Assert(!IsolatePrototypes() || ((this->GetFlags() & IsPrototypeFlag) == 0));

if (this->GetFlags() & IsPrototypeFlag)

{

scriptContext->InvalidateProtoCaches(propertyId);

}

SetPropertyUpdateSideEffect(instance, propertyId, value, possibleSideEffects);

return true;

}

DynamicTypeHandler\* PathTypeHandlerBase::ConvertToTypeWithItemAttributes(DynamicObject\* instance)

{

return JavascriptArray::Is(instance) ?

ConvertToES5ArrayType(instance) : ConvertToDictionaryType(instance);

}

void PathTypeHandlerBase::ShrinkSlotAndInlineSlotCapacity()

{

if (!GetIsInlineSlotCapacityLocked())

{

PathTypeHandlerBase \* rootTypeHandler = GetRootPathTypeHandler();

bool shrunk = false;

uint16 maxPathLength = 0;

if (rootTypeHandler->GetMaxPathLength(&maxPathLength))

{

uint16 newInlineSlotCapacity =

IsObjectHeaderInlinedTypeHandler()

? RoundUpObjectHeaderInlinedInlineSlotCapacity(maxPathLength)

: RoundUpInlineSlotCapacity(maxPathLength);

if (newInlineSlotCapacity < GetInlineSlotCapacity())

{

rootTypeHandler->ShrinkSlotAndInlineSlotCapacity(newInlineSlotCapacity);

shrunk = true;

}

}

if (!shrunk)

{

rootTypeHandler->LockInlineSlotCapacity();

}

}

#if DBG

PathTypeHandlerBase \* rootTypeHandler = GetRootPathTypeHandler();

rootTypeHandler->VerifyInlineSlotCapacityIsLocked();

#endif

}

void PathTypeHandlerBase::EnsureInlineSlotCapacityIsLocked()

{

EnsureInlineSlotCapacityIsLocked(true);

#if DBG

VerifyInlineSlotCapacityIsLocked();

#endif

}

void PathTypeHandlerBase::VerifyInlineSlotCapacityIsLocked()

{

VerifyInlineSlotCapacityIsLocked(true);

}

SimplePathTypeHandler \*PathTypeHandlerBase::DeoptimizeObjectHeaderInlining(JavascriptLibrary \*const library)

{

Assert(IsObjectHeaderInlinedTypeHandler());

// Clone the type Path here to evolve separately

uint16 pathLength = typePath->pathLength;

TypePath \* clonedPath = TypePath::New(library->GetRecycler(), pathLength);

for (PropertyIndex i = 0; i < pathLength; i++)

{

clonedPath->assignments[i] = typePath->assignments[i];

clonedPath->AddInternal(clonedPath->assignments[i]);

}

// We don't copy the fixed fields, as we will be sharing this type anyways later and the fixed fields vector has to be invalidated.

SimplePathTypeHandler \*const clonedTypeHandler =

SimplePathTypeHandler::New(

library->GetScriptContext(),

clonedPath,

GetPathLength(),

static\_cast<PropertyIndex>(GetSlotCapacity()),

GetInlineSlotCapacity() - GetObjectHeaderInlinableSlotCapacity(),

sizeof(DynamicObject),

false,

false);

clonedTypeHandler->SetMayBecomeShared();

return clonedTypeHandler;

}

void PathTypeHandlerBase::SetPrototype(DynamicObject\* instance, RecyclableObject\* newPrototype)

{

ConvertToSimpleDictionaryType(instance, GetPathLength())->SetPrototype(instance, newPrototype);

}

void PathTypeHandlerBase::SetIsPrototype(DynamicObject\* instance)

{

// Don't return if IsPrototypeFlag is set, because we may still need to do a type transition and

// set fixed bits. If this handler is shared, this instance may not even be a prototype yet.

// In this case we may need to convert to a non-shared type handler.

if (!ChangeTypeOnProto() && !(GetIsOrMayBecomeShared() && IsolatePrototypes()))

{

SetFlags(IsPrototypeFlag);

return;

}

#ifdef ENABLE\_DEBUG\_CONFIG\_OPTIONS

DynamicType\* oldType = instance->GetDynamicType();

RecyclerWeakReference<DynamicObject>\* oldSingletonInstance = GetSingletonInstance();

#endif

if ((GetIsOrMayBecomeShared() && IsolatePrototypes()))

{

// The type coming in may not be shared or even locked (i.e. might have been created via DynamicObject::ChangeType()).

// In that case the type handler change below won't change the type on the object, so we have to force it.

DynamicType\* oldType = instance->GetDynamicType();

ConvertToSimpleDictionaryType(instance, GetPathLength());

if (ChangeTypeOnProto() && instance->GetDynamicType() == oldType)

{

instance->ChangeType();

}

}

else

{

#ifdef ENABLE\_DEBUG\_CONFIG\_OPTIONS

DynamicType\* oldType = instance->GetDynamicType();

RecyclerWeakReference<DynamicObject>\* oldSingletonInstance = GetSingletonInstance();

TraceFixedFieldsBeforeSetIsProto(instance, this, oldType, oldSingletonInstance);

#endif

if (ChangeTypeOnProto())

{

// If this handler is shared and we don't isolate prototypes, it's possible that the handler has

// the prototype flag, but this instance may not yet be a prototype and may not have taken

// the required type transition. It would be nice to have a reliable flag on the object

// indicating whether it's a prototype to avoid multiple type transitions if the same object

// with shared type handler is used as prototype multiple times.

if (((GetFlags() & IsPrototypeFlag) == 0) || (GetIsShared() && !IsolatePrototypes()))

{

// We're about to split out the type. If the original type was shared the handler better be shared as well.

// Otherwise, the handler would lose track of being shared between different types and instances.

Assert(!instance->HasSharedType() || instance->GetDynamicType()->GetTypeHandler()->GetIsShared());

instance->ChangeType();

Assert(!instance->HasLockedType() && !instance->HasSharedType());

}

}

}

DynamicTypeHandler\* typeHandler = GetCurrentTypeHandler(instance);

if (typeHandler != this)

{

typeHandler->SetIsPrototype(instance);

}

else

{

SetFlags(IsPrototypeFlag);

#ifdef ENABLE\_DEBUG\_CONFIG\_OPTIONS

TraceFixedFieldsAfterSetIsProto(instance, this, typeHandler, oldType, instance->GetDynamicType(), oldSingletonInstance);

#endif

}

}

bool PathTypeHandlerBase::HasSingletonInstance() const

{

Assert(HasSingletonInstanceOnlyIfNeeded());

if (!FixPropsOnPathTypes())

{

return false;

}

return this->typePath->HasSingletonInstance() && GetPathLength() >= this->typePath->GetMaxInitializedLength();

}

void PathTypeHandlerBase::DoShareTypeHandler(ScriptContext\* scriptContext)

{

DoShareTypeHandlerInternal<true>(scriptContext);

}

template <bool invalidateFixedFields>

void PathTypeHandlerBase::DoShareTypeHandlerInternal(ScriptContext\* scriptContext)

{

Assert((GetFlags() & (IsLockedFlag | MayBecomeSharedFlag | IsSharedFlag)) == (IsLockedFlag | MayBecomeSharedFlag));

Assert(!IsolatePrototypes() || !GetIsOrMayBecomeShared() || !GetIsPrototype());

#ifdef SUPPORT\_FIXED\_FIELDS\_ON\_PATH\_TYPES

// If this handler is becoming shared we need to remove the singleton instance (so that it can be collected

// if no longer referenced by anything else) and invalidate any fixed fields.

if (FixPropsOnPathTypes())

{

if (invalidateFixedFields)

{

if (this->typePath->GetMaxInitializedLength() < GetPathLength())

{

this->typePath->SetMaxInitializedLength(GetPathLength());

}

for (PropertyIndex index = 0; index < this->GetPathLength(); index++)

{

InvalidateFixedFieldAt(this->typePath->GetPropertyIdUnchecked(index)->GetPropertyId(), index, scriptContext);

}

}

Assert(HasOnlyInitializedNonFixedProperties());

Assert(HasSingletonInstanceOnlyIfNeeded());

if (HasSingletonInstance())

{

this->typePath->ClearSingletonInstance();

}

}

#endif

}

void PathTypeHandlerBase::InvalidateFixedFieldAt(Js::PropertyId propertyId, Js::PropertyIndex index, ScriptContext\* scriptContext)

{

if (!FixPropsOnPathTypes())

{

return;

}

// We are adding a new value where some other instance already has an existing value. If this is a fixed

// field we must clear the bit. If the value was hard coded in the JIT-ed code, we must invalidate the guards.

if (this->typePath->GetIsUsedFixedFieldAt(index, GetPathLength()))

{

// We may be a second instance chasing the singleton and invalidating fixed fields along the way.

// Assert(newTypeHandler->typePath->GetSingletonInstance() == instance);

// Invalidate any JIT-ed code that hard coded this method. No need to invalidate store field

// inline caches (which might quietly overwrite this fixed fields, because they have never been populated.

#if ENABLE\_NATIVE\_CODEGEN

scriptContext->GetThreadContext()->InvalidatePropertyGuards(propertyId);

#endif

}

// If we're overwriting an existing value of this property, we don't consider the new one fixed.

// This also means that it's ok to populate the inline caches for this property from now on.

this->typePath->ClearIsFixedFieldAt(index, GetPathLength());

}

void PathTypeHandlerBase::AddBlankFieldAt(Js::PropertyId propertyId, Js::PropertyIndex index, ScriptContext\* scriptContext)

{

if (!FixPropsOnPathTypes())

{

return;

}

if (index >= this->typePath->GetMaxInitializedLength())

{

// We are a adding a property where no instance property has been set before. We rely on properties being

// added in order of indexes to be sure that we don't leave any uninitialized properties interspersed with

// initialized ones, which could lead to incorrect behavior. See comment in TypePath::Branch.

AssertMsg(index == this->typePath->GetMaxInitializedLength(), "Adding properties out of order?");

this->typePath->AddBlankFieldAt(index, GetPathLength());

}

else

{

InvalidateFixedFieldAt(propertyId, index, scriptContext);

// We have now reached the most advanced instance along this path. If this instance is not the singleton instance,

// then the former singleton instance (if any) is no longer a singleton. This instance could be the singleton

// instance, if we just happen to set (overwrite) its last property.

if (index + 1 == this->typePath->GetMaxInitializedLength())

{

// If we cleared the singleton instance while some fields remained fixed, the instance would

// be collectible, and yet some code would expect to see values and call methods on it. We rely on the

// fact that we always add properties to (pre-initialized) type handlers in the order they appear

// on the type path. By the time we reach the singleton instance, all fixed fields will have been invalidated.

// Otherwise, some fields could remain fixed (or even uninitialized) and we would have to spin off a loop here

// to invalidate any remaining fixed fields

Assert(HasSingletonInstanceOnlyIfNeeded());

this->typePath->ClearSingletonInstance();

}

}

}

bool PathTypeHandlerBase::ProcessFixedFieldChange(DynamicObject\* instance, PropertyId propertyId, PropertyIndex slotIndex, Var value, bool isNonFixed,const PropertyRecord \* propertyRecord)

{

Assert(!instance->GetTypeHandler()->GetIsShared());

// We don't want fixed properties on external objects, either external properties or expando properties.

// See DynamicObject::ResetObject for more information.

Assert(!instance->IsExternal() || isNonFixed);

if (!FixPropsOnPathTypes())

{

return true;

}

bool populateInlineCache = true;

PathTypeHandler\* newTypeHandler = (PathTypeHandler\*)instance->GetTypeHandler();

if (slotIndex >= newTypeHandler->typePath->GetMaxInitializedLength())

{

// We are a adding a property where no instance property has been set before. We rely on properties being

// added in order of indexes to be sure that we don't leave any uninitialized properties interspersed with

// initialized ones, which could lead to incorrect behavior. See comment in TypePath::Branch.

AssertMsg(slotIndex == newTypeHandler->typePath->GetMaxInitializedLength(), "Adding properties out of order?");

// Consider: It would be nice to assert the slot is actually null. However, we sometimes pre-initialize to

// undefined or even some other special illegal value (for let or const, currently == null)

// Assert(instance->GetSlot(index) == nullptr);

if (ShouldFixAnyProperties() && CanBeSingletonInstance(instance))

{

bool markAsFixed = !isNonFixed && !IsInternalPropertyId(propertyId) &&

(JavascriptFunction::Is(value) ? ShouldFixMethodProperties() || ShouldFixAccessorProperties() :

(ShouldFixDataProperties() && CheckHeuristicsForFixedDataProps(instance, propertyRecord, propertyId, value)));

// Mark the newly added field as fixed and prevent population of inline caches.

newTypeHandler->typePath->AddSingletonInstanceFieldAt(instance, slotIndex, markAsFixed, newTypeHandler->GetPathLength());

}

else

{

newTypeHandler->typePath->AddSingletonInstanceFieldAt(slotIndex, newTypeHandler->GetPathLength());

}

populateInlineCache = false;

}

else

{

newTypeHandler->InvalidateFixedFieldAt(propertyId, slotIndex, instance->GetScriptContext());

// We have now reached the most advanced instance along this path. If this instance is not the singleton instance,

// then the former singleton instance (if any) is no longer a singleton. This instance could be the singleton

// instance, if we just happen to set (overwrite) its last property.

if (slotIndex + 1 == newTypeHandler->typePath->GetMaxInitializedLength())

{

// If we cleared the singleton instance while some fields remained fixed, the instance would

// be collectible, and yet some code would expect to see values and call methods on it. We rely on the

// fact that we always add properties to (pre-initialized) type handlers in the order they appear

// on the type path. By the time we reach the singleton instance, all fixed fields will have been invalidated.

// Otherwise, some fields could remain fixed (or even uninitialized) and we would have to spin off a loop here

// to invalidate any remaining fixed fields

auto singletonWeakRef = newTypeHandler->typePath->GetSingletonInstance();

if (singletonWeakRef != nullptr && instance != singletonWeakRef->Get())

{

Assert(newTypeHandler->HasSingletonInstanceOnlyIfNeeded());

newTypeHandler->typePath->ClearSingletonInstance();

}

}

}

// If we branched and this is the singleton instance, we need to remove it from this type handler. The only time

// this can happen is when another not fully initialized instance is ahead of this one on the current path.

auto singletonWeakRef = this->typePath->GetSingletonInstance();

if (newTypeHandler->typePath != this->typePath && singletonWeakRef != nullptr && singletonWeakRef->Get() == instance)

{

// If this is the singleton instance, there shouldn't be any other initialized instance ahead of it on the old path.

Assert(GetPathLength() >= this->typePath->GetMaxInitializedLength());

Assert(HasSingletonInstanceOnlyIfNeeded());

this->typePath->ClearSingletonInstance();

}

return populateInlineCache;

}

bool PathTypeHandlerBase::TryUseFixedProperty(PropertyRecord const \* propertyRecord, Var \* pProperty, FixedPropertyKind propertyType, ScriptContext \* requestContext)

{

bool result = TryGetFixedProperty<false, true>(propertyRecord, pProperty, propertyType, requestContext);

TraceUseFixedProperty(propertyRecord, pProperty, result, L"PathTypeHandler", requestContext);

return result;

}

bool PathTypeHandlerBase::TryUseFixedAccessor(PropertyRecord const \* propertyRecord, Var \* pAccessor, FixedPropertyKind propertyType, bool getter, ScriptContext \* requestContext)

{

if (PHASE\_VERBOSE\_TRACE1(Js::FixedMethodsPhase) || PHASE\_VERBOSE\_TESTTRACE1(Js::FixedMethodsPhase) ||

PHASE\_VERBOSE\_TRACE1(Js::UseFixedDataPropsPhase) || PHASE\_VERBOSE\_TESTTRACE1(Js::UseFixedDataPropsPhase))

{

Output::Print(L"FixedFields: attempt to use fixed accessor %s from PathTypeHandler returned false.\n", propertyRecord->GetBuffer());

if (this->HasSingletonInstance() && this->GetSingletonInstance()->Get()->GetScriptContext() != requestContext)

{

Output::Print(L"FixedFields: Cross Site Script Context is used for property %s. \n", propertyRecord->GetBuffer());

}

Output::Flush();

}

return false;

}

#if DBG

bool PathTypeHandlerBase::CanStorePropertyValueDirectly(const DynamicObject\* instance, PropertyId propertyId, bool allowLetConst)

{

Assert(!allowLetConst);

// We pass Constants::NoProperty for ActivationObjects for functions with same named formals, but we don't

// use PathTypeHandlers for those.

Assert(propertyId != Constants::NoProperty);

Js::PropertyIndex index = GetPropertyIndex(propertyId);

if (index != Constants::NoSlot)

{

if (FixPropsOnPathTypes())

{

return index < this->typePath->GetMaxInitializedLength() && !this->typePath->GetIsFixedFieldAt(index, this->GetPathLength());

}

else

{

return true;

}

}

else

{

AssertMsg(false, "Asking about a property this type handler doesn't know about?");

return false;

}

}

bool PathTypeHandlerBase::HasOnlyInitializedNonFixedProperties()

{

#ifdef SUPPORT\_FIXED\_FIELDS\_ON\_PATH\_TYPES

if (this->typePath->GetMaxInitializedLength() < GetPathLength())

{

return false;

}

for (PropertyIndex index = 0; index < this->GetPathLength(); index++)

{

if (this->typePath->GetIsFixedFieldAt(index, this->GetPathLength()))

{

return false;

}

}

#endif

return true;

}

bool PathTypeHandlerBase::CheckFixedProperty(PropertyRecord const \* propertyRecord, Var \* pProperty, ScriptContext \* requestContext)

{

return TryGetFixedProperty<true, false>(propertyRecord, pProperty, (Js::FixedPropertyKind)(Js::FixedPropertyKind::FixedMethodProperty | Js::FixedPropertyKind::FixedDataProperty), requestContext);

}

bool PathTypeHandlerBase::HasAnyFixedProperties() const

{

int pathLength = GetPathLength();

for (PropertyIndex i = 0; i < pathLength; i++)

{

if (this->typePath->GetIsFixedFieldAt(i, pathLength))

{

return true;

}

}

return false;

}

#endif

template <bool allowNonExistent, bool markAsUsed>

bool PathTypeHandlerBase::TryGetFixedProperty(PropertyRecord const \* propertyRecord, Var \* pProperty, Js::FixedPropertyKind propertyType, ScriptContext \* requestContext)

{

if (!FixPropsOnPathTypes())

{

return false;

}

PropertyIndex index = this->typePath->Lookup(propertyRecord->GetPropertyId(), GetPathLength());

if (index == Constants::NoSlot)

{

AssertMsg(allowNonExistent, "Trying to get a fixed function instance for a non-existent property?");

return false;

}

Var value = this->typePath->GetSingletonFixedFieldAt(index, GetPathLength(), requestContext);

if (value && ((IsFixedMethodProperty(propertyType) && JavascriptFunction::Is(value)) || IsFixedDataProperty(propertyType)))

{

\*pProperty = value;

if (markAsUsed)

{

this->typePath->SetIsUsedFixedFieldAt(index, GetPathLength());

}

return true;

}

else

{

return false;

}

}

PathTypeHandlerBase\* PathTypeHandlerBase::GetRootPathTypeHandler()

{

PathTypeHandlerBase\* rootTypeHandler = this;

while (rootTypeHandler->predecessorType != nullptr)

{

rootTypeHandler = PathTypeHandlerBase::FromTypeHandler(rootTypeHandler->predecessorType->GetTypeHandler());

}

Assert(rootTypeHandler->predecessorType == nullptr);

return rootTypeHandler;

}

#ifdef ENABLE\_DEBUG\_CONFIG\_OPTIONS

void PathTypeHandlerBase::DumpFixedFields() const {

if (FixPropsOnPathTypes())

{

for (PropertyIndex i = 0; i < GetPathLength(); i++)

{

Output::Print(L" %s %d%d%d,", typePath->GetPropertyId(i)->GetBuffer(),

i < this->typePath->GetMaxInitializedLength() ? 1 : 0,

this->typePath->GetIsFixedFieldAt(i, GetPathLength()) ? 1 : 0,

this->typePath->GetIsUsedFixedFieldAt(i, GetPathLength()) ? 1 : 0);

}

}

else

{

for (PropertyIndex i = 0; i < GetPathLength(); i++)

{

Output::Print(L" %s %d%d%d,", typePath->GetPropertyId(i)->GetBuffer(), 1, 0, 0);

}

}

}

void PathTypeHandlerBase::TraceFixedFieldsBeforeTypeHandlerChange(

const wchar\_t\* conversionName, const wchar\_t\* oldTypeHandlerName, const wchar\_t\* newTypeHandlerName,

DynamicObject\* instance, DynamicTypeHandler\* oldTypeHandler,

DynamicType\* oldType, RecyclerWeakReference<DynamicObject>\* oldSingletonInstanceBefore)

{

if (PHASE\_VERBOSE\_TRACE1(FixMethodPropsPhase))

{

Output::Print(L"FixedFields: %s 0x%p from %s to %s:\n", conversionName, instance, oldTypeHandlerName, newTypeHandlerName);

Output::Print(L" before: type = 0x%p, type handler = 0x%p, old singleton = 0x%p(0x%p)\n",

oldType, oldTypeHandler, oldSingletonInstanceBefore, oldSingletonInstanceBefore != nullptr ? oldSingletonInstanceBefore->Get() : nullptr);

Output::Print(L" fixed fields:");

oldTypeHandler->DumpFixedFields();

Output::Print(L"\n");

}

if (PHASE\_VERBOSE\_TESTTRACE1(FixMethodPropsPhase))

{

Output::Print(L"FixedFields: %s instance from %s to %s:\n", conversionName, oldTypeHandlerName, newTypeHandlerName);

Output::Print(L" old singleton before %s null \n", oldSingletonInstanceBefore == nullptr ? L"==" : L"!=");

Output::Print(L" fixed fields before:");

oldTypeHandler->DumpFixedFields();

Output::Print(L"\n");

}

}

void PathTypeHandlerBase::TraceFixedFieldsAfterTypeHandlerChange(

DynamicObject\* instance, DynamicTypeHandler\* oldTypeHandler, DynamicTypeHandler\* newTypeHandler,

DynamicType\* oldType, DynamicType\* newType, RecyclerWeakReference<DynamicObject>\* oldSingletonInstanceBefore)

{

if (PHASE\_VERBOSE\_TRACE1(FixMethodPropsPhase))

{

RecyclerWeakReference<DynamicObject>\* oldSingletonInstanceAfter = oldTypeHandler->GetSingletonInstance();

RecyclerWeakReference<DynamicObject>\* newSingletonInstanceAfter = newTypeHandler->GetSingletonInstance();

Output::Print(L" after: type = 0x%p, type handler = 0x%p, old singleton = 0x%p(0x%p), new singleton = 0x%p(0x%p)\n",

newType, newTypeHandler, oldSingletonInstanceAfter, oldSingletonInstanceAfter != nullptr ? oldSingletonInstanceAfter->Get() : nullptr,

newSingletonInstanceAfter, newSingletonInstanceAfter != nullptr ? newSingletonInstanceAfter->Get() : nullptr);

Output::Print(L" fixed fields:");

newTypeHandler->DumpFixedFields();

Output::Print(L"\n");

Output::Flush();

}

if (PHASE\_VERBOSE\_TESTTRACE1(FixMethodPropsPhase))

{

Output::Print(L" type %s, typeHandler %s, old singleton after %s null (%s), new singleton after %s null\n",

oldTypeHandler != newTypeHandler ? L"changed" : L"unchanged",

oldType != newType ? L"changed" : L"unchanged",

oldTypeHandler->GetSingletonInstance() == nullptr ? L"==" : L"!=",

oldSingletonInstanceBefore != oldTypeHandler->GetSingletonInstance() ? L"changed" : L"unchanged",

newTypeHandler->GetSingletonInstance() == nullptr ? L"==" : L"!=");

Output::Print(L" fixed fields after:");

newTypeHandler->DumpFixedFields();

Output::Print(L"\n");

Output::Flush();

}

}

void PathTypeHandlerBase::TraceFixedFieldsBeforeSetIsProto(

DynamicObject\* instance, DynamicTypeHandler\* oldTypeHandler, DynamicType\* oldType, RecyclerWeakReference<DynamicObject>\* oldSingletonInstanceBefore)

{

if (PHASE\_VERBOSE\_TRACE1(FixMethodPropsPhase))

{

Output::Print(L"FixedFields: PathTypeHandler::SetIsPrototype(0x%p):\n", instance);

Output::Print(L" before: type = 0x%p, type handler = 0x%p, old singleton = 0x%p(0x%p)\n",

oldType, oldTypeHandler, oldSingletonInstanceBefore, oldSingletonInstanceBefore != nullptr ? oldSingletonInstanceBefore->Get() : nullptr);

Output::Print(L" fixed fields:");

oldTypeHandler->DumpFixedFields();

Output::Print(L"\n");

}

if (PHASE\_VERBOSE\_TESTTRACE1(FixMethodPropsPhase))

{

Output::Print(L"FixedFields: PathTypeHandler::SetIsPrototype():\n");

Output::Print(L" old singleton before %s null \n", oldSingletonInstanceBefore == nullptr ? L"==" : L"!=");

Output::Print(L" fixed fields before:");

oldTypeHandler->DumpFixedFields();

Output::Print(L"\n");

}

}

void PathTypeHandlerBase::TraceFixedFieldsAfterSetIsProto(

DynamicObject\* instance, DynamicTypeHandler\* oldTypeHandler, DynamicTypeHandler\* newTypeHandler,

DynamicType\* oldType, DynamicType\* newType, RecyclerWeakReference<DynamicObject>\* oldSingletonInstanceBefore)

{

if (PHASE\_VERBOSE\_TRACE1(FixMethodPropsPhase))

{

RecyclerWeakReference<DynamicObject>\* oldSingletonInstanceAfter = oldTypeHandler->GetSingletonInstance();

RecyclerWeakReference<DynamicObject>\* newSingletonInstanceAfter = newTypeHandler->GetSingletonInstance();

Output::Print(L" after: type = 0x%p, type handler = 0x%p, old singleton = 0x%p(0x%p), new singleton = 0x%p(0x%p)\n",

instance->GetType(), newTypeHandler,

oldSingletonInstanceAfter, oldSingletonInstanceAfter != nullptr ? oldSingletonInstanceAfter->Get() : nullptr,

newSingletonInstanceAfter, newSingletonInstanceAfter != nullptr ? newSingletonInstanceAfter->Get() : nullptr);

Output::Print(L" fixed fields:");

newTypeHandler->DumpFixedFields();

Output::Print(L"\n");

Output::Flush();

}

if (PHASE\_VERBOSE\_TESTTRACE1(FixMethodPropsPhase))

{

Output::Print(L" type %s, old singleton after %s null (%s)\n",

oldType != newType ? L"changed" : L"unchanged",

oldSingletonInstanceBefore == nullptr ? L"==" : L"!=",

oldSingletonInstanceBefore != oldTypeHandler->GetSingletonInstance() ? L"changed" : L"unchanged");

Output::Print(L" fixed fields after:");

newTypeHandler->DumpFixedFields();

Output::Print(L"\n");

Output::Flush();

}

}

#endif

SimplePathTypeHandler \* SimplePathTypeHandler::New(ScriptContext \* scriptContext, TypePath\* typePath, uint16 pathLength, uint16 inlineSlotCapacity, uint16 offsetOfInlineSlots, bool isLocked, bool isShared, DynamicType\* predecessorType)

{

return New(scriptContext, typePath, pathLength, max(pathLength, inlineSlotCapacity), inlineSlotCapacity, offsetOfInlineSlots, isLocked, isShared, predecessorType);

}

SimplePathTypeHandler \* SimplePathTypeHandler::New(ScriptContext \* scriptContext, TypePath\* typePath, uint16 pathLength, const PropertyIndex slotCapacity, uint16 inlineSlotCapacity, uint16 offsetOfInlineSlots, bool isLocked, bool isShared, DynamicType\* predecessorType)

{

Assert(typePath != nullptr);

#ifdef PROFILE\_TYPES

scriptContext->simplePathTypeHandlerCount++;

#endif

return RecyclerNew(scriptContext->GetRecycler(), SimplePathTypeHandler, typePath, pathLength, slotCapacity, inlineSlotCapacity, offsetOfInlineSlots, isLocked, isShared, predecessorType);

}

SimplePathTypeHandler \* SimplePathTypeHandler::New(ScriptContext \* scriptContext, SimplePathTypeHandler \* typeHandler, bool isLocked, bool isShared)

{

Assert(typeHandler != nullptr);

return RecyclerNew(scriptContext->GetRecycler(), SimplePathTypeHandler, typeHandler->GetTypePath(), typeHandler->GetPathLength(), typeHandler->GetInlineSlotCapacity(), typeHandler->GetOffsetOfInlineSlots(), isLocked, isShared);

}

SimplePathTypeHandler::SimplePathTypeHandler(TypePath\* typePath, uint16 pathLength, const PropertyIndex slotCapacity, uint16 inlineSlotCapacity, uint16 offsetOfInlineSlots, bool isLocked, bool isShared, DynamicType\* predecessorType) :

PathTypeHandlerBase(typePath, pathLength, slotCapacity, inlineSlotCapacity, offsetOfInlineSlots, isLocked, isShared, predecessorType),

successorPropertyRecord(nullptr),

successorTypeWeakRef(nullptr)

{

}

void SimplePathTypeHandler::ShrinkSlotAndInlineSlotCapacity(uint16 newInlineSlotCapacity)

{

Assert(!this->GetIsInlineSlotCapacityLocked());

this->SetInlineSlotCapacity(newInlineSlotCapacity);

this->SetSlotCapacity(newInlineSlotCapacity);

this->SetIsInlineSlotCapacityLocked();

if (this->successorPropertyRecord)

{

DynamicType \* type = successorTypeWeakRef->Get();

if (type)

{

PathTypeHandlerBase::FromTypeHandler(type->GetTypeHandler())->ShrinkSlotAndInlineSlotCapacity(newInlineSlotCapacity);

}

}

}

void SimplePathTypeHandler::LockInlineSlotCapacity()

{

Assert(!GetIsInlineSlotCapacityLocked());

SetIsInlineSlotCapacityLocked();

if (!successorPropertyRecord)

{

return;

}

DynamicType \* type = successorTypeWeakRef->Get();

if (type)

{

type->GetTypeHandler()->LockInlineSlotCapacity();

}

}

void SimplePathTypeHandler::EnsureInlineSlotCapacityIsLocked(bool startFromRoot)

{

if (startFromRoot)

{

GetRootPathTypeHandler()->EnsureInlineSlotCapacityIsLocked(false);

return;

}

Assert(!startFromRoot);

if (!GetIsInlineSlotCapacityLocked())

{

SetIsInlineSlotCapacityLocked();

if (successorPropertyRecord)

{

DynamicType \* type = successorTypeWeakRef->Get();

if (type)

{

DynamicTypeHandler\* successorTypeHandler = type->GetTypeHandler();

successorTypeHandler->IsPathTypeHandler() ?

PathTypeHandler::FromTypeHandler(successorTypeHandler)->EnsureInlineSlotCapacityIsLocked(false) :

successorTypeHandler->EnsureInlineSlotCapacityIsLocked();

}

}

}

}

void SimplePathTypeHandler::VerifyInlineSlotCapacityIsLocked(bool startFromRoot)

{

if (startFromRoot)

{

GetRootPathTypeHandler()->VerifyInlineSlotCapacityIsLocked(false);

return;

}

Assert(!startFromRoot);

Assert(GetIsInlineSlotCapacityLocked());

if (!successorPropertyRecord)

{

return;

}

DynamicType \* type = successorTypeWeakRef->Get();

if (type)

{

DynamicTypeHandler\* successorTypeHandler = type->GetTypeHandler();

successorTypeHandler->IsPathTypeHandler() ?

PathTypeHandler::FromTypeHandler(successorTypeHandler)->VerifyInlineSlotCapacityIsLocked(false) :

successorTypeHandler->VerifyInlineSlotCapacityIsLocked();

}

}

bool SimplePathTypeHandler::GetMaxPathLength(uint16 \* maxPathLength)

{

if (GetPathLength() > \*maxPathLength)

{

\*maxPathLength = GetPathLength();

}

if (!successorPropertyRecord)

{

return true;

}

DynamicType \* type = successorTypeWeakRef->Get();

if (type)

{

if (!type->GetTypeHandler()->IsPathTypeHandler())

{

return false;

}

if (!PathTypeHandlerBase::FromTypeHandler(type->GetTypeHandler())->GetMaxPathLength(maxPathLength))

{

return false;

}

}

return true;

}

bool SimplePathTypeHandler::GetSuccessor(const PropertyRecord\* propertyRecord, RecyclerWeakReference<DynamicType> \*\* typeWeakRef)

{

if (successorPropertyRecord != propertyRecord)

{

\*typeWeakRef = nullptr;

return false;

}

\*typeWeakRef = successorTypeWeakRef;

return true;

}

void SimplePathTypeHandler::SetSuccessor(DynamicType \* type, const PropertyRecord\* propertyRecord, RecyclerWeakReference<DynamicType> \* typeWeakRef, ScriptContext \* scriptContext)

{

if (!successorPropertyRecord || successorPropertyRecord == propertyRecord || !successorTypeWeakRef->Get())

{

successorPropertyRecord = propertyRecord;

successorTypeWeakRef = typeWeakRef;

return;

}

// This is an interesting transition from the fixed fields perspective. If there are any other types using this type handler

// (which can happen if we don't isolate prototypes but force type change on becoming proto), they will continue to do so. So

// we will have two different type handlers at the exact same point in type path evolution sharing the same type path, and

// consequently all fixed field info as well. This is fine, because fixed field management is done at the type path level.

PathTypeHandler \* newTypeHandler = PathTypeHandler::New(scriptContext, GetTypePath(), GetPathLength(), static\_cast<PropertyIndex>(GetSlotCapacity()), GetInlineSlotCapacity(), GetOffsetOfInlineSlots(), true, true, GetPredecessorType());

newTypeHandler->SetSuccessor(type, this->successorPropertyRecord, this->successorTypeWeakRef, scriptContext);

newTypeHandler->SetSuccessor(type, propertyRecord, typeWeakRef, scriptContext);

newTypeHandler->SetFlags(IsPrototypeFlag, GetFlags());

newTypeHandler->CopyPropertyTypes(PropertyTypesWritableDataOnly | PropertyTypesWritableDataOnlyDetection | PropertyTypesInlineSlotCapacityLocked, this->GetPropertyTypes());

// We don't transfer any fixed field data because we assume the type path remains the same.

Assert(newTypeHandler->GetTypePath() == this->GetTypePath());

Assert(type->typeHandler == this);

type->typeHandler = newTypeHandler;

#ifdef PROFILE\_TYPES

scriptContext->convertSimplePathToPathCount++;

#endif

}

PathTypeHandler \* PathTypeHandler::New(ScriptContext \* scriptContext, TypePath\* typePath, uint16 pathLength, uint16 inlineSlotCapacity, uint16 offsetOfInlineSlots, bool isLocked, bool isShared, DynamicType\* predecessorType)

{

return New(scriptContext, typePath, pathLength, max(pathLength, inlineSlotCapacity), inlineSlotCapacity, offsetOfInlineSlots, isLocked, isShared, predecessorType);

}

PathTypeHandler \* PathTypeHandler::New(ScriptContext \* scriptContext, TypePath\* typePath, uint16 pathLength, const PropertyIndex slotCapacity, uint16 inlineSlotCapacity, uint16 offsetOfInlineSlots, bool isLocked, bool isShared, DynamicType\* predecessorType)

{

Assert(typePath != nullptr);

#ifdef PROFILE\_TYPES

scriptContext->pathTypeHandlerCount++;

#endif

return RecyclerNew(scriptContext->GetRecycler(), PathTypeHandler, typePath, pathLength, slotCapacity, inlineSlotCapacity, offsetOfInlineSlots, isLocked, isShared, predecessorType);

}

PathTypeHandler \* PathTypeHandler::New(ScriptContext \* scriptContext, PathTypeHandler \* typeHandler, bool isLocked, bool isShared)

{

Assert(typeHandler != nullptr);

return RecyclerNew(scriptContext->GetRecycler(), PathTypeHandler, typeHandler->GetTypePath(), typeHandler->GetPathLength(), static\_cast<PropertyIndex>(typeHandler->GetSlotCapacity()), typeHandler->GetInlineSlotCapacity(), typeHandler->GetOffsetOfInlineSlots(), isLocked, isShared);

}

PathTypeHandler::PathTypeHandler(TypePath\* typePath, uint16 pathLength, const PropertyIndex slotCapacity, uint16 inlineSlotCapacity, uint16 offsetOfInlineSlots, bool isLocked, bool isShared, DynamicType\* predecessorType) :

PathTypeHandlerBase(typePath, pathLength, slotCapacity, inlineSlotCapacity, offsetOfInlineSlots, isLocked, isShared, predecessorType),

propertySuccessors(nullptr)

{

}

void PathTypeHandler::ShrinkSlotAndInlineSlotCapacity(uint16 newInlineSlotCapacity)

{

Assert(!this->GetIsInlineSlotCapacityLocked());

this->SetInlineSlotCapacity(newInlineSlotCapacity);

// Slot capacity should also be shrunk when the inlineSlotCapacity is shrunk.

this->SetSlotCapacity(newInlineSlotCapacity);

this->SetIsInlineSlotCapacityLocked();

if (this->propertySuccessors)

{

this->propertySuccessors->Map([newInlineSlotCapacity](PropertyId, RecyclerWeakReference<DynamicType> \* typeWeakReference)

{

DynamicType \* type = typeWeakReference->Get();

if (type)

{

PathTypeHandlerBase::FromTypeHandler(type->GetTypeHandler())->ShrinkSlotAndInlineSlotCapacity(newInlineSlotCapacity);

}

});

}

}

void PathTypeHandler::LockInlineSlotCapacity()

{

Assert(!GetIsInlineSlotCapacityLocked());

SetIsInlineSlotCapacityLocked();

if (!propertySuccessors || propertySuccessors->Count() == 0)

{

return;

}

this->propertySuccessors->Map([](const PropertyId, RecyclerWeakReference<DynamicType>\* typeWeakReference)

{

DynamicType \* type = typeWeakReference->Get();

if (!type)

{

return;

}

type->GetTypeHandler()->LockInlineSlotCapacity();

});

}

void PathTypeHandler::EnsureInlineSlotCapacityIsLocked(bool startFromRoot)

{

if (startFromRoot)

{

GetRootPathTypeHandler()->EnsureInlineSlotCapacityIsLocked(false);

return;

}

Assert(!startFromRoot);

if (!GetIsInlineSlotCapacityLocked())

{

SetIsInlineSlotCapacityLocked();

if (propertySuccessors && propertySuccessors->Count() > 0)

{

this->propertySuccessors->Map([](const PropertyId, RecyclerWeakReference<DynamicType> \* typeWeakReference)

{

DynamicType \* type = typeWeakReference->Get();

if (!type)

{

return;

}

DynamicTypeHandler\* successorTypeHandler = type->GetTypeHandler();

successorTypeHandler->IsPathTypeHandler() ?

PathTypeHandler::FromTypeHandler(successorTypeHandler)->EnsureInlineSlotCapacityIsLocked(false) :

successorTypeHandler->EnsureInlineSlotCapacityIsLocked();

});

}

}

}

void PathTypeHandler::VerifyInlineSlotCapacityIsLocked(bool startFromRoot)

{

if (startFromRoot)

{

GetRootPathTypeHandler()->VerifyInlineSlotCapacityIsLocked(false);

return;

}

Assert(!startFromRoot);

Assert(GetIsInlineSlotCapacityLocked());

if (!propertySuccessors || propertySuccessors->Count() == 0)

{

return;

}

this->propertySuccessors->Map([](const PropertyId, RecyclerWeakReference<DynamicType> \* typeWeakReference)

{

DynamicType \* type = typeWeakReference->Get();

if (!type)

{

return;

}

DynamicTypeHandler\* successorTypeHandler = type->GetTypeHandler();

successorTypeHandler->IsPathTypeHandler() ?

PathTypeHandler::FromTypeHandler(successorTypeHandler)->VerifyInlineSlotCapacityIsLocked(false) :

successorTypeHandler->VerifyInlineSlotCapacityIsLocked();

});

}

bool PathTypeHandler::GetMaxPathLength(uint16 \* maxPathLength)

{

if (GetPropertyCount() > \*maxPathLength)

{

\*maxPathLength = GetPathLength();

}

if (!propertySuccessors || propertySuccessors->Count() == 0)

{

return true;

}

bool result = true;

this->propertySuccessors->MapUntil([&result, maxPathLength](PropertyId, RecyclerWeakReference<DynamicType> \* typeWeakReference) -> bool

{

DynamicType \* type = typeWeakReference->Get();

if (!type)

{

return false;

}

if (!type->GetTypeHandler()->IsPathTypeHandler())

{

result = false;

return true;

}

if (!PathTypeHandlerBase::FromTypeHandler(type->GetTypeHandler())->GetMaxPathLength(maxPathLength))

{

result = false;

return true;

}

return false;

});

return result;

}

bool PathTypeHandler::GetSuccessor(const PropertyRecord\* propertyRecord, RecyclerWeakReference<DynamicType> \*\* typeWeakRef)

{

if (!propertySuccessors || !propertySuccessors->TryGetValue(propertyRecord->GetPropertyId(), typeWeakRef))

{

\*typeWeakRef = nullptr;

return false;

}

return true;

}

void PathTypeHandler::SetSuccessor(DynamicType \* type, const PropertyRecord\* propertyRecord, RecyclerWeakReference<DynamicType> \* typeWeakRef, ScriptContext \* scriptContext)

{

if (!propertySuccessors)

{

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

propertySuccessors = RecyclerNew(recycler, PropertySuccessorsMap, recycler, 3);

}

propertySuccessors->Item(propertyRecord->GetPropertyId(), typeWeakRef);

}

}

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

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

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

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

#pragma once

namespace Js

{

class PathTypeHandlerBase : public DynamicTypeHandler

{

friend class DynamicObject;

friend class SimplePathTypeHandler;

friend class PathTypeHandler;

private:

TypePath\* typePath;

DynamicType\* predecessorType; // Strong reference to predecessor type so that predecessor types remain in the cache even though they might not be used

public:

DEFINE\_GETCPPNAME();

protected:

PathTypeHandlerBase(TypePath\* typePath, uint16 pathLength, const PropertyIndex slotCapacity, uint16 inlineSlotCapacity, uint16 offsetOfInlineSlots, bool isLocked = false, bool isShared = false, DynamicType\* predecessorType = nullptr);

DEFINE\_VTABLE\_CTOR\_NO\_REGISTER(PathTypeHandlerBase, DynamicTypeHandler, typePath(0));

public:

virtual BOOL IsLockable() const override { return true; }

virtual BOOL IsSharable() const override { return true; }

virtual void DoShareTypeHandler(ScriptContext\* scriptContext) override;

static bool UsePathTypeHandlerForObjectLiteral(const PropertyIdArray \*const propIds, ScriptContext \*const scriptContext, bool \*const check\_\_proto\_\_Ref = nullptr);

static DynamicType\* CreateTypeForNewScObject(ScriptContext\* scriptContext, DynamicType\* type, const Js::PropertyIdArray \*propIds, bool shareType);

static DynamicType\* CreateNewScopeObject(ScriptContext\* scriptContext, DynamicType\* type, const Js::PropertyIdArray \*propIds, PropertyAttributes extraAttributes = PropertyNone, uint extraAttributesSlotCount = UINT\_MAX);

static PathTypeHandlerBase \* FromTypeHandler(DynamicTypeHandler \* const typeHandler) { Assert(typeHandler->IsPathTypeHandler()); return static\_cast<PathTypeHandlerBase\*>(typeHandler); }

virtual int GetPropertyCount() override;

virtual PropertyId GetPropertyId(ScriptContext\* scriptContext, PropertyIndex index) override;

virtual PropertyId GetPropertyId(ScriptContext\* scriptContext, BigPropertyIndex index) override;

virtual PropertyIndex GetPropertyIndex(const PropertyRecord\* propertyRecord) override;

virtual bool GetPropertyEquivalenceInfo(PropertyRecord const\* propertyRecord, PropertyEquivalenceInfo& info) override;

virtual bool IsObjTypeSpecEquivalent(const Type\* type, const TypeEquivalenceRecord& record, uint& failedPropertyIndex) override;

virtual bool IsObjTypeSpecEquivalent(const Type\* type, const EquivalentPropertyEntry\* entry) override;

virtual BOOL FindNextProperty(ScriptContext\* scriptContext, PropertyIndex& index, JavascriptString\*\* propertyString,

PropertyId\* propertyId, PropertyAttributes\* attributes, Type\* type, DynamicType \*typeToEnumerate, bool requireEnumerable, bool enumSymbols = false) override;

virtual BOOL HasProperty(DynamicObject\* instance, PropertyId propertyId, \_\_out\_opt bool \*noRedecl = nullptr) override;

virtual BOOL HasProperty(DynamicObject\* instance, JavascriptString\* propertyNameString) override;

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

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

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

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

virtual BOOL DeleteProperty(DynamicObject\* instance, PropertyId propertyId, PropertyOperationFlags flags) override;

virtual BOOL IsFixedProperty(const DynamicObject\* instance, PropertyId propertyId) override;

virtual BOOL IsEnumerable(DynamicObject\* instance, PropertyId propertyId) override;

virtual BOOL IsWritable(DynamicObject\* instance, PropertyId propertyId) override;

virtual BOOL IsConfigurable(DynamicObject\* instance, PropertyId propertyId) override;

virtual BOOL SetEnumerable(DynamicObject\* instance, PropertyId propertyId, BOOL value) override;

virtual BOOL SetWritable(DynamicObject\* instance, PropertyId propertyId, BOOL value) override;

virtual BOOL SetConfigurable(DynamicObject\* instance, PropertyId propertyId, BOOL value) override;

virtual BOOL SetAccessors(DynamicObject\* instance, PropertyId propertyId, Var getter, Var setter, PropertyOperationFlags flags = PropertyOperation\_None) override;

virtual BOOL PreventExtensions(DynamicObject \*instance) override;

virtual BOOL Seal(DynamicObject\* instance) override;

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

virtual BOOL SetAttributes(DynamicObject\* instance, PropertyId propertyId, PropertyAttributes attributes) override;

virtual BOOL GetAttributesWithPropertyIndex(DynamicObject \* instance, PropertyId propertyId, BigPropertyIndex index, PropertyAttributes \* attributes) override;

virtual void ResetTypeHandler(DynamicObject \* instance) override;

virtual void SetAllPropertiesToUndefined(DynamicObject\* instance, bool invalidateFixedFields) override;

virtual void MarshalAllPropertiesToScriptContext(DynamicObject\* instance, ScriptContext\* targetScriptContext, bool invalidateFixedFields) override;

virtual DynamicTypeHandler\* ConvertToTypeWithItemAttributes(DynamicObject\* instance) override;

virtual BOOL AllPropertiesAreEnumerable() sealed { return true; }

virtual BOOL IsPathTypeHandler() const { return TRUE; }

virtual void ShrinkSlotAndInlineSlotCapacity() override;

virtual void LockInlineSlotCapacity() override { Assert(false); };

virtual void EnsureInlineSlotCapacityIsLocked() override;

virtual void VerifyInlineSlotCapacityIsLocked() override;

virtual void EnsureInlineSlotCapacityIsLocked(bool startFromRoot) = 0;

virtual void VerifyInlineSlotCapacityIsLocked(bool startFromRoot) = 0;

SimplePathTypeHandler \*DeoptimizeObjectHeaderInlining(JavascriptLibrary \*const library);

virtual void SetPrototype(DynamicObject\* instance, RecyclableObject\* newPrototype) override;

virtual void SetIsPrototype(DynamicObject\* instance) override;

#if DBG

virtual bool SupportsPrototypeInstances() const { return !IsolatePrototypes(); }

#endif

virtual bool HasSingletonInstance() const override sealed;

virtual bool TryUseFixedProperty(PropertyRecord const \* propertyRecord, Var \* pProperty, FixedPropertyKind propertyType, ScriptContext \* requestContext) override;

virtual bool TryUseFixedAccessor(PropertyRecord const \* propertyRecord, Var \* pAccessor, FixedPropertyKind propertyType, bool getter, ScriptContext \* requestContext) override;

#if DBG

virtual bool CanStorePropertyValueDirectly(const DynamicObject\* instance, PropertyId propertyId, bool allowLetConst) override;

bool HasOnlyInitializedNonFixedProperties();

virtual bool CheckFixedProperty(PropertyRecord const \* propertyRecord, Var \* pProperty, ScriptContext \* requestContext) override;

virtual bool HasAnyFixedProperties() const override;

#endif

#ifdef ENABLE\_DEBUG\_CONFIG\_OPTIONS

virtual void DumpFixedFields() const override;

static void TraceFixedFieldsBeforeTypeHandlerChange(

const wchar\_t\* conversionName, const wchar\_t\* oldTypeHandlerName, const wchar\_t\* newTypeHandlerName,

DynamicObject\* instance, DynamicTypeHandler\* oldTypeHandler, DynamicType\* oldType, RecyclerWeakReference<DynamicObject>\* oldSingletonInstanceBefore);

static void TraceFixedFieldsAfterTypeHandlerChange(

DynamicObject\* instance, DynamicTypeHandler\* oldTypeHandler, DynamicTypeHandler\* newTypeHandler,

DynamicType\* oldType, DynamicType\* newType, RecyclerWeakReference<DynamicObject>\* oldSingletonInstanceBefore);

static void TraceFixedFieldsBeforeSetIsProto(

DynamicObject\* instance, DynamicTypeHandler\* oldTypeHandler, DynamicType\* oldType, RecyclerWeakReference<DynamicObject>\* oldSingletonInstanceBefore);

static void TraceFixedFieldsAfterSetIsProto(

DynamicObject\* instance, DynamicTypeHandler\* oldTypeHandler, DynamicTypeHandler\* newTypeHandler,

DynamicType\* oldType, DynamicType\* newType, RecyclerWeakReference<DynamicObject>\* oldSingletonInstanceBefore);

#endif

private:

static bool FixPropsOnPathTypes()

{

#ifdef SUPPORT\_FIXED\_FIELDS\_ON\_PATH\_TYPES

return CONFIG\_FLAG(FixPropsOnPathTypes);

#else

return false;

#endif

}

template <bool allowNonExistent, bool markAsUsed>

bool TryGetFixedProperty(PropertyRecord const \* propertyRecord, Var \* pProperty, Js::FixedPropertyKind propertyType, ScriptContext \* requestContext);

public:

virtual RecyclerWeakReference<DynamicObject>\* GetSingletonInstance() const override sealed { return HasSingletonInstance() ? this->typePath->GetSingletonInstance() : nullptr; }

virtual void SetSingletonInstanceUnchecked(RecyclerWeakReference<DynamicObject>\* instance) override

{

Assert(!GetIsShared());

this->typePath->SetSingletonInstance(instance, GetPathLength());

}

virtual void ClearSingletonInstance() { Assert(false); }

#if DBG

bool HasSingletonInstanceOnlyIfNeeded() const

{

#ifdef SUPPORT\_FIXED\_FIELDS\_ON\_PATH\_TYPES

return AreSingletonInstancesNeeded() || !this->typePath->HasSingletonInstance();

#else

return true;

#endif

}

#endif

private:

template <bool invalidateFixedFields> void DoShareTypeHandlerInternal(ScriptContext\* scriptContext);

void CopySingletonInstance(DynamicObject\* instance, DynamicTypeHandler\* typeHandler);

void InvalidateFixedFieldAt(Js::PropertyId propertyId, Js::PropertyIndex index, ScriptContext\* scriptContext);

void AddBlankFieldAt(Js::PropertyId propertyId, Js::PropertyIndex index, ScriptContext\* scriptContext);

bool ProcessFixedFieldChange(DynamicObject\* instance, PropertyId propertyId, PropertyIndex slotIndex, Var value, bool isNonFixed, const PropertyRecord \* propertyRecord = nullptr);

private:

template <typename T>

T\* ConvertToTypeHandler(DynamicObject\* instance);

DynamicType\* PromoteType(DynamicObject\* instance, const PropertyRecord\* propertyId, PropertyIndex\* propertyIndex);

DictionaryTypeHandler\* ConvertToDictionaryType(DynamicObject\* instance);

ES5ArrayTypeHandler\* ConvertToES5ArrayType(DynamicObject\* instance);

template <typename T> T\*

ConvertToSimpleDictionaryType(DynamicObject\* instance, int propertyCapacity, bool mayBecomeShared = false);

SimpleDictionaryTypeHandler\* ConvertToSimpleDictionaryType(DynamicObject\* instance, int propertyCapacity, bool mayBecomeShared = false)

{

return ConvertToSimpleDictionaryType<SimpleDictionaryTypeHandler>(instance, propertyCapacity, mayBecomeShared);

}

BOOL AddPropertyInternal(DynamicObject \* instance, PropertyId propertyId, Js::Var value, PropertyValueInfo\* info, PropertyOperationFlags flags, SideEffects possibleSideEffects);

BOOL AddProperty(DynamicObject\* instance, PropertyId propertyId, Var value, PropertyAttributes attributes, PropertyValueInfo\* info, PropertyOperationFlags flags, SideEffects possibleSideEffects);

BOOL SetPropertyInternal(DynamicObject\* instance, PropertyId propertyId, Var value, PropertyValueInfo\* info, PropertyOperationFlags flags, SideEffects possibleSideEffects);

virtual BOOL FreezeImpl(DynamicObject\* instance, bool isConvertedType) override;

// Checks whether conversion to shared type is needed and performs it, then calls actual operation on the shared type.

// Template method used for PreventExtensions, Seal, Freeze.

// FType is functor/lambda to perform actual forced operation (such as PreventExtensionsInternal) on the shared type.

template<typename FType>

BOOL ConvertToSharedNonExtensibleTypeIfNeededAndCallOperation(DynamicObject\* instance, const PropertyRecord\* operationInternalPropertyId, FType operation);

template <bool isObjectLiteral>

DynamicType\* PromoteType(DynamicType\* type, const PropertyRecord\* propertyId, bool shareType, ScriptContext\* scriptContext, DynamicObject\* object = nullptr, PropertyIndex\* propertyIndex = nullptr);

PropertyIndex GetPropertyIndex(PropertyId propertyId);

protected:

virtual bool GetSuccessor(const PropertyRecord\* propertyRecord, RecyclerWeakReference<DynamicType> \*\* typeWeakRef) = 0;

virtual void SetSuccessor(DynamicType \* type, const PropertyRecord\* propertyRecord, RecyclerWeakReference<DynamicType> \* typeWeakRef, ScriptContext \* scriptContext) = 0;

uint16 GetPathLength() const { return GetUnusedBytesValue(); }

TypePath \* GetTypePath() const { return typePath; }

DynamicType \* GetPredecessorType() const { return predecessorType; }

PathTypeHandlerBase\* GetRootPathTypeHandler();

public:

virtual void ShrinkSlotAndInlineSlotCapacity(uint16 newInlineSlotCapacity) = 0;

virtual bool GetMaxPathLength(uint16 \* maxPathLength) = 0;

};

typedef SimpleDictionaryTypeHandlerBase<PropertyIndex, const PropertyRecord\*, true> SimpleDictionaryTypeHandlerWithNontExtensibleSupport;

class SimplePathTypeHandler sealed : public PathTypeHandlerBase

{

private:

const PropertyRecord \* successorPropertyRecord;

RecyclerWeakReference<DynamicType> \* successorTypeWeakRef;

public:

DEFINE\_GETCPPNAME();

private:

SimplePathTypeHandler(TypePath\* typePath, uint16 pathLength, const PropertyIndex slotCapacity, uint16 inlineSlotCapacity, uint16 offsetOfInlineSlots, bool isLocked = false, bool isShared = false, DynamicType\* predecessorType = nullptr);

DEFINE\_VTABLE\_CTOR\_NO\_REGISTER(SimplePathTypeHandler, PathTypeHandlerBase, successorPropertyRecord(nullptr), successorTypeWeakRef(nullptr));

protected:

virtual bool GetSuccessor(const PropertyRecord\* propertyRecord, RecyclerWeakReference<DynamicType> \*\* typeWeakRef) override;

virtual void SetSuccessor(DynamicType \* type, const PropertyRecord\* propertyRecord, RecyclerWeakReference<DynamicType> \* typeWeakRef, ScriptContext \* scriptContext) override;

public:

virtual void ShrinkSlotAndInlineSlotCapacity(uint16 newInlineSlotCapacity) override;

virtual void LockInlineSlotCapacity() override;

virtual bool GetMaxPathLength(uint16 \* maxPathLength) override;

virtual void EnsureInlineSlotCapacityIsLocked(bool startFromRoot) override;

virtual void VerifyInlineSlotCapacityIsLocked(bool startFromRoot) override;

static SimplePathTypeHandler \* New(ScriptContext \* scriptContext, TypePath\* typePath, uint16 pathLength, uint16 inlineSlotCapacity, uint16 offsetOfInlineSlots, bool isLocked = false, bool isShared = false, DynamicType\* predecessorType = nullptr);

static SimplePathTypeHandler \* New(ScriptContext \* scriptContext, TypePath\* typePath, uint16 pathLength, const PropertyIndex slotCapacity, uint16 inlineSlotCapacity, uint16 offsetOfInlineSlots, bool isLocked = false, bool isShared = false, DynamicType\* predecessorType = nullptr);

static SimplePathTypeHandler \* SimplePathTypeHandler::New(ScriptContext \* scriptContext, SimplePathTypeHandler \* typeHandler, bool isLocked, bool isShared);

};

class PathTypeHandler sealed : public PathTypeHandlerBase

{

friend class SimplePathTypeHandler;

private:

typedef JsUtil::WeakReferenceDictionary<PropertyId, DynamicType, DictionarySizePolicy<PowerOf2Policy, 1>> PropertySuccessorsMap;

PropertySuccessorsMap \* propertySuccessors;

public:

DEFINE\_GETCPPNAME();

private:

PathTypeHandler(TypePath\* typePath, uint16 pathLength, const PropertyIndex slotCapacity, uint16 inlineSlotCapacity, uint16 offsetOfInlineSlots, bool isLocked = false, bool isShared = false, DynamicType\* predecessorType = nullptr);

DEFINE\_VTABLE\_CTOR\_NO\_REGISTER(PathTypeHandler, PathTypeHandlerBase, propertySuccessors(nullptr));

protected:

virtual bool GetSuccessor(const PropertyRecord\* propertyRecord, RecyclerWeakReference<DynamicType> \*\* typeWeakRef) override;

virtual void SetSuccessor(DynamicType \* type, const PropertyRecord\* propertyRecord, RecyclerWeakReference<DynamicType> \* typeWeakRef, ScriptContext \* scriptContext) override;

public:

virtual void ShrinkSlotAndInlineSlotCapacity(uint16 newInlineSlotCapacity) override;

virtual void LockInlineSlotCapacity() override;

virtual bool GetMaxPathLength(uint16 \* maxPathLength) override;

virtual void EnsureInlineSlotCapacityIsLocked(bool startFromRoot) override;

virtual void VerifyInlineSlotCapacityIsLocked(bool startFromRoot) override;

static PathTypeHandler \* New(ScriptContext \* scriptContext, TypePath\* typePath, uint16 pathLength, uint16 inlineSlotCapacity, uint16 offsetOfInlineSlots, bool isLocked = false, bool isShared = false, DynamicType\* predecessorType = nullptr);

static PathTypeHandler \* New(ScriptContext \* scriptContext, TypePath\* typePath, uint16 pathLength, const PropertyIndex slotCapacity, uint16 inlineSlotCapacity, uint16 offsetOfInlineSlots, bool isLocked = false, bool isShared = false, DynamicType\* predecessorType = nullptr);

static PathTypeHandler \* New(ScriptContext \* scriptContext, PathTypeHandler \* typeHandler, bool isLocked, bool isShared);

};

}

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

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

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

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

#include "RuntimeTypePch.h"

namespace Js

{

PropertyDescriptor::PropertyDescriptor() :

writableSpecified(false),

enumerableSpecified(false),

configurableSpecified(false),

valueSpecified(false),

getterSpecified(false),

setterSpecified(false),

Writable(false),

Enumerable(false),

Configurable(false),

Value(nullptr),

Getter(nullptr),

Setter(nullptr),

originalVar(nullptr),

fromProxy(false)

{

}

void PropertyDescriptor::SetEnumerable(bool value)

{

Enumerable = value;

enumerableSpecified = true;

}

void PropertyDescriptor::SetWritable(bool value)

{

Writable = value;

writableSpecified = true;

}

void PropertyDescriptor::SetConfigurable(bool value)

{

Configurable = value;

configurableSpecified = true;

}

void PropertyDescriptor::SetValue(Var value)

{

this->Value = value;

this->valueSpecified = true;

}

void PropertyDescriptor::SetGetter(Var getter)

{

this->Getter = getter;

this->getterSpecified = true;

}

void PropertyDescriptor::SetSetter(Var setter)

{

this->Setter = setter;

this->setterSpecified = true;

}

PropertyAttributes PropertyDescriptor::GetAttributes() const

{

PropertyAttributes attributes = PropertyNone;

if (this->configurableSpecified && this->Configurable)

{

attributes |= PropertyConfigurable;

}

if (this->enumerableSpecified && this->Enumerable)

{

attributes |= PropertyEnumerable;

}

if (this->writableSpecified && this->Writable)

{

attributes |= PropertyWritable;

}

return attributes;

}

void PropertyDescriptor::SetAttributes(PropertyAttributes attributes, PropertyAttributes mask)

{

if (mask & PropertyConfigurable)

{

this->SetConfigurable(PropertyNone != (attributes & PropertyConfigurable));

}

if (mask & PropertyEnumerable)

{

this->SetEnumerable(PropertyNone != (attributes & PropertyEnumerable));

}

if (mask & PropertyWritable)

{

this->SetWritable(PropertyNone != (attributes & PropertyWritable));

}

}

void PropertyDescriptor::MergeFrom(const PropertyDescriptor& descriptor)

{

if (descriptor.configurableSpecified)

{

this->SetConfigurable(descriptor.Configurable);

}

if (descriptor.enumerableSpecified)

{

this->SetEnumerable(descriptor.Enumerable);

}

if (descriptor.writableSpecified)

{

this->SetWritable(descriptor.Writable);

}

if (descriptor.valueSpecified)

{

this->SetValue(descriptor.Value);

}

if (descriptor.getterSpecified)

{

this->SetGetter(descriptor.Getter);

}

if (descriptor.setterSpecified)

{

this->SetSetter(descriptor.Setter);

}

}

}

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

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

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

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

#pragma once

namespace Js

{

struct PropertyDescriptor

{

public:

PropertyDescriptor();

private:

Var Value;

Var Getter;

Var Setter;

Var originalVar;

bool writableSpecified;

bool enumerableSpecified;

bool configurableSpecified;

bool valueSpecified;

bool getterSpecified;

bool setterSpecified;

bool Writable;

bool Enumerable;

bool Configurable;

bool fromProxy;

public:

bool IsDataDescriptor() const { return writableSpecified | valueSpecified;}

bool IsAccessorDescriptor() const { return getterSpecified | setterSpecified;}

bool IsGenericDescriptor() const { return !IsAccessorDescriptor() && !IsDataDescriptor(); }

void SetEnumerable(bool value);

void SetWritable(bool value);

void SetConfigurable(bool value);

void SetValue(Var value);

Var GetValue() const { return Value; }

void SetGetter(Var getter);

Var GetGetter() const { Assert(getterSpecified || Getter == nullptr); return Getter; }

void SetSetter(Var setter);

Var GetSetter() const { Assert(setterSpecified || Setter == nullptr); return Setter; }

PropertyAttributes GetAttributes() const;

bool IsFromProxy() const { return fromProxy; }

void SetFromProxy(bool value) { fromProxy = value; }

void SetOriginal(Var orginal) { originalVar = orginal; }

Var GetOriginal() const { return originalVar; }

bool ValueSpecified() const { return valueSpecified; }

bool WritableSpecified() const { return writableSpecified; };

bool ConfigurableSpecified() const { return configurableSpecified; }

bool EnumerableSpecified() const { return enumerableSpecified; }

bool GetterSpecified() const { return getterSpecified; }

bool SetterSpecified() const { return setterSpecified; }

bool IsWritable() const { Assert(writableSpecified); return Writable; }

bool IsEnumerable() const { Assert(enumerableSpecified); return Enumerable; }

bool IsConfigurable() const { Assert(configurableSpecified); return Configurable; }

// Set configurable/enumerable/writable.

// attributes: attribute values.

// mask: specified which attributes to set. If an attribute is not in the mask.

void SetAttributes(PropertyAttributes attributes, PropertyAttributes mask = ~PropertyNone);

// Merge from descriptor parameter into this but only fields specified by descriptor parameter.

void MergeFrom(const PropertyDescriptor& descriptor);

};

}

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

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

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

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

#pragma once

namespace Js

{

template <typename Ranges>

struct PropertyIndexRangesBase

{

static void VerifySlotCapacity(int requestedCapacity);

};

template <typename TPropertyIndex>

struct PropertyIndexRanges

{

};

template <>

struct PropertyIndexRanges<PropertyIndex> : public PropertyIndexRangesBase<PropertyIndexRanges<PropertyIndex>>

{

static const PropertyIndex MaxValue = Constants::PropertyIndexMax;

static const PropertyIndex NoSlots = Constants::NoSlot;

};

template <>

struct PropertyIndexRanges<BigPropertyIndex> : public PropertyIndexRangesBase<PropertyIndexRanges<BigPropertyIndex>>

{

static const BigPropertyIndex MaxValue = 0x3FFFFFFF;

static const BigPropertyIndex NoSlots = Constants::NoBigSlot;

};

};

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

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

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

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

#include "RuntimeTypePch.h"

#include "Library\JavascriptSymbol.h"

#include "Library\JavascriptSymbolObject.h"

DEFINE\_VALIDATE\_HAS\_VTABLE\_CTOR(Js::RecyclableObject);

namespace Js

{

void PropertyValueInfo::SetCacheInfo(PropertyValueInfo\* info, InlineCache \*const inlineCache)

{

Assert(info);

Assert(inlineCache);

info->functionBody = nullptr;

info->inlineCache = inlineCache;

info->polymorphicInlineCache = nullptr;

info->inlineCacheIndex = Js::Constants::NoInlineCacheIndex;

info->allowResizingPolymorphicInlineCache = false;

}

void PropertyValueInfo::SetCacheInfo(

PropertyValueInfo\* info,

FunctionBody \*const functionBody,

InlineCache \*const inlineCache,

const InlineCacheIndex inlineCacheIndex,

const bool allowResizingPolymorphicInlineCache)

{

Assert(info);

Assert(functionBody);

Assert(inlineCache);

Assert(inlineCacheIndex < functionBody->GetInlineCacheCount());

info->functionBody = functionBody;

info->inlineCache = inlineCache;

info->polymorphicInlineCache = nullptr;

info->inlineCacheIndex = inlineCacheIndex;

info->allowResizingPolymorphicInlineCache = allowResizingPolymorphicInlineCache;

}

void PropertyValueInfo::SetCacheInfo(

PropertyValueInfo\* info,

FunctionBody \*const functionBody,

PolymorphicInlineCache \*const polymorphicInlineCache,

const InlineCacheIndex inlineCacheIndex,

const bool allowResizingPolymorphicInlineCache)

{

Assert(info);

Assert(functionBody);

Assert(polymorphicInlineCache);

Assert(inlineCacheIndex < functionBody->GetInlineCacheCount());

info->functionBody = functionBody;

info->inlineCache = nullptr;

info->polymorphicInlineCache = polymorphicInlineCache;

info->inlineCacheIndex = inlineCacheIndex;

info->allowResizingPolymorphicInlineCache = allowResizingPolymorphicInlineCache;

}

void PropertyValueInfo::ClearCacheInfo(PropertyValueInfo\* info)

{

if (info != NULL)

{

info->functionBody = nullptr;

info->inlineCache = nullptr;

info->polymorphicInlineCache = nullptr;

info->inlineCacheIndex = Constants::NoInlineCacheIndex;

info->allowResizingPolymorphicInlineCache = true;

}

}

#if DBG || defined(PROFILE\_TYPES)

// Used only by the GlobalObject, because it's typeHandler can't be fully initialized

// with the globalobject which is currently being created.

RecyclableObject::RecyclableObject(DynamicType \* type, ScriptContext \* scriptContext) : type(type)

{

#if DBG\_EXTRAFIELD

dtorCalled = false;

#ifdef HEAP\_ENUMERATION\_VALIDATION

m\_heapEnumValidationCookie = 0;

#endif

#endif

Assert(type->GetTypeId() == TypeIds\_GlobalObject);

RecordAllocation(scriptContext);

}

void RecyclableObject::RecordAllocation(ScriptContext \* scriptContext)

{

#ifdef PROFILE\_TYPES

TypeId typeId = this->GetType()->GetTypeId();

if (typeId < sizeof(scriptContext->instanceCount)/sizeof(int))

{

scriptContext->instanceCount[typeId]++;

}

#endif

}

#endif

RecyclableObject::RecyclableObject(Type \* type) : type(type)

{

#if DBG\_EXTRAFIELD

dtorCalled = false;

#ifdef HEAP\_ENUMERATION\_VALIDATION

m\_heapEnumValidationCookie = 0;

#endif

#endif

#if DBG || defined(PROFILE\_TYPES)

RecordAllocation(type->GetScriptContext());

#endif

}

RecyclableObject\* RecyclableObject::GetPrototype() const

{

Type\* type = GetType();

if (!type->HasSpecialPrototype())

{

return type->GetPrototype();

}

return const\_cast<RecyclableObject\*>(this)->GetPrototypeSpecial();

}

RecyclableObject\* RecyclableObject::GetPrototypeSpecial()

{

AssertMsg(GetType()->GetTypeId() == TypeIds\_Null, "Do not use this function.");

return nullptr;

}

JavascriptMethod RecyclableObject::GetEntryPoint() const

{

return this->GetType()->GetEntryPoint();

}

Recycler\* RecyclableObject::GetRecycler() const

{

return this->GetLibrary()->GetRecycler();

}

void RecyclableObject::SetIsPrototype()

{

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

{

DynamicObject\* dynamicThis = DynamicObject::FromVar(this);

dynamicThis->SetIsPrototype(); // Call the DynamicObject::SetIsPrototype

}

}

bool RecyclableObject::HasOnlyWritableDataProperties()

{

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

{

DynamicObject\* obj = DynamicObject::FromVar(this);

return obj->GetTypeHandler()->GetHasOnlyWritableDataProperties() &&

(!obj->HasObjectArray() || obj->GetObjectArrayOrFlagsAsArray()->HasOnlyWritableDataProperties());

}

return true;

}

void RecyclableObject::ClearWritableDataOnlyDetectionBit()

{

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

{

DynamicObject\* obj = DynamicObject::FromVar(this);

obj->GetTypeHandler()->ClearWritableDataOnlyDetectionBit();

if (obj->HasObjectArray())

{

obj->GetObjectArrayOrFlagsAsArray()->ClearWritableDataOnlyDetectionBit();

}

}

}

bool RecyclableObject::IsWritableDataOnlyDetectionBitSet()

{

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

{

DynamicObject\* obj = DynamicObject::FromVar(this);

return obj->GetTypeHandler()->IsWritableDataOnlyDetectionBitSet() ||

(obj->HasObjectArray() && obj->GetObjectArrayOrFlagsAsArray()->IsWritableDataOnlyDetectionBitSet());

}

return false;

}

RecyclableObject\* RecyclableObject::GetProxiedObjectForHeapEnum()

{

Assert(this->GetScriptContext()->IsHeapEnumInProgress());

return NULL;

}

BOOL RecyclableObject::IsExternal() const

{

Assert(this->IsExternalVirtual() == this->GetType()->IsExternal());

return this->GetType()->IsExternal();

}

BOOL RecyclableObject::SkipsPrototype() const

{

Assert(this->DbgSkipsPrototype() == this->GetType()->SkipsPrototype());

return this->GetType()->SkipsPrototype();

}

uint32

RecyclableObject::GetOffsetOfType()

{

return offsetof(RecyclableObject, type);

}

RecyclableObject \* RecyclableObject::CloneToScriptContext(ScriptContext\* requestContext)

{

switch (JavascriptOperators::GetTypeId(this))

{

case TypeIds\_Undefined:

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

case TypeIds\_Null:

return requestContext->GetLibrary()->GetNull();

case TypeIds\_Number:

return RecyclableObject::FromVar(JavascriptNumber::CloneToScriptContext(this, requestContext));

default:

AssertMsg(FALSE, "shouldn't clone for other types");

Js::JavascriptError::ThrowError(requestContext, VBSERR\_InternalError);

}

}

#if defined(PROFILE\_RECYCLER\_ALLOC) && defined(RECYCLER\_DUMP\_OBJECT\_GRAPH)

bool RecyclableObject::DumpObjectFunction(type\_info const \* typeinfo, bool isArray, void \* objectAddress)

{

if (isArray)

{

// Don't deal with array

return false;

}

Output::Print(L"%S{%x} %p", typeinfo->name(), ((RecyclableObject \*)objectAddress)->GetTypeId(), objectAddress);

return true;

}

#endif

BOOL RecyclableObject::SetPropertyWithAttributes(PropertyId propertyId, Var value, PropertyAttributes attributes, PropertyValueInfo\* info, PropertyOperationFlags flags, SideEffects possibleSideEffects)

{

// TODO: It appears as though this is never called. Some types (such as JavascriptNumber) don't override this, but they

// also don't expect properties to be set on them. Need to review this and see if we can make this pure virtual or

// Assert(false) here. In any case, this should be SetProperty, not InitProperty.

Assert(false);

bool isForce = (flags & PropertyOperation\_Force) != 0;

bool throwIfNotExtensible = (flags & PropertyOperation\_ThrowIfNotExtensible) != 0;

if (!isForce)

{

// throwIfNotExtensible is only relevant to DynamicObjects

Assert(!throwIfNotExtensible);

}

return

this->InitProperty(propertyId, value, flags) &&

this->SetAttributes(propertyId, attributes);

}

void RecyclableObject::ThrowIfCannotDefineProperty(PropertyId propId, PropertyDescriptor descriptor)

{

// Do nothing

}

BOOL RecyclableObject::GetDefaultPropertyDescriptor(PropertyDescriptor& descriptor)

{

// By default, when GetOwnPropertyDescriptor is called for a nonexistent property,

// return undefined.

return false;

}

HRESULT RecyclableObject::QueryObjectInterface(REFIID riid, void \*\*ppvObj)

{

Assert(!this->GetScriptContext()->GetThreadContext()->IsScriptActive());

return E\_NOINTERFACE;

}

RecyclableObject\* RecyclableObject::GetThisObjectOrUnWrap()

{

if (WithScopeObject::Is(this))

{

return WithScopeObject::FromVar(this)->GetWrappedObject();

}

return this;

}

// In order to avoid a branch, every object has an entry point if it gets called like a

// function - however, if it can't be called like a function, it's set to DefaultEntryPoint

// which will emit an error.

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

{

ARGUMENTS(args, callInfo);

TypeId typeId = function->GetTypeId();

rtErrors err = typeId == TypeIds\_Undefined || typeId == TypeIds\_Null ? JSERR\_NeedObject : JSERR\_NeedFunction;

JavascriptError::ThrowTypeError(function->GetScriptContext(), err

/\* TODO-ERROR: args.Info.Count > 0? args[0] : nullptr); \*/);

}

Var RecyclableObject::DefaultExternalEntryPoint(RecyclableObject\* function, CallInfo callInfo, Var\* arguments)

{

TypeId typeId = function->GetTypeId();

rtErrors err = typeId == TypeIds\_Undefined || typeId == TypeIds\_Null ? JSERR\_NeedObject : JSERR\_NeedFunction;

JavascriptError::ThrowTypeError(function->GetScriptContext(), err

/\* TODO-ERROR: args.Info.Count > 0? args[0] : nullptr); \*/);

}

BOOL RecyclableObject::HasProperty(PropertyId propertyId)

{

return false;

}

BOOL RecyclableObject::HasOwnProperty(PropertyId propertyId)

{

return false;

}

BOOL RecyclableObject::HasOwnPropertyNoHostObject(PropertyId propertyId)

{

return HasOwnProperty(propertyId);

}

BOOL RecyclableObject::GetProperty(Var originalInstance, PropertyId propertyId, Var\* value, PropertyValueInfo\* info, ScriptContext\* requestContext)

{

return false;

}

BOOL RecyclableObject::GetProperty(Var originalInstance, JavascriptString\* propertyNameString, Var\* value, PropertyValueInfo\* info, ScriptContext\* requestContext)

{

return false;

}

BOOL RecyclableObject::GetInternalProperty(Var originalInstance, PropertyId propertyId, Var\* value, PropertyValueInfo\* info, ScriptContext\* requestContext)

{

return false;

}

BOOL RecyclableObject::GetPropertyReference(Var originalInstance, PropertyId propertyId, Var\* value, PropertyValueInfo\* info, ScriptContext\* requestContext)

{

return false;

}

BOOL RecyclableObject::SetProperty(PropertyId propertyId, Var value, PropertyOperationFlags flags, PropertyValueInfo\* info)

{

return false;

}

BOOL RecyclableObject::SetProperty(JavascriptString\* propertyNameString, Var value, PropertyOperationFlags flags, PropertyValueInfo\* info)

{

return false;

}

BOOL RecyclableObject::SetInternalProperty(PropertyId propertyId, Var value, PropertyOperationFlags flags, PropertyValueInfo\* info)

{

return false;

}

BOOL RecyclableObject::InitProperty(PropertyId propertyId, Var value, PropertyOperationFlags flags, PropertyValueInfo\* info)

{

return false;

}

BOOL RecyclableObject::InitPropertyScoped(PropertyId propertyId, Var value)

{

return false;

}

BOOL RecyclableObject::InitFuncScoped(PropertyId propertyId, Var value)

{

return false;

}

BOOL RecyclableObject::EnsureProperty(PropertyId propertyId)

{

return false;

}

BOOL RecyclableObject::EnsureNoRedeclProperty(PropertyId propertyId)

{

return false;

}

BOOL RecyclableObject::DeleteProperty(PropertyId propertyId, PropertyOperationFlags flags)

{

return true;

}

BOOL RecyclableObject::IsFixedProperty(PropertyId propertyId)

{

return false;

}

BOOL RecyclableObject::HasItem(uint32 index)

{

return false;

}

BOOL RecyclableObject::HasOwnItem(uint32 index)

{

return false;

}

BOOL RecyclableObject::GetItem(Var originalInstance, uint32 index, Var\* value, ScriptContext \* requestContext)

{

return false;

}

BOOL RecyclableObject::GetItemReference(Var originalInstance, uint32 index, Var\* value, ScriptContext \* requestContext)

{

return false;

}

BOOL RecyclableObject::SetItem(uint32 index, Var value, PropertyOperationFlags flags)

{

return false;

}

BOOL RecyclableObject::DeleteItem(uint32 index, PropertyOperationFlags flags)

{

return true;

}

BOOL RecyclableObject::GetEnumerator(BOOL enumNonEnumerable, Var\* enumerator, ScriptContext \* requestContext, bool preferSnapshotSemantics, bool enumSymbols)

{

return false;

}

BOOL RecyclableObject::ToPrimitive(JavascriptHint hint, Var\* value, ScriptContext \* scriptContext)

{

\*value = NULL;

return false;

}

BOOL RecyclableObject::SetAccessors(PropertyId propertyId, Var getter, Var setter, PropertyOperationFlags flags)

{

return false;

}

BOOL RecyclableObject::GetAccessors(PropertyId propertyId, Var\* getter, Var\* setter, ScriptContext \* requestContext)

{

return false;

}

BOOL RecyclableObject::StrictEquals(Var aRight, BOOL\* value, ScriptContext \* requestContext)

{

//StrictEquals is handled in JavascriptOperators::StrictEqual

Throw::InternalError();

}

#pragma fenv\_access (on)

BOOL RecyclableObject::Equals(Var aRight, BOOL\* value, ScriptContext \* requestContext)

{

Var aLeft = this;

if (aLeft == aRight)

{

//In ES5 mode strict equals (===) on same instance of object type VariantDate succeeds.

//Hence equals needs to succeed.

goto ReturnTrue;

}

double dblLeft, dblRight;

TypeId leftType = this->GetTypeId();

TypeId rightType = JavascriptOperators::GetTypeId(aRight);

int redoCount = 0;

Redo:

if (redoCount == 2)

{

goto ReturnFalse;

}

switch (leftType)

{

case TypeIds\_Undefined:

case TypeIds\_Null:

switch (rightType)

{

case TypeIds\_Integer:

case TypeIds\_Number:

case TypeIds\_Symbol:

goto ReturnFalse;

case TypeIds\_Undefined:

case TypeIds\_Null:

goto ReturnTrue;

default:

// Falsy objects are == null and == undefined.

\*value = RecyclableObject::FromVar(aRight)->GetType()->IsFalsy();

return TRUE;

}

case TypeIds\_Integer:

switch (rightType)

{

case TypeIds\_Undefined:

case TypeIds\_Null:

case TypeIds\_Symbol:

goto ReturnFalse;

case TypeIds\_Integer:

// We already did a check to see if aLeft == aRight above, but we need to check again in case there was a redo.

\*value = aLeft == aRight;

return TRUE;

case TypeIds\_Int64Number:

{

int leftValue = TaggedInt::ToInt32(aLeft);

\_\_int64 rightValue = JavascriptInt64Number::FromVar(aRight)->GetValue();

\*value = leftValue == rightValue;

Assert(!(\*value)); // currently it cannot be true. more for future extension if we allow arithmetic calculation

return TRUE;

}

case TypeIds\_UInt64Number:

{

\_\_int64 leftValue = TaggedInt::ToInt32(aLeft);

unsigned \_\_int64 rightValue = JavascriptInt64Number::FromVar(aRight)->GetValue();

// TODO: yongqu to review whether we need to check for neg value

\*value = (/\*leftValue >= 0 && \*/(unsigned \_\_int64)leftValue == rightValue);

Assert(!(\*value)); // currently it cannot be true. more for future extension if we allow arithmetic calculation

return TRUE;

}

case TypeIds\_Number:

dblLeft = TaggedInt::ToDouble(aLeft);

dblRight = JavascriptNumber::GetValue(aRight);

goto CompareDoubles;

case TypeIds\_Boolean:

case TypeIds\_String:

dblLeft = TaggedInt::ToDouble(aLeft);

dblRight = JavascriptConversion::ToNumber(aRight, requestContext);

goto CompareDoubles;

default:

goto RedoRight;

}

break;

case TypeIds\_Int64Number:

switch (rightType)

{

case TypeIds\_Integer:

{

\_\_int64 leftValue = JavascriptInt64Number::FromVar(aLeft)->GetValue();

int rightValue = TaggedInt::ToInt32(aRight);

\*value = leftValue == rightValue;

Assert(!(\*value)); // currently it cannot be true. more for future extension if we allow arithmetic calculation

return TRUE;

}

case TypeIds\_Number:

dblLeft = (double)JavascriptInt64Number::FromVar(aLeft)->GetValue();

dblRight = JavascriptNumber::GetValue(aRight);

goto CompareDoubles;

case TypeIds\_Int64Number:

{

\_\_int64 leftValue = JavascriptInt64Number::FromVar(aLeft)->GetValue();

\_\_int64 rightValue = JavascriptInt64Number::FromVar(aRight)->GetValue();

\*value = leftValue == rightValue;

return TRUE;

}

case TypeIds\_UInt64Number:

{

\_\_int64 leftValue = JavascriptInt64Number::FromVar(aLeft)->GetValue();

unsigned \_\_int64 rightValue = JavascriptInt64Number::FromVar(aRight)->GetValue();

// TODO: yongqu to review whether we need to check for neg value

\*value = (/\* leftValue >= 0 && \*/(unsigned \_\_int64)leftValue == rightValue);

return TRUE;

}

}

break;

case TypeIds\_UInt64Number:

switch (rightType)

{

case TypeIds\_Integer:

{

unsigned \_\_int64 leftValue = JavascriptUInt64Number::FromVar(aLeft)->GetValue();

\_\_int64 rightValue = TaggedInt::ToInt32(aRight);

// TODO: yongqu to review whether we need to check for neg value

\*value = rightValue >= 0 && leftValue == (unsigned \_\_int64)rightValue;

Assert(!(\*value)); // currently it cannot be true. more for future extension if we allow arithmetic calculation

return TRUE;

}

case TypeIds\_Number:

dblLeft = (double)JavascriptUInt64Number::FromVar(aLeft)->GetValue();

dblRight = JavascriptNumber::GetValue(aRight);

goto CompareDoubles;

case TypeIds\_Int64Number:

{

unsigned \_\_int64 leftValue = JavascriptUInt64Number::FromVar(aLeft)->GetValue();

\_\_int64 rightValue = JavascriptInt64Number::FromVar(aRight)->GetValue();

// TODO: yongqu to review whether we need to check for neg value

\*value = (/\* rightValue >= 0 && \*/leftValue == (unsigned \_\_int64)rightValue);

return TRUE;

}

case TypeIds\_UInt64Number:

{

unsigned \_\_int64 leftValue = JavascriptUInt64Number::FromVar(aLeft)->GetValue();

unsigned \_\_int64 rightValue = JavascriptInt64Number::FromVar(aRight)->GetValue();

\*value = leftValue == rightValue;

return TRUE;

}

}

break;

case TypeIds\_Number:

switch (rightType)

{

case TypeIds\_Undefined:

case TypeIds\_Null:

case TypeIds\_Symbol:

goto ReturnFalse;

case TypeIds\_Integer:

dblLeft = JavascriptNumber::GetValue(aLeft);

dblRight = TaggedInt::ToDouble(aRight);

goto CompareDoubles;

case TypeIds\_Number:

dblLeft = JavascriptNumber::GetValue(aLeft);

dblRight = JavascriptNumber::GetValue(aRight);

goto CompareDoubles;

case TypeIds\_Boolean:

case TypeIds\_String:

dblLeft = JavascriptNumber::GetValue(aLeft);

dblRight = JavascriptConversion::ToNumber(aRight, requestContext);

goto CompareDoubles;

default:

goto RedoRight;

}

break;

case TypeIds\_String:

switch (rightType)

{

case TypeIds\_Undefined:

case TypeIds\_Null:

case TypeIds\_Symbol:

goto ReturnFalse;

case TypeIds\_String:

goto CompareStrings;

case TypeIds\_Number:

case TypeIds\_Integer:

case TypeIds\_Boolean:

dblLeft = JavascriptConversion::ToNumber(aLeft, requestContext);

dblRight = JavascriptConversion::ToNumber(aRight, requestContext);

goto CompareDoubles;

default:

goto RedoRight;

}

case TypeIds\_Boolean:

switch (rightType)

{

case TypeIds\_Undefined:

case TypeIds\_Null:

case TypeIds\_Symbol:

goto ReturnFalse;

case TypeIds\_Boolean:

\*value = JavascriptBoolean::FromVar(aLeft)->GetValue() == JavascriptBoolean::FromVar(aRight)->GetValue();

return TRUE;

case TypeIds\_Number:

case TypeIds\_Integer:

case TypeIds\_String:

dblLeft = JavascriptConversion::ToNumber(aLeft, requestContext);

dblRight = JavascriptConversion::ToNumber(aRight, requestContext);

goto CompareDoubles;

default:

goto RedoRight;

}

break;

case TypeIds\_Symbol:

switch (rightType)

{

case TypeIds\_Undefined:

case TypeIds\_Null:

case TypeIds\_Number:

case TypeIds\_Integer:

case TypeIds\_String:

case TypeIds\_Boolean:

goto ReturnFalse;

case TypeIds\_Symbol:

\*value = JavascriptSymbol::FromVar(aLeft)->GetValue() == JavascriptSymbol::FromVar(aRight)->GetValue();

return TRUE;

case TypeIds\_SymbolObject:

\*value = JavascriptSymbol::FromVar(aLeft)->GetValue() == JavascriptSymbolObject::FromVar(aRight)->GetValue();

return TRUE;

default:

goto RedoRight;

}

break;

case TypeIds\_Function:

if (rightType == TypeIds\_Function)

{

// In ES5 in certain cases (ES5 10.6.14(strict), 13.2.19(strict), 15.3.4.5.20-21) we return a function that throws type error.

// For different scenarios we return different instances of the function, which differ by exception/error message.

// According to ES5, this is the same [[ThrowTypeError]] (thrower) internal function, thus they should be equal.

if (JavascriptFunction::FromVar(aLeft)->IsThrowTypeErrorFunction() &&

JavascriptFunction::FromVar(aRight)->IsThrowTypeErrorFunction())

{

goto ReturnTrue;

}

goto ReturnFalse;

}

// Fall through to do normal object comparison on function object.

default:

switch (rightType)

{

case TypeIds\_Undefined:

case TypeIds\_Null:

// Falsy objects are == null and == undefined.

\*value = this->type->IsFalsy();

return TRUE;

case TypeIds\_Boolean:

case TypeIds\_Integer:

case TypeIds\_Number:

case TypeIds\_String:

case TypeIds\_Symbol:

goto RedoLeft;

default:

goto ReturnFalse;

}

}

RedoLeft:

aLeft = JavascriptConversion::ToPrimitive(aLeft, JavascriptHint::None, requestContext);

leftType = JavascriptOperators::GetTypeId(aLeft);

redoCount++;

goto Redo;

RedoRight:

aRight = JavascriptConversion::ToPrimitive(aRight, JavascriptHint::None, requestContext);

rightType = JavascriptOperators::GetTypeId(aRight);

redoCount++;

goto Redo;

CompareStrings:

\*value = JavascriptString::Equals(aLeft, aRight);

return TRUE;

CompareDoubles:

\*value = dblLeft == dblRight;

return TRUE;

ReturnFalse:

\*value = FALSE;

return TRUE;

ReturnTrue:

\*value = TRUE;

return TRUE;

}

RecyclableObject\* RecyclableObject::ToObject(ScriptContext \* requestContext)

{

AssertMsg(JavascriptOperators::IsObject(this), "bad type object in conversion ToObject");

Assert(!CrossSite::NeedMarshalVar(this, requestContext));

return this;

}

Var RecyclableObject::GetTypeOfString(ScriptContext \* requestContext)

{

return requestContext->GetLibrary()->GetUnknownDisplayString();

}

Var RecyclableObject::InvokePut(Arguments args)

{

// Handle x(y) = z.

// Native jscript object behavior: throw an error in all such cases.

JavascriptError::ThrowReferenceError(GetScriptContext(), JSERR\_CantAsgCall);

}

BOOL RecyclableObject::GetRemoteTypeId(TypeId \* typeId)

{

return FALSE;

}

DynamicObject\* RecyclableObject::GetRemoteObject()

{

return NULL;

}

Var RecyclableObject::GetHostDispatchVar()

{

Assert(FALSE);

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

}

JavascriptString\* RecyclableObject::GetClassName(ScriptContext \* requestContext)

{

// we don't need this when not handling fastDOM.

Assert(0);

return NULL;

}

BOOL RecyclableObject::HasInstance(Var instance, ScriptContext\* scriptContext, IsInstInlineCache\* inlineCache)

{

JavascriptError::ThrowTypeError(scriptContext, JSERR\_Operand\_Invalid\_NeedFunction, L"instanceof" /\* TODO-ERROR: get arg name - aClass \*/);

}

} // namespace Js

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

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

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

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

#pragma once

namespace Js {

// Cache property index and IsWritable info for UpdatePatch

class PropertyValueInfo

{

enum CacheInfoFlag

{

preventFalseReferenceFlag = 0x1, // avoid false positive for GC

disablePrototypeCacheFlag = 0x2,

enableStoreFieldCacheFlag = 0x4,

defaultInfoFlags = preventFalseReferenceFlag | enableStoreFieldCacheFlag

};

private:

RecyclableObject\* m\_instance; // Slot owner instance

PropertyIndex m\_propertyIndex; // Slot index on m\_instance for the property, or NoSlot to indicate the object can't cache

PropertyAttributes m\_attributes; // Attributes of the property -- only Writable is used

InlineCacheFlags flags;

CacheInfoFlag cacheInfoFlag;

InlineCache\* inlineCache;

PolymorphicInlineCache\* polymorphicInlineCache;

FunctionBody \* functionBody;

uint inlineCacheIndex;

bool allowResizingPolymorphicInlineCache;

void Set(RecyclableObject\* instance, PropertyIndex propertyIndex, PropertyAttributes attributes, InlineCacheFlags flags)

{

m\_instance = instance;

m\_propertyIndex = propertyIndex;

m\_attributes = attributes;

this->flags = flags;

}

void SetInfoFlag(CacheInfoFlag newFlag) { cacheInfoFlag = (CacheInfoFlag)(cacheInfoFlag | newFlag); }

void ClearInfoFlag(CacheInfoFlag newFlag) { cacheInfoFlag = (CacheInfoFlag)(cacheInfoFlag & ~newFlag); }

BOOL IsInfoFlagSet(CacheInfoFlag checkFlag) const { return (cacheInfoFlag & checkFlag) == checkFlag; }

public:

PropertyValueInfo()

: m\_instance(NULL), m\_propertyIndex(Constants::NoSlot), m\_attributes(PropertyNone), flags(InlineCacheNoFlags),

cacheInfoFlag(CacheInfoFlag::defaultInfoFlags), inlineCache(NULL), polymorphicInlineCache(NULL), functionBody(NULL),

inlineCacheIndex(Constants::NoInlineCacheIndex),

allowResizingPolymorphicInlineCache(true)

{

}

RecyclableObject\* GetInstance() const { return m\_instance; }

PropertyIndex GetPropertyIndex() const { return m\_propertyIndex; }

bool IsWritable() const { return (m\_attributes & PropertyWritable) != 0; }

bool IsEnumerable() const { return (m\_attributes & PropertyEnumerable) != 0; }

bool IsNoCache() const { return m\_instance && m\_propertyIndex == Constants::NoSlot; }

void AddFlags(InlineCacheFlags newFlag) { flags = (InlineCacheFlags)(flags | newFlag); }

InlineCacheFlags GetFlags() const { return flags; }

PropertyAttributes GetAttributes() const { return m\_attributes; }

// Set property index and IsWritable cache info

static void Set(PropertyValueInfo\* info, RecyclableObject\* instance, PropertyIndex propertyIndex, PropertyAttributes attributes = PropertyWritable,

InlineCacheFlags flags = InlineCacheNoFlags)

{

if (info)

{

info->Set(instance, propertyIndex, attributes, flags);

}

}

static void SetCacheInfo(PropertyValueInfo\* info, InlineCache \*const inlineCache);

static void SetCacheInfo(PropertyValueInfo\* info, FunctionBody \*const functionBody, InlineCache \*const inlineCache, const InlineCacheIndex inlineCacheIndex, const bool allowResizingPolymorphicInlineCache);

static void SetCacheInfo(PropertyValueInfo\* info, FunctionBody \*const functionBody, PolymorphicInlineCache \*const polymorphicInlineCache, const InlineCacheIndex inlineCacheIndex, const bool allowResizingPolymorphicInlineCache);

static void ClearCacheInfo(PropertyValueInfo\* info);

\_\_inline InlineCache \* GetInlineCache() const

{

return this->inlineCache;

}

\_\_inline PolymorphicInlineCache \* GetPolymorphicInlineCache() const

{

return this->polymorphicInlineCache;

}

\_\_inline FunctionBody \* GetFunctionBody() const

{

return this->functionBody;

}

\_\_inline uint GetInlineCacheIndex() const

{

return this->inlineCacheIndex;

}

bool AllowResizingPolymorphicInlineCache() const

{

return allowResizingPolymorphicInlineCache;

}

// Set to indicate the instance can't cache property index / IsWritable

static void SetNoCache(PropertyValueInfo\* info, RecyclableObject\* instance)

{

Set(info, instance, Constants::NoSlot, PropertyNone, InlineCacheNoFlags);

}

static void DisablePrototypeCache(PropertyValueInfo\* info, RecyclableObject\* instance)

{

if (info)

{

info->SetInfoFlag(disablePrototypeCacheFlag);

}

}

static bool PrototypeCacheDisabled(const PropertyValueInfo\* info)

{

return (info != NULL) && !!info->IsInfoFlagSet(disablePrototypeCacheFlag);

}

static void DisableStoreFieldCache(PropertyValueInfo\* info)

{

if (info)

{

info->ClearInfoFlag(enableStoreFieldCacheFlag);

}

}

static bool IsStoreFieldCacheEnabled(const PropertyValueInfo\* info)

{

return (info != NULL) && !!info->IsInfoFlagSet(enableStoreFieldCacheFlag);

}

bool IsStoreFieldCacheEnabled() const

{

return IsStoreFieldCacheEnabled(this);

}

};

enum SideEffects : byte

{

SideEffects\_None = 0,

SideEffects\_MathFunc = 0x1,

SideEffects\_ValueOf = 0x2,

SideEffects\_ToString = 0x4,

SideEffects\_Accessor = 0x8,

SideEffects\_ToPrimitive = SideEffects\_ValueOf | SideEffects\_ToString,

SideEffects\_Any = SideEffects\_MathFunc | SideEffects\_ValueOf | SideEffects\_ToString | SideEffects\_Accessor

};

// int32 is used in JIT code to pass the flag

// Used to tweak type system methods behavior.

// Normally, use: PropertyOperation\_None.

enum PropertyOperationFlags : int32

{

PropertyOperation\_None = 0x00,

PropertyOperation\_StrictMode = 0x01,

PropertyOperation\_Root = 0x02, // Operation doesn't specify base

// In particular, used by SetProperty/WithAttributes to throw, rather than return false, when then instance object is not extensible.

PropertyOperation\_ThrowIfNotExtensible = 0x04,

// Intent: avoid any checks and force the operation.

// In particular, used by SetProperty/WithAttributes to force adding a property when an object is not extensible.

PropertyOperation\_Force = 0x08,

// Initializing a property with a special internal value, which the user's code will never see.

PropertyOperation\_SpecialValue = 0x10,

// Pre-initializing a property value before the user's code actually does.

PropertyOperation\_PreInit = 0x20,

// Don't mark this fields as fixed in the type handler.

PropertyOperation\_NonFixedValue = 0x40,

PropertyOperation\_PreInitSpecialValue = PropertyOperation\_PreInit | PropertyOperation\_SpecialValue,

PropertyOperation\_StrictModeRoot = PropertyOperation\_StrictMode | PropertyOperation\_Root,

// No need to check for undeclared let/const (as this operation is initializing the let/const)

PropertyOperation\_AllowUndecl = 0x80,

// No need to check for undeclared let/const in case of console scope (as this operation is initializing the let/const)

PropertyOperation\_AllowUndeclInConsoleScope = 0x100,

PropertyOperation\_ThrowIfNonWritable = 0x300

};

class RecyclableObject : public FinalizableObject

{

friend class JavascriptOperators;

#if DBG

public:

DECLARE\_VALIDATE\_VTABLE\_REGISTERED\_NOBASE(RecyclableObject);

#endif

#if DBG || defined(PROFILE\_TYPES)

protected:

RecyclableObject(DynamicType \* type, ScriptContext \* scriptContext);

private:

void RecordAllocation(ScriptContext \* scriptContext);

#endif

protected:

Type \* type;

DEFINE\_VTABLE\_CTOR\_NOBASE(RecyclableObject);

virtual RecyclableObject\* GetPrototypeSpecial();

RecyclableObject() { /\* Do nothing, needed by the vtable ctor for ForInObjectEnumeratorWrapper \*/ }

public:

static bool Is(Var aValue);

static RecyclableObject\* FromVar(Var varValue);

RecyclableObject(Type \* type);

static DWORD GetTypeOffset() { return offsetof(RecyclableObject, type); }

#if DBG\_EXTRAFIELD

// This dtor should only be call when OOM occurs and RecyclableObject ctor has completed

// as the base class, or we have a stack instance

~RecyclableObject() { dtorCalled = true; }

#endif

ScriptContext\* GetScriptContext() const;

TypeId GetTypeId() const;

RecyclableObject\* GetPrototype() const;

JavascriptMethod GetEntryPoint() const;

JavascriptLibrary\* GetLibrary() const;

Recycler\* GetRecycler() const;

void SetIsPrototype();

// Is this object known to have only writable data properties

// (i.e. no accessors or non-writable properties)?

bool HasOnlyWritableDataProperties();

void ClearWritableDataOnlyDetectionBit();

bool IsWritableDataOnlyDetectionBitSet();

\_\_inline Type \* GetType() const { return type; }

// In order to avoid a branch, every object has an entry point if it gets called like a

// function - however, if it can't be called like a function, it's set to DefaultEntryPoint

// which will emit an error.

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

static Var DefaultExternalEntryPoint(RecyclableObject\* function, CallInfo callInfo, Var\* arguments);

virtual PropertyId GetPropertyId(PropertyIndex index) { return Constants::NoProperty; }

virtual PropertyId GetPropertyId(BigPropertyIndex index) { return Constants::NoProperty; }

virtual PropertyIndex GetPropertyIndex(PropertyId propertyId) { return Constants::NoSlot; }

virtual int GetPropertyCount() { return 0; }

virtual BOOL HasProperty(PropertyId propertyId);

virtual BOOL HasOwnProperty( PropertyId propertyId);

virtual BOOL HasOwnPropertyNoHostObject( PropertyId propertyId);

virtual BOOL HasOwnPropertyCheckNoRedecl( PropertyId propertyId) { Assert(FALSE); return FALSE; }

virtual BOOL UseDynamicObjectForNoHostObjectAccess() { return FALSE; }

virtual DescriptorFlags GetSetter(PropertyId propertyId, Var\* setterValue, PropertyValueInfo\* info, ScriptContext\* requestContext) { return None; }

virtual DescriptorFlags GetSetter(JavascriptString\* propertyNameString, Var\* setterValue, PropertyValueInfo\* info, ScriptContext\* requestContext) { return None; }

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

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

virtual BOOL GetInternalProperty(Var instance, PropertyId internalPropertyId, Var\* value, PropertyValueInfo\* info, ScriptContext\* requestContext);

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

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

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

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

virtual BOOL SetInternalProperty(PropertyId internalPropertyId, Var value, PropertyOperationFlags flags, PropertyValueInfo\* info);

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

virtual BOOL EnsureProperty(PropertyId propertyId);

virtual BOOL EnsureNoRedeclProperty(PropertyId propertyId);

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

virtual BOOL InitPropertyScoped(PropertyId propertyId, Var value);

virtual BOOL InitFuncScoped(PropertyId propertyId, Var value);

virtual BOOL DeleteProperty(PropertyId propertyId, PropertyOperationFlags flags);

virtual BOOL IsFixedProperty(PropertyId propertyId);

virtual BOOL HasItem(uint32 index);

virtual BOOL HasOwnItem(uint32 index);

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

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

virtual DescriptorFlags GetItemSetter(uint32 index, Var\* setterValue, ScriptContext\* requestContext) { return None; }

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

virtual BOOL DeleteItem(uint32 index, PropertyOperationFlags flags);

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

virtual BOOL ToPrimitive(JavascriptHint hint, Var\* value, ScriptContext \* requestContext);

virtual BOOL SetAccessors(PropertyId propertyId, Var getter, Var setter, PropertyOperationFlags flags = PropertyOperation\_None);

virtual BOOL Equals(Var other, BOOL\* value, ScriptContext\* requestContext);

virtual BOOL StrictEquals(Var other, BOOL\* value, ScriptContext\* requestContext);

virtual BOOL IsWritable(PropertyId propertyId) { return false; }

virtual BOOL IsConfigurable(PropertyId propertyId) { return false; }

virtual BOOL IsEnumerable(PropertyId propertyId) { return false; }

virtual BOOL IsExtensible() { return false; }

virtual BOOL PreventExtensions() { return false; }; // Sets [[Extensible]] flag of instance to false

virtual void ThrowIfCannotDefineProperty(PropertyId propId, PropertyDescriptor descriptor);

virtual void ThrowIfCannotGetOwnPropertyDescriptor(PropertyId propId) {}

virtual BOOL GetDefaultPropertyDescriptor(PropertyDescriptor& descriptor);

virtual BOOL Seal() { return false; } // Seals the instance, no additional property can be added or deleted

virtual BOOL Freeze() { return false; } // Freezes the instance, no additional property can be added or deleted or written

virtual BOOL IsSealed() { return false; }

virtual BOOL IsFrozen() { return false; }

virtual BOOL SetWritable(PropertyId propertyId, BOOL value) { return false; }

virtual BOOL SetConfigurable(PropertyId propertyId, BOOL value) { return false; }

virtual BOOL SetEnumerable(PropertyId propertyId, BOOL value) { return false; }

virtual BOOL SetAttributes(PropertyId propertyId, PropertyAttributes attributes) { return false; }

virtual BOOL GetSpecialPropertyName(uint32 index, Var \*propertyName, ScriptContext \* requestContext) { return false; }

virtual uint GetSpecialPropertyCount() const { return 0; }

virtual PropertyId const \* GetSpecialPropertyIds() const { return nullptr; }

virtual RecyclableObject\* GetThisObjectOrUnWrap(); // Due to the withScope object there are times we need to unwrap

virtual BOOL HasInstance(Var instance, ScriptContext\* scriptContext, IsInstInlineCache\* inlineCache = NULL);

BOOL SkipsPrototype() const;

BOOL CanHaveInterceptors() const;

BOOL IsExternal() const;

// Used only in JsVarToExtension where it may be during dispose and the type is not available

virtual BOOL IsExternalVirtual() const { return FALSE; }

virtual RecyclableObject\* GetConfigurablePrototype(ScriptContext \* requestContext) { return GetPrototype(); }

virtual Js::JavascriptString\* GetClassName(ScriptContext \* requestContext);

virtual RecyclableObject\* GetProxiedObjectForHeapEnum();

#if DBG

virtual bool CanStorePropertyValueDirectly(PropertyId propertyId, bool allowLetConst) { Assert(false); return false; };

#endif

virtual void RemoveFromPrototype(ScriptContext \* requestContext) { AssertMsg(false, "Shouldn't call this implementation."); }

virtual void AddToPrototype(ScriptContext \* requestContext) { AssertMsg(false, "Shouldn't call this implementation."); }

virtual void SetPrototype(RecyclableObject\* newPrototype) { AssertMsg(false, "Shouldn't call this implementation."); }

virtual BOOL ToString(Js::Var\* value, Js::ScriptContext\* scriptContext) { AssertMsg(FALSE, "Do not use this function."); return false; }

// don't need cross-site: in HostDispatch it's IDispatchEx based; in CustomExternalObject we have marshalling code explicitly.

virtual Var GetNamespaceParent(Js::Var aChild) { return nullptr; }

virtual HRESULT QueryObjectInterface(REFIID riid, void \*\*ppvObj);

virtual BOOL GetDiagValueString(StringBuilder<ArenaAllocator>\* stringBuilder, ScriptContext\* requestContext);

virtual BOOL GetDiagTypeString(StringBuilder<ArenaAllocator>\* stringBuilder, ScriptContext\* requestContext);

virtual RecyclableObject\* ToObject(ScriptContext \* requestContext);

virtual Var GetTypeOfString(ScriptContext\* requestContext);

// don't need cross-site: only supported in HostDispatch.

virtual Var InvokePut(Arguments args);

virtual BOOL GetRemoteTypeId(TypeId\* typeId);

// Only implemented by the HostDispatch object for cross-thread support

// Only supports a subset of entry points to be called remotely.

// For a list of supported entry points see the BuiltInOperation enum defined in JscriptInfo.idl

virtual BOOL InvokeBuiltInOperationRemotely(JavascriptMethod entryPoint, Arguments args, Var\* result) { return FALSE; };

// don't need cross-site: only supported in HostDispatch.

virtual DynamicObject\* GetRemoteObject();

// don't need cross-site: get the HostDispatch for global object/module root. don't need marshalling.

virtual Var GetHostDispatchVar();

virtual RecyclableObject \* CloneToScriptContext(ScriptContext\* requestContext);

// If dtor is called, that means that OOM happened (mostly), then the vtable might not be initialized

// to the base class', so we can't assert.

virtual void Finalize(bool isShutdown) override {

#ifdef DBG\_EXTRAFIELD

AssertMsg(dtorCalled, "Can't allocate a finalizable object without implementing Finalize");

#endif

}

virtual void Dispose(bool isShutdown) override {

#ifdef DBG\_EXTRAFIELD

AssertMsg(dtorCalled, "Can't allocate a finalizable object without implementing Dispose");

#endif

}

virtual void Mark(Recycler \*recycler) override { AssertMsg(false, "Mark called on object that isn't TrackableObject"); }

static uint32 GetOffsetOfType();

virtual void InvalidateCachedScope() { return; }

virtual BOOL HasDeferredTypeHandler() const { return false; }

#if DBG

public:

// Used to Assert that the object may safely be cast to a DynamicObject

virtual bool DbgIsDynamicObject() const { return false; }

virtual BOOL DbgSkipsPrototype() const { return FALSE; }

virtual BOOL DbgCanHaveInterceptors() const { return false; }

#endif

#if defined(PROFILE\_RECYCLER\_ALLOC) && defined(RECYCLER\_DUMP\_OBJECT\_GRAPH)

public:

static bool DumpObjectFunction(type\_info const \* typeinfo, bool isArray, void \* objectAddress);

#endif

private:

#if DBG\_EXTRAFIELD

bool dtorCalled;

#endif

friend class LowererMD;

friend class LowererMDArch;

friend struct InlineCache;

#ifdef HEAP\_ENUMERATION\_VALIDATION

private:

UINT m\_heapEnumValidationCookie;

public:

void SetHeapEnumValidationCookie(int cookie ) { m\_heapEnumValidationCookie = cookie; }

int GetHeapEnumValidationCookie() { return m\_heapEnumValidationCookie; }

#endif

};

}

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

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

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

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

#pragma once

namespace Js

{

// These function needs to be in INL file for static lib

#if INT32VAR

inline bool RecyclableObject::Is(Var aValue)

{

AssertMsg(aValue != nullptr, "RecyclableObject::Is aValue is null");

return (((uintptr)aValue) >> VarTag\_Shift) == 0;

}

#else

inline bool RecyclableObject::Is(Var aValue)

{

AssertMsg(aValue != nullptr, "RecyclableObject::Is aValue is null");

return (((uintptr)aValue) & AtomTag) == AtomTag\_Object;

}

#endif

inline RecyclableObject\* RecyclableObject::FromVar(const Js::Var aValue)

{

AssertMsg(AtomTag\_Object == 0, "Ensure GC objects do not need to be marked");

AssertMsg(Is(aValue), "Ensure instance is a RecyclableObject");

AssertMsg(!TaggedNumber::Is(aValue), "Tagged value being used as RecyclableObject");

return reinterpret\_cast<RecyclableObject \*>(aValue);

}

inline TypeId RecyclableObject::GetTypeId() const

{

return this->GetType()->GetTypeId();

}

inline JavascriptLibrary\* RecyclableObject::GetLibrary() const

{

return this->GetType()->GetLibrary();

}

inline ScriptContext\* RecyclableObject::GetScriptContext() const

{

return this->GetLibrary()->GetScriptContext();

}

inline BOOL RecyclableObject::CanHaveInterceptors() const

{

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

Assert(this->DbgCanHaveInterceptors() == this->GetType()->CanHaveInterceptors());

#endif

return this->GetType()->CanHaveInterceptors();

}

};

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

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

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

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

#include "RuntimeTypePch.h"

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

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

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

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

#pragma once

#include "Runtime.h"

#include "Language\CacheOperators.h"

#include "Language\InlineCachePointerArray.h"

#include "Types\WithScopeObject.h"

#include "Types\TypePropertyCache.h"

#include "Types\MissingPropertyTypeHandler.h"

#include "Types\PathTypeHandler.h"

#include "Types\PropertyIndexRanges.h"

#include "Types\SimpleDictionaryPropertyDescriptor.h"

#include "Types\SimpleDictionaryTypeHandler.h"

#include "Types\SimpleDictionaryUnorderedTypeHandler.h"

#include "Types\DictionaryPropertyDescriptor.h"

#include "Types\DictionaryTypeHandler.h"

#include "Types\ES5ArrayTypeHandler.h"

#include "Types\DynamicObjectEnumerator.h"

#include "Types\DynamicObjectSnapshotEnumerator.h"

#include "Types\DynamicObjectSnapshotEnumeratorWPCache.h"

// .inl files

#include "Language\CacheOperators.inl"

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

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

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

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

#include "RuntimeTypePch.h"

namespace Js

{

ScriptFunctionType::ScriptFunctionType(ScriptFunctionType \* type)

: DynamicType(type), entryPointInfo(type->GetEntryPointInfo())

{}

ScriptFunctionType::ScriptFunctionType(ScriptContext\* scriptContext, RecyclableObject\* prototype,

JavascriptMethod entryPoint, ProxyEntryPointInfo \* entryPointInfo, DynamicTypeHandler \* typeHandler,

bool isLocked, bool isShared)

: DynamicType(scriptContext, TypeIds\_Function, prototype, entryPoint, typeHandler, isLocked, isShared),

entryPointInfo(entryPointInfo)

{

}

ScriptFunctionType \* ScriptFunctionType::New(FunctionProxy \* proxy, bool isShared)

{

Assert(proxy->GetFunctionProxy() == proxy);

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

JavascriptLibrary \* library = scriptContext->GetLibrary();

DynamicObject \* functionPrototype = proxy->IsAsync() ? library->GetAsyncFunctionPrototype() : library->GetFunctionPrototype();

JavascriptMethod address = (JavascriptMethod)proxy->GetDefaultEntryPointInfo()->address;

return RecyclerNew(scriptContext->GetRecycler(), ScriptFunctionType,

scriptContext, functionPrototype,

address,

proxy->GetDefaultEntryPointInfo(),

library->ScriptFunctionTypeHandler(proxy->IsLambda() || proxy->IsAsync() || proxy->IsClassMethod(), proxy->GetIsAnonymousFunction()),

isShared, isShared);

}

};

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

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

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

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

#pragma once

namespace Js

{

class ScriptFunctionType : public DynamicType

{

public:

static ScriptFunctionType \* New(FunctionProxy \* proxy, bool isShared);

static DWORD GetEntryPointInfoOffset() { return offsetof(ScriptFunctionType, entryPointInfo); }

ProxyEntryPointInfo \* GetEntryPointInfo() const { return entryPointInfo; }

void SetEntryPointInfo(ProxyEntryPointInfo \* entryPointInfo) { this->entryPointInfo = entryPointInfo; }

private:

ScriptFunctionType(ScriptFunctionType \* type);

ScriptFunctionType(ScriptContext\* scriptContext, RecyclableObject\* prototype,

JavascriptMethod entryPoint, ProxyEntryPointInfo \* entryPointInfo, DynamicTypeHandler \* typeHandler,

bool isLocked, bool isShared);

ProxyEntryPointInfo \* entryPointInfo;

friend class ScriptFunction;

friend class JavascriptLibrary;

};

};

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

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

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

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

#pragma once

namespace Js

{

template <typename TPropertyIndex>

class SimpleDictionaryPropertyDescriptor

{

public:

SimpleDictionaryPropertyDescriptor() :

propertyIndex(NoSlots), Attributes(PropertyDynamicTypeDefaults),

preventFalseReference(true), isInitialized(false), isFixed(false), usedAsFixed(false) { }

SimpleDictionaryPropertyDescriptor(TPropertyIndex inPropertyIndex) :

propertyIndex(inPropertyIndex), Attributes(PropertyDynamicTypeDefaults),

preventFalseReference(true), isInitialized(false), isFixed(false), usedAsFixed(false) { }

SimpleDictionaryPropertyDescriptor(TPropertyIndex inPropertyIndex, PropertyAttributes attributes) :

propertyIndex(inPropertyIndex), Attributes(attributes),

preventFalseReference(true), isInitialized(false), isFixed(false), usedAsFixed(false) { }

// SimpleDictionaryPropertyDescriptor is allocated by a dictionary along with the PropertyRecord

// so it can not allocate as leaf, tag the lower bit to prevent false reference

bool preventFalseReference:1;

bool isInitialized: 1;

bool isFixed:1;

bool usedAsFixed:1;

PropertyAttributes Attributes;

TPropertyIndex propertyIndex;

bool HasNonLetConstGlobal() const

{

return (this->Attributes & PropertyLetConstGlobal) == 0;

}

private:

static const TPropertyIndex NoSlots = PropertyIndexRanges<TPropertyIndex>::NoSlots;

};

}

namespace JsUtil

{

template <typename TPropertyIndex>

class ValueEntry<Js::SimpleDictionaryPropertyDescriptor<TPropertyIndex> >: public BaseValueEntry<Js::SimpleDictionaryPropertyDescriptor<TPropertyIndex>>

{

public:

void Clear()

{

this->value = 0;

}

};

}

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

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

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

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

#include "RuntimeTypePch.h"

#include "Library\ForInObjectEnumerator.h"

namespace Js

{

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

// Helper methods to deal with differing TMapKey and TPropertyKey types.

// Used by both SimpleDictionaryTypeHandler and DictionaryTypeHandler.

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

PropertyId TMapKey\_GetPropertyId(ScriptContext\* scriptContext, const PropertyRecord\* key)

{

return key->GetPropertyId();

}

PropertyId TMapKey\_GetPropertyId(ScriptContext\* scriptContext, JavascriptString\* key)

{

return scriptContext->GetOrAddPropertyIdTracked(key->GetSz(), key->GetLength());

}

bool TMapKey\_IsSymbol(const PropertyRecord\* key, ScriptContext\* scriptContext)

{

return key->IsSymbol();

}

bool TMapKey\_IsSymbol(JavascriptString\* key, ScriptContext\* scriptContext)

{

// Property indexed via string cannot be a symbol.

return false;

}

bool TMapKey\_IsSymbol(PropertyId key, ScriptContext\* scriptContext)

{

return scriptContext->GetPropertyName(key)->IsSymbol();

}

template<typename TMapKey>

TMapKey TMapKey\_ConvertKey(ScriptContext\* scriptContext, const PropertyRecord\* key)

{

return key;

}

template<>

JavascriptString\* TMapKey\_ConvertKey(ScriptContext\* scriptContext, const PropertyRecord\* key)

{

// String keyed type handlers can't handle InternalPropertyIds because they have no string representation

// so assert that no code paths convert InternalPropertyIds to PropertyStrings.

Assert(!IsInternalPropertyId(key->GetPropertyId()));

// The same is true for symbols - we should not be converting a symbol property into a PropertyString.

Assert(!key->IsSymbol());

return scriptContext->GetPropertyString(key->GetPropertyId());

}

template<typename TMapKey>

TMapKey TMapKey\_ConvertKey(ScriptContext\* scriptContext, JavascriptString\* key)

{

return key;

}

template<>

const PropertyRecord\* TMapKey\_ConvertKey(ScriptContext\* scriptContext, JavascriptString\* key)

{

PropertyRecord const \* propertyRecord;

if (VirtualTableInfo<Js::PropertyString>::HasVirtualTable(key))

{

propertyRecord = ((PropertyString\*)key)->GetPropertyRecord();

}

else

{

scriptContext->GetOrAddPropertyRecord(key->GetString(), key->GetLength(), &propertyRecord);

}

return propertyRecord;

}

bool TPropertyKey\_IsInternalPropertyId(JavascriptString\* key)

{

// WARNING: This will return false for PropertyStrings that are actually InternalPropertyIds

Assert(!VirtualTableInfo<PropertyString>::HasVirtualTable(key) || !IsInternalPropertyId(((PropertyString\*)key)->GetPropertyRecord()->GetPropertyId()));

return false;

}

bool TPropertyKey\_IsInternalPropertyId(const PropertyRecord\* key)

{

return IsInternalPropertyId(key->GetPropertyId()) ? true : false;

}

bool TPropertyKey\_IsInternalPropertyId(PropertyId key)

{

return IsInternalPropertyId(key) ? true : false;

}

template <typename TMapKey>

bool TMapKey\_IsJavascriptString()

{

return false;

}

template <>

bool TMapKey\_IsJavascriptString<JavascriptString\*>()

{

return true;

}

template <typename TPropertyKey>

bool TPropertyKey\_IsJavascriptString()

{

return false;

}

template<>

bool TPropertyKey\_IsJavascriptString<JavascriptString\*>()

{

return true;

}

template <typename TPropertyKey>

PropertyId TPropertyKey\_GetOptionalPropertyId(ScriptContext\* scriptContext, TPropertyKey key)

{

return key;

}

template <>

PropertyId TPropertyKey\_GetOptionalPropertyId(ScriptContext\* scriptContext, const PropertyRecord\* key)

{

return key->GetPropertyId();

}

template <>

PropertyId TPropertyKey\_GetOptionalPropertyId(ScriptContext\* scriptContext, JavascriptString\* key)

{

const PropertyRecord\* propertyRecord = nullptr;

scriptContext->FindPropertyRecord(key, &propertyRecord);

return propertyRecord != nullptr ? propertyRecord->GetPropertyId() : Constants::NoProperty;

}

JavascriptString\* TMapKey\_OptionalConvertPropertyIdToPropertyRecord(ScriptContext\* scriptContext, JavascriptString\* propertyString)

{

return propertyString;

}

const PropertyRecord\* TMapKey\_OptionalConvertPropertyIdToPropertyRecord(ScriptContext\* scriptContext, PropertyId propertyId)

{

return scriptContext->GetPropertyName(propertyId);

}

template <typename TPropertyKey>

PropertyId TPropertyKey\_GetUpdateSideEffectPropertyId(PropertyId propertyId, TPropertyKey propertyKey);

template <>

PropertyId TPropertyKey\_GetUpdateSideEffectPropertyId<PropertyId>(PropertyId propertyId, PropertyId propertyKey)

{

Assert(propertyId != Js::Constants::NoProperty);

Assert(propertyId == propertyKey);

return propertyKey;

}

template <>

PropertyId TPropertyKey\_GetUpdateSideEffectPropertyId<JavascriptString \*>(PropertyId propertyId, JavascriptString \* propertyKey)

{

if (propertyId != Js::Constants::NoProperty)

{

return propertyId;

}

JsUtil::CharacterBuffer<WCHAR> propertyStr(propertyKey->GetString(), propertyKey->GetLength());

if (BuiltInPropertyRecords::valueOf.Equals(propertyStr))

{

return PropertyIds::valueOf;

}

if (BuiltInPropertyRecords::toString.Equals(propertyStr))

{

return PropertyIds::toString;

}

return Js::Constants::NoProperty;

}

#if DBG

template <typename TPropertyKey>

bool TPropertyKey\_IsNumeric(TPropertyKey key)

{

return false;

}

template <>

bool TPropertyKey\_IsNumeric(const PropertyRecord\* key)

{

return key->IsNumeric();

}

#endif

#ifdef ENABLE\_DEBUG\_CONFIG\_OPTIONS

const wchar\_t\* TMapKey\_GetBuffer(const PropertyRecord\* key)

{

return key->GetBuffer();

}

const wchar\_t\* TMapKey\_GetBuffer(JavascriptString\* key)

{

return key->GetSz();

}

#endif

// Round up requested property capacity and cap by max range value.

template <typename Ranges>

void PropertyIndexRangesBase<Ranges>::VerifySlotCapacity(int requestedCapacity)

{

Assert(requestedCapacity <= static\_cast<int>(Ranges::MaxValue)); // Should never request more than max range value

}

template <typename TPropertyIndex, typename TMapKey, bool IsNotExtensibleSupported>

SimpleDictionaryTypeHandlerBase<TPropertyIndex, TMapKey, IsNotExtensibleSupported> \* SimpleDictionaryTypeHandlerBase<TPropertyIndex, TMapKey, IsNotExtensibleSupported>::New(Recycler \* recycler, int initialCapacity, uint16 inlineSlotCapacity, uint16 offsetOfInlineSlots, bool isLocked, bool isShared)

{

PropertyIndexRangesType::VerifySlotCapacity(initialCapacity);

return RecyclerNew(recycler, SimpleDictionaryTypeHandlerBase, recycler, initialCapacity, inlineSlotCapacity, offsetOfInlineSlots, isLocked, isShared);

}

template <typename TPropertyIndex, typename TMapKey, bool IsNotExtensibleSupported>

SimpleDictionaryTypeHandlerBase<TPropertyIndex, TMapKey, IsNotExtensibleSupported> \* SimpleDictionaryTypeHandlerBase<TPropertyIndex, TMapKey, IsNotExtensibleSupported>::New(ScriptContext \* scriptContext, SimplePropertyDescriptor const\* propertyDescriptors, int propertyCount, uint16 inlineSlotCapacity, uint16 offsetOfInlineSlots, bool isLocked, bool isShared)

{

PropertyIndexRangesType::VerifySlotCapacity(propertyCount);

return RecyclerNew(scriptContext->GetRecycler(), SimpleDictionaryTypeHandlerBase, scriptContext, propertyDescriptors, propertyCount, propertyCount, inlineSlotCapacity, offsetOfInlineSlots, isLocked, isShared);

}

template <typename TPropertyIndex, typename TMapKey, bool IsNotExtensibleSupported>

SimpleDictionaryTypeHandlerBase<TPropertyIndex, TMapKey, IsNotExtensibleSupported>::SimpleDictionaryTypeHandlerBase(Recycler\* recycler) :

// We can do slotCapacity roundup here because this constructor is always creating type handler for a new object.

DynamicTypeHandler(1),

nextPropertyIndex(0),

singletonInstance(nullptr),

isUnordered(false),

numDeletedProperties(0)

{

SetIsInlineSlotCapacityLocked();

propertyMap = RecyclerNew(recycler, SimplePropertyDescriptorMap, recycler, this->GetSlotCapacity());

}

template <typename TPropertyIndex, typename TMapKey, bool IsNotExtensibleSupported>

SimpleDictionaryTypeHandlerBase<TPropertyIndex, TMapKey, IsNotExtensibleSupported>::SimpleDictionaryTypeHandlerBase(ScriptContext \* scriptContext, SimplePropertyDescriptor const\* propertyDescriptors, int propertyCount, int slotCapacity, uint16 inlineSlotCapacity, uint16 offsetOfInlineSlots, bool isLocked, bool isShared) :

// Do not RoundUp passed in slotCapacity. This may be called by ConvertTypeHandler for an existing DynamicObject and should use the real existing slotCapacity.

DynamicTypeHandler(slotCapacity, inlineSlotCapacity, offsetOfInlineSlots, DefaultFlags | (isLocked ? IsLockedFlag : 0) | (isShared ? (MayBecomeSharedFlag | IsSharedFlag) : 0)),

nextPropertyIndex(0),

singletonInstance(nullptr),

isUnordered(false),

hasNamelessPropertyId(false),

numDeletedProperties(0)

{

SetIsInlineSlotCapacityLocked();

Assert(slotCapacity <= MaxPropertyIndexSize);

propertyMap = RecyclerNew(scriptContext->GetRecycler(), SimplePropertyDescriptorMap, scriptContext->GetRecycler(), propertyCount);

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

{

Add(propertyDescriptors[i].Id, propertyDescriptors[i].Attributes, false, false, false, scriptContext);

}

}

template <typename TPropertyIndex, typename TMapKey, bool IsNotExtensibleSupported>

SimpleDictionaryTypeHandlerBase<TPropertyIndex, TMapKey, IsNotExtensibleSupported>::SimpleDictionaryTypeHandlerBase(Recycler \* recycler, int slotCapacity, uint16 inlineSlotCapacity, uint16 offsetOfInlineSlots, bool isLocked, bool isShared) :

// Do not RoundUp passed in slotCapacity. This may be called by ConvertTypeHandler for an existing DynamicObject and should use the real existing slotCapacity.

DynamicTypeHandler(slotCapacity, inlineSlotCapacity, offsetOfInlineSlots, DefaultFlags | (isLocked ? IsLockedFlag : 0) | (isShared ? (MayBecomeSharedFlag | IsSharedFlag) : 0)),

nextPropertyIndex(0),

singletonInstance(nullptr),

isUnordered(false),

hasNamelessPropertyId(false),

numDeletedProperties(0)

{

SetIsInlineSlotCapacityLocked();

Assert(slotCapacity <= MaxPropertyIndexSize);

propertyMap = RecyclerNew(recycler, SimplePropertyDescriptorMap, recycler, this->GetSlotCapacity());

}

template <typename TPropertyIndex, typename TMapKey, bool IsNotExtensibleSupported>

SimpleDictionaryTypeHandlerBase<TPropertyIndex, TMapKey, IsNotExtensibleSupported>::SimpleDictionaryTypeHandlerBase(Recycler\* recycler, int slotCapacity, int propertyCapacity, uint16 inlineSlotCapacity, uint16 offsetOfInlineSlots, bool isLocked = false, bool isShared = false) :

// Do not RoundUp passed in slotCapacity. This may be called by ConvertTypeHandler for an existing DynamicObject and should use the real existing slotCapacity.

DynamicTypeHandler(slotCapacity, inlineSlotCapacity, offsetOfInlineSlots, DefaultFlags | (isLocked ? IsLockedFlag : 0) | (isShared ? (MayBecomeSharedFlag | IsSharedFlag) : 0)),

nextPropertyIndex(0),

singletonInstance(nullptr),

isUnordered(false),

hasNamelessPropertyId(false),

numDeletedProperties(0)

{

SetIsInlineSlotCapacityLocked();

Assert(slotCapacity <= MaxPropertyIndexSize);

propertyMap = RecyclerNew(recycler, SimplePropertyDescriptorMap, recycler, propertyCapacity);

}

template <typename TPropertyIndex, typename TMapKey, bool IsNotExtensibleSupported>

void SimpleDictionaryTypeHandlerBase<TPropertyIndex, TMapKey, IsNotExtensibleSupported>::DoShareTypeHandler(ScriptContext\* scriptContext)

{

Assert((GetFlags() & (IsLockedFlag | MayBecomeSharedFlag | IsSharedFlag)) == (IsLockedFlag | MayBecomeSharedFlag));

Assert(HasSingletonInstanceOnlyIfNeeded());

// If this handler is becoming shared we need to remove the singleton instance (so that it can be collected

// if no longer referenced by anything else) and invalidate any fixed fields.

// The propertyMap dictionary is guaranteed to have contiguous entries because we never remove entries from it.

for (int index = 0; index < propertyMap->Count(); index++)

{

TMapKey propertyKey = propertyMap->GetKeyAt(index);

SimpleDictionaryPropertyDescriptor<TPropertyIndex>\* const descriptor = propertyMap->GetReferenceAt(index);

descriptor->isInitialized = true;

InvalidateFixedField(propertyKey, descriptor, scriptContext);

}

this->singletonInstance = nullptr;

}

template <typename TPropertyIndex, typename TMapKey, bool IsNotExtensibleSupported>

template <bool check\_\_proto\_\_>

DynamicType\* SimpleDictionaryTypeHandlerBase<TPropertyIndex, TMapKey, IsNotExtensibleSupported>::InternalCreateTypeForNewScObject(ScriptContext\* scriptContext, DynamicType\* type, const Js::PropertyIdArray \*propIds, bool shareType)

{

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

uint count = propIds->count;

Assert(count <= static\_cast<uint>(MaxPropertyIndexSize));

SimpleDictionaryTypeHandlerBase\* typeHandler = SimpleDictionaryTypeHandlerBase::New(recycler, count,

type->GetTypeHandler()->GetInlineSlotCapacity(), type->GetTypeHandler()->GetOffsetOfInlineSlots(), true, shareType);

if (!shareType) typeHandler->SetMayBecomeShared();

for (uint i = 0; i < count; i++)

{

//

// For a function with same named parameters,

// property id Constants::NoProperty will be passed for all the dupes except the last one

// We need to allocate space for dupes, but don't add those to map

//

PropertyId propertyId = propIds->elements[i];

const PropertyRecord\* propertyRecord = propertyId == Constants::NoProperty ? NULL : scriptContext->GetPropertyName(propertyId);

PropertyAttributes attr = PropertyRecord::DefaultAttributesForPropertyId(propertyId, check\_\_proto\_\_);

typeHandler->Add(propertyRecord, attr, shareType, false, false, scriptContext);

}

Assert((typeHandler->GetFlags() & IsPrototypeFlag) == 0);

#ifdef PROFILE\_OBJECT\_LITERALS

scriptContext->objectLiteralSimpleDictionaryCount++;

#endif

return RecyclerNew(recycler, DynamicType, type, typeHandler, /\* isLocked = \*/ true, /\* isShared = \*/ shareType);

}

template <typename TPropertyIndex, typename TMapKey, bool IsNotExtensibleSupported>

DynamicType\* SimpleDictionaryTypeHandlerBase<TPropertyIndex, TMapKey, IsNotExtensibleSupported>::CreateTypeForNewScObject(ScriptContext\* scriptContext, DynamicType\* type, const Js::PropertyIdArray \*propIds, bool shareType, bool check\_\_proto\_\_)

{

if (check\_\_proto\_\_)

{

return InternalCreateTypeForNewScObject<true>(scriptContext, type, propIds, shareType);

}

else

{

return InternalCreateTypeForNewScObject<false>(scriptContext, type, propIds, shareType);

}

}

template <typename TPropertyIndex, typename TMapKey, bool IsNotExtensibleSupported>

int SimpleDictionaryTypeHandlerBase<TPropertyIndex, TMapKey, IsNotExtensibleSupported>::GetPropertyCount()

{

return propertyMap->Count();

}

template <typename TPropertyIndex, typename TMapKey, bool IsNotExtensibleSupported>

bool SimpleDictionaryTypeHandlerBase<TPropertyIndex, TMapKey, IsNotExtensibleSupported>::SupportsSwitchingToUnordered(

const ScriptContext \*const scriptContext) const

{

Assert(scriptContext);

return

!isUnordered &&

CONFIG\_FLAG(DeletedPropertyReuseThreshold) > 0;

}

template <typename TPropertyIndex, typename TMapKey, bool IsNotExtensibleSupported>

SimpleDictionaryUnorderedTypeHandler<TPropertyIndex, TMapKey, IsNotExtensibleSupported> \*SimpleDictionaryTypeHandlerBase<TPropertyIndex, TMapKey, IsNotExtensibleSupported>::AsUnordered()

{

return static\_cast<SimpleDictionaryUnorderedTypeHandler<TPropertyIndex, TMapKey, IsNotExtensibleSupported> \*>(this);

}

template <typename TPropertyIndex, typename TMapKey, bool IsNotExtensibleSupported>

void SimpleDictionaryTypeHandlerBase<TPropertyIndex, TMapKey, IsNotExtensibleSupported>::SetNumDeletedProperties(const byte n)

{

numDeletedProperties = n;

}

template <typename TPropertyIndex, typename TMapKey, bool IsNotExtensibleSupported>

template <typename U, typename UMapKey>

U\* SimpleDictionaryTypeHandlerBase<TPropertyIndex, TMapKey, IsNotExtensibleSupported>::ConvertToTypeHandler(DynamicObject\* instance)

{

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

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

U\* newTypeHandler = RecyclerNew(recycler, U, recycler, GetSlotCapacity(), GetInlineSlotCapacity(), GetOffsetOfInlineSlots());

// We expect the new type handler to start off marked as having only writable data properties.

Assert(newTypeHandler->GetHasOnlyWritableDataProperties());

#ifdef ENABLE\_DEBUG\_CONFIG\_OPTIONS

DynamicType\* oldType = instance->GetDynamicType();

RecyclerWeakReference<DynamicObject>\* oldSingletonInstance = GetSingletonInstance();

TraceFixedFieldsBeforeTypeHandlerChange(L"SimpleDictionaryTypeHandler", L"[Simple]DictionaryTypeHandler", instance, this, oldType, oldSingletonInstance);

#endif

// We assume the new type handler is not shared. Hence it's ok to set this instance as the handler's singleton instance.

Assert(HasSingletonInstanceOnlyIfNeeded());

if (AreSingletonInstancesNeeded())

{

RecyclerWeakReference<DynamicObject>\* curSingletonInstance = this->singletonInstance;

if (curSingletonInstance != nullptr && curSingletonInstance->Get() == instance)

{

newTypeHandler->SetSingletonInstance(curSingletonInstance);

}

else

{

newTypeHandler->SetSingletonInstance(instance->CreateWeakReferenceToSelf());

}

}

// If we are a prototype or may become a prototype we must transfer used as fixed bits.

// See point 4 in PathTypeHandlerBase::ConvertToSimpleDictionaryType.

bool isGlobalObject = instance->GetTypeId() == TypeIds\_GlobalObject;

bool isTypeLocked = instance->GetDynamicType()->GetIsLocked();

bool isOrMayBecomeShared = GetIsOrMayBecomeShared();

Assert(!isOrMayBecomeShared || !IsolatePrototypes() || ((this->GetFlags() & IsPrototypeFlag) == 0));

// For the global object we don't emit a type check before a hard-coded use of a fixed field. Therefore a type transition isn't sufficient to

// invalidate any used fixed fields, and we must continue tracking them on the new type handler. If the type isn't locked, we may not change the

// type of the instance, and we must also track the used fixed fields on the new handler.

bool transferUsedAsFixed = isGlobalObject || !isTypeLocked || ((this->GetFlags() & IsPrototypeFlag) != 0 || (isOrMayBecomeShared && !IsolatePrototypes())) || PHASE\_FORCE1(Js::FixDataPropsPhase);

SimpleDictionaryPropertyDescriptor<TPropertyIndex> descriptor;

TMapKey propertyKey;

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

{

descriptor = propertyMap->GetValueAt(i);

propertyKey = propertyMap->GetKeyAt(i);

// newTH->nextPropertyIndex will be less than desc.propertyIndex, when we have function with same name parameters

if (newTypeHandler->nextPropertyIndex < (U::PropertyIndexType)descriptor.propertyIndex)

{

newTypeHandler->nextPropertyIndex = (U::PropertyIndexType)descriptor.propertyIndex;

}

Assert(newTypeHandler->nextPropertyIndex == descriptor.propertyIndex);

Assert(!GetIsShared() || !descriptor.isFixed);

newTypeHandler->Add(TMapKey\_ConvertKey<UMapKey>(scriptContext, propertyKey), descriptor.Attributes, descriptor.isInitialized, descriptor.isFixed, transferUsedAsFixed && descriptor.usedAsFixed, scriptContext);

}

newTypeHandler->nextPropertyIndex = (U::PropertyIndexType)nextPropertyIndex;

newTypeHandler->SetNumDeletedProperties(numDeletedProperties);

ClearSingletonInstance();

AssertMsg((newTypeHandler->GetFlags() & IsPrototypeFlag) == 0, "Why did we create a brand new type handler with a prototype flag set?");

newTypeHandler->SetFlags(IsPrototypeFlag, this->GetFlags());

newTypeHandler->ChangeFlags(IsExtensibleFlag | IsSealedOnceFlag | IsFrozenOnceFlag, this->GetFlags());

// Any new type handler we expect to see here should have inline slot capacity locked. If this were to change, we would need

// to update our shrinking logic (see PathTypeHandlerBase::ShrinkSlotAndInlineSlotCapacity).

Assert(newTypeHandler->GetIsInlineSlotCapacityLocked());

newTypeHandler->SetPropertyTypes(PropertyTypesWritableDataOnly | PropertyTypesWritableDataOnlyDetection, this->GetPropertyTypes());

newTypeHandler->SetInstanceTypeHandler(instance);

// We assumed that we don't need to transfer used as fixed bits unless we are a prototype, which is only valid if we also changed the type.

Assert(transferUsedAsFixed || (instance->GetType() != oldType && oldType->GetTypeId() != TypeIds\_GlobalObject));

Assert(!newTypeHandler->HasSingletonInstance() || !instance->HasSharedType());

#ifdef ENABLE\_DEBUG\_CONFIG\_OPTIONS

TraceFixedFieldsAfterTypeHandlerChange(instance, this, newTypeHandler, oldType, oldSingletonInstance);

#endif

return newTypeHandler;

}

template <typename TPropertyIndex, typename TMapKey, bool IsNotExtensibleSupported>

DictionaryTypeHandlerBase<TPropertyIndex>\* SimpleDictionaryTypeHandlerBase<TPropertyIndex, TMapKey, IsNotExtensibleSupported>::ConvertToDictionaryType(DynamicObject\* instance)

{

DictionaryTypeHandlerBase<TPropertyIndex>\* newTypeHandler = ConvertToTypeHandler<DictionaryTypeHandlerBase<TPropertyIndex>, const PropertyRecord\*>(instance);

#ifdef PROFILE\_TYPES

instance->GetScriptContext()->convertSimpleDictionaryToDictionaryCount++;

#endif

return newTypeHandler;

}

template <typename TPropertyIndex, typename TMapKey, bool IsNotExtensibleSupported>

ES5ArrayTypeHandlerBase<TPropertyIndex>\* SimpleDictionaryTypeHandlerBase<TPropertyIndex, TMapKey, IsNotExtensibleSupported>::ConvertToES5ArrayType(DynamicObject\* instance)

{

ES5ArrayTypeHandlerBase<TPropertyIndex>\* newTypeHandler = ConvertToTypeHandler<ES5ArrayTypeHandlerBase<TPropertyIndex>, const PropertyRecord\*>(instance);

#ifdef PROFILE\_TYPES

instance->GetScriptContext()->convertSimpleDictionaryToDictionaryCount++;

#endif

return newTypeHandler;

}

template <typename TPropertyIndex, typename TMapKey, bool IsNotExtensibleSupported>

SimpleDictionaryTypeHandlerBase<TPropertyIndex, TMapKey, IsNotExtensibleSupported>\* SimpleDictionaryTypeHandlerBase<TPropertyIndex, TMapKey, IsNotExtensibleSupported>::ConvertToNonSharedSimpleDictionaryType(DynamicObject\* instance)

{

// Although an unordered type handler is never actually shared, it can be flagged as shared by type snapshot enumeration

// to freeze the initial type handler before enumeration commences

SimpleDictionaryTypeHandlerBase<TPropertyIndex, TMapKey, IsNotExtensibleSupported>\* newTypeHandler =

isUnordered

? ConvertToSimpleDictionaryUnorderedTypeHandler<TPropertyIndex, TMapKey, IsNotExtensibleSupported>(instance)

: ConvertToTypeHandler<SimpleDictionaryTypeHandlerBase<TPropertyIndex, TMapKey, IsNotExtensibleSupported>, TMapKey>(instance);

#ifdef PROFILE\_TYPES

instance->GetScriptContext()->convertSimpleSharedDictionaryToNonSharedCount++;

#endif

return newTypeHandler;

}

template <typename TPropertyIndex, typename TMapKey, bool IsNotExtensibleSupported>

template <typename NewTPropertyIndex, typename NewTMapKey, bool NewIsNotExtensibleSupported>

SimpleDictionaryUnorderedTypeHandler<NewTPropertyIndex, NewTMapKey, NewIsNotExtensibleSupported>\* SimpleDictionaryTypeHandlerBase<TPropertyIndex, TMapKey, IsNotExtensibleSupported>::ConvertToSimpleDictionaryUnorderedTypeHandler(DynamicObject\* instance)

{

CompileAssert(sizeof(NewTPropertyIndex) >= sizeof(TPropertyIndex));

Assert(instance);

SimpleDictionaryUnorderedTypeHandler<NewTPropertyIndex, NewTMapKey, NewIsNotExtensibleSupported> \*const newTypeHandler =

ConvertToTypeHandler<SimpleDictionaryUnorderedTypeHandler<NewTPropertyIndex, NewTMapKey, NewIsNotExtensibleSupported>, NewTMapKey>(instance);

if(isUnordered)

{

newTypeHandler->CopyUnorderedStateFrom(\*AsUnordered());

}

else

{

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

{

SimpleDictionaryPropertyDescriptor<TPropertyIndex> descriptor(propertyMap->GetValueAt(i));

if(descriptor.Attributes & PropertyDeleted)

{

newTypeHandler->TryRegisterDeletedPropertyIndex(instance, descriptor.propertyIndex);

}

}

}

return newTypeHandler;

}

template <typename TPropertyIndex, typename TMapKey, bool IsNotExtensibleSupported>

PropertyId SimpleDictionaryTypeHandlerBase<TPropertyIndex, TMapKey, IsNotExtensibleSupported>::GetPropertyId(ScriptContext\* scriptContext, PropertyIndex index)

{

if (index < propertyMap->Count() && !(propertyMap->GetValueAt(index).Attributes & (PropertyDeleted | PropertyLetConstGlobal)))

{

return TMapKey\_GetPropertyId(scriptContext, propertyMap->GetKeyAt(index));

}

else

{

return Constants::NoProperty;

}

}

template <typename TPropertyIndex, typename TMapKey, bool IsNotExtensibleSupported>

PropertyId SimpleDictionaryTypeHandlerBase<TPropertyIndex, TMapKey, IsNotExtensibleSupported>::GetPropertyId(ScriptContext\* scriptContext, BigPropertyIndex index)

{

if (index < propertyMap->Count() && !(propertyMap->GetValueAt(index).Attributes & (PropertyDeleted | PropertyLetConstGlobal)))

{

return TMapKey\_GetPropertyId(scriptContext, propertyMap->GetKeyAt(index));

}

else

{

return Constants::NoProperty;

}

}

template <typename TPropertyIndex, typename TMapKey, bool IsNotExtensibleSupported>

BOOL SimpleDictionaryTypeHandlerBase<TPropertyIndex, TMapKey, IsNotExtensibleSupported>::FindNextProperty(ScriptContext\* scriptContext, PropertyIndex& index, JavascriptString\*\* propertyStringName,

PropertyId\* propertyId, PropertyAttributes\* attributes, Type\* type, DynamicType \*typeToEnumerate, bool requireEnumerable, bool enumSymbols)

{

Assert(propertyStringName);

Assert(propertyId);

Assert(type);

Assert(typeToEnumerate);

if(type == typeToEnumerate)

{

for( ; index < propertyMap->Count(); ++index )

{

SimpleDictionaryPropertyDescriptor<TPropertyIndex> descriptor(propertyMap->GetValueAt(index));

if( !(descriptor.Attributes & (PropertyDeleted | PropertyLetConstGlobal)) && (!requireEnumerable || (descriptor.Attributes & PropertyEnumerable)))

{

TMapKey key = propertyMap->GetKeyAt(index);

// Skip this property if it is a symbol and we are not including symbol properties

if (!enumSymbols && TMapKey\_IsSymbol(key, scriptContext))

{

continue;

}

if (attributes != nullptr)

{

\*attributes = descriptor.Attributes;

}

\*propertyId = TMapKey\_GetPropertyId(scriptContext, key);

PropertyString\* propertyString = type->GetScriptContext()->GetPropertyString(\*propertyId);

\*propertyStringName = propertyString;

if (descriptor.Attributes & PropertyWritable)

{

uint16 inlineOrAuxSlotIndex;

bool isInlineSlot;

PropertyIndexToInlineOrAuxSlotIndex(descriptor.propertyIndex, &inlineOrAuxSlotIndex, &isInlineSlot);

propertyString->UpdateCache(type, inlineOrAuxSlotIndex, isInlineSlot, descriptor.isInitialized && !descriptor.isFixed);

}

else

{

#ifdef DEBUG

PropertyCache const\* cache = propertyString->GetPropertyCache();

Assert(!cache || cache->type != type);

#endif

}

return TRUE;

}

}

return FALSE;

}

// Need to enumerate a different type than the current one. This is because type snapshot enumerate is enabled and the

// object's type changed since enumeration began, so need to enumerate properties of the initial type.

DynamicTypeHandler \*const typeHandlerToEnumerate = typeToEnumerate->GetTypeHandler();

for(

;

typeHandlerToEnumerate->FindNextProperty(

scriptContext,

index,

propertyStringName,

propertyId,

attributes,

typeToEnumerate,

typeToEnumerate,

requireEnumerable,

enumSymbols);

++index)

{

SimpleDictionaryPropertyDescriptor<TPropertyIndex> descriptor;

bool hasValue = false;

if (\*propertyId != Constants::NoProperty)

{

PropertyRecord const\* propertyRecord = type->GetScriptContext()->GetPropertyName(\*propertyId);

AssertMsg(enumSymbols || !propertyRecord->IsSymbol(),

"typeHandlerToEnumerate->FindNextProperty call above should not have returned us a symbol if we are not enumerating symbols");

hasValue = propertyMap->TryGetValue(propertyRecord, &descriptor);

}

else if (\*propertyStringName != nullptr)

{

hasValue = propertyMap->TryGetValue(\*propertyStringName, &descriptor);

}

if (hasValue &&

!(descriptor.Attributes & (PropertyDeleted | PropertyLetConstGlobal)) &&

(!requireEnumerable || descriptor.Attributes & PropertyEnumerable))

{

if (attributes != nullptr)

{

\*attributes = descriptor.Attributes;

}

if(descriptor.Attributes & PropertyWritable)

{

uint16 inlineOrAuxSlotIndex;

bool isInlineSlot;

PropertyIndexToInlineOrAuxSlotIndex(descriptor.propertyIndex, &inlineOrAuxSlotIndex, &isInlineSlot);

if (VirtualTableInfo<PropertyString>::HasVirtualTable(\*propertyStringName))

{

PropertyString\* propertyString = (PropertyString\*)(\*propertyStringName);

propertyString->UpdateCache(type, inlineOrAuxSlotIndex, isInlineSlot, descriptor.isInitialized && !descriptor.isFixed);

}

}

else

{

#ifdef DEBUG

if (VirtualTableInfo<PropertyString>::HasVirtualTable(\*propertyStringName))

{

PropertyString\* propertyString = (PropertyString\*)(\*propertyStringName);

PropertyCache const\* cache = propertyString->GetPropertyCache();

Assert(!cache || cache->type != type);

}

#endif

}

return TRUE;

}

}

return FALSE;

}

// Note on template specializations:

// C++ doesn't allow us to specify partially specialized template member function and requires all parameters,

// like this: template<bool B> PropertyIndex SimpleDictionaryTypeHandlerBase<PropertyIndex, B>::GetPropertyIndex().

// Since we don't care about the boolean in this template method, just delegate to the other function.

#define DefineUnusedSpecialization\_FindNextProperty\_BigPropertyIndex(T, S) \

template <> BOOL SimpleDictionaryTypeHandlerBase<BigPropertyIndex, T, S>::FindNextProperty(ScriptContext\* scriptContext, PropertyIndex& index, JavascriptString\*\* propertyString, PropertyId\* propertyId, PropertyAttributes\* attributes, Type\* type, DynamicType \*typeToEnumerate, bool requireEnumerable, bool enumSymbols) { Throw::InternalError(); }

DefineUnusedSpecialization\_FindNextProperty\_BigPropertyIndex(const PropertyRecord\*, false)

DefineUnusedSpecialization\_FindNextProperty\_BigPropertyIndex(const PropertyRecord\*, true)

DefineUnusedSpecialization\_FindNextProperty\_BigPropertyIndex(JavascriptString\*, false)

DefineUnusedSpecialization\_FindNextProperty\_BigPropertyIndex(JavascriptString\*, true)

#undef DefineUnusedSpecialization\_FindNextProperty\_BigPropertyIndex

template <typename TPropertyIndex, typename TMapKey, bool IsNotExtensibleSupported>

BOOL SimpleDictionaryTypeHandlerBase<TPropertyIndex, TMapKey, IsNotExtensibleSupported>::FindNextProperty(ScriptContext\* scriptContext, BigPropertyIndex& index, JavascriptString\*\* propertyString,

PropertyId\* propertyId, PropertyAttributes\* attributes, Type\* type, DynamicType \*typeToEnumerate, bool requireEnumerable, bool enumSymbols)

{

PropertyIndex local = (PropertyIndex)index;

Assert(index <= Constants::UShortMaxValue || index == Constants::NoBigSlot);

BOOL result = this->FindNextProperty(scriptContext, local, propertyString, propertyId, attributes, type, typeToEnumerate, requireEnumerable, enumSymbols);

index = local;

return result;

}

template <typename TPropertyIndex, typename TMapKey, bool IsNotExtensibleSupported>

\_\_inline BOOL SimpleDictionaryTypeHandlerBase<TPropertyIndex, TMapKey, IsNotExtensibleSupported>::FindNextProperty\_BigPropertyIndex(ScriptContext\* scriptContext, TPropertyIndex& index,

JavascriptString\*\* propertyStringName, PropertyId\* propertyId, PropertyAttributes\* attributes, Type\* type, DynamicType \*typeToEnumerate, bool requireEnumerable, bool enumSymbols)

{

Assert(propertyStringName);

Assert(propertyId);

Assert(type);

Assert(typeToEnumerate);

if(type == typeToEnumerate)

{

for( ; index < propertyMap->Count(); ++index )

{

SimpleDictionaryPropertyDescriptor<TPropertyIndex> descriptor(propertyMap->GetValueAt(index));

if( !(descriptor.Attributes & (PropertyDeleted | PropertyLetConstGlobal)) && (!requireEnumerable || (descriptor.Attributes & PropertyEnumerable)))

{

auto key = propertyMap->GetKeyAt(index);

// Skip this property if it is a symbol and we are not including symbol properties

if (!enumSymbols && TMapKey\_IsSymbol(key, scriptContext))

{

continue;

}

if (attributes != nullptr)

{

\*attributes = descriptor.Attributes;

}

\*propertyId = TMapKey\_GetPropertyId(scriptContext, key);

\*propertyStringName = type->GetScriptContext()->GetPropertyString(\*propertyId);

return TRUE;

}

}

return FALSE;

}

// Need to enumerate a different type than the current one. This is because type snapshot enumerate is enabled and the

// object's type changed since enumeration began, so need to enumerate properties of the initial type.

DynamicTypeHandler \*const typeHandlerToEnumerate = typeToEnumerate->GetTypeHandler();

for(

;

typeHandlerToEnumerate->FindNextProperty(

scriptContext,

index,

propertyStringName,

propertyId,

attributes,

typeToEnumerate,

typeToEnumerate,

requireEnumerable,

enumSymbols);

++index)

{

SimpleDictionaryPropertyDescriptor<TPropertyIndex> descriptor;

bool hasValue = false;

if (\*propertyId != Constants::NoProperty)

{

PropertyRecord const\* propertyRecord = type->GetScriptContext()->GetPropertyName(\*propertyId);

AssertMsg(enumSymbols || !propertyRecord->IsSymbol(),

"typeHandlerToEnumerate->FindNextProperty call above should not have returned us a symbol if we are not enumerating symbols");

hasValue = propertyMap->TryGetValue(propertyRecord, &descriptor);

}

else if (\*propertyStringName != nullptr)

{

hasValue = propertyMap->TryGetValue(\*propertyStringName, &descriptor);

}

if (hasValue &&

!(descriptor.Attributes & (PropertyDeleted | PropertyLetConstGlobal)) &&

(!requireEnumerable || descriptor.Attributes & PropertyEnumerable))

{

if (attributes != nullptr)

{

\*attributes = descriptor.Attributes;

}

#ifdef DEBUG

if (VirtualTableInfo<PropertyString>::HasVirtualTable(\*propertyStringName))

{

PropertyCache const\* cache = ((PropertyString\*)(\*propertyStringName))->GetPropertyCache();

Assert(!cache || cache->type != type);

}

#endif

return TRUE;

}

}

return FALSE;

}

template <>

BOOL SimpleDictionaryTypeHandlerBase<BigPropertyIndex, const PropertyRecord\*, false>::FindNextProperty(ScriptContext\* scriptContext, BigPropertyIndex& index, JavascriptString\*\* propertyString,

PropertyId\* propertyId, PropertyAttributes\* attributes, Type\* type, DynamicType \*typeToEnumerate, bool requireEnumerable, bool enumSymbols)

{

return this->FindNextProperty\_BigPropertyIndex(scriptContext, index, propertyString, propertyId, attributes, type, typeToEnumerate, requireEnumerable, enumSymbols);

}

template <>

BOOL SimpleDictionaryTypeHandlerBase<BigPropertyIndex, const PropertyRecord\*, true>::FindNextProperty(ScriptContext\* scriptContext, BigPropertyIndex& index, JavascriptString\*\* propertyString,

PropertyId\* propertyId, PropertyAttributes\* attributes, Type\* type, DynamicType \*typeToEnumerate, bool requireEnumerable, bool enumSymbols)

{

return this->FindNextProperty\_BigPropertyIndex(scriptContext, index, propertyString, propertyId, attributes, type, typeToEnumerate, requireEnumerable, enumSymbols);

}

template <>

BOOL SimpleDictionaryTypeHandlerBase<BigPropertyIndex, JavascriptString\*, false>::FindNextProperty(ScriptContext\* scriptContext, BigPropertyIndex& index, JavascriptString\*\* propertyString,

PropertyId\* propertyId, PropertyAttributes\* attributes, Type\* type, DynamicType \*typeToEnumerate, bool requireEnumerable, bool enumSymbols)

{

return this->FindNextProperty\_BigPropertyIndex(scriptContext, index, propertyString, propertyId, attributes, type, typeToEnumerate, requireEnumerable, enumSymbols);

}

template <>

BOOL SimpleDictionaryTypeHandlerBase<BigPropertyIndex, JavascriptString\*, true>::FindNextProperty(ScriptContext\* scriptContext, BigPropertyIndex& index, JavascriptString\*\* propertyString,

PropertyId\* propertyId, PropertyAttributes\* attributes, Type\* type, DynamicType \*typeToEnumerate, bool requireEnumerable, bool enumSymbols)

{

return this->FindNextProperty\_BigPropertyIndex(scriptContext, index, propertyString, propertyId, attributes, type, typeToEnumerate, requireEnumerable, enumSymbols);

}

template <typename TPropertyIndex>

\_\_inline PropertyIndex DisallowBigPropertyIndex(TPropertyIndex index)

{

if (index <= Constants::PropertyIndexMax)

{

return (PropertyIndex)index;

}

return Constants::NoSlot;

}

template <typename TPropertyIndex, typename TMapKey, bool IsNotExtensibleSupported>

template <bool allowLetConstGlobal>

\_\_inline PropertyIndex SimpleDictionaryTypeHandlerBase<TPropertyIndex, TMapKey, IsNotExtensibleSupported>::GetPropertyIndex\_Internal(const PropertyRecord\* propertyRecord)

{

SimpleDictionaryPropertyDescriptor<TPropertyIndex>\* descriptor;

if (propertyMap->TryGetReference(propertyRecord, &descriptor) && !(descriptor->Attributes & (PropertyDeleted | (!allowLetConstGlobal ? PropertyLetConstGlobal : 0))))

{

return DisallowBigPropertyIndex(descriptor->propertyIndex);

}

return Constants::NoSlot;

}

template <typename TPropertyIndex, typename TMapKey, bool IsNotExtensibleSupported>

PropertyIndex SimpleDictionaryTypeHandlerBase<TPropertyIndex, TMapKey, IsNotExtensibleSupported>::GetPropertyIndex(const PropertyRecord\* propertyRecord)

{

return this->GetPropertyIndex\_Internal<false>(propertyRecord);

}

template <typename TPropertyIndex, typename TMapKey, bool IsNotExtensibleSupported>

PropertyIndex SimpleDictionaryTypeHandlerBase<TPropertyIndex, TMapKey, IsNotExtensibleSupported>::GetRootPropertyIndex(const PropertyRecord\* propertyRecord)

{

return this->GetPropertyIndex\_Internal<true>(propertyRecord);

}

template <typename TPropertyIndex, typename TMapKey, bool IsNotExtensibleSupported>

bool SimpleDictionaryTypeHandlerBase<TPropertyIndex, TMapKey, IsNotExtensibleSupported>::GetPropertyEquivalenceInfo(PropertyRecord const\* propertyRecord, PropertyEquivalenceInfo& info)

{

SimpleDictionaryPropertyDescriptor<TPropertyIndex>\* descriptor;

if (propertyMap->TryGetReference(propertyRecord, &descriptor) && !(descriptor->Attributes & PropertyDeleted))

{

Js::PropertyIndex absSlotIndex = DisallowBigPropertyIndex(descriptor->propertyIndex);

info.slotIndex = AdjustSlotIndexForInlineSlots(absSlotIndex);

info.isAuxSlot = absSlotIndex > GetInlineSlotCapacity();

info.isWritable = !!(descriptor->Attributes & PropertyWritable);

}

else

{

info.slotIndex = Constants::NoSlot;

info.isAuxSlot = false;

info.isWritable = false;

}

return info.slotIndex != Constants::NoSlot;

}

template <typename TPropertyIndex, typename TMapKey, bool IsNotExtensibleSupported>

bool SimpleDictionaryTypeHandlerBase<TPropertyIndex, TMapKey, IsNotExtensibleSupported>::IsObjTypeSpecEquivalent(const Type\* type, const TypeEquivalenceRecord& record, uint& failedPropertyIndex)

{

uint propertyCount = record.propertyCount;

Js::EquivalentPropertyEntry\* properties = record.properties;

for (uint pi = 0; pi < propertyCount; pi++)

{

const EquivalentPropertyEntry\* refInfo = &properties[pi];

if (!this->IsObjTypeSpecEquivalentImpl<false>(type, refInfo))

{

failedPropertyIndex = pi;

return false;

}

}

return true;

}

template <typename TPropertyIndex, typename TMapKey, bool IsNotExtensibleSupported>

bool SimpleDictionaryTypeHandlerBase<TPropertyIndex, TMapKey, IsNotExtensibleSupported>::IsObjTypeSpecEquivalent(const Type\* type, const EquivalentPropertyEntry \*entry)

{

return this->IsObjTypeSpecEquivalentImpl<true>(type, entry);

}

template <typename TPropertyIndex, typename TMapKey, bool IsNotExtensibleSupported>

template <bool doLock>

bool SimpleDictionaryTypeHandlerBase<TPropertyIndex, TMapKey, IsNotExtensibleSupported>::IsObjTypeSpecEquivalentImpl(const Type\* type, const EquivalentPropertyEntry \*entry)

{

TPropertyIndex absSlotIndex = Constants::NoSlot;

PropertyIndex relSlotIndex = Constants::NoSlot;

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

const PropertyRecord\* propertyRecord =

doLock ? scriptContext->GetPropertyNameLocked(entry->propertyId) : scriptContext->GetPropertyName(entry->propertyId);

SimpleDictionaryPropertyDescriptor<TPropertyIndex>\* descriptor;

if (this->propertyMap->TryGetReference(propertyRecord, &descriptor) && !(descriptor->Attributes & PropertyDeleted))

{

absSlotIndex = descriptor->propertyIndex;

if (absSlotIndex <= Constants::PropertyIndexMax)

{

relSlotIndex = AdjustValidSlotIndexForInlineSlots(static\_cast<PropertyIndex>(absSlotIndex));

}

}

if (relSlotIndex != Constants::NoSlot)

{

if (relSlotIndex != entry->slotIndex || ((absSlotIndex >= GetInlineSlotCapacity()) != entry->isAuxSlot))

{

return false;

}

if (entry->mustBeWritable && (!(descriptor->Attributes & PropertyWritable) || !descriptor->isInitialized || descriptor->isFixed))

{

return false;

}

}

else

{

if (entry->slotIndex != Constants::NoSlot || entry->mustBeWritable)

{

return false;

}

}

return true;

}

// The following template specialization is required in order to provide an implementation of

// Add for the linker to find that TypePathHandler uses. The following definition should have sufficed.

template<>

template<>

void SimpleDictionaryTypeHandlerBase<PropertyIndex, const PropertyRecord\*, false>::Add(

const PropertyRecord\* propertyRecord,

PropertyAttributes attributes,

ScriptContext\* const scriptContext)

{

Add(propertyRecord, attributes, true, false, false, scriptContext);

}

template <typename TPropertyIndex, typename TMapKey, bool IsNotExtensibleSupported>

template <typename TPropertyKey>

void SimpleDictionaryTypeHandlerBase<TPropertyIndex, TMapKey, IsNotExtensibleSupported>::Add(

TPropertyKey propertyKey,

PropertyAttributes attributes,

ScriptContext \*const scriptContext)

{

Add(propertyKey, attributes, true, false, false, scriptContext);

}

template <typename TPropertyIndex, typename TMapKey, bool IsNotExtensibleSupported>

template <typename TPropertyKey>

void SimpleDictionaryTypeHandlerBase<TPropertyIndex, TMapKey, IsNotExtensibleSupported>::Add(

TPropertyKey propertyKey,

PropertyAttributes attributes,

bool isInitialized, bool isFixed, bool usedAsFixed,

ScriptContext \*const scriptContext)

{

Assert(this->GetSlotCapacity() <= MaxPropertyIndexSize); // slotCapacity should never exceed MaxPropertyIndexSize

Assert(nextPropertyIndex < this->GetSlotCapacity()); // nextPropertyIndex must be ready

Add(nextPropertyIndex++, propertyKey, attributes, isInitialized, isFixed, usedAsFixed, scriptContext);

}

template <typename TPropertyIndex, typename TMapKey, bool IsNotExtensibleSupported>

template <typename TPropertyKey>

void SimpleDictionaryTypeHandlerBase<TPropertyIndex, TMapKey, IsNotExtensibleSupported>::Add(

TPropertyIndex propertyIndex,

TPropertyKey propertyKey,

PropertyAttributes attributes,

ScriptContext \*const scriptContext)

{

Add(propertyIndex, propertyKey, attributes, true, false, false, scriptContext);

}

template <typename TPropertyIndex, typename TMapKey, bool IsNotExtensibleSupported>

template <typename TPropertyKey>

void SimpleDictionaryTypeHandlerBase<TPropertyIndex, TMapKey, IsNotExtensibleSupported>::Add(

TPropertyIndex propertyIndex,

TPropertyKey propertyKey,

PropertyAttributes attributes,

bool isInitialized, bool isFixed, bool usedAsFixed,

ScriptContext \*const scriptContext)

{

//

// For a function with same named parameters,

// property id Constants::NoProperty will be passed for all the dups except the last one

// We need to allocate space for dups, but don't add those to map

if (propertyKey != NULL)

{

SimpleDictionaryPropertyDescriptor<TPropertyIndex> descriptor(propertyIndex, attributes);

Assert((!isFixed && !usedAsFixed) || (!TPropertyKey\_IsInternalPropertyId(propertyKey) && this->singletonInstance != nullptr));

if (TPropertyKey\_IsInternalPropertyId(propertyKey) || TMapKey\_IsSymbol(propertyKey, scriptContext))

{

Assert(!TMapKey\_IsJavascriptString<TMapKey>());

hasNamelessPropertyId = true;

}

descriptor.isInitialized = isInitialized;

descriptor.isFixed = isFixed;

descriptor.usedAsFixed = usedAsFixed;

propertyMap->Add(TMapKey\_ConvertKey<TMapKey>(scriptContext, propertyKey), descriptor);

}

if (!(attributes & PropertyWritable))

{

this->ClearHasOnlyWritableDataProperties();

if (GetFlags() & IsPrototypeFlag)

{

scriptContext->InvalidateStoreFieldCaches(TMapKey\_GetPropertyId(scriptContext, propertyKey));

scriptContext->GetLibrary()->NoPrototypeChainsAreEnsuredToHaveOnlyWritableDataProperties();

}

}

}

template <typename TPropertyIndex, typename TMapKey, bool IsNotExtensibleSupported>

BOOL SimpleDictionaryTypeHandlerBase<TPropertyIndex, TMapKey, IsNotExtensibleSupported>::HasProperty(DynamicObject\* instance, PropertyId propertyId, bool \*noRedecl)

{

return HasProperty\_Internal<false>(instance, propertyId, noRedecl, nullptr);

}

template <typename TPropertyIndex, typename TMapKey, bool IsNotExtensibleSupported>

BOOL SimpleDictionaryTypeHandlerBase<TPropertyIndex, TMapKey, IsNotExtensibleSupported>::HasRootProperty(DynamicObject\* instance, PropertyId propertyId, bool \*noRedecl, bool \*pDeclaredProperty)

{

return HasProperty\_Internal<true>(instance, propertyId, noRedecl, pDeclaredProperty);

}

template <typename TPropertyIndex, typename TMapKey, bool IsNotExtensibleSupported>

template <bool allowLetConstGlobal>

BOOL SimpleDictionaryTypeHandlerBase<TPropertyIndex, TMapKey, IsNotExtensibleSupported>::HasProperty\_Internal(DynamicObject\* instance, PropertyId propertyId, bool \*noRedecl, bool \*pDeclaredProperty)

{

// HasProperty is called with NoProperty in JavascriptDispatch.cpp to for undeferral of the

// deferred type system that DOM objects use. Allow NoProperty for this reason, but only

// here in HasProperty.

if (propertyId == Constants::NoProperty)

{

return false;

}

SimpleDictionaryPropertyDescriptor<TPropertyIndex>\* descriptor;

Assert(propertyId != Constants::NoProperty);

PropertyRecord const\* propertyRecord = instance->GetScriptContext()->GetPropertyName(propertyId);

if (propertyMap->TryGetReference(propertyRecord, &descriptor))

{

if ((descriptor->Attributes & PropertyDeleted) || (!allowLetConstGlobal && !descriptor->HasNonLetConstGlobal()))

{

return false;

}

if (noRedecl && descriptor->Attributes & PropertyNoRedecl)

{

\*noRedecl = true;

}

if (pDeclaredProperty && descriptor->Attributes & (PropertyNoRedecl | PropertyDeclaredGlobal))

{

\*pDeclaredProperty = true;

}

return true;

}

// Check numeric propertyRecord only if objectArray available

if (instance->HasObjectArray() && propertyRecord->IsNumeric())

{

return SimpleDictionaryTypeHandlerBase<TPropertyIndex, TMapKey, IsNotExtensibleSupported>::HasItem(instance, propertyRecord->GetNumericValue());

}

return false;

}

template <typename TPropertyIndex, typename TMapKey, bool IsNotExtensibleSupported>

BOOL SimpleDictionaryTypeHandlerBase<TPropertyIndex, TMapKey, IsNotExtensibleSupported>::HasProperty(DynamicObject\* instance, JavascriptString\* propertyNameString)

{

AssertMsg(!PropertyRecord::IsPropertyNameNumeric(propertyNameString->GetString(), propertyNameString->GetLength()),

"Numeric property names should have been converted to uint or PropertyRecord\* before calling GetSetter");

JsUtil::CharacterBuffer<WCHAR> propertyName(propertyNameString->GetString(), propertyNameString->GetLength());

SimpleDictionaryPropertyDescriptor<TPropertyIndex>\* descriptor;

if (propertyMap->TryGetReference(propertyName, &descriptor))

{

if (descriptor->Attributes & (PropertyDeleted | PropertyLetConstGlobal))

{

return false;

}

return true;

}

return false;

}

template <typename TPropertyIndex, typename TMapKey, bool IsNotExtensibleSupported>

BOOL SimpleDictionaryTypeHandlerBase<TPropertyIndex, TMapKey, IsNotExtensibleSupported>::GetRootProperty(DynamicObject\* instance, Var originalInstance, PropertyId propertyId, Var\* value, PropertyValueInfo\* info, ScriptContext\* requestContext)

{

AssertMsg(RootObjectBase::Is(instance), "Instance must be a root object!");

return GetProperty\_Internal<true>(instance, originalInstance, propertyId, value, info, requestContext);

}

template <typename TPropertyIndex, typename TMapKey, bool IsNotExtensibleSupported>

BOOL SimpleDictionaryTypeHandlerBase<TPropertyIndex, TMapKey, IsNotExtensibleSupported>::GetProperty(DynamicObject\* instance, Var originalInstance, PropertyId propertyId, Var\* value, PropertyValueInfo\* info, ScriptContext\* requestContext)

{

return GetProperty\_Internal<false>(instance, originalInstance, propertyId, value, info, requestContext);

}

template <typename TPropertyIndex, typename TMapKey, bool IsNotExtensibleSupported>

template <bool allowLetConstGlobal>

BOOL SimpleDictionaryTypeHandlerBase<TPropertyIndex, TMapKey, IsNotExtensibleSupported>::GetProperty\_Internal(DynamicObject\* instance, Var originalInstance, PropertyId propertyId, Var\* value, PropertyValueInfo\* info, ScriptContext\* requestContext)

{

SimpleDictionaryPropertyDescriptor<TPropertyIndex>\* descriptor;

Assert(propertyId != Constants::NoProperty);

PropertyRecord const\* propertyRecord = instance->GetScriptContext()->GetPropertyName(propertyId);

if (propertyMap->TryGetReference(propertyRecord, &descriptor))

{

return GetPropertyFromDescriptor<allowLetConstGlobal>(instance, descriptor, value, info);

}

// Check numeric propertyRecord only if objectArray available

if (instance->HasObjectArray() && propertyRecord->IsNumeric())

{

return SimpleDictionaryTypeHandlerBase<TPropertyIndex, TMapKey, IsNotExtensibleSupported>::GetItem(instance, originalInstance, propertyRecord->GetNumericValue(), value, requestContext);

}

return false;

}

template <typename TPropertyIndex, typename TMapKey, bool IsNotExtensibleSupported>

BOOL SimpleDictionaryTypeHandlerBase<TPropertyIndex, TMapKey, IsNotExtensibleSupported>::GetProperty(DynamicObject\* instance, Var originalInstance, JavascriptString\* propertyNameString, Var\* value, PropertyValueInfo\* info, ScriptContext\* requestContext)

{

AssertMsg(!PropertyRecord::IsPropertyNameNumeric(propertyNameString->GetString(), propertyNameString->GetLength()),

"Numeric property names should have been converted to uint or PropertyRecord\* before calling GetSetter");

JsUtil::CharacterBuffer<WCHAR> propertyName(propertyNameString->GetString(), propertyNameString->GetLength());

SimpleDictionaryPropertyDescriptor<TPropertyIndex>\* descriptor;

if (propertyMap->TryGetReference(propertyName, &descriptor))

{

return GetPropertyFromDescriptor<false>(instance, descriptor, value, info);

}

return false;

}

template <typename TPropertyIndex, typename TMapKey, bool IsNotExtensibleSupported>

template <bool allowLetConstGlobal>

BOOL SimpleDictionaryTypeHandlerBase<TPropertyIndex, TMapKey, IsNotExtensibleSupported>::GetPropertyFromDescriptor(DynamicObject\* instance, SimpleDictionaryPropertyDescriptor<TPropertyIndex>\* descriptor, Var\* value, PropertyValueInfo\* info)

{

if (descriptor->Attributes & (PropertyDeleted | (allowLetConstGlobal ? 0 : PropertyLetConstGlobal)))

{

return false;

}

if (descriptor->propertyIndex != NoSlots)

{

\*value = instance->GetSlot(descriptor->propertyIndex);

SetPropertyValueInfo(info, instance, descriptor->propertyIndex, descriptor->Attributes);

if (!descriptor->isInitialized || descriptor->isFixed)

{

PropertyValueInfo::DisableStoreFieldCache(info);

}

}

else

{

\*value = instance->GetLibrary()->GetUndefined();

}

return true;

}

template <typename TPropertyIndex, typename TMapKey, bool IsNotExtensibleSupported>

BOOL SimpleDictionaryTypeHandlerBase<TPropertyIndex, TMapKey, IsNotExtensibleSupported>::SetProperty(DynamicObject\* instance, JavascriptString\* propertyNameString, Var value, PropertyOperationFlags flags, PropertyValueInfo\* info)

{

return SetProperty\_JavascriptString(instance, propertyNameString, value, flags, info, TemplateParameter::Box<TMapKey>());

}

template <typename TPropertyIndex, typename TMapKey, bool IsNotExtensibleSupported>

BOOL SimpleDictionaryTypeHandlerBase<TPropertyIndex, TMapKey, IsNotExtensibleSupported>::SetProperty\_JavascriptString(DynamicObject\* instance, JavascriptString\* propertyNameString, Var value, PropertyOperationFlags flags, PropertyValueInfo\* info, TemplateParameter::Box<const PropertyRecord\*>)

{

// Either the property exists in the dictionary, in which case a PropertyRecord for it exists,

// or we have to add it to the dictionary, in which case we need to get or create a PropertyRecord.

// Thus, just get or create one and call the PropertyId overload of SetProperty.

PropertyRecord const \* propertyRecord;

instance->GetScriptContext()->GetOrAddPropertyRecord(propertyNameString->GetString(), propertyNameString->GetLength(), &propertyRecord);

return SimpleDictionaryTypeHandlerBase<TPropertyIndex, TMapKey, IsNotExtensibleSupported>::SetProperty(instance, propertyRecord->GetPropertyId(), value, flags, info);

}

template <typename TPropertyIndex, typename TMapKey, bool IsNotExtensibleSupported>

BOOL SimpleDictionaryTypeHandlerBase<TPropertyIndex, TMapKey, IsNotExtensibleSupported>::SetProperty\_JavascriptString(DynamicObject\* instance, JavascriptString\* propertyNameString, Var value, PropertyOperationFlags flags, PropertyValueInfo\* info, TemplateParameter::Box<JavascriptString\*>)

{

AssertMsg(!PropertyRecord::IsPropertyNameNumeric(propertyNameString->GetString(), propertyNameString->GetLength()),

"Numeric property names should have been converted to uint or PropertyRecord\* before calling GetSetter");

JsUtil::CharacterBuffer<WCHAR> propertyName(propertyNameString->GetString(), propertyNameString->GetLength());

SimpleDictionaryPropertyDescriptor<TPropertyIndex>\* descriptor;

if (propertyMap->TryGetReference(propertyName, &descriptor))

{

return SetPropertyFromDescriptor<false>(instance, Constants::NoProperty, propertyNameString, descriptor, value, flags, info);

}

return this->AddProperty(instance, propertyNameString, value, PropertyDynamicTypeDefaults, info, flags, SideEffects\_Any);

}

#define DefineUnusedSpecialization\_SetProperty\_JavascriptString(T,S) \

template<> BOOL SimpleDictionaryTypeHandlerBase<T, const PropertyRecord\*, S>::SetProperty\_JavascriptString(DynamicObject\* instance, JavascriptString\* propertyNameString, Var value, PropertyOperationFlags flags, PropertyValueInfo\* info, TemplateParameter::Box<JavascriptString\*>) { Throw::InternalError(); }

DefineUnusedSpecialization\_SetProperty\_JavascriptString(PropertyIndex, true)

DefineUnusedSpecialization\_SetProperty\_JavascriptString(PropertyIndex, false)

DefineUnusedSpecialization\_SetProperty\_JavascriptString(BigPropertyIndex, true)

DefineUnusedSpecialization\_SetProperty\_JavascriptString(BigPropertyIndex, false)

#undef DefineUnusedSpecialization\_SetProperty\_JavascriptString

template <typename TPropertyIndex, typename TMapKey, bool IsNotExtensibleSupported>

template <bool allowLetConstGlobal>

BOOL SimpleDictionaryTypeHandlerBase<TPropertyIndex, TMapKey, IsNotExtensibleSupported>::SetProperty\_Internal(DynamicObject\* instance, PropertyId propertyId, Var value, PropertyOperationFlags flags, PropertyValueInfo\* info)

{

SimpleDictionaryPropertyDescriptor<TPropertyIndex>\* descriptor;

Assert(propertyId != Constants::NoProperty);

PropertyRecord const\* propertyRecord = instance->GetScriptContext()->GetPropertyName(propertyId);

// It can be the case that propertyRecord is a symbol property and it has the same string description as

// another property which is in the propertyMap. If we are in a string-keyed type handler, the propertyMap

// will find that normal property when we call TryGetReference with the symbol propertyRecord. However,

// we don't want to update that descriptor with value since the two properties are actually different.

// In fact, we can't store a symbol in a string-keyed type handler at all since the string description

// is not used for symbols. Instead, we want to skip searching for the descriptor if we are in a string-keyed

// type handler. When we call AddProperty with the symbol propertyRecord, it should convert us to a

// const PropertyRecord\* - keyed type handler anyway.

if (!(TMapKey\_IsJavascriptString<TMapKey>() && propertyRecord->IsSymbol())

&& propertyMap->TryGetReference(propertyRecord, &descriptor))

{

return SetPropertyFromDescriptor<allowLetConstGlobal>(instance, propertyId, propertyId, descriptor, value, flags, info);

}

// Always check numeric propertyId. This may create objectArray.

if (propertyRecord->IsNumeric())

{

return this->SetItem(instance, propertyRecord->GetNumericValue(), value, flags);

}

return this->AddProperty(instance, propertyRecord, value, PropertyDynamicTypeDefaults, info, flags, SideEffects\_Any);

}

template <typename TPropertyIndex, typename TMapKey, bool IsNotExtensibleSupported>

BOOL SimpleDictionaryTypeHandlerBase<TPropertyIndex, TMapKey, IsNotExtensibleSupported>::SetProperty(DynamicObject\* instance, PropertyId propertyId, Var value, PropertyOperationFlags flags, PropertyValueInfo\* info)

{

return SetProperty\_Internal<false>(instance, propertyId, value, flags, info);

}

template <typename TPropertyIndex, typename TMapKey, bool IsNotExtensibleSupported>

BOOL SimpleDictionaryTypeHandlerBase<TPropertyIndex, TMapKey, IsNotExtensibleSupported>::SetRootProperty(DynamicObject\* instance, PropertyId propertyId, Var value, PropertyOperationFlags flags, PropertyValueInfo\* info)

{

AssertMsg(RootObjectBase::Is(instance), "Instance must be a root object!");

return SetProperty\_Internal<true>(instance, propertyId, value, flags, info);

}

template <typename TPropertyIndex, typename TMapKey, bool IsNotExtensibleSupported>

template <bool allowLetConstGlobal, typename TPropertyKey>

BOOL SimpleDictionaryTypeHandlerBase<TPropertyIndex, TMapKey, IsNotExtensibleSupported>::SetPropertyFromDescriptor(DynamicObject\* instance, PropertyId propertyId, TPropertyKey propertyKey, SimpleDictionaryPropertyDescriptor<TPropertyIndex>\* descriptor, Var value, PropertyOperationFlags flags, PropertyValueInfo\* info)

{

Assert(instance);

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

bool throwIfNotExtensible = (flags & (PropertyOperation\_ThrowIfNotExtensible | PropertyOperation\_StrictMode)) != 0;

if (!allowLetConstGlobal && (descriptor->Attributes & PropertyLetConstGlobal))

{

// We have a shadowing case here. An undeclared global property

// is being added after a let/const was already declared.

//

// SimpleDictionaryTypeHandlerBase does not handle shadowed globals

// because we do not want to add another property index field to the

// property descriptors. Instead convert to DictionaryTypeHandler

// where it will reuse one of the getter/setter fields on its

// property descriptor type.

return

ConvertToDictionaryType(instance)

->SetProperty(

instance,

propertyId,

value,

flags,

info);

}

if (descriptor->Attributes & PropertyDeleted)

{

if(GetIsLocked())

{

return

ConvertToNonSharedSimpleDictionaryType(instance)->SetProperty(instance, propertyKey, value, flags, info);

}

if(isUnordered)

{

TPropertyIndex propertyIndex;

if(AsUnordered()->TryUndeleteProperty(instance, descriptor->propertyIndex, &propertyIndex))

{

Assert(PropertyRecordStringHashComparer<TMapKey>::Equals(propertyMap->GetKeyAt(propertyIndex), TMapKey\_OptionalConvertPropertyIdToPropertyRecord(scriptContext, propertyKey)));

descriptor = propertyMap->GetReferenceAt(propertyIndex);

}

}

if (IsNotExtensibleSupported)

{

bool isForce = (flags & PropertyOperation\_Force) != 0;

if (!isForce)

{

if (!this->VerifyIsExtensible(scriptContext, throwIfNotExtensible))

{

return FALSE;

}

}

}

if(SupportsSwitchingToUnordered(scriptContext))

{

--numDeletedProperties;

}

descriptor->Attributes = PropertyDynamicTypeDefaults;

instance->SetHasNoEnumerableProperties(false);

propertyId = TPropertyKey\_GetOptionalPropertyId(instance->GetScriptContext(), propertyKey);

if (propertyId != Constants::NoProperty)

{

scriptContext->InvalidateProtoCaches(propertyId);

}

descriptor->Attributes = PropertyDynamicTypeDefaults;

}

else if (!(descriptor->Attributes & PropertyWritable) && !(flags & PropertyOperation\_AllowUndeclInConsoleScope))

{

JavascriptError::ThrowCantAssignIfStrictMode(flags, scriptContext);

// Since we separate LdFld and StFld caches there is no point in caching for StFld with non-writable properties, except perhaps

// to prepopulate the type property cache (which we do share between LdFld and StFld), for potential future field loads. This

// would require additional handling in CacheOperators::CachePropertyWrite, such that for !info-IsWritable() we don't populate

// the local cache (that would be illegal), but still populate the type's property cache.

PropertyValueInfo::SetNoCache(info, instance);

return false;

}

if (descriptor->propertyIndex != NoSlots)

{

if ((descriptor->Attributes & PropertyNoRedecl) && !(flags & PropertyOperation\_AllowUndecl))

{

Assert(scriptContext->GetConfig()->IsLetAndConstEnabled());

if (scriptContext->IsUndeclBlockVar(instance->GetSlot(descriptor->propertyIndex)) && !(flags & PropertyOperation\_AllowUndeclInConsoleScope))

{

JavascriptError::ThrowReferenceError(scriptContext, JSERR\_UseBeforeDeclaration);

}

}

DynamicObject\* localSingletonInstance = this->singletonInstance != nullptr ? this->singletonInstance->Get() : nullptr;

Assert(this->singletonInstance == nullptr || localSingletonInstance == instance);

if (!descriptor->isInitialized)

{

if ((flags & PropertyOperation\_PreInit) == 0)

{

// Consider: It would be nice to assert the slot is actually null. However, we sometimes pre-initialize to undefined or even

// some other special illegal value (for let or const), currently == null.

descriptor->isInitialized = true;

if (localSingletonInstance == instance &&

!TPropertyKey\_IsInternalPropertyId(propertyKey) &&

(flags & (PropertyOperation\_NonFixedValue | PropertyOperation\_SpecialValue)) == 0)

{

Assert(!GetIsShared());

Assert(value != nullptr);

// We don't want fixed properties on external objects. See DynamicObject::ResetObject for more information.

Assert(!instance->IsExternal());

descriptor->isFixed = (JavascriptFunction::Is(value) ? ShouldFixMethodProperties() : (ShouldFixDataProperties() && CheckHeuristicsForFixedDataProps(instance, propertyId, value)));

}

}

}

else

{

InvalidateFixedField(TMapKey\_OptionalConvertPropertyIdToPropertyRecord(scriptContext, propertyKey), descriptor, instance->GetScriptContext());

}

SetSlotUnchecked(instance, descriptor->propertyIndex, value);

if (descriptor->isInitialized && !descriptor->isFixed)

{

SetPropertyValueInfo(info, instance, descriptor->propertyIndex, descriptor->Attributes);

}

else

{

PropertyValueInfo::SetNoCache(info, instance);

}

}

propertyId = TPropertyKey\_GetUpdateSideEffectPropertyId(propertyId, propertyKey);

if (propertyId != Constants::NoProperty)

{

SetPropertyUpdateSideEffect(instance, propertyId, value, SideEffects\_Any);

}

return true;

}

template <typename TPropertyIndex, typename TMapKey, bool IsNotExtensibleSupported>

DescriptorFlags SimpleDictionaryTypeHandlerBase<TPropertyIndex, TMapKey, IsNotExtensibleSupported>::GetSetter(DynamicObject\* instance, PropertyId propertyId, Var\* setterValue, PropertyValueInfo\* info, ScriptContext\* requestContext)

{

return GetSetter\_Internal<false>(instance, propertyId, setterValue, info, requestContext);

}

template <typename TPropertyIndex, typename TMapKey, bool IsNotExtensibleSupported>

DescriptorFlags SimpleDictionaryTypeHandlerBase<TPropertyIndex, TMapKey, IsNotExtensibleSupported>::GetRootSetter(DynamicObject\* instance, PropertyId propertyId, Var\* setterValue, PropertyValueInfo\* info, ScriptContext\* requestContext)

{

AssertMsg(RootObjectBase::Is(instance), "Instance must be a root object!");

return GetSetter\_Internal<true>(instance, propertyId, setterValue, info, requestContext);

}

template <typename TPropertyIndex, typename TMapKey, bool IsNotExtensibleSupported>

template <bool allowLetConstGlobal>

DescriptorFlags SimpleDictionaryTypeHandlerBase<TPropertyIndex, TMapKey, IsNotExtensibleSupported>::GetSetter\_Internal(DynamicObject\* instance, PropertyId propertyId, Var\* setterValue, PropertyValueInfo\* info, ScriptContext\* requestContext)

{

SimpleDictionaryPropertyDescriptor<TPropertyIndex>\* descriptor;

Assert(propertyId != Constants::NoProperty);

PropertyRecord const\* propertyRecord = instance->GetScriptContext()->GetPropertyName(propertyId);

if (propertyMap->TryGetReference(propertyRecord, &descriptor))

{

return GetSetterFromDescriptor<allowLetConstGlobal>(descriptor);

}

if (propertyRecord->IsNumeric())

{

return SimpleDictionaryTypeHandlerBase<TPropertyIndex, TMapKey, IsNotExtensibleSupported>::GetItemSetter(instance, propertyRecord->GetNumericValue(), setterValue, requestContext);

}

return None;

}

template <typename TPropertyIndex, typename TMapKey, bool IsNotExtensibleSupported>

DescriptorFlags SimpleDictionaryTypeHandlerBase<TPropertyIndex, TMapKey, IsNotExtensibleSupported>::GetSetter(DynamicObject\* instance, JavascriptString\* propertyNameString, Var\* setterValue, PropertyValueInfo\* info, ScriptContext\* requestContext)

{

AssertMsg(!PropertyRecord::IsPropertyNameNumeric(propertyNameString->GetString(), propertyNameString->GetLength()),

"Numeric property names should have been converted to uint or PropertyRecord\* before calling GetSetter");

JsUtil::CharacterBuffer<WCHAR> propertyName(propertyNameString->GetString(), propertyNameString->GetLength());

SimpleDictionaryPropertyDescriptor<TPropertyIndex>\* descriptor;

if (propertyMap->TryGetReference(propertyName, &descriptor))

{

return GetSetterFromDescriptor<false>(descriptor);

}

return None;

}

template <typename TPropertyIndex, typename TMapKey, bool IsNotExtensibleSupported>

template <bool allowLetConstGlobal>

DescriptorFlags SimpleDictionaryTypeHandlerBase<TPropertyIndex, TMapKey, IsNotExtensibleSupported>::GetSetterFromDescriptor(SimpleDictionaryPropertyDescriptor<TPropertyIndex>\* descriptor)

{

if (descriptor->Attributes & (PropertyDeleted | (!allowLetConstGlobal ? PropertyLetConstGlobal : 0)))

{

return None;

}

if (descriptor->Attributes & PropertyLetConstGlobal)

{

if (descriptor->Attributes & PropertyConst)

{

return (DescriptorFlags)(Const|Data);

}

Assert(descriptor->Attributes & PropertyLet);

return WritableData;

}

if (descriptor->Attributes & PropertyWritable)

{

return WritableData;

}

if (descriptor->Attributes & PropertyConst)

{

return (DescriptorFlags)(Const|Data);

}

return Data;

}

template <typename TPropertyIndex, typename TMapKey, bool IsNotExtensibleSupported>

BOOL SimpleDictionaryTypeHandlerBase<TPropertyIndex, TMapKey, IsNotExtensibleSupported>::DeleteProperty(DynamicObject\* instance, PropertyId propertyId, PropertyOperationFlags propertyOperationFlags)

{

return DeleteProperty\_Internal<false>(instance, propertyId, propertyOperationFlags);

}

template <typename TPropertyIndex, typename TMapKey, bool IsNotExtensibleSupported>

BOOL SimpleDictionaryTypeHandlerBase<TPropertyIndex, TMapKey, IsNotExtensibleSupported>::DeleteRootProperty(DynamicObject\* instance, PropertyId propertyId, PropertyOperationFlags propertyOperationFlags)

{

AssertMsg(RootObjectBase::Is(instance), "Instance must be a root object!");

return DeleteProperty\_Internal<true>(instance, propertyId, propertyOperationFlags);

}

template <typename TPropertyIndex, typename TMapKey, bool IsNotExtensibleSupported>

template <bool allowLetConstGlobal>

BOOL SimpleDictionaryTypeHandlerBase<TPropertyIndex, TMapKey, IsNotExtensibleSupported>::DeleteProperty\_Internal(DynamicObject\* instance, PropertyId propertyId, PropertyOperationFlags propertyOperationFlags)

{

if(!GetIsLocked())

{

#ifdef ENABLE\_DEBUG\_CONFIG\_OPTIONS

if (CONFIG\_FLAG(ForceStringKeyedSimpleDictionaryTypeHandler) &&

!TMapKey\_IsJavascriptString<TMapKey>() &&

!isUnordered && !hasNamelessPropertyId)

{

return ConvertToSimpleDictionaryUnorderedTypeHandler<TPropertyIndex, JavascriptString\*, IsNotExtensibleSupported>(instance)

->DeleteProperty(instance, propertyId, propertyOperationFlags);

}

#endif

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

SimpleDictionaryPropertyDescriptor<TPropertyIndex>\* descriptor;

Assert(propertyId != Constants::NoProperty);

PropertyRecord const\* propertyRecord = instance->GetScriptContext()->GetPropertyName(propertyId);

if (propertyMap->TryGetReference(propertyRecord, &descriptor))

{

if (descriptor->Attributes & (PropertyDeleted | (!allowLetConstGlobal ? PropertyLetConstGlobal : 0)))

{

// If PropertyLetConstGlobal is present then we have a let/const and no global property,

// since SimpleDictionaryTypeHandler does not support shadowing which means it can only

// have one or the other. Therefore return true for no property found if allowLetConstGlobal

// is false. If allowLetConstGlobal is true we will enter the else if branch below and

// return false since let/const variables cannot be deleted.

return true;

}

else if (!(descriptor->Attributes & PropertyConfigurable) ||

(allowLetConstGlobal && (descriptor->Attributes & PropertyLetConstGlobal)))

{

JavascriptError::ThrowCantDeleteIfStrictMode(propertyOperationFlags, scriptContext, propertyRecord->GetBuffer());

return false;

}

Assert(!(descriptor->Attributes & PropertyLetConstGlobal));

Var undefined = scriptContext->GetLibrary()->GetUndefined();

if (descriptor->propertyIndex != NoSlots)

{

if (SupportsSwitchingToUnordered(scriptContext))

{

++numDeletedProperties;

if (numDeletedProperties >= CONFIG\_FLAG(DeletedPropertyReuseThreshold))

{

// This type handler is being used as a hashtable. Start reusing deleted property indexes for new

// property IDs. After this, enumeration order is nondeterministic.

// Also use JavascriptString\* as the property map key so that PropertyRecords can be avoided

// entirely where possible.

// Check if prototype chain has enumerable properties, according to logic used in

// ForInObjectEnumerator::Initialize(). If there are enumerable properties in the

// prototype chain, then enumerating this object's properties will require keeping

// track of properties so that shadowed properties are not included, but doing so

// currently requires converting the property to a PropertyRecord with a PropertyId

// for use in a bit vector that tracks shadowing. To avoid having a string keyed

// type handler hit this, only convert to the string keyed type handler if the

// prototype chain does not have enumerable properties.

bool fConvertToStringKeyedHandler =

!hasNamelessPropertyId &&

ForInObjectEnumerator::GetFirstPrototypeWithEnumerableProperties(instance) == nullptr;

if (fConvertToStringKeyedHandler)

{

PHASE\_PRINT\_TESTTRACE1(Js::TypeHandlerTransitionPhase, L"Transitioning to string keyed SimpleDictionaryUnorderedTypeHandler\n");

// if TMapKey is already JavascriptString\* we will not get here because we'd

// already be unordered and SupportsSwitchingToUnordered would have returned false

return ConvertToSimpleDictionaryUnorderedTypeHandler<TPropertyIndex, JavascriptString\*, IsNotExtensibleSupported>(instance)

->DeleteProperty(instance, propertyId, propertyOperationFlags);

}

else

{

PHASE\_PRINT\_TESTTRACE1(Js::TypeHandlerTransitionPhase, L"Transitioning to PropertyRecord keyed SimpleDictionaryUnorderedTypeHandler\n");

return ConvertToSimpleDictionaryUnorderedTypeHandler<TPropertyIndex, TMapKey, IsNotExtensibleSupported>(instance)

->DeleteProperty(instance, propertyId, propertyOperationFlags);

}

}

}

Assert(this->singletonInstance == nullptr || instance == this->singletonInstance->Get());

InvalidateFixedField(propertyRecord, descriptor, instance->GetScriptContext());

if (this->GetFlags() & IsPrototypeFlag)

{

scriptContext->InvalidateProtoCaches(propertyId);

}

// If this is an unordered type handler, register the deleted property index so that it can be reused for

// other property IDs added later

if(!isUnordered ||

!AsUnordered()->TryRegisterDeletedPropertyIndex(instance, descriptor->propertyIndex))

{

SetSlotUnchecked(instance, descriptor->propertyIndex, undefined);

}

}

descriptor->Attributes = PropertyDeletedDefaults;

// Change the type so as we can invalidate the cache in fast path jit

instance->ChangeType();

SetPropertyUpdateSideEffect(instance, propertyId, nullptr, SideEffects\_Any);

return true;

}

// Check for a numeric propertyRecord only if objectArray available

if (instance->HasObjectArray() && propertyRecord->IsNumeric())

{

return SimpleDictionaryTypeHandlerBase<TPropertyIndex, TMapKey, IsNotExtensibleSupported>::DeleteItem(instance, propertyRecord->IsNumeric(), propertyOperationFlags);

}

}

else

{

return ConvertToNonSharedSimpleDictionaryType(instance)->DeleteProperty\_Internal<allowLetConstGlobal>(instance, propertyId, propertyOperationFlags);

}

return true;

}

template <typename TPropertyIndex, typename TMapKey, bool IsNotExtensibleSupported>

BOOL SimpleDictionaryTypeHandlerBase<TPropertyIndex, TMapKey, IsNotExtensibleSupported>::IsFixedProperty(const DynamicObject\* instance, PropertyId propertyId)

{

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

SimpleDictionaryPropertyDescriptor<TPropertyIndex>\* descriptor;

Assert(propertyId != Constants::NoProperty);

PropertyRecord const\* propertyRecord = scriptContext->GetPropertyName(propertyId);

if (propertyMap->TryGetReference(propertyRecord, &descriptor))

{

return descriptor->isFixed;

}

else

{

AssertMsg(false, "Asking about a property this type handler doesn't know about?");

return false;

}

}

template <typename TPropertyIndex, typename TMapKey, bool IsNotExtensibleSupported>

BOOL SimpleDictionaryTypeHandlerBase<TPropertyIndex, TMapKey, IsNotExtensibleSupported>::IsEnumerable(DynamicObject\* instance, PropertyId propertyId)

{

SimpleDictionaryPropertyDescriptor<TPropertyIndex>\* descriptor;

Assert(propertyId != Constants::NoProperty);

PropertyRecord const\* propertyRecord = instance->GetScriptContext()->GetPropertyName(propertyId);

if (propertyMap->TryGetReference(propertyRecord, &descriptor))

{

if (descriptor->Attributes & PropertyLetConstGlobal)

{

return true;

}

return descriptor->Attributes & PropertyEnumerable;

}

return true;

}

template <typename TPropertyIndex, typename TMapKey, bool IsNotExtensibleSupported>

BOOL SimpleDictionaryTypeHandlerBase<TPropertyIndex, TMapKey, IsNotExtensibleSupported>::IsWritable(DynamicObject\* instance, PropertyId propertyId)

{

SimpleDictionaryPropertyDescriptor<TPropertyIndex>\* descriptor;

Assert(propertyId != Constants::NoProperty);

PropertyRecord const\* propertyRecord = instance->GetScriptContext()->GetPropertyName(propertyId);

if (propertyMap->TryGetReference(propertyRecord, &descriptor))

{

if (descriptor->Attributes & PropertyLetConstGlobal)

{

return !(descriptor->Attributes & PropertyConst);

}

return descriptor->Attributes & PropertyWritable;

}

return true;

}

template <typename TPropertyIndex, typename TMapKey, bool IsNotExtensibleSupported>

BOOL SimpleDictionaryTypeHandlerBase<TPropertyIndex, TMapKey, IsNotExtensibleSupported>::IsConfigurable(DynamicObject\* instance, PropertyId propertyId)

{

SimpleDictionaryPropertyDescriptor<TPropertyIndex>\* descriptor;

Assert(propertyId != Constants::NoProperty);

PropertyRecord const\* propertyRecord = instance->GetScriptContext()->GetPropertyName(propertyId);

if (propertyMap->TryGetReference(propertyRecord, &descriptor))

{

if (descriptor->Attributes & PropertyLetConstGlobal)

{

AssertMsg(RootObjectBase::Is(instance), "Instance must be a root object!");

return true;

}

return descriptor->Attributes & PropertyConfigurable;

}

return true;

}

//

// Set an attribute bit. Return true if change is made.

//

template <typename TPropertyIndex, typename TMapKey, bool IsNotExtensibleSupported>

BOOL SimpleDictionaryTypeHandlerBase<TPropertyIndex, TMapKey, IsNotExtensibleSupported>::SetAttribute(DynamicObject\* instance, SimpleDictionaryPropertyDescriptor<TPropertyIndex>\* descriptor, PropertyAttributes attribute)

{

Assert(!(descriptor->Attributes & PropertyLetConstGlobal));

if (descriptor->Attributes & PropertyDeleted)

{

return false;

}

PropertyAttributes attributes = descriptor->Attributes;

attributes |= attribute;

if (attributes == descriptor->Attributes)

{

return false;

}

if (GetIsLocked())

{

PropertyAttributes oldAttributes = descriptor->Attributes;

descriptor->Attributes = attributes;

ConvertToNonSharedSimpleDictionaryType(instance); // This changes TypeHandler, but non-necessarily Type.

descriptor->Attributes = oldAttributes;

}

else

{

descriptor->Attributes = attributes;

}

return true;

}

//

// Clear an attribute bit. Return true if change is made.

//

template <typename TPropertyIndex, typename TMapKey, bool IsNotExtensibleSupported>

BOOL SimpleDictionaryTypeHandlerBase<TPropertyIndex, TMapKey, IsNotExtensibleSupported>::ClearAttribute(DynamicObject\* instance, SimpleDictionaryPropertyDescriptor<TPropertyIndex>\* descriptor, PropertyAttributes attribute)

{

Assert(!(descriptor->Attributes & PropertyLetConstGlobal));

if (descriptor->Attributes & PropertyDeleted)

{

return false;

}

PropertyAttributes attributes = descriptor->Attributes;

attributes &= ~attribute;

if (attributes == descriptor->Attributes)

{

return false;

}

if (GetIsLocked())

{

PropertyAttributes oldAttributes = descriptor->Attributes;

descriptor->Attributes = attributes;

ConvertToNonSharedSimpleDictionaryType(instance); // This changes TypeHandler, but non-necessarily Type.

descriptor->Attributes = oldAttributes;

}

else

{

descriptor->Attributes = attributes;

}

return true;

}

template <typename TPropertyIndex, typename TMapKey, bool IsNotExtensibleSupported>

BOOL SimpleDictionaryTypeHandlerBase<TPropertyIndex, TMapKey, IsNotExtensibleSupported>::SetEnumerable(DynamicObject\* instance, PropertyId propertyId, BOOL value)

{

SimpleDictionaryPropertyDescriptor<TPropertyIndex>\* descriptor;

Assert(propertyId != Constants::NoProperty);

PropertyRecord const\* propertyRecord = instance->GetScriptContext()->GetPropertyName(propertyId);

if (!propertyMap->TryGetReference(propertyRecord, &descriptor))

{

// Upgrade type handler if set objectArray item attribute.

// Only check numeric propertyId if objectArray available.

if (instance->HasObjectArray() && propertyRecord->IsNumeric())

{

return SimpleDictionaryTypeHandlerBase<TPropertyIndex, TMapKey, IsNotExtensibleSupported>::ConvertToTypeWithItemAttributes(instance)

->SetEnumerable(instance, propertyId, value);

}

return true;

}

if (descriptor->Attributes & PropertyLetConstGlobal)

{

return true;

}

if (value)

{

if (SetAttribute(instance, descriptor, PropertyEnumerable))

{

instance->SetHasNoEnumerableProperties(false);

}

}

else

{

ClearAttribute(instance, descriptor, PropertyEnumerable);

}

return true;

}

template <typename TPropertyIndex, typename TMapKey, bool IsNotExtensibleSupported>

BOOL SimpleDictionaryTypeHandlerBase<TPropertyIndex, TMapKey, IsNotExtensibleSupported>::SetWritable(DynamicObject\* instance, PropertyId propertyId, BOOL value)

{

SimpleDictionaryPropertyDescriptor<TPropertyIndex>\* descriptor;

Assert(propertyId != Constants::NoProperty);

PropertyRecord const\* propertyRecord = instance->GetScriptContext()->GetPropertyName(propertyId);

if (!propertyMap->TryGetReference(propertyRecord, &descriptor))

{

// Upgrade type handler if set objectArray item attribute.

// Only check numeric propertyId if objectArray available.

if (instance->HasObjectArray() && propertyRecord->IsNumeric())

{

return SimpleDictionaryTypeHandlerBase<TPropertyIndex, TMapKey, IsNotExtensibleSupported>::ConvertToTypeWithItemAttributes(instance)

->SetWritable(instance, propertyId, value);

}

return true;

}

if (descriptor->Attributes & PropertyLetConstGlobal)

{

return true;

}

const Type\* oldType = instance->GetType();

if (value)

{

if (SetAttribute(instance, descriptor, PropertyWritable))

{

instance->ChangeTypeIf(oldType); // Ensure type change to invalidate caches

}

}

else

{

if (ClearAttribute(instance, descriptor, PropertyWritable))

{

instance->ChangeTypeIf(oldType); // Ensure type change to invalidate caches

// Clearing the attribute may have changed the type handler, so make sure

// we access the current one.

DynamicTypeHandler \*const typeHandler = GetCurrentTypeHandler(instance);

typeHandler->ClearHasOnlyWritableDataProperties();

if(typeHandler->GetFlags() & IsPrototypeFlag)

{

instance->GetScriptContext()->InvalidateStoreFieldCaches(propertyId);

instance->GetLibrary()->NoPrototypeChainsAreEnsuredToHaveOnlyWritableDataProperties();

}

}

}

return true;

}

template <typename TPropertyIndex, typename TMapKey, bool IsNotExtensibleSupported>

BOOL SimpleDictionaryTypeHandlerBase<TPropertyIndex, TMapKey, IsNotExtensibleSupported>::SetConfigurable(DynamicObject\* instance, PropertyId propertyId, BOOL value)

{

SimpleDictionaryPropertyDescriptor<TPropertyIndex>\* descriptor;

Assert(propertyId != Constants::NoProperty);

PropertyRecord const\* propertyRecord = instance->GetScriptContext()->GetPropertyName(propertyId);

if (!propertyMap->TryGetReference(propertyRecord, &descriptor))

{

// Upgrade type handler if set objectArray item attribute.

// Only check numeric propertyId if objectArray available.

if (instance->HasObjectArray() && propertyRecord->IsNumeric())

{

return SimpleDictionaryTypeHandlerBase<TPropertyIndex, TMapKey, IsNotExtensibleSupported>::ConvertToTypeWithItemAttributes(instance)

->SetConfigurable(instance, propertyId, value);

}

return true;

}

if (descriptor->Attributes & PropertyLetConstGlobal)

{

return true;

}

if (value)

{

SetAttribute(instance, descriptor, PropertyConfigurable);

}

else

{

ClearAttribute(instance, descriptor, PropertyConfigurable);

}

return true;

}

template <typename TPropertyIndex, typename TMapKey, bool IsNotExtensibleSupported>

BOOL SimpleDictionaryTypeHandlerBase<TPropertyIndex, TMapKey, IsNotExtensibleSupported>::PreventExtensions(DynamicObject\* instance)

{

if (IsNotExtensibleSupported)

{

if ((this->GetFlags() & IsExtensibleFlag) == 0)

{

// Already not extensible => no further change needed.

return TRUE;

}

if (!GetIsLocked())

{

// If the type is not shared with others, we can just change it by itself.

return PreventExtensionsInternal(instance);

}

}

return ConvertToDictionaryType(instance)->PreventExtensions(instance);

}

template <typename TPropertyIndex, typename TMapKey, bool IsNotExtensibleSupported>

BOOL SimpleDictionaryTypeHandlerBase<TPropertyIndex, TMapKey, IsNotExtensibleSupported>::PreventExtensionsInternal(DynamicObject\* instance)

{

AssertMsg(IsNotExtensibleSupported, "This method must not be called for SimpleDictionaryTypeHandler<TPropertyIndex, IsNotExtensibleSupported = false>");

this->ClearFlags(IsExtensibleFlag);

ArrayObject \* objectArray = instance->GetObjectArray();

if (objectArray)

{

objectArray->PreventExtensions();

}

return TRUE;

}

template <typename TPropertyIndex, typename TMapKey, bool IsNotExtensibleSupported>

BOOL SimpleDictionaryTypeHandlerBase<TPropertyIndex, TMapKey, IsNotExtensibleSupported>::Seal(DynamicObject\* instance)

{

if (IsNotExtensibleSupported)

{

if (this->GetFlags() & IsSealedOnceFlag)

{

// Already sealed => no further change needed.

return TRUE;

}

if (!GetIsLocked() && !instance->HasObjectArray())

{

// If there is object array, we need to convert both type handler and array to ES5.

// Otherwise, if the type is not shared with others, we can just change it by itself.

return this->SealInternal(instance);

}

}

return ConvertToDictionaryType(instance)->Seal(instance);

}

template <typename TPropertyIndex, typename TMapKey, bool IsNotExtensibleSupported>

BOOL SimpleDictionaryTypeHandlerBase<TPropertyIndex, TMapKey, IsNotExtensibleSupported>::SealInternal(DynamicObject\* instance)

{

AssertMsg(IsNotExtensibleSupported, "This method must not be called for SimpleDictionaryTypeHandler<TPropertyIndex, IsNotExtensibleSupported = false>");

this->ChangeFlags(IsExtensibleFlag | IsSealedOnceFlag, 0 | IsSealedOnceFlag);

//Set [[Configurable]] flag of each property to false

SimpleDictionaryPropertyDescriptor<TPropertyIndex> \*descriptor = nullptr;

for (TPropertyIndex index = 0; index < propertyMap->Count(); index++)

{

descriptor = propertyMap->GetReferenceAt(index);

if (!(descriptor->Attributes & PropertyLetConstGlobal))

{

descriptor->Attributes &= (~PropertyConfigurable);

}

}

ArrayObject \* objectArray = instance->GetObjectArray();

if (objectArray)

{

objectArray->Seal();

}

return TRUE;

}

template <typename TPropertyIndex, typename TMapKey, bool IsNotExtensibleSupported>

BOOL SimpleDictionaryTypeHandlerBase<TPropertyIndex, TMapKey, IsNotExtensibleSupported>::FreezeImpl(DynamicObject\* instance, bool isConvertedType)

{

if (IsNotExtensibleSupported)

{

if (this->GetFlags() & IsFrozenOnceFlag)

{

// Already frozen => no further change needed.

return TRUE;

}

if (!GetIsLocked() && !instance->HasObjectArray())

{

// If there is object array, we need to convert both type handler and array to ES5.

// Otherwise, if the type is not shared with others, we can just change it by itself.

// If the type is not shared with others, we can just change it by itself.

return FreezeInternal(instance, isConvertedType);

}

}

return ConvertToDictionaryType(instance)->Freeze(instance);

}

template <typename TPropertyIndex, typename TMapKey, bool IsNotExtensibleSupported>

BOOL SimpleDictionaryTypeHandlerBase<TPropertyIndex, TMapKey, IsNotExtensibleSupported>::FreezeInternal(DynamicObject\* instance, bool isConvertedType)

{

AssertMsg(IsNotExtensibleSupported, "This method must not be called for SimpleDictionaryTypeHandler<TPropertyIndex, IsNotExtensibleSupported = false>");

this->ChangeFlags(IsExtensibleFlag | IsSealedOnceFlag | IsFrozenOnceFlag,

0 | IsSealedOnceFlag | IsFrozenOnceFlag);

//Set [[Writable]] flag of each property to false except for setter\getters

//Set [[Configurable]] flag of each property to false

SimpleDictionaryPropertyDescriptor<TPropertyIndex> \*descriptor = nullptr;

for (TPropertyIndex index = 0; index < propertyMap->Count(); index++)

{

descriptor = propertyMap->GetReferenceAt(index);

if (!(descriptor->Attributes & PropertyLetConstGlobal))

{

descriptor->Attributes &= ~(PropertyWritable | PropertyConfigurable);

}

}

if (!isConvertedType)

{

// Change of [[Writable]] property requires cache invalidation, hence ChangeType

instance->ChangeType();

}

ArrayObject \* objectArray = instance->GetObjectArray();

if (objectArray)

{

objectArray->Freeze();

}

this->ClearHasOnlyWritableDataProperties();

if (GetFlags() & IsPrototypeFlag)

{

InvalidateStoreFieldCachesForAllProperties(instance->GetScriptContext());

instance->GetLibrary()->NoPrototypeChainsAreEnsuredToHaveOnlyWritableDataProperties();

}

return TRUE;

}

template <typename TPropertyIndex, typename TMapKey, bool IsNotExtensibleSupported>

BOOL SimpleDictionaryTypeHandlerBase<TPropertyIndex, TMapKey, IsNotExtensibleSupported>::IsSealed(DynamicObject\* instance)

{

if (!IsNotExtensibleSupported)

{

return false;

}

BYTE flags = this->GetFlags();

if (flags & IsSealedOnceFlag)

{

// Once sealed, there is no way to undo seal.

// But note: still, it can also be sealed when the flag is not set.

return true;

}

if (flags & IsExtensibleFlag)

{

return false;

}

SimpleDictionaryPropertyDescriptor<TPropertyIndex> \*descriptor = nullptr;

for (TPropertyIndex index = 0; index < propertyMap->Count(); index++)

{

descriptor = propertyMap->GetReferenceAt(index);

if ((!(descriptor->Attributes & PropertyDeleted) && !(descriptor->Attributes & PropertyLetConstGlobal)))

{

if (descriptor->Attributes & PropertyConfigurable)

{

// [[Configurable]] must be false for all (existing) properties.

return false;

}

}

}

ArrayObject \* objectArray = instance->GetObjectArray();

if (objectArray && !objectArray->IsSealed())

{

return false;

}

// Since we've determined that the object was sealed, set the flag to avoid further checks into all properties

// (once sealed there is no way to go back to un-sealed).

this->SetFlags(IsSealedOnceFlag);

return true;

}

template <typename TPropertyIndex, typename TMapKey, bool IsNotExtensibleSupported>

BOOL SimpleDictionaryTypeHandlerBase<TPropertyIndex, TMapKey, IsNotExtensibleSupported>::IsFrozen(DynamicObject\* instance)

{

if (!IsNotExtensibleSupported)

{

return false;

}

BYTE flags = this->GetFlags();

if (flags & IsFrozenOnceFlag)

{

// Once frozen, there is no way to undo freeze.

// But note: still, it can also be frozen when the flag is not set.

return true;

}

if (this->GetFlags() & IsExtensibleFlag)

{

return false;

}

SimpleDictionaryPropertyDescriptor<TPropertyIndex> \*descriptor = nullptr;

for (TPropertyIndex index = 0; index < propertyMap->Count(); index++)

{

descriptor = propertyMap->GetReferenceAt(index);

if ((!(descriptor->Attributes & PropertyDeleted) && !(descriptor->Attributes & PropertyLetConstGlobal)))

{

// [[Configurable]] and [[Configurable]] must be false for all (existing) properties.

// IE9 compatibility: keep IE9 behavior (also check deleted properties)

if (descriptor->Attributes & PropertyConfigurable)

{

return false;

}

if (descriptor->Attributes & PropertyWritable)

{

return false;

}

}

}

// Use IsObjectArrayFrozen() to skip "length" [[Writable]] check

ArrayObject \* objectArray = instance->GetObjectArray();

if (objectArray && !objectArray->IsObjectArrayFrozen())

{

return false;

}

// Since we've determined that the object was frozen, set the flag to avoid further checks into all properties

// (once frozen there is no way to go back to un-frozen).

this->SetFlags(IsSealedOnceFlag | IsFrozenOnceFlag);

return true;

}

template <typename TPropertyIndex, typename TMapKey, bool IsNotExtensibleSupported>

BOOL SimpleDictionaryTypeHandlerBase<TPropertyIndex, TMapKey, IsNotExtensibleSupported>::SetAccessors(DynamicObject\* instance, PropertyId propertyId, Var getter, Var setter, PropertyOperationFlags flags)

{

return ConvertToDictionaryType(instance)->SetAccessors(instance, propertyId, getter, setter, flags);

}

template <typename TPropertyIndex, typename TMapKey, bool IsNotExtensibleSupported>

BOOL SimpleDictionaryTypeHandlerBase<TPropertyIndex, TMapKey, IsNotExtensibleSupported>::SetPropertyWithAttributes(DynamicObject\* instance, PropertyId propertyId, Var value, PropertyAttributes attributes, PropertyValueInfo\* info, PropertyOperationFlags flags, SideEffects possibleSideEffects)

{

AnalysisAssert(instance);

SimpleDictionaryPropertyDescriptor<TPropertyIndex>\* descriptor;

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

bool throwIfNotExtensible = (flags & PropertyOperation\_ThrowIfNotExtensible) != 0;

Assert(propertyId != Constants::NoProperty);

PropertyRecord const\* propertyRecord = instance->GetScriptContext()->GetPropertyName(propertyId);

if (propertyMap->TryGetReference(propertyRecord, &descriptor))

{

if ((attributes & PropertyLetConstGlobal) != (descriptor->Attributes & PropertyLetConstGlobal))

{

// We have a shadowing case here. Either a let/const is being declared

// that shadows an undeclared global property or an undeclared global

// property is being added after a let/const was already declared.

//

// SimpleDictionaryTypeHandlerBase does not handle shadowed globals

// because we do not want to add another property index field to the

// property descriptors. Instead convert to DictionaryTypeHandler

// where it will reuse one of the getter/setter fields on its

// property descriptor type.

//

// An exception is in the language service that will sometimes execute

// the glo function twice causing a const or let appear to shadow itself.

// In this case setting the property is also right.

return

ConvertToDictionaryType(instance)

->SetPropertyWithAttributes(

instance,

propertyId,

value,

attributes,

info,

flags,

possibleSideEffects);

}

if (descriptor->Attributes & PropertyDeleted && !(descriptor->Attributes & PropertyLetConstGlobal))

{

Assert((descriptor->Attributes & PropertyLetConstGlobal) == 0);

if(GetIsLocked())

{

return

ConvertToNonSharedSimpleDictionaryType(instance)

->SetPropertyWithAttributes(

instance,

propertyId,

value,

attributes,

info,

flags,

possibleSideEffects);

}

if(isUnordered)

{

TPropertyIndex propertyIndex;

if(AsUnordered()->TryUndeleteProperty(instance, descriptor->propertyIndex, &propertyIndex))

{

Assert(PropertyRecordStringHashComparer<TMapKey>::Equals(propertyMap->GetKeyAt(propertyIndex), propertyRecord));

descriptor = propertyMap->GetReferenceAt(propertyIndex);

}

}

if (IsNotExtensibleSupported)

{

bool isForce = (flags & PropertyOperation\_Force) != 0;

if (!isForce)

{

if (!this->VerifyIsExtensible(scriptContext, throwIfNotExtensible))

{

return FALSE;

}

}

}

if(SupportsSwitchingToUnordered(scriptContext))

{

--numDeletedProperties;

}

scriptContext->InvalidateProtoCaches(propertyId);

descriptor->Attributes = PropertyDynamicTypeDefaults;

}

if (descriptor->Attributes != attributes)

{

if (GetIsLocked())

{

return

ConvertToNonSharedSimpleDictionaryType(instance)

->SetPropertyWithAttributes(

instance,

propertyId,

value,

attributes,

info,

flags,

possibleSideEffects);

}

else

{

descriptor->Attributes = attributes;

}

}

if (descriptor->propertyIndex != NoSlots)

{

DynamicObject\* localSingletonInstance = this->singletonInstance != nullptr ? this->singletonInstance->Get() : nullptr;

Assert(this->singletonInstance == nullptr || localSingletonInstance == instance);

if (!descriptor->isInitialized)

{

if ((flags & PropertyOperation\_PreInit) == 0)

{

// Consider: It would be nice to assert the slot is actually null. However, we sometimes pre-initialize to undefined or even

// some other special illegal value (for let or const), currently == null.

descriptor->isInitialized = true;

if (localSingletonInstance == instance && !IsInternalPropertyId(propertyId) && (flags & (PropertyOperation\_SpecialValue | PropertyOperation\_NonFixedValue)) == 0)

{

Assert(!GetIsShared());

Assert(value != nullptr);

// We don't want fixed properties on external objects. See DynamicObject::ResetObject for more information.

Assert(!instance->IsExternal());

descriptor->isFixed = (JavascriptFunction::Is(value) ? ShouldFixMethodProperties() : (ShouldFixDataProperties() && CheckHeuristicsForFixedDataProps(instance, propertyId, value)));

}

}

}

else

{

InvalidateFixedField(propertyRecord, descriptor, instance->GetScriptContext());

}

SetSlotUnchecked(instance, descriptor->propertyIndex, value);

if (descriptor->isInitialized && !descriptor->isFixed)

{

SetPropertyValueInfo(info, instance, descriptor->propertyIndex, descriptor->Attributes);

}

else

{

PropertyValueInfo::SetNoCache(info, instance);

}

}

if (descriptor->Attributes & PropertyEnumerable)

{

instance->SetHasNoEnumerableProperties(false);

}

if (!(descriptor->Attributes & PropertyWritable))

{

this->ClearHasOnlyWritableDataProperties();

if(GetFlags() & IsPrototypeFlag)

{

instance->GetScriptContext()->InvalidateStoreFieldCaches(propertyId);

instance->GetLibrary()->NoPrototypeChainsAreEnsuredToHaveOnlyWritableDataProperties();

}

}

SetPropertyUpdateSideEffect(instance, propertyId, value, possibleSideEffects);

return true;

}

// Always check numeric propertyRecord. May create objectArray.

if (propertyRecord->IsNumeric())

{

return SimpleDictionaryTypeHandlerBase<TPropertyIndex, TMapKey, IsNotExtensibleSupported>::SetItemWithAttributes(instance, propertyRecord->GetNumericValue(), value, attributes);

}

return this->AddProperty(instance, propertyRecord, value, attributes, info, flags, possibleSideEffects);

}

// We need to override SetItem as JavascriptArray (in contrary to ES5Array doesn't have checks for object being not extensible).

// So, check here.

// Note that we don't need to override SetItemWithAttributes because for that one base class implementation calls ConvertToTypeWithItemAttributes

// which converts both this type and its objectArray to DictionaryTypeHandler/ES5ArrayTypeHandler.

template <typename TPropertyIndex, typename TMapKey, bool IsNotExtensibleSupported>

BOOL SimpleDictionaryTypeHandlerBase<TPropertyIndex, TMapKey, IsNotExtensibleSupported>::SetItem(DynamicObject\* instance, uint32 index, Var value, PropertyOperationFlags flags)

{

if (IsNotExtensibleSupported)

{

// When adding a new property && we are not extensible:

// - if (!objectArray) => do not even get into creating new objectArrray

// (anyhow, if we were to create one, we would need one supporting non-extensible, i.e. ES5Array).

// - else the array was created earlier and will handle the operation

// (it would be non-extensible ES5 array as array must match object's IsExtensible).

bool isExtensible = (this->GetFlags() & IsExtensibleFlag) != 0;

if (!isExtensible && !instance->HasObjectArray()) // Note: Setitem && !HasObjectArray => attempt to add a new item.

{

bool throwIfNotExtensible = (flags & (PropertyOperation\_StrictMode | PropertyOperation\_ThrowIfNotExtensible)) != 0;

if (throwIfNotExtensible)

{

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

JavascriptError::ThrowTypeError(scriptContext, JSERR\_NonExtensibleObject);

}

return FALSE;

}

}

return instance->SetObjectArrayItem(index, value, flags); // I.e. \_\_super::SetItem(...).

}

template <typename TPropertyIndex, typename TMapKey, bool IsNotExtensibleSupported>

void SimpleDictionaryTypeHandlerBase<TPropertyIndex, TMapKey, IsNotExtensibleSupported>::EnsureSlotCapacity(DynamicObject \* instance)

{

Assert(this->GetSlotCapacity() < MaxPropertyIndexSize); // Otherwise we can't grow this handler's capacity. We should've evolved to bigger handler or OOM.

// This check should be done by caller of this function.

//if (slotCapacity <= nextPropertyIndex)

{

// A Dictionary type is expected to have more properties

// grow exponentially rather linearly to avoid the realloc and moves,

// however use a small exponent to avoid waste

int newSlotCapacity = (nextPropertyIndex + 1);

newSlotCapacity += (newSlotCapacity>>2);

if (newSlotCapacity > MaxPropertyIndexSize)

{

newSlotCapacity = MaxPropertyIndexSize;

}

newSlotCapacity = RoundUpSlotCapacity(newSlotCapacity, GetInlineSlotCapacity());

Assert(newSlotCapacity <= MaxPropertyIndexSize);

instance->EnsureSlots(this->GetSlotCapacity(), newSlotCapacity, instance->GetScriptContext(), this);

this->SetSlotCapacity(newSlotCapacity);

}

}

template <typename TPropertyIndex, typename TMapKey, bool IsNotExtensibleSupported>

BOOL SimpleDictionaryTypeHandlerBase<TPropertyIndex, TMapKey, IsNotExtensibleSupported>::SetAttributes(DynamicObject\* instance, PropertyId propertyId, PropertyAttributes attributes)

{

if (!GetIsLocked())

{

SimpleDictionaryPropertyDescriptor<TPropertyIndex>\* descriptor;

Assert(propertyId != Constants::NoProperty);

PropertyRecord const\* propertyRecord = instance->GetScriptContext()->GetPropertyName(propertyId);

if (propertyMap->TryGetReference(propertyRecord, &descriptor))

{

if (attributes & PropertyLetConstGlobal)

{

Assert(!(descriptor->Attributes & PropertyLetConstGlobal));

// Need to implement type transition to DictionaryTypeHandler in the case of

// shadowing a var or global property with a let in a new script body.

Throw::NotImplemented();

}

if (descriptor->Attributes & PropertyLetConstGlobal)

{

Assert(!(attributes & PropertyLetConstGlobal));

Assert(false);

}

if (descriptor->Attributes & PropertyDeleted)

{

return false;

}

descriptor->Attributes = (descriptor->Attributes & ~PropertyDynamicTypeDefaults) | (attributes & PropertyDynamicTypeDefaults);

if (attributes & PropertyEnumerable)

{

instance->SetHasNoEnumerableProperties(false);

}

if (!(descriptor->Attributes & PropertyWritable))

{

this->ClearHasOnlyWritableDataProperties();

if(GetFlags() & IsPrototypeFlag)

{

instance->GetScriptContext()->InvalidateStoreFieldCaches(propertyId);

instance->GetLibrary()->NoPrototypeChainsAreEnsuredToHaveOnlyWritableDataProperties();

}

}

return true;

}

// Check numeric propertyRecord only if objectArray available

if (instance->HasObjectArray() && propertyRecord->IsNumeric())

{

return SimpleDictionaryTypeHandlerBase<TPropertyIndex, TMapKey, IsNotExtensibleSupported>::SetItemAttributes(instance, propertyRecord->GetNumericValue(), attributes);

}

}

else

{

return ConvertToNonSharedSimpleDictionaryType(instance)->SetAttributes(instance, propertyId, attributes);

}

return false;

}

template <typename TPropertyIndex, typename TMapKey, bool IsNotExtensibleSupported>

BOOL SimpleDictionaryTypeHandlerBase<TPropertyIndex, TMapKey, IsNotExtensibleSupported>::GetAttributesWithPropertyIndex(DynamicObject \* instance, PropertyId propertyId, BigPropertyIndex index, PropertyAttributes \* attributes)

{

// this might get value that are deleted from the dictionary, but that should be nulled out

SimpleDictionaryPropertyDescriptor<TPropertyIndex> const \* descriptor;

if (!propertyMap->TryGetValueAt(index, &descriptor))

{

return false;

}

Assert(descriptor->propertyIndex == index);

if (descriptor->Attributes & PropertyDeleted)

{

return false;

}

\*attributes = descriptor->Attributes & PropertyDynamicTypeDefaults;

return true;

}

template <typename TPropertyIndex, typename TMapKey, bool IsNotExtensibleSupported>

template <typename TPropertyKey>

BOOL SimpleDictionaryTypeHandlerBase<TPropertyIndex, TMapKey, IsNotExtensibleSupported>::AddProperty(DynamicObject\* instance, TPropertyKey propertyKey, Var value, PropertyAttributes attributes,

PropertyValueInfo\* info, PropertyOperationFlags flags, SideEffects possibleSideEffects)

{

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

#if DBG

// Only Assert that the propertyMap doesn't contain propertyKey if TMapKey is string and propertyKey is not a symbol.

if (!TMapKey\_IsJavascriptString<TMapKey>() || !TMapKey\_IsSymbol(propertyKey, scriptContext))

{

SimpleDictionaryPropertyDescriptor<TPropertyIndex>\* descriptor;

Assert(!propertyMap->TryGetReference(propertyKey, &descriptor));

}

Assert(!TPropertyKey\_IsNumeric(propertyKey));

#endif

#ifdef ENABLE\_DEBUG\_CONFIG\_OPTIONS

if (CONFIG\_FLAG(ForceStringKeyedSimpleDictionaryTypeHandler) &&

!TMapKey\_IsJavascriptString<TMapKey>() &&

!isUnordered && !hasNamelessPropertyId &&

!TPropertyKey\_IsInternalPropertyId(propertyKey) &&

!TMapKey\_IsSymbol(propertyKey, scriptContext))

{

return ConvertToSimpleDictionaryUnorderedTypeHandler<TPropertyIndex, JavascriptString\*, IsNotExtensibleSupported>(instance)

->AddProperty(instance, propertyKey, value, attributes, info, flags, possibleSideEffects);

}

#endif

if (IsNotExtensibleSupported)

{

bool isForce = (flags & PropertyOperation\_Force) != 0;

if (!isForce)

{

bool throwIfNotExtensible = (flags & (PropertyOperation\_ThrowIfNotExtensible | PropertyOperation\_StrictMode)) != 0;

if (!this->VerifyIsExtensible(scriptContext, throwIfNotExtensible))

{

return FALSE;

}

}

}

SimpleDictionaryTypeHandlerBase \* typeHandler = this;

if (GetIsLocked())

{

typeHandler = ConvertToNonSharedSimpleDictionaryType(instance);

}

if (TMapKey\_IsJavascriptString<TMapKey>() &&

(TPropertyKey\_IsInternalPropertyId(propertyKey) || TMapKey\_IsSymbol(propertyKey, scriptContext)))

{

PHASE\_PRINT\_TESTTRACE1(Js::TypeHandlerTransitionPhase, L"Transitioning from string keyed to PropertyRecord keyed SimpleDictionaryUnorderedTypeHandler\n");

// String keyed type handler cannot store InternalPropertyRecords since they have no string representation

return ConvertToSimpleDictionaryUnorderedTypeHandler<TPropertyIndex, const PropertyRecord\*, IsNotExtensibleSupported>(instance)

->AddProperty(instance, propertyKey, value, attributes, info, flags, possibleSideEffects);

}

if (this->GetSlotCapacity() <= nextPropertyIndex)

{

if (this->GetSlotCapacity() >= MaxPropertyIndexSize)

{

BigSimpleDictionaryTypeHandler\* newTypeHandler = ConvertToBigSimpleDictionaryTypeHandler(instance);

return newTypeHandler->AddProperty(instance, propertyKey, value, attributes, info, flags, possibleSideEffects);

}

typeHandler->EnsureSlotCapacity(instance);

}

Assert((flags & PropertyOperation\_SpecialValue) != 0 || value != nullptr);

Assert(!typeHandler->GetIsShared());

Assert(typeHandler->singletonInstance == nullptr || typeHandler->singletonInstance->Get() == instance);

bool markAsInitialized = ((flags & PropertyOperation\_PreInit) == 0);

bool markAsFixed = markAsInitialized && !TPropertyKey\_IsInternalPropertyId(propertyKey) && (flags & (PropertyOperation\_NonFixedValue | PropertyOperation\_SpecialValue)) == 0 &&

typeHandler->singletonInstance != nullptr && typeHandler->singletonInstance->Get() == instance

&& (JavascriptFunction::Is(value) ? ShouldFixMethodProperties() : (ShouldFixDataProperties() && CheckHeuristicsForFixedDataProps(instance, propertyKey, value)));

TPropertyIndex index;

if (typeHandler->isUnordered &&

typeHandler->AsUnordered()->TryReuseDeletedPropertyIndex(instance, &index))

{

// We are reusing a deleted property index for the new property ID. Update the property map.

Assert(typeHandler->propertyMap->GetValueAt(index).propertyIndex == index);

TMapKey deletedPropertyKey = typeHandler->propertyMap->GetKeyAt(index);

typeHandler->propertyMap->Remove(deletedPropertyKey);

typeHandler->Add(index, propertyKey, attributes, markAsInitialized, markAsFixed, false, scriptContext);

}

else

{

index = nextPropertyIndex;

typeHandler->Add(propertyKey, attributes, markAsInitialized, markAsFixed, false, scriptContext);

}

if (attributes & PropertyEnumerable)

{

instance->SetHasNoEnumerableProperties(false);

}

SetSlotUnchecked(instance, index, value);

// It's ok to populate inline cache here even if this handler isn't shared yet, because we don't have property add

// inline cache flavor for SimpleDictionaryTypeHandlers. This ensures that a) no new instance reaches this handler

// without us knowing, and b) the inline cache doesn't inadvertently become polymorphic.

if (markAsInitialized && !markAsFixed)

{

SetPropertyValueInfo(info, instance, index, attributes);

}

else

{

PropertyValueInfo::SetNoCache(info, instance);

}

PropertyId propertyId = TPropertyKey\_GetOptionalPropertyId(scriptContext, propertyKey);

if (propertyId != Constants::NoProperty)

{

if ((typeHandler->GetFlags() & IsPrototypeFlag)

|| (!IsInternalPropertyId(propertyId)

&& JavascriptOperators::HasProxyOrPrototypeInlineCacheProperty(instance, propertyId)))

{

// We don't evolve dictionary types when adding a field, so we need to invalidate prototype caches.

// We only have to do this though if the current type is used as a prototype, or the current property

// is found on the prototype chain.

scriptContext->InvalidateProtoCaches(propertyId);

}

SetPropertyUpdateSideEffect(instance, propertyId, value, possibleSideEffects);

}

return true;

}

template <typename TPropertyIndex, typename TMapKey, bool IsNotExtensibleSupported>

void SimpleDictionaryTypeHandlerBase<TPropertyIndex, TMapKey, IsNotExtensibleSupported>::SetAllPropertiesToUndefined(DynamicObject\* instance, bool invalidateFixedFields)

{

// Note: This method is currently only called from ResetObject, which in turn only applies to external objects.

// Before using for other purposes, make sure the assumptions made here make sense in the new context. In particular,

// the invalidateFixedFields == false is only correct if a) the object is known not to have any, or b) the type of the

// object has changed and/or property guards have already been invalidated through some other means.

int propertyCount = this->propertyMap->Count();

if (IsNotExtensibleSupported)

{

// The Var for window is reused across navigation. we shouldn't preserve the IsExtensibleFlag when we don't keep

// the expandoes. Reset the IsExtensibleFlag in cleanup scenario should be good enough

// to cover all the preventExtension/Freeze/Seal scenarios.

ChangeFlags(IsExtensibleFlag | IsSealedOnceFlag | IsFrozenOnceFlag, IsExtensibleFlag);

}

if (invalidateFixedFields)

{

Js::ScriptContext\* scriptContext = instance->GetScriptContext();

for (int propertyIndex = 0; propertyIndex < propertyCount; propertyIndex++)

{

const TMapKey propertyRecord = this->propertyMap->GetKeyAt(propertyIndex);

SimpleDictionaryPropertyDescriptor<TPropertyIndex>\* descriptor = this->propertyMap->GetReferenceAt(propertyIndex);

InvalidateFixedField(propertyRecord, descriptor, scriptContext);

}

}

Js::RecyclableObject\* undefined = instance->GetLibrary()->GetUndefined();

int slotCount = this->nextPropertyIndex;

for (int slotIndex = 0; slotIndex < slotCount; slotIndex++)

{

SetSlotUnchecked(instance, slotIndex, undefined);

}

}

template <typename TPropertyIndex, typename TMapKey, bool IsNotExtensibleSupported>

void SimpleDictionaryTypeHandlerBase<TPropertyIndex, TMapKey, IsNotExtensibleSupported>::MarshalAllPropertiesToScriptContext(DynamicObject\* instance, ScriptContext\* targetScriptContext, bool invalidateFixedFields)

{

// Note: This method is currently only called from ResetObject, which in turn only applies to external objects.

// Before using for other purposes, make sure the assumptions made here make sense in the new context. In particular,

// the invalidateFixedFields == false is only correct if a) the object is known not to have any, or b) the type of the

// object has changed and/or property guards have already been invalidated through some other means.

int propertyCount = this->propertyMap->Count();

if (invalidateFixedFields)

{

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

for (int propertyIndex = 0; propertyIndex < propertyCount; propertyIndex++)

{

const TMapKey propertyRecord = this->propertyMap->GetKeyAt(propertyIndex);

SimpleDictionaryPropertyDescriptor<TPropertyIndex>\* descriptor = this->propertyMap->GetReferenceAt(propertyIndex);

InvalidateFixedField(propertyRecord, descriptor, scriptContext);

}

}

int slotCount = this->nextPropertyIndex;

for (int slotIndex = 0; slotIndex < slotCount; slotIndex++)

{

SetSlotUnchecked(instance, slotIndex, CrossSite::MarshalVar(targetScriptContext, GetSlot(instance, slotIndex)));

}

}

template <typename TPropertyIndex, typename TMapKey, bool IsNotExtensibleSupported>

DynamicTypeHandler\* SimpleDictionaryTypeHandlerBase<TPropertyIndex, TMapKey, IsNotExtensibleSupported>::ConvertToTypeWithItemAttributes(DynamicObject\* instance)

{

return JavascriptArray::Is(instance) ?

ConvertToES5ArrayType(instance) : ConvertToDictionaryType(instance);

}

template <typename TPropertyIndex, typename TMapKey, bool IsNotExtensibleSupported>

void SimpleDictionaryTypeHandlerBase<TPropertyIndex, TMapKey, IsNotExtensibleSupported>::SetIsPrototype(DynamicObject\* instance)

{

// Don't return if IsPrototypeFlag is set, because we may still need to do a type transition and

// set fixed bits. If this handler is shared, this instance may not even be a prototype yet.

// In this case we may need to convert to a non-shared type handler.

if (!ChangeTypeOnProto() && !(GetIsOrMayBecomeShared() && IsolatePrototypes()))

{

SetFlags(IsPrototypeFlag);

return;

}

Assert(!GetIsShared() || this->singletonInstance == nullptr);

Assert(this->singletonInstance == nullptr || this->singletonInstance->Get() == instance);

SetIsPrototype(instance, false);

}

template <typename TPropertyIndex, typename TMapKey, bool IsNotExtensibleSupported>

void SimpleDictionaryTypeHandlerBase<TPropertyIndex, TMapKey, IsNotExtensibleSupported>::SetIsPrototype(DynamicObject\* instance, bool hasNewType)

{

const auto setFixedFlags = [instance](TMapKey propertyKey, SimpleDictionaryPropertyDescriptor<TPropertyIndex>\* const descriptor, bool hasNewType)

{

if (TPropertyKey\_IsInternalPropertyId(propertyKey))

{

return;

}

if (!(descriptor->Attributes & PropertyDeleted) && !(descriptor->Attributes & PropertyLetConstGlobal))

{

// See PathTypeHandlerBase::ConvertToSimpleDictionaryType for rules governing fixed field bits during type

// handler transitions. In addition, we know that the current instance is not yet a prototype.

if (descriptor->propertyIndex != NoSlots)

{

// Consider: If we decide to fix all types of properties, we could skip loading the value from the instance.

if (descriptor->isInitialized)

{

Var value = instance->GetSlot(descriptor->propertyIndex);

// Even though the handler says the property is initialized the particular instance may not yet have

// a value for this property. This should only happen if the handler is shared.

if (value != nullptr)

{

if (hasNewType)

{

// Since we have a non-shared type handler, it's ok to fix all fields at their current values, as long as

// we've also taken a type transition. Otherwise populated load field caches would still be valid and

// would need to be explicitly invalidated if the property value changes.

// saravind:If the instance is used by a CrossSiteObject, then we are conservative and do not mark any field as fixed in that instance.

// We need to relax this in the future and support fixed fields for Cross Site Context usage

descriptor->isFixed = (JavascriptFunction::Is(value) ? ShouldFixMethodProperties() : (ShouldFixDataProperties() && CheckHeuristicsForFixedDataProps(instance, propertyKey, value)));

// Since we have a new type we can clear all used as fixed bits. That's because any instance field loads

// will have been invalidated by the type transition, and there are no proto fields loads from this object

// because it is just now becoming a proto.

descriptor->usedAsFixed = false;

}

}

}

else

{

Assert(!descriptor->isFixed && !descriptor->usedAsFixed);

}

}

}

};

bool isShared = GetIsShared();

if (GetIsOrMayBecomeShared() && IsolatePrototypes())

{

Type\* oldType = instance->GetType();

ConvertToNonSharedSimpleDictionaryType(instance)->SetIsPrototype(instance, instance->GetType() != oldType);

}

else

{

#ifdef ENABLE\_DEBUG\_CONFIG\_OPTIONS

DynamicType\* oldType = instance->GetDynamicType();

RecyclerWeakReference<DynamicObject>\* oldSingletonInstance = GetSingletonInstance();

TraceFixedFieldsBeforeSetIsProto(instance, this, oldType, oldSingletonInstance);

#endif

if (!hasNewType && ChangeTypeOnProto())

{

// We're about to split out the type. If the original type was shared the handler better be shared as well.

// Otherwise, the handler would lose track of being shared between different types and instances.

Assert(!instance->HasSharedType() || instance->GetDynamicType()->GetTypeHandler()->GetIsShared());

// Forcing a type transition allows us to fix all fields (even those that were previously marked as non-fixed).

instance->ChangeType();

Assert(!instance->HasSharedType());

hasNewType = true;

}

if (!isShared)

{

Assert(this->singletonInstance == nullptr || this->singletonInstance->Get() == instance);

Assert(HasSingletonInstanceOnlyIfNeeded());

if (AreSingletonInstancesNeeded() && this->singletonInstance == nullptr)

{

this->singletonInstance = instance->CreateWeakReferenceToSelf();

}

// We don't want fixed properties on external objects. See DynamicObject::ResetObject for more information.

if (!instance->IsExternal())

{

// If this type handler is not shared by any types or instances we can simply mark all existing properties as fixed.

// The propertyMap dictionary is guaranteed to have contiguous entries because we never remove entries from it.

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

{

TMapKey propertyKey = propertyMap->GetKeyAt(i);

SimpleDictionaryPropertyDescriptor<TPropertyIndex>\* const descriptor = propertyMap->GetReferenceAt(i);

setFixedFlags(propertyKey, descriptor, hasNewType);

}

}

}

SetFlags(IsPrototypeFlag);

#ifdef ENABLE\_DEBUG\_CONFIG\_OPTIONS

TraceFixedFieldsAfterSetIsProto(instance, this, this, oldType, oldSingletonInstance);

#endif

}

}

template <typename TPropertyIndex, typename TMapKey, bool IsNotExtensibleSupported>

bool SimpleDictionaryTypeHandlerBase<TPropertyIndex, TMapKey, IsNotExtensibleSupported>::HasSingletonInstance() const

{

return this->singletonInstance != nullptr;

}

template <typename TPropertyIndex, typename TMapKey, bool IsNotExtensibleSupported>

bool SimpleDictionaryTypeHandlerBase<TPropertyIndex, TMapKey, IsNotExtensibleSupported>::TryUseFixedProperty(PropertyRecord const \* propertyRecord, Var \* pProperty, FixedPropertyKind propertyType, ScriptContext \* requestContext)

{

bool result = TryGetFixedProperty<false, true>(propertyRecord, pProperty, propertyType, requestContext);

TraceUseFixedProperty(propertyRecord, pProperty, result, L"SimpleDictionaryTypeHandler", requestContext);

return result;

}

template <typename TPropertyIndex, typename TMapKey, bool IsNotExtensibleSupported>

bool SimpleDictionaryTypeHandlerBase<TPropertyIndex, TMapKey, IsNotExtensibleSupported>::TryUseFixedAccessor(PropertyRecord const \* propertyRecord, Var \* pAccessor, FixedPropertyKind propertyType, bool getter, ScriptContext \* requestContext)

{

if (PHASE\_VERBOSE\_TRACE1(Js::FixedMethodsPhase) || PHASE\_VERBOSE\_TESTTRACE1(Js::FixedMethodsPhase) ||

PHASE\_VERBOSE\_TRACE1(Js::UseFixedDataPropsPhase) || PHASE\_VERBOSE\_TESTTRACE1(Js::UseFixedDataPropsPhase))

{

Output::Print(L"FixedFields: attempt to use fixed accessor %s from SimpleDictionaryTypeHandler returned false.\n", propertyRecord->GetBuffer());

if (this->HasSingletonInstance() && this->GetSingletonInstance()->Get()->GetScriptContext() != requestContext)

{

Output::Print(L"FixedFields: Cross Site Script Context is used for property %s. \n", propertyRecord->GetBuffer());

}

Output::Flush();

}

return false;

}

#if DBG

template <typename TPropertyIndex, typename TMapKey, bool IsNotExtensibleSupported>

bool SimpleDictionaryTypeHandlerBase<TPropertyIndex, TMapKey, IsNotExtensibleSupported>::CanStorePropertyValueDirectly(const DynamicObject\* instance, PropertyId propertyId, bool allowLetConst)

{

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

SimpleDictionaryPropertyDescriptor<TPropertyIndex>\* descriptor;

// We pass Constants::NoProperty for ActivationObjects for functions with same named formals.

if (propertyId == Constants::NoProperty)

{

return true;

}

PropertyRecord const\* propertyRecord = scriptContext->GetPropertyName(propertyId);

if (propertyMap->TryGetReference(propertyRecord, &descriptor))

{

if (allowLetConst && (descriptor->Attributes & PropertyLetConstGlobal))

{

return true;

}

else

{

AssertMsg(!(descriptor->Attributes & PropertyLetConstGlobal), "Asking about a global property this type handler doesn't have?");

return descriptor->isInitialized && !descriptor->isFixed;

}

}

else

{

AssertMsg(false, "Asking about a property this type handler doesn't know about?");

return false;

}

}

template <typename TPropertyIndex, typename TMapKey, bool IsNotExtensibleSupported>

bool SimpleDictionaryTypeHandlerBase<TPropertyIndex, TMapKey, IsNotExtensibleSupported>::CheckFixedProperty(PropertyRecord const \* propertyRecord, Var \* pProperty, ScriptContext \* requestContext)

{

return TryGetFixedProperty<true, false>(propertyRecord, pProperty, (FixedPropertyKind) (Js::FixedPropertyKind::FixedMethodProperty | Js::FixedPropertyKind::FixedDataProperty), requestContext);

}

template <typename TPropertyIndex, typename TMapKey, bool IsNotExtensibleSupported>

bool SimpleDictionaryTypeHandlerBase<TPropertyIndex, TMapKey, IsNotExtensibleSupported>::HasAnyFixedProperties() const

{

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

{

SimpleDictionaryPropertyDescriptor<TPropertyIndex> descriptor = propertyMap->GetValueAt(i);

if (descriptor.isFixed)

{

return true;

}

}

return false;

}

#endif

template <typename TPropertyIndex, typename TMapKey, bool IsNotExtensibleSupported>

template <bool allowNonExistent, bool markAsUsed>

bool SimpleDictionaryTypeHandlerBase<TPropertyIndex, TMapKey, IsNotExtensibleSupported>::TryGetFixedProperty(PropertyRecord const \* propertyRecord, Var \* pProperty, FixedPropertyKind propertyType, ScriptContext \* requestContext)

{

// Note: This function is not thread-safe and cannot be called from the JIT thread. That's why we collect and

// cache any fixed function instances during work item creation on the main thread.

DynamicObject\* localSingletonInstance = this->singletonInstance != nullptr ? this->singletonInstance->Get() : nullptr;

if (localSingletonInstance != nullptr && localSingletonInstance->GetScriptContext() == requestContext)

{

SimpleDictionaryPropertyDescriptor<TPropertyIndex>\* descriptor;

if (propertyMap->TryGetReference(propertyRecord, &descriptor))

{

if (descriptor->isFixed)

{

AssertMsg(!(descriptor->Attributes & PropertyLetConstGlobal), "can't have fixed global let/const");

Assert(!IsInternalPropertyId(propertyRecord->GetPropertyId()));

Var value = localSingletonInstance->GetSlot(descriptor->propertyIndex);

if (value && ((IsFixedMethodProperty(propertyType) && JavascriptFunction::Is(value)) || IsFixedDataProperty(propertyType)))

{

\*pProperty = value;

if (markAsUsed)

{

descriptor->usedAsFixed = true;

}

return true;

}

}

}

else

{

// If we're unordered, we may have removed this descriptor from the map and replaced it with a descriptor for a

// different property. When we do that, we change the type of the instance, but the old type (which may still be

// in some inline cache) still points to the same type handler.

AssertMsg(allowNonExistent || isUnordered, "Trying to get a fixed function instance for a non-existent property?");

}

}

return false;

}

template <typename TPropertyIndex, typename TMapKey, bool IsNotExtensibleSupported>

template <typename TPropertyKey>

void SimpleDictionaryTypeHandlerBase<TPropertyIndex, TMapKey, IsNotExtensibleSupported>::InvalidateFixedField(const TPropertyKey propertyKey, SimpleDictionaryPropertyDescriptor<TPropertyIndex>\* descriptor, ScriptContext\* scriptContext)

{

Assert(descriptor->isInitialized);

descriptor->isFixed = false;

if (descriptor->usedAsFixed)

{

#if ENABLE\_NATIVE\_CODEGEN

PropertyId propertyId = TMapKey\_GetPropertyId(scriptContext, propertyKey);

scriptContext->GetThreadContext()->InvalidatePropertyGuards(propertyId);

#endif

descriptor->usedAsFixed = false;

}

}

#if DBG

template <typename TPropertyIndex, typename TMapKey, bool IsNotExtensibleSupported>

bool SimpleDictionaryTypeHandlerBase<TPropertyIndex, TMapKey, IsNotExtensibleSupported>::IsLetConstGlobal(DynamicObject\* instance, PropertyId propertyId)

{

SimpleDictionaryPropertyDescriptor<TPropertyIndex>\* descriptor;

PropertyRecord const\* propertyRecord = instance->GetScriptContext()->GetPropertyName(propertyId);

if (propertyMap->TryGetReference(propertyRecord, &descriptor) && (descriptor->Attributes & PropertyLetConstGlobal))

{

return true;

}

return false;

}

#endif

template <typename TPropertyIndex, typename TMapKey, bool IsNotExtensibleSupported>

bool SimpleDictionaryTypeHandlerBase<TPropertyIndex, TMapKey, IsNotExtensibleSupported>::NextLetConstGlobal(int& index, RootObjectBase\* instance, const PropertyRecord\*\* propertyRecord, Var\* value, bool\* isConst)

{

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

for (; index < propertyMap->Count(); index++)

{

SimpleDictionaryPropertyDescriptor<TPropertyIndex> descriptor = propertyMap->GetValueAt(index);

if (descriptor.Attributes & PropertyLetConstGlobal)

{

\*propertyRecord = TMapKey\_ConvertKey<const PropertyRecord\*>(scriptContext, propertyMap->GetKeyAt(index));

\*value = instance->GetSlot(descriptor.propertyIndex);

\*isConst = (descriptor.Attributes & PropertyConst) != 0;

index += 1;

return true;

}

}

return false;

}

#ifdef ENABLE\_DEBUG\_CONFIG\_OPTIONS

template <typename TPropertyIndex, typename TMapKey, bool IsNotExtensibleSupported>

void SimpleDictionaryTypeHandlerBase<TPropertyIndex, TMapKey, IsNotExtensibleSupported>::DumpFixedFields() const {

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

{

SimpleDictionaryPropertyDescriptor<TPropertyIndex> descriptor = propertyMap->GetValueAt(i);

TMapKey propertyKey = propertyMap->GetKeyAt(i);

Output::Print(L" %s %d%d%d,", TMapKey\_GetBuffer(propertyKey),

descriptor.isInitialized ? 1 : 0, descriptor.isFixed ? 1 : 0, descriptor.usedAsFixed ? 1 : 0);

}

}

template <typename TPropertyIndex, typename TMapKey, bool IsNotExtensibleSupported>

void SimpleDictionaryTypeHandlerBase<TPropertyIndex, TMapKey, IsNotExtensibleSupported>::TraceFixedFieldsBeforeTypeHandlerChange(

const wchar\_t\* oldTypeHandlerName, const wchar\_t\* newTypeHandlerName,

DynamicObject\* instance, DynamicTypeHandler\* oldTypeHandler,

DynamicType\* oldType, RecyclerWeakReference<DynamicObject>\* oldSingletonInstanceBefore)

{

if (PHASE\_VERBOSE\_TRACE1(FixMethodPropsPhase))

{

Output::Print(L"FixedFields: converting 0x%p from %s to %s:\n", instance, oldTypeHandlerName, newTypeHandlerName);

Output::Print(L" before: type = 0x%p, type handler = 0x%p, old singleton = 0x%p(0x%p)\n",

oldType, oldTypeHandler, oldSingletonInstanceBefore, oldSingletonInstanceBefore != nullptr ? oldSingletonInstanceBefore->Get() : nullptr);

Output::Print(L" fixed fields:");

oldTypeHandler->DumpFixedFields();

Output::Print(L"\n");

}

if (PHASE\_VERBOSE\_TESTTRACE1(FixMethodPropsPhase))

{

Output::Print(L"FixedFields: converting instance from %s to %s:\n", oldTypeHandlerName, newTypeHandlerName);

Output::Print(L" old singleton before %s null \n", oldSingletonInstanceBefore == nullptr ? L"==" : L"!=");

Output::Print(L" fixed fields before:");

oldTypeHandler->DumpFixedFields();

Output::Print(L"\n");

}

}

template <typename TPropertyIndex, typename TMapKey, bool IsNotExtensibleSupported>

void SimpleDictionaryTypeHandlerBase<TPropertyIndex, TMapKey, IsNotExtensibleSupported>::TraceFixedFieldsAfterTypeHandlerChange(

DynamicObject\* instance, DynamicTypeHandler\* oldTypeHandler, DynamicTypeHandler\* newTypeHandler,

DynamicType\* oldType, RecyclerWeakReference<DynamicObject>\* oldSingletonInstanceBefore)

{

if (PHASE\_VERBOSE\_TRACE1(FixMethodPropsPhase))

{

RecyclerWeakReference<DynamicObject>\* oldSingletonInstanceAfter = oldTypeHandler->GetSingletonInstance();

RecyclerWeakReference<DynamicObject>\* newSingletonInstanceAfter = newTypeHandler->GetSingletonInstance();

Output::Print(L" after: type = 0x%p, type handler = 0x%p, old singleton = 0x%p(0x%p), new singleton = 0x%p(0x%p)\n",

instance->GetType(), newTypeHandler, oldSingletonInstanceAfter, oldSingletonInstanceAfter != nullptr ? oldSingletonInstanceAfter->Get() : nullptr,

newSingletonInstanceAfter, newSingletonInstanceAfter != nullptr ? newSingletonInstanceAfter->Get() : nullptr);

Output::Print(L" fixed fields:");

newTypeHandler->DumpFixedFields();

Output::Print(L"\n");

Output::Flush();

}

if (PHASE\_VERBOSE\_TESTTRACE1(FixMethodPropsPhase))

{

Output::Print(L" type %s, typeHandler %s, old singleton after %s null (%s), new singleton after %s null\n",

oldTypeHandler != newTypeHandler ? L"changed" : L"unchanged",

oldType != instance->GetType() ? L"changed" : L"unchanged",

oldTypeHandler->GetSingletonInstance() == nullptr ? L"==" : L"!=",

oldSingletonInstanceBefore != oldTypeHandler->GetSingletonInstance() ? L"changed" : L"unchanged",

newTypeHandler->GetSingletonInstance() == nullptr ? L"==" : L"!=");

Output::Print(L" fixed fields after:");

newTypeHandler->DumpFixedFields();

Output::Print(L"\n");

Output::Flush();

}

}

template <typename TPropertyIndex, typename TMapKey, bool IsNotExtensibleSupported>

void SimpleDictionaryTypeHandlerBase<TPropertyIndex, TMapKey, IsNotExtensibleSupported>::TraceFixedFieldsBeforeSetIsProto(

DynamicObject\* instance, DynamicTypeHandler\* oldTypeHandler, DynamicType\* oldType, RecyclerWeakReference<DynamicObject>\* oldSingletonInstanceBefore)

{

if (PHASE\_VERBOSE\_TRACE1(FixMethodPropsPhase))

{

Output::Print(L"FixedFields: PathTypeHandler::SetIsPrototype(0x%p):\n", instance);

Output::Print(L" before: type = 0x%p, old singleton: 0x%p(0x%p)\n",

oldType, oldSingletonInstanceBefore, oldSingletonInstanceBefore != nullptr ? oldSingletonInstanceBefore->Get() : nullptr);

Output::Print(L" fixed fields:");

oldTypeHandler->DumpFixedFields();

Output::Print(L"\n");

}

if (PHASE\_VERBOSE\_TESTTRACE1(FixMethodPropsPhase))

{

Output::Print(L"FixedFields: PathTypeHandler::SetIsPrototype():\n");

Output::Print(L" old singleton before %s null \n", oldSingletonInstanceBefore == nullptr ? L"==" : L"!=");

Output::Print(L" fixed fields before:");

oldTypeHandler->DumpFixedFields();

Output::Print(L"\n");

}

}

template <typename TPropertyIndex, typename TMapKey, bool IsNotExtensibleSupported>

void SimpleDictionaryTypeHandlerBase<TPropertyIndex, TMapKey, IsNotExtensibleSupported>::TraceFixedFieldsAfterSetIsProto(

DynamicObject\* instance, DynamicTypeHandler\* oldTypeHandler, DynamicTypeHandler\* newTypeHandler,

DynamicType\* oldType, RecyclerWeakReference<DynamicObject>\* oldSingletonInstanceBefore)

{

if (PHASE\_VERBOSE\_TRACE1(FixMethodPropsPhase))

{

RecyclerWeakReference<DynamicObject>\* oldSingletonInstanceAfter = oldTypeHandler->GetSingletonInstance();

RecyclerWeakReference<DynamicObject>\* newSingletonInstanceAfter = newTypeHandler->GetSingletonInstance();

Output::Print(L" after: type = 0x%p, type handler = 0x%p, old singleton = 0x%p(0x%p), new singleton = 0x%p(0x%p)\n",

instance->GetType(), newTypeHandler,

oldSingletonInstanceAfter, oldSingletonInstanceAfter != nullptr ? oldSingletonInstanceAfter->Get() : nullptr,

newSingletonInstanceAfter, newSingletonInstanceAfter != nullptr ? newSingletonInstanceAfter->Get() : nullptr);

Output::Print(L" fixed fields:");

newTypeHandler->DumpFixedFields();

Output::Print(L"\n");

Output::Flush();

}

if (PHASE\_VERBOSE\_TESTTRACE1(FixMethodPropsPhase))

{

Output::Print(L" type %s, old singleton after %s null (%s)\n",

oldType != instance->GetType() ? L"changed" : L"unchanged",

oldSingletonInstanceBefore == nullptr ? L"==" : L"!=",

oldSingletonInstanceBefore != oldTypeHandler->GetSingletonInstance() ? L"changed" : L"unchanged");

Output::Print(L" fixed fields after:");

newTypeHandler->DumpFixedFields();

Output::Print(L"\n");

Output::Flush();

}

}

#endif

template <typename TPropertyIndex, typename TMapKey, bool IsNotExtensibleSupported>

typename SimpleDictionaryTypeHandlerBase<TPropertyIndex, TMapKey, IsNotExtensibleSupported>::BigSimpleDictionaryTypeHandler\* SimpleDictionaryTypeHandlerBase<TPropertyIndex, TMapKey, IsNotExtensibleSupported>::ConvertToBigSimpleDictionaryTypeHandler(DynamicObject\* instance)

{

return

isUnordered

? ConvertToSimpleDictionaryUnorderedTypeHandler<BigPropertyIndex, TMapKey, false>(instance)

: ConvertToTypeHandler<BigSimpleDictionaryTypeHandler, TMapKey>(instance);

}

template <>

BigSimpleDictionaryTypeHandler\* SimpleDictionaryTypeHandlerBase<BigPropertyIndex, const PropertyRecord\*, false>::ConvertToBigSimpleDictionaryTypeHandler(DynamicObject\* instance)

{

Throw::OutOfMemory();

}

template <>

BigSimpleDictionaryTypeHandler\* SimpleDictionaryTypeHandlerBase<BigPropertyIndex, const PropertyRecord\*, true>::ConvertToBigSimpleDictionaryTypeHandler(DynamicObject\* instance)

{

Throw::OutOfMemory();

}

template <typename TPropertyIndex, typename TMapKey, bool IsNotExtensibleSupported>

void SimpleDictionaryTypeHandlerBase<TPropertyIndex, TMapKey, IsNotExtensibleSupported>::SetPropertyValueInfo(PropertyValueInfo\* info, RecyclableObject\* instance, TPropertyIndex propIndex, PropertyAttributes attributes, InlineCacheFlags flags)

{

PropertyValueInfo::Set(info, instance, propIndex, attributes, flags);

}

template <>

void SimpleDictionaryTypeHandlerBase<BigPropertyIndex, const PropertyRecord\*, false>::SetPropertyValueInfo(PropertyValueInfo\* info, RecyclableObject\* instance, BigPropertyIndex propIndex, PropertyAttributes attributes, InlineCacheFlags flags)

{

PropertyValueInfo::SetNoCache(info, instance);

}

template <>

void SimpleDictionaryTypeHandlerBase<BigPropertyIndex, const PropertyRecord\*, true>::SetPropertyValueInfo(PropertyValueInfo\* info, RecyclableObject\* instance, BigPropertyIndex propIndex, PropertyAttributes attributes, InlineCacheFlags flags)

{

PropertyValueInfo::SetNoCache(info, instance);

}

template <>

void SimpleDictionaryTypeHandlerBase<BigPropertyIndex, JavascriptString\*, false>::SetPropertyValueInfo(PropertyValueInfo\* info, RecyclableObject\* instance, BigPropertyIndex propIndex, PropertyAttributes attributes, InlineCacheFlags flags)

{

PropertyValueInfo::SetNoCache(info, instance);

}

template <>

void SimpleDictionaryTypeHandlerBase<BigPropertyIndex, JavascriptString\*, true>::SetPropertyValueInfo(PropertyValueInfo\* info, RecyclableObject\* instance, BigPropertyIndex propIndex, PropertyAttributes attributes, InlineCacheFlags flags)

{

PropertyValueInfo::SetNoCache(info, instance);

}

template class SimpleDictionaryTypeHandlerBase<PropertyIndex, const PropertyRecord\*, false>;

template class SimpleDictionaryTypeHandlerBase<PropertyIndex, const PropertyRecord\*, true>;

template class SimpleDictionaryTypeHandlerBase<BigPropertyIndex, const PropertyRecord\*, false>;

template class SimpleDictionaryTypeHandlerBase<BigPropertyIndex, const PropertyRecord\*, true>;

template class SimpleDictionaryTypeHandlerBase<PropertyIndex, JavascriptString\*, false>;

template class SimpleDictionaryTypeHandlerBase<PropertyIndex, JavascriptString\*, true>;

template class SimpleDictionaryTypeHandlerBase<BigPropertyIndex, JavascriptString\*, false>;

template class SimpleDictionaryTypeHandlerBase<BigPropertyIndex, JavascriptString\*, true>;

}

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

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

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

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

//

// This type handler supports storage of a map of property Id to index along with

// property descriptors for every property.

//

// It can be shared across objects as long as no properties are

// added, deleted or changed. Once the property structure is changed the type is marked as

// non-shared. In non-shared state, any property structural changes can happen without creating

// a new type or handler.

//

// Type transition to DictionaryTypeHandler happens on the use of setters.

//

#pragma once

namespace Js

{

template <typename TMapKey>

struct PropertyMapKeyTraits

{

};

template <>

struct PropertyMapKeyTraits<const PropertyRecord\*>

{

template <typename TKey, typename TValue>

class Entry : public JsUtil::SimpleDictionaryEntry<TKey, TValue> { };

static bool IsStringTypeHandler() { return false; };

};

template <>

struct PropertyMapKeyTraits<JavascriptString\*>

{

template <typename TKey, typename TValue>

class Entry : public JsUtil::DictionaryEntry<TKey, TValue> { };

static bool IsStringTypeHandler() { return true; };

};

// Template parameters:

// - TPropertyIndex: property index type: PropertyIndex, BigPropertyIndex, etc.

// - TMapKey: key type for property map: PropertyRecord\* const, JavascriptString\*

// - IsNotExtensibleSupported: whether the following features are supported preventExtensions, seal, freeze.

template <typename TPropertyIndex, typename TMapKey, bool IsNotExtensibleSupported>

class SimpleDictionaryTypeHandlerBase : public DynamicTypeHandler

{

private:

friend class NullTypeHandlerBase;

friend class DeferredTypeHandlerBase;

friend class PathTypeHandlerBase;

template<size\_t size>

friend class SimpleTypeHandler;

template <typename TPropertyIndex, typename TMapKey, bool IsNotExtensibleSupported> friend class SimpleDictionaryTypeHandlerBase;

// Explicit non leaf allocator now that the key is non-leaf

typedef JsUtil::BaseDictionary<TMapKey, SimpleDictionaryPropertyDescriptor<TPropertyIndex>, RecyclerNonLeafAllocator, DictionarySizePolicy<PowerOf2Policy, 1>, PropertyRecordStringHashComparer, PropertyMapKeyTraits<TMapKey>::Entry>

SimplePropertyDescriptorMap;

typedef SimplePropertyDescriptorMap PropertyDescriptorMapType; // alias used by diagnostics

protected:

SimplePropertyDescriptorMap\* propertyMap;

private:

RecyclerWeakReference<DynamicObject>\* singletonInstance;

TPropertyIndex nextPropertyIndex;

protected:

// Determines whether this instance is actually a SimpleDictionaryUnorderedTypeHandler

bool isUnordered : 1;

// Tracks if an InternalPropertyRecord or symbol has been added to this type; will prevent conversion to string-keyed type handler

bool hasNamelessPropertyId : 1;

private:

// Number of deleted properties in the property map

byte numDeletedProperties;

public:

DEFINE\_GETCPPNAME();

protected:

SimpleDictionaryTypeHandlerBase(Recycler \* recycler);

SimpleDictionaryTypeHandlerBase(Recycler \* recycler, int slotCapacity, uint16 inlineSlotCapacity, uint16 offsetOfInlineSlots, bool isLocked = false, bool isShared = false);

SimpleDictionaryTypeHandlerBase(ScriptContext \* scriptContext, SimplePropertyDescriptor const\* propertyDescriptors, int propertyCount, int slotCapacity, uint16 inlineSlotCapacity, uint16 offsetOfInlineSlots, bool isLocked = false, bool isShared = false);

SimpleDictionaryTypeHandlerBase(Recycler\* recycler, int slotCapacity, int propertyCapacity, uint16 inlineSlotCapacity, uint16 offsetOfInlineSlots, bool isLocked = false, bool isShared = false);

DEFINE\_VTABLE\_CTOR\_NO\_REGISTER(SimpleDictionaryTypeHandlerBase, DynamicTypeHandler);

typedef PropertyIndexRanges<TPropertyIndex> PropertyIndexRangesType;

static const TPropertyIndex MaxPropertyIndexSize = PropertyIndexRangesType::MaxValue;

static const TPropertyIndex NoSlots = PropertyIndexRangesType::NoSlots;

public:

typedef TPropertyIndex PropertyIndexType;

// Create a new type handler for a future DynamicObject. This is for public usage. "initialCapacity" indicates desired slotCapacity, subject to alignment round up.

static SimpleDictionaryTypeHandlerBase \* New(Recycler \* recycler, int initialCapacity, uint16 inlineSlotCapacity, uint16 offsetOfInlineSlots, bool isLocked = false, bool isShared = false);

// Create a new type handler for a future DynamicObject. This is for public usage. "propertyCount" indicates desired slotCapacity, subject to alignment round up.

static SimpleDictionaryTypeHandlerBase \* New(ScriptContext \* scriptContext, SimplePropertyDescriptor const\* propertyDescriptors, int propertyCount, uint16 inlineSlotCapacity, uint16 offsetOfInlineSlots, bool isLocked = false, bool isShared = false);

static DynamicType\* CreateTypeForNewScObject(ScriptContext\* scriptContext, DynamicType\* type, const Js::PropertyIdArray \*propIds, bool shareType, bool check\_\_proto\_\_);

virtual BOOL IsStringTypeHandler() const override { return PropertyMapKeyTraits<TMapKey>::IsStringTypeHandler(); }

virtual BOOL IsLockable() const override { return true; }

virtual BOOL IsSharable() const override { return true; }

virtual void DoShareTypeHandler(ScriptContext\* scriptContext) override;

virtual int GetPropertyCount() override;

virtual PropertyId GetPropertyId(ScriptContext\* scriptContext, PropertyIndex index) override;

virtual PropertyId GetPropertyId(ScriptContext\* scriptContext, BigPropertyIndex index) override;

virtual PropertyIndex GetPropertyIndex(const PropertyRecord\* propertyRecord) override;

virtual bool GetPropertyEquivalenceInfo(PropertyRecord const\* propertyRecord, PropertyEquivalenceInfo& info) override;

virtual bool IsObjTypeSpecEquivalent(const Type\* type, const TypeEquivalenceRecord& record, uint& failedPropertyIndex) override;

virtual bool IsObjTypeSpecEquivalent(const Type\* type, const EquivalentPropertyEntry\* entry) override;

virtual BOOL FindNextProperty(ScriptContext\* scriptContext, PropertyIndex& index, JavascriptString\*\* propertyString,

PropertyId\* propertyId, PropertyAttributes\* attributes, Type\* type, DynamicType \*typeToEnumerate, bool requireEnumerable, bool enumSymbols = false) override;

virtual BOOL FindNextProperty(ScriptContext\* scriptContext, BigPropertyIndex& index, JavascriptString\*\* propertyString,

PropertyId\* propertyId, PropertyAttributes\* attributes, Type\* type, DynamicType \*typeToEnumerate, bool requireEnumerable, bool enumSymbols = false) override;

virtual BOOL HasProperty(DynamicObject\* instance, PropertyId propertyId, bool \*noRedecl = nullptr) override;

virtual BOOL HasProperty(DynamicObject\* instance, JavascriptString\* propertyNameString) override;

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

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

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

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

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

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

virtual BOOL DeleteProperty(DynamicObject\* instance, PropertyId propertyId, PropertyOperationFlags flags) override sealed;

virtual PropertyIndex GetRootPropertyIndex(const PropertyRecord\* propertyRecord) override;

virtual BOOL HasRootProperty(DynamicObject\* instance, PropertyId propertyId, bool \*noRedecl, bool \*pDeclaredProperty = nullptr) override;

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

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

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

virtual BOOL DeleteRootProperty(DynamicObject\* instance, PropertyId propertyId, PropertyOperationFlags flags) override;

virtual BOOL IsSimpleDictionaryTypeHandler() const {return TRUE;}

#if DBG

virtual bool IsLetConstGlobal(DynamicObject\* instance, PropertyId propertyId) override;

#endif

virtual bool NextLetConstGlobal(int& index, RootObjectBase\* instance, const PropertyRecord\*\* propertyRecord, Var\* value, bool\* isConst) override;

virtual BOOL IsFixedProperty(const DynamicObject\* instance, PropertyId propertyId) override;

virtual BOOL IsEnumerable(DynamicObject\* instance, PropertyId propertyId) override;

virtual BOOL IsWritable(DynamicObject\* instance, PropertyId propertyId) override;

virtual BOOL IsConfigurable(DynamicObject\* instance, PropertyId propertyId) override;

virtual BOOL SetEnumerable(DynamicObject\* instance, PropertyId propertyId, BOOL value) override;

virtual BOOL SetWritable(DynamicObject\* instance, PropertyId propertyId, BOOL value) override;

virtual BOOL SetConfigurable(DynamicObject\* instance, PropertyId propertyId, BOOL value) override sealed;

virtual BOOL SetAccessors(DynamicObject\* instance, PropertyId propertyId, Var getter, Var setter, PropertyOperationFlags flags = PropertyOperation\_None) override;

virtual BOOL PreventExtensions(DynamicObject \*instance) override;

virtual BOOL Seal(DynamicObject\* instance) override;

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

virtual BOOL IsSealed(DynamicObject\* instance) override;

virtual BOOL IsFrozen(DynamicObject\* instance) override;

virtual BOOL SetAttributes(DynamicObject\* instance, PropertyId propertyId, PropertyAttributes attributes) override sealed;

virtual BOOL GetAttributesWithPropertyIndex(DynamicObject \* instance, PropertyId propertyId, BigPropertyIndex index, PropertyAttributes \* attributes) override;

virtual BOOL SetItem(DynamicObject\* instance, uint32 index, Var value, PropertyOperationFlags flags) override sealed;

virtual void SetAllPropertiesToUndefined(DynamicObject\* instance, bool invalidateFixedFields) override;

virtual void MarshalAllPropertiesToScriptContext(DynamicObject\* instance, ScriptContext\* targetScriptContext, bool invalidateFixedFields) override;

virtual DynamicTypeHandler\* ConvertToTypeWithItemAttributes(DynamicObject\* instance) override;

virtual void SetIsPrototype(DynamicObject\* instance) override;

#if DBG

virtual bool SupportsPrototypeInstances() const { return true; }

#endif

virtual bool HasSingletonInstance() const override sealed;

virtual bool TryUseFixedProperty(PropertyRecord const \* propertyRecord, Var \* pProperty, FixedPropertyKind propertyType, ScriptContext \* requestContext) override;

virtual bool TryUseFixedAccessor(PropertyRecord const \* propertyRecord, Var \* pAccessor, FixedPropertyKind propertyType, bool getter, ScriptContext \* requestContext) override;

#if DBG

virtual bool CanStorePropertyValueDirectly(const DynamicObject\* instance, PropertyId propertyId, bool allowLetConst) override;

virtual bool CheckFixedProperty(PropertyRecord const \* propertyRecord, Var \* pProperty, ScriptContext \* requestContext) override;

virtual bool HasAnyFixedProperties() const override;

#endif

#ifdef ENABLE\_DEBUG\_CONFIG\_OPTIONS

virtual void DumpFixedFields() const override;

static void TraceFixedFieldsBeforeTypeHandlerChange(

const wchar\_t\* oldTypeHandlerName, const wchar\_t\* newTypeHandlerName,

DynamicObject\* instance, DynamicTypeHandler\* oldTypeHandler, DynamicType\* oldType, RecyclerWeakReference<DynamicObject>\* oldSingletonInstanceBefore);

static void TraceFixedFieldsAfterTypeHandlerChange(

DynamicObject\* instance, DynamicTypeHandler\* oldTypeHandler, DynamicTypeHandler\* newTypeHandler,

DynamicType\* oldType, RecyclerWeakReference<DynamicObject>\* oldSingletonInstanceBefore);

static void TraceFixedFieldsBeforeSetIsProto(

DynamicObject\* instance, DynamicTypeHandler\* oldTypeHandler, DynamicType\* oldType, RecyclerWeakReference<DynamicObject>\* oldSingletonInstanceBefore);

static void TraceFixedFieldsAfterSetIsProto(

DynamicObject\* instance, DynamicTypeHandler\* oldTypeHandler, DynamicTypeHandler\* newTypeHandler,

DynamicType\* oldType, RecyclerWeakReference<DynamicObject>\* oldSingletonInstanceBefore);

#endif

private:

typedef SimpleDictionaryTypeHandlerBase<BigPropertyIndex, TMapKey, false> BigSimpleDictionaryTypeHandler;

template <bool doLock>

bool IsObjTypeSpecEquivalentImpl(const Type\* type, const EquivalentPropertyEntry \*entry);

template <bool allowNonExistent, bool markAsUsed>

bool TryGetFixedProperty(PropertyRecord const \* propertyRecord, Var \* pProperty, FixedPropertyKind propertyType, ScriptContext \* requestContext);

public:

virtual RecyclerWeakReference<DynamicObject>\* GetSingletonInstance() const sealed { Assert(HasSingletonInstanceOnlyIfNeeded()); return this->singletonInstance; }

virtual void SetSingletonInstanceUnchecked(RecyclerWeakReference<DynamicObject>\* instance) override

{

Assert(!GetIsShared());

Assert(this->singletonInstance == nullptr);

this->singletonInstance = instance;

}

virtual void ClearSingletonInstance() override sealed

{

Assert(HasSingletonInstanceOnlyIfNeeded());

this->singletonInstance = nullptr;

}

#if DBG

bool HasSingletonInstanceOnlyIfNeeded() const

{

return AreSingletonInstancesNeeded() || this->singletonInstance == nullptr;

}

#endif

private:

void SetIsPrototype(DynamicObject\* instance, bool hasNewType);

template <typename TPropertyKey>

void InvalidateFixedField(const TPropertyKey propertyKey, SimpleDictionaryPropertyDescriptor<TPropertyIndex>\* descriptor, ScriptContext\* scriptContext);

bool SupportsSwitchingToUnordered(const ScriptContext \*const scriptContext) const;

SimpleDictionaryUnorderedTypeHandler<TPropertyIndex, TMapKey, IsNotExtensibleSupported> \*AsUnordered();

void SetNumDeletedProperties(const byte n);

template <typename U, typename UMapKey>

U\* ConvertToTypeHandler(DynamicObject\* instance);

template <typename TPropertyKey>

void Add(TPropertyKey propertyKey, PropertyAttributes attributes, ScriptContext \*const scriptContext);

template <typename TPropertyKey>

void Add(TPropertyKey propertyKey, PropertyAttributes attributes, bool isInitialized, bool isFixed, bool usedAsFixed, ScriptContext \*const scriptContext);

template <typename TPropertyKey>

void Add(TPropertyIndex propertyIndex, TPropertyKey propertyKey, PropertyAttributes attributes, ScriptContext \*const scriptContext);

template <typename TPropertyKey>

void Add(TPropertyIndex propertyIndex, TPropertyKey propertyKey, PropertyAttributes attributes, bool isIntiailized, bool isFixed, bool usedAsFixed, ScriptContext \*const scriptContext);

DictionaryTypeHandlerBase<TPropertyIndex>\* ConvertToDictionaryType(DynamicObject\* instance);

ES5ArrayTypeHandlerBase<TPropertyIndex>\* ConvertToES5ArrayType(DynamicObject\* instance);

SimpleDictionaryTypeHandlerBase\* ConvertToNonSharedSimpleDictionaryType(DynamicObject\* instance);

template <typename NewTPropertyIndex, typename NewTMapKey, bool NewIsNotExtensibleSupported> SimpleDictionaryUnorderedTypeHandler<NewTPropertyIndex, NewTMapKey, NewIsNotExtensibleSupported>\* ConvertToSimpleDictionaryUnorderedTypeHandler(DynamicObject\* instance);

BOOL SetAttribute(DynamicObject\* instance, SimpleDictionaryPropertyDescriptor<TPropertyIndex>\* descriptor, PropertyAttributes attribute);

BOOL ClearAttribute(DynamicObject\* instance, SimpleDictionaryPropertyDescriptor<TPropertyIndex>\* descriptor, PropertyAttributes attribute);

void EnsureSlotCapacity(DynamicObject \* instance);

template <typename TPropertyKey>

BOOL AddProperty(DynamicObject\* instance, TPropertyKey propertyKey, Var value, PropertyAttributes attributes, PropertyValueInfo\* info, PropertyOperationFlags flags, SideEffects possibleSideEffects);

virtual BOOL FreezeImpl(DynamicObject\* instance, bool isConvertedType) override;

template <bool allowLetConstGlobal>

\_\_inline BOOL HasProperty\_Internal(DynamicObject\* instance, PropertyId propertyId, bool \*noRedecl, bool \*pDeclaredProperty);

template <bool allowLetConstGlobal>

\_\_inline PropertyIndex GetPropertyIndex\_Internal(const PropertyRecord\* propertyRecord);

template <bool allowLetConstGlobal>

\_\_inline BOOL GetProperty\_Internal(DynamicObject\* instance, Var originalInstance, PropertyId propertyId, Var\* value, PropertyValueInfo\* info, ScriptContext\* requestContext);

template <bool allowLetConstGlobal>

\_\_inline BOOL SetProperty\_Internal(DynamicObject\* instance, PropertyId propertyId, Var value, PropertyOperationFlags flags, PropertyValueInfo\* info);

template <bool allowLetConstGlobal>

\_\_inline DescriptorFlags GetSetter\_Internal(DynamicObject\* instance, PropertyId propertyId, Var\* setterValue, PropertyValueInfo\* info, ScriptContext\* requestContext);

template <bool allowLetConstGlobal>

\_\_inline BOOL DeleteProperty\_Internal(DynamicObject\* instance, PropertyId propertyId, PropertyOperationFlags flags);

template <bool allowLetConstGlobal>

\_\_inline BOOL GetPropertyFromDescriptor(DynamicObject\* instance, SimpleDictionaryPropertyDescriptor<TPropertyIndex>\* descriptor, Var\* value, PropertyValueInfo\* info);

template <bool allowLetConstGlobal, typename TPropertyKey>

\_\_inline BOOL SetPropertyFromDescriptor(DynamicObject\* instance, PropertyId propertyId, TPropertyKey propertyKey, SimpleDictionaryPropertyDescriptor<TPropertyIndex>\* descriptor, Var value, PropertyOperationFlags flags, PropertyValueInfo\* info);

template <bool allowLetConstGlobal>

\_\_inline DescriptorFlags GetSetterFromDescriptor(SimpleDictionaryPropertyDescriptor<TPropertyIndex>\* descriptor);

BOOL SetProperty\_JavascriptString(DynamicObject\* instance, JavascriptString\* propertyNameString, Var value, PropertyOperationFlags flags, PropertyValueInfo\* info, TemplateParameter::Box<const PropertyRecord\*>);

BOOL SetProperty\_JavascriptString(DynamicObject\* instance, JavascriptString\* propertyNameString, Var value, PropertyOperationFlags flags, PropertyValueInfo\* info, TemplateParameter::Box<JavascriptString\*>);

BigSimpleDictionaryTypeHandler\* ConvertToBigSimpleDictionaryTypeHandler(DynamicObject\* instance);

void SetPropertyValueInfo(PropertyValueInfo\* info, RecyclableObject\* instance, TPropertyIndex propIndex, PropertyAttributes attributes, InlineCacheFlags flags = InlineCacheNoFlags);

BOOL PreventExtensionsInternal(DynamicObject\* instance);

BOOL SealInternal(DynamicObject\* instance);

BOOL FreezeInternal(DynamicObject\* instance, bool isConvertedType = false);

// This was added to work around not being able to specify partial template specialization of member function.

BOOL FindNextProperty\_BigPropertyIndex(ScriptContext\* scriptContext, TPropertyIndex& index, JavascriptString\*\* propertyString,

PropertyId\* propertyId, PropertyAttributes\* attributes, Type\* type, DynamicType \*typeToEnumerate, bool requireEnumerable, bool enumSymbols = false);

template <bool check\_\_proto\_\_>

static DynamicType\* InternalCreateTypeForNewScObject(ScriptContext\* scriptContext, DynamicType\* type, const Js::PropertyIdArray \*propIds, bool shareType);

};

}

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

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

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

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

#include "RuntimeTypePch.h"

namespace Js

{

template<class TPropertyIndex, class TMapKey, bool IsNotExtensibleSupported>

SimpleDictionaryUnorderedTypeHandler<TPropertyIndex, TMapKey, IsNotExtensibleSupported>::SimpleDictionaryUnorderedTypeHandler(Recycler \* recycler, int slotCapacity, uint16 inlineSlotCapacity, uint16 offsetOfInlineSlots)

: SimpleDictionaryTypeHandlerBase(recycler, slotCapacity, inlineSlotCapacity, offsetOfInlineSlots),

deletedPropertyIndex(NoSlots)

{

isUnordered = true;

}

template<class TPropertyIndex, class TMapKey, bool IsNotExtensibleSupported>

SimpleDictionaryUnorderedTypeHandler<TPropertyIndex, TMapKey, IsNotExtensibleSupported>::SimpleDictionaryUnorderedTypeHandler(ScriptContext \* scriptContext, SimplePropertyDescriptor\* propertyDescriptors, int propertyCount, int slotCapacity, uint16 inlineSlotCapacity, uint16 offsetOfInlineSlots)

: SimpleDictionaryTypeHandlerBase(scriptContext, propertyDescriptors, propertyCount, slotCapacity, inlineSlotCapacity, offsetOfInlineSlots),

deletedPropertyIndex(NoSlots)

{

isUnordered = true;

}

template<class TPropertyIndex, class TMapKey, bool IsNotExtensibleSupported>

SimpleDictionaryUnorderedTypeHandler<TPropertyIndex, TMapKey, IsNotExtensibleSupported>::SimpleDictionaryUnorderedTypeHandler(Recycler\* recycler, int slotCapacity, int propertyCapacity, uint16 inlineSlotCapacity, uint16 offsetOfInlineSlots)

: SimpleDictionaryTypeHandlerBase(recycler, slotCapacity, propertyCapacity, inlineSlotCapacity, offsetOfInlineSlots),

deletedPropertyIndex(NoSlots)

{

isUnordered = true;

}

template<class TPropertyIndex, class TMapKey, bool IsNotExtensibleSupported>

bool SimpleDictionaryUnorderedTypeHandler<TPropertyIndex, TMapKey, IsNotExtensibleSupported>::IsReusablePropertyIndex(const TPropertyIndex propertyIndex)

{

// When reusing a deleted property index, we will treat the property index as the dictionary index, so they must be the

// same. Also, property indexes stored in the object's slot corresponding to a deleted property are tagged so that they

// don't look like pointers. If the property index is too large, it will not be free-listed.

return

static\_cast<int>(propertyIndex) >= 0 &&

static\_cast<int>(propertyIndex) < propertyMap->Count() &&

propertyMap->GetValueAt(propertyIndex).propertyIndex == propertyIndex &&

!TaggedInt::IsOverflow(propertyIndex);

}

template<class TPropertyIndex, class TMapKey, bool IsNotExtensibleSupported>

bool SimpleDictionaryUnorderedTypeHandler<TPropertyIndex, TMapKey, IsNotExtensibleSupported>::TryRegisterDeletedPropertyIndex(

DynamicObject \*const object,

const TPropertyIndex propertyIndex)

{

Assert(object);

if(!IsReusablePropertyIndex(propertyIndex))

{

return false;

}

Assert(!TaggedInt::IsOverflow(NoSlots)); // the last deleted property's slot in the chain is going to store NoSlots as a tagged int

SetSlotUnchecked(object, propertyIndex, TaggedInt::ToVarUnchecked(deletedPropertyIndex));

deletedPropertyIndex = propertyIndex;

return true;

}

template<class TPropertyIndex, class TMapKey, bool IsNotExtensibleSupported>

bool SimpleDictionaryUnorderedTypeHandler<TPropertyIndex, TMapKey, IsNotExtensibleSupported>::TryReuseDeletedPropertyIndex(

DynamicObject \*const object,

TPropertyIndex \*const propertyIndex)

{

Assert(object);

Assert(propertyIndex);

if(deletedPropertyIndex == NoSlots)

{

return false;

}

Assert(propertyMap->GetValueAt(deletedPropertyIndex).propertyIndex == deletedPropertyIndex);

Assert(propertyMap->GetValueAt(deletedPropertyIndex).Attributes & PropertyDeleted);

\*propertyIndex = deletedPropertyIndex;

deletedPropertyIndex = static\_cast<TPropertyIndex>(TaggedInt::ToInt32(object->GetSlot(deletedPropertyIndex)));

return true;

}

template<class TPropertyIndex, class TMapKey, bool IsNotExtensibleSupported>

bool SimpleDictionaryUnorderedTypeHandler<TPropertyIndex, TMapKey, IsNotExtensibleSupported>::TryUndeleteProperty(

DynamicObject \*const object,

const TPropertyIndex existingPropertyIndex,

TPropertyIndex \*const propertyIndex)

{

Assert(object);

Assert(propertyIndex);

if(!IsReusablePropertyIndex(existingPropertyIndex))

{

return false;

}

Assert(propertyMap->GetValueAt(existingPropertyIndex).propertyIndex == existingPropertyIndex);

Assert(propertyMap->GetValueAt(existingPropertyIndex).Attributes & PropertyDeleted);

const bool reused = TryReuseDeletedPropertyIndex(object, propertyIndex);

Assert(reused); // at least one property index must have been free-listed since we're adding an existing deleted property

if(\*propertyIndex == existingPropertyIndex)

{

// The deleted property index that is being added is the first deleted property index in the free-list

return true;

}

// We're trying to add a deleted property index that is currently somewhere in the middle of the free-list chain. To

// avoid rebuilding the free-list, swap the property descriptor with the one for the first deleted property index in the

// free-list. Since we also need to make sure that each descriptor's property index is the same as its dictionary entry

// index, we need to remove them from the dictionary and add them back in the same order, which actually adds them in

// reverse order. This relies on the fact that BaseDictionary first reuses the last-deleted entry index in its

// free-listing strategy. Should remove this dependence in the future.

TMapKey propertyKeyToPreserve = propertyMap->GetKeyAt(\*propertyIndex);

SimpleDictionaryPropertyDescriptor<TPropertyIndex> descriptorToPreserve = propertyMap->GetValueAt(\*propertyIndex);

descriptorToPreserve.propertyIndex = existingPropertyIndex;

TMapKey propertyKeyToReuse = propertyMap->GetKeyAt(existingPropertyIndex);

SimpleDictionaryPropertyDescriptor<TPropertyIndex> descriptorToReuse = propertyMap->GetValueAt(existingPropertyIndex);

descriptorToReuse.propertyIndex = \*propertyIndex;

propertyMap->Remove(propertyKeyToPreserve);

propertyMap->Remove(propertyKeyToReuse);

int dictionaryIndex = propertyMap->Add(propertyKeyToPreserve, descriptorToPreserve);

Assert(dictionaryIndex == existingPropertyIndex);

dictionaryIndex = propertyMap->Add(propertyKeyToReuse, descriptorToReuse);

Assert(dictionaryIndex == \*propertyIndex);

return true;

}

template class SimpleDictionaryUnorderedTypeHandler<PropertyIndex, const PropertyRecord\*, false>;

template class SimpleDictionaryUnorderedTypeHandler<PropertyIndex, const PropertyRecord\*, true>;

template class SimpleDictionaryUnorderedTypeHandler<BigPropertyIndex, const PropertyRecord\*, false>;

template class SimpleDictionaryUnorderedTypeHandler<BigPropertyIndex, const PropertyRecord\*, true>;

template class SimpleDictionaryUnorderedTypeHandler<PropertyIndex, JavascriptString\*, false>;

template class SimpleDictionaryUnorderedTypeHandler<PropertyIndex, JavascriptString\*, true>;

template class SimpleDictionaryUnorderedTypeHandler<BigPropertyIndex, JavascriptString\*, false>;

template class SimpleDictionaryUnorderedTypeHandler<BigPropertyIndex, JavascriptString\*, true>;

}

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

// 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

{

// This is an unordered type handler (enumeration order of properties is nondeterministic). It is used when an object using

// a SimpleDictionaryTypeHandler is determined to be used like a hashtable, to prevent unbounded memory growth.

//

// An object that is used as a hashtable will typically have a number of property adds and deletes, where the added

// properties often don't have the same property name as a previously deleted property. Since SimpleDictionaryTypeHandler

// does not remove deleted properties from its property map, and preserves the slot in the object (to preserve

// enumeration order if a deleted property is added back), the property map and object continue to grow in size as the

// script continues to add more property IDs to the object, even if there are corresponding deletes on different property

// IDs to make room.

//

// At some point, SimpleDictionaryTypeHandler determines that it needs to stop the unbounded growth and start to reuse

// property indexes from deleted properties for new properties, even if the property ID is different. That is when it

// transitions into this unordered type handler.

template<class TPropertyIndex, class TMapKey, bool IsNotExtensibleSupported>

class SimpleDictionaryUnorderedTypeHandler sealed : public SimpleDictionaryTypeHandlerBase<TPropertyIndex, TMapKey, IsNotExtensibleSupported>

{

template <typename TPropertyIndex, typename TMapKey, bool IsNotExtensibleSupported> friend class SimpleDictionaryUnorderedTypeHandler;

template <typename TPropertyIndex, typename TMapKey, bool IsNotExtensibleSupported> friend class SimpleDictionaryTypeHandlerBase;

private:

// A deleted property ID that will be reused for the next property add. The object's slot corresponding to this property

// ID will have a tagged int that is the next deleted property ID in the chain.

TPropertyIndex deletedPropertyIndex;

public:

DEFINE\_GETCPPNAME();

public:

SimpleDictionaryUnorderedTypeHandler(Recycler \* recycler, int slotCapacity, uint16 inlineSlotCapacity, uint16 offsetOfInlineSlots);

SimpleDictionaryUnorderedTypeHandler(ScriptContext \* scriptContext, SimplePropertyDescriptor\* propertyDescriptors, int propertyCount, int slotCapacity, uint16 inlineSlotCapacity, uint16 offsetOfInlineSlots);

SimpleDictionaryUnorderedTypeHandler(Recycler\* recycler, int slotCapacity, int propertyCapacity, uint16 inlineSlotCapacity, uint16 offsetOfInlineSlots);

DEFINE\_VTABLE\_CTOR\_NO\_REGISTER(SimpleDictionaryUnorderedTypeHandler, SimpleDictionaryTypeHandlerBase);

private:

template<class OtherTPropertyIndex, class OtherTMapKey, bool OtherIsNotExtensibleSupported>

void CopyUnorderedStateFrom(const SimpleDictionaryUnorderedTypeHandler<OtherTPropertyIndex, OtherTMapKey, OtherIsNotExtensibleSupported> &other)

{

CompileAssert(sizeof(TPropertyIndex) >= sizeof(OtherTPropertyIndex));

if (other.deletedPropertyIndex != PropertyIndexRanges<OtherTPropertyIndex>::NoSlots)

{

deletedPropertyIndex = other.deletedPropertyIndex;

}

}

bool IsReusablePropertyIndex(const TPropertyIndex propertyIndex);

bool TryRegisterDeletedPropertyIndex(DynamicObject \*const object, const TPropertyIndex propertyIndex);

bool TryReuseDeletedPropertyIndex(DynamicObject \*const object, TPropertyIndex \*const propertyIndex);

bool TryUndeleteProperty(

DynamicObject \*const object,

const TPropertyIndex existingPropertyIndex,

TPropertyIndex \*const propertyIndex);

};

}

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

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

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

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

#pragma once

namespace Js

{

class SimplePropertyDescriptor

{

public:

SimplePropertyDescriptor() : Id(NULL), preventFalseReference(NULL) { Attributes = PropertyDynamicTypeDefaults; }

SimplePropertyDescriptor(const PropertyRecord\* id) : Id(id), preventFalseReference(NULL) { Attributes = PropertyDynamicTypeDefaults; }

SimplePropertyDescriptor(const PropertyRecord\* id, PropertyAttributes attributes) : Id(id), preventFalseReference(NULL) { Attributes = attributes; }

const PropertyRecord\* Id;

union

{

PropertyAttributes Attributes;

void\* preventFalseReference; // SimplePropertyDescriptor can be declared on stack. Always zero out to avoid this becoming a memory address reference.

};

};

CompileAssert(sizeof(SimplePropertyDescriptor) == 2 \* sizeof(Var));

}

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

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

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

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

#include "RuntimeTypePch.h"

#include "Types\NullTypeHandler.h"

#include "Types\SimpleTypeHandler.h"

namespace Js

{

template<size\_t size>

SimpleTypeHandler<size>::SimpleTypeHandler(SimpleTypeHandler<size> \* typeHandler)

: DynamicTypeHandler(sizeof(descriptors) / sizeof(SimplePropertyDescriptor),

typeHandler->GetInlineSlotCapacity(), typeHandler->GetOffsetOfInlineSlots()), propertyCount(typeHandler->propertyCount)

{

Assert(typeHandler->GetIsInlineSlotCapacityLocked());

SetIsInlineSlotCapacityLocked();

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

{

descriptors[i] = typeHandler->descriptors[i];

}

}

template<size\_t size>

SimpleTypeHandler<size>::SimpleTypeHandler(Recycler\*) :

DynamicTypeHandler(sizeof(descriptors) / sizeof(SimplePropertyDescriptor)),

propertyCount(0)

{

SetIsInlineSlotCapacityLocked();

}

template<size\_t size>

SimpleTypeHandler<size>::SimpleTypeHandler(const PropertyRecord\* id, PropertyAttributes attributes, PropertyTypes propertyTypes, uint16 inlineSlotCapacity, uint16 offsetOfInlineSlots) :

DynamicTypeHandler(sizeof(descriptors) / sizeof(SimplePropertyDescriptor),

inlineSlotCapacity, offsetOfInlineSlots, DefaultFlags | IsLockedFlag | MayBecomeSharedFlag | IsSharedFlag), propertyCount(1)

{

Assert((attributes & PropertyDeleted) == 0);

descriptors[0].Id = id;

descriptors[0].Attributes = attributes;

Assert((propertyTypes & (PropertyTypesAll & ~PropertyTypesWritableDataOnly)) == 0);

SetPropertyTypes(PropertyTypesWritableDataOnly, propertyTypes);

SetIsInlineSlotCapacityLocked();

}

template<size\_t size>

SimpleTypeHandler<size>::SimpleTypeHandler(SimplePropertyDescriptor const (&SharedFunctionPropertyDescriptors)[size], PropertyTypes propertyTypes, uint16 inlineSlotCapacity, uint16 offsetOfInlineSlots) :

DynamicTypeHandler(sizeof(descriptors) / sizeof(SimplePropertyDescriptor),

inlineSlotCapacity, offsetOfInlineSlots, DefaultFlags | IsLockedFlag | MayBecomeSharedFlag | IsSharedFlag), propertyCount(size)

{

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

{

Assert((SharedFunctionPropertyDescriptors[i].Attributes & PropertyDeleted) == 0);

descriptors[i].Id = SharedFunctionPropertyDescriptors[i].Id;

descriptors[i].Attributes = SharedFunctionPropertyDescriptors[i].Attributes;

}

Assert((propertyTypes & (PropertyTypesAll & ~PropertyTypesWritableDataOnly)) == 0);

SetPropertyTypes(PropertyTypesWritableDataOnly, propertyTypes);

SetIsInlineSlotCapacityLocked();

}

template<size\_t size>

SimpleTypeHandler<size> \* SimpleTypeHandler<size>::ConvertToNonSharedSimpleType(DynamicObject\* instance)

{

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

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

CompileAssert(\_countof(descriptors) == size);

SimpleTypeHandler \* newTypeHandler = RecyclerNew(recycler, SimpleTypeHandler, this);

// Consider: Add support for fixed fields to SimpleTypeHandler when

// non-shared. Here we could set the instance as the singleton instance on the newly

// created handler.

newTypeHandler->SetFlags(IsPrototypeFlag | HasKnownSlot0Flag, this->GetFlags());

Assert(newTypeHandler->GetIsInlineSlotCapacityLocked());

newTypeHandler->SetPropertyTypes(PropertyTypesWritableDataOnly | PropertyTypesWritableDataOnlyDetection, this->GetPropertyTypes());

newTypeHandler->SetInstanceTypeHandler(instance);

return newTypeHandler;

}

template<size\_t size>

template <typename T>

T\* SimpleTypeHandler<size>::ConvertToTypeHandler(DynamicObject\* instance)

{

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

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

#if DBG

DynamicType\* oldType = instance->GetDynamicType();

#endif

T\* newTypeHandler = RecyclerNew(recycler, T, recycler, SimpleTypeHandler<size>::GetSlotCapacity(), GetInlineSlotCapacity(), GetOffsetOfInlineSlots());

Assert(HasSingletonInstanceOnlyIfNeeded());

bool const hasSingletonInstance = newTypeHandler->SetSingletonInstanceIfNeeded(instance);

// If instance has a shared type, the type handler change below will induce a type transition, which

// guarantees that any existing fast path field stores (which could quietly overwrite a fixed field

// on this instance) will be invalidated. It is safe to mark all fields as fixed.

bool const allowFixedFields = hasSingletonInstance && instance->HasLockedType();

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

{

Var value = instance->GetSlot(i);

Assert(value != nullptr || IsInternalPropertyId(descriptors[i].Id->GetPropertyId()));

bool markAsFixed = allowFixedFields && !IsInternalPropertyId(descriptors[i].Id->GetPropertyId()) &&

(JavascriptFunction::Is(value) ? ShouldFixMethodProperties() : false);

newTypeHandler->Add(descriptors[i].Id, descriptors[i].Attributes, true, markAsFixed, false, scriptContext);

}

newTypeHandler->SetFlags(IsPrototypeFlag | HasKnownSlot0Flag, this->GetFlags());

// We don't expect to convert to a PathTypeHandler. If we change this later we need to review if we want the new type handler to have locked

// inline slot capacity, or if we want to allow shrinking of the SimpleTypeHandler's inline slot capacity.

Assert(!newTypeHandler->IsPathTypeHandler());

Assert(newTypeHandler->GetIsInlineSlotCapacityLocked());

newTypeHandler->SetPropertyTypes(PropertyTypesWritableDataOnly | PropertyTypesWritableDataOnlyDetection, this->GetPropertyTypes());

newTypeHandler->SetInstanceTypeHandler(instance);

#if DBG

// If we marked fields as fixed we had better forced a type transition.

Assert(!allowFixedFields || instance->GetDynamicType() != oldType);

#endif

return newTypeHandler;

}

template<size\_t size>

DictionaryTypeHandler\* SimpleTypeHandler<size>::ConvertToDictionaryType(DynamicObject\* instance)

{

DictionaryTypeHandler\* newTypeHandler = ConvertToTypeHandler<DictionaryTypeHandler>(instance);

#ifdef PROFILE\_TYPES

instance->GetScriptContext()->convertSimpleToDictionaryCount++;

#endif

return newTypeHandler;

}

template<size\_t size>

SimpleDictionaryTypeHandler\* SimpleTypeHandler<size>::ConvertToSimpleDictionaryType(DynamicObject\* instance)

{

SimpleDictionaryTypeHandler\* newTypeHandler = ConvertToTypeHandler<SimpleDictionaryTypeHandler >(instance);

#ifdef PROFILE\_TYPES

instance->GetScriptContext()->convertSimpleToSimpleDictionaryCount++;

#endif

return newTypeHandler;

}

template<size\_t size>

ES5ArrayTypeHandler\* SimpleTypeHandler<size>::ConvertToES5ArrayType(DynamicObject\* instance)

{

ES5ArrayTypeHandler\* newTypeHandler = ConvertToTypeHandler<ES5ArrayTypeHandler>(instance);

#ifdef PROFILE\_TYPES

instance->GetScriptContext()->convertSimpleToDictionaryCount++;

#endif

return newTypeHandler;

}

template<size\_t size>

int SimpleTypeHandler<size>::GetPropertyCount()

{

return propertyCount;

}

template<size\_t size>

PropertyId SimpleTypeHandler<size>::GetPropertyId(ScriptContext\* scriptContext, PropertyIndex index)

{

if (index < propertyCount && !(descriptors[index].Attributes & PropertyDeleted))

{

return descriptors[index].Id->GetPropertyId();

}

else

{

return Constants::NoProperty;

}

}

template<size\_t size>

PropertyId SimpleTypeHandler<size>::GetPropertyId(ScriptContext\* scriptContext, BigPropertyIndex index)

{

if (index < propertyCount && !(descriptors[index].Attributes & PropertyDeleted))

{

return descriptors[index].Id->GetPropertyId();

}

else

{

return Constants::NoProperty;

}

}

template<size\_t size>

BOOL SimpleTypeHandler<size>::FindNextProperty(ScriptContext\* scriptContext, PropertyIndex& index, JavascriptString\*\* propertyStringName,

PropertyId\* propertyId, PropertyAttributes\* attributes, Type\* type, DynamicType \*typeToEnumerate, bool requireEnumerable, bool enumSymbols)

{

Assert(propertyStringName);

Assert(propertyId);

Assert(type);

for( ; index < propertyCount; ++index )

{

PropertyAttributes attribs = descriptors[index].Attributes;

if( !(attribs & PropertyDeleted) && (!requireEnumerable || (attribs & PropertyEnumerable)))

{

const PropertyRecord\* propertyRecord = descriptors[index].Id;

// Skip this property if it is a symbol and we are not including symbol properties

if (!enumSymbols && propertyRecord->IsSymbol())

{

continue;

}

if (attributes != nullptr)

{

\*attributes = attribs;

}

\*propertyId = propertyRecord->GetPropertyId();

PropertyString\* propertyString = type->GetScriptContext()->GetPropertyString(\*propertyId);

\*propertyStringName = propertyString;

if (attribs & PropertyWritable)

{

uint16 inlineOrAuxSlotIndex;

bool isInlineSlot;

PropertyIndexToInlineOrAuxSlotIndex(index, &inlineOrAuxSlotIndex, &isInlineSlot);

propertyString->UpdateCache(type, inlineOrAuxSlotIndex, isInlineSlot, true);

}

else

{

#ifdef DEBUG

PropertyCache const\* cache = propertyString->GetPropertyCache();

Assert(!cache || cache->type != type);

#endif

}

return TRUE;

}

}

return FALSE;

}

template<size\_t size>

PropertyIndex SimpleTypeHandler<size>::GetPropertyIndex(PropertyRecord const\* propertyRecord)

{

int index;

if (GetDescriptor(propertyRecord->GetPropertyId(), &index) && !(descriptors[index].Attributes & PropertyDeleted))

{

return (PropertyIndex)index;

}

return Constants::NoSlot;

}

template<size\_t size>

bool SimpleTypeHandler<size>::GetPropertyEquivalenceInfo(PropertyRecord const\* propertyRecord, PropertyEquivalenceInfo& info)

{

int index;

if (GetDescriptor(propertyRecord->GetPropertyId(), &index) && !(descriptors[index].Attributes & PropertyDeleted))

{

info.slotIndex = AdjustSlotIndexForInlineSlots((PropertyIndex)index);

info.isWritable = !!(descriptors[index].Attributes & PropertyWritable);

return true;

}

else

{

info.slotIndex = Constants::NoSlot;

info.isWritable = true;

return false;

}

}

template<size\_t size>

bool SimpleTypeHandler<size>::IsObjTypeSpecEquivalent(const Type\* type, const TypeEquivalenceRecord& record, uint& failedPropertyIndex)

{

Js::EquivalentPropertyEntry\* properties = record.properties;

for (uint pi = 0; pi < record.propertyCount; pi++)

{

const EquivalentPropertyEntry\* refInfo = &properties[pi];

if (!this->SimpleTypeHandler<size>::IsObjTypeSpecEquivalent(type, refInfo))

{

failedPropertyIndex = pi;

return false;

}

}

return true;

}

template<size\_t size>

bool SimpleTypeHandler<size>::IsObjTypeSpecEquivalent(const Type\* type, const EquivalentPropertyEntry \*entry)

{

if (this->propertyCount > 0)

{

for (int i = 0; i < this->propertyCount; i++)

{

SimplePropertyDescriptor\* descriptor = &this->descriptors[i];

Js::PropertyId propertyId = descriptor->Id->GetPropertyId();

if (entry->propertyId == propertyId && !(descriptor->Attributes & PropertyDeleted))

{

Js::PropertyIndex relSlotIndex = AdjustValidSlotIndexForInlineSlots(static\_cast<PropertyIndex>(0));

if (relSlotIndex != entry->slotIndex ||

entry->isAuxSlot != (GetInlineSlotCapacity() == 0) ||

(entry->mustBeWritable && !(descriptor->Attributes & PropertyWritable)))

{

return false;

}

}

else

{

if (entry->slotIndex != Constants::NoSlot || entry->mustBeWritable)

{

return false;

}

}

}

}

else

{

if (entry->slotIndex != Constants::NoSlot || entry->mustBeWritable)

{

return false;

}

}

return true;

}

template<size\_t size>

BOOL SimpleTypeHandler<size>::HasProperty(DynamicObject\* instance, PropertyId propertyId, \_\_out\_opt bool \*noRedecl)

{

if (noRedecl != nullptr)

{

\*noRedecl = false;

}

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

{

if (descriptors[i].Id->GetPropertyId() == propertyId)

{

if (descriptors[i].Attributes & PropertyDeleted)

{

return false;

}

if (noRedecl && descriptors[i].Attributes & PropertyNoRedecl)

{

\*noRedecl = true;

}

return true;

}

}

// Check numeric propertyId only if objectArray available

uint32 indexVal;

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

if (instance->HasObjectArray() && scriptContext->IsNumericPropertyId(propertyId, &indexVal))

{

return SimpleTypeHandler<size>::HasItem(instance, indexVal);

}

return false;

}

template<size\_t size>

BOOL SimpleTypeHandler<size>::HasProperty(DynamicObject\* instance, JavascriptString\* propertyNameString)

{

AssertMsg(!PropertyRecord::IsPropertyNameNumeric(propertyNameString->GetString(), propertyNameString->GetLength()),

"Numeric property names should have been converted to uint or PropertyRecord\* before calling GetSetter");

JsUtil::CharacterBuffer<WCHAR> propertyName(propertyNameString->GetString(), propertyNameString->GetLength());

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

{

if (descriptors[i].Id->Equals(propertyName))

{

return !(descriptors[i].Attributes & PropertyDeleted);

}

}

return false;

}

template<size\_t size>

BOOL SimpleTypeHandler<size>::GetProperty(DynamicObject\* instance, Var originalInstance, PropertyId propertyId, Var\* value, PropertyValueInfo\* info, ScriptContext\* requestContext)

{

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

{

if (descriptors[i].Id->GetPropertyId() == propertyId)

{

if (descriptors[i].Attributes & PropertyDeleted)

{

return false;

}

\*value = instance->GetSlot(i);

PropertyValueInfo::Set(info, instance, static\_cast<PropertyIndex>(i), descriptors[i].Attributes);

return true;

}

}

// Check numeric propertyId only if objectArray available

uint32 indexVal;

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

if (instance->HasObjectArray() && scriptContext->IsNumericPropertyId(propertyId, &indexVal))

{

return SimpleTypeHandler<size>::GetItem(instance, originalInstance, indexVal, value, scriptContext);

}

return false;

}

template<size\_t size>

BOOL SimpleTypeHandler<size>::GetProperty(DynamicObject\* instance, Var originalInstance, JavascriptString\* propertyNameString, Var\* value, PropertyValueInfo\* info, ScriptContext\* requestContext)

{

AssertMsg(!PropertyRecord::IsPropertyNameNumeric(propertyNameString->GetString(), propertyNameString->GetLength()),

"Numeric property names should have been converted to uint or PropertyRecord\* before calling GetSetter");

JsUtil::CharacterBuffer<WCHAR> propertyName(propertyNameString->GetString(), propertyNameString->GetLength());

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

{

if (descriptors[i].Id->Equals(propertyName))

{

if (descriptors[i].Attributes & PropertyDeleted)

{

return false;

}

\*value = instance->GetSlot(i);

PropertyValueInfo::Set(info, instance, static\_cast<PropertyIndex>(i), descriptors[i].Attributes);

return true;

}

}

return false;

}

template<size\_t size>

BOOL SimpleTypeHandler<size>::SetProperty(DynamicObject\* instance, PropertyId propertyId, Var value, PropertyOperationFlags flags, PropertyValueInfo\* info)

{

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

int index;

if (GetDescriptor(propertyId, &index))

{

if (descriptors[index].Attributes & PropertyDeleted)

{

// A locked type should not have deleted properties

Assert(!GetIsLocked());

descriptors[index].Attributes = PropertyDynamicTypeDefaults;

instance->SetHasNoEnumerableProperties(false);

}

else if (!(descriptors[index].Attributes & PropertyWritable))

{

JavascriptError::ThrowCantAssignIfStrictMode(flags, scriptContext);

PropertyValueInfo::Set(info, instance, static\_cast<PropertyIndex>(index), descriptors[index].Attributes); // Try to cache property info even if not writable

return false;

}

SetSlotUnchecked(instance, index, value);

PropertyValueInfo::Set(info, instance, static\_cast<PropertyIndex>(index), descriptors[index].Attributes);

SetPropertyUpdateSideEffect(instance, propertyId, value, SideEffects\_Any);

return true;

}

// Always check numeric propertyId. This may create objectArray.

uint32 indexVal;

if (scriptContext->IsNumericPropertyId(propertyId, &indexVal))

{

return SimpleTypeHandler<size>::SetItem(instance, indexVal, value, flags);

}

return this->AddProperty(instance, propertyId, value, PropertyDynamicTypeDefaults, info, flags, SideEffects\_Any);

}

template<size\_t size>

BOOL SimpleTypeHandler<size>::SetProperty(DynamicObject\* instance, JavascriptString\* propertyNameString, Var value, PropertyOperationFlags flags, PropertyValueInfo\* info)

{

// Either the property exists in the dictionary, in which case a PropertyRecord for it exists,

// or we have to add it to the dictionary, in which case we need to get or create a PropertyRecord.

// Thus, just get or create one and call the PropertyId overload of SetProperty.

PropertyRecord const\* propertyRecord;

instance->GetScriptContext()->GetOrAddPropertyRecord(propertyNameString->GetString(), propertyNameString->GetLength(), &propertyRecord);

return SimpleTypeHandler<size>::SetProperty(instance, propertyRecord->GetPropertyId(), value, flags, info);

}

template<size\_t size>

DescriptorFlags SimpleTypeHandler<size>::GetSetter(DynamicObject\* instance, PropertyId propertyId, Var\* setterValue, PropertyValueInfo\* info, ScriptContext\* requestContext)

{

int index;

PropertyValueInfo::SetNoCache(info, instance);

if (GetDescriptor(propertyId, &index))

{

if (descriptors[index].Attributes & PropertyDeleted)

{

return None;

}

return (descriptors[index].Attributes & PropertyWritable) ? WritableData : Data;

}

uint32 indexVal;

if (instance->GetScriptContext()->IsNumericPropertyId(propertyId, &indexVal))

{

return SimpleTypeHandler<size>::GetItemSetter(instance, indexVal, setterValue, requestContext);

}

return None;

}

template<size\_t size>

DescriptorFlags SimpleTypeHandler<size>::GetSetter(DynamicObject\* instance, JavascriptString\* propertyNameString, Var\* setterValue, PropertyValueInfo\* info, ScriptContext\* requestContext)

{

AssertMsg(!PropertyRecord::IsPropertyNameNumeric(propertyNameString->GetString(), propertyNameString->GetLength()),

"Numeric property names should have been converted to uint or PropertyRecord\* before calling GetSetter");

JsUtil::CharacterBuffer<WCHAR> propertyName(propertyNameString->GetString(), propertyNameString->GetLength());

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

{

if (descriptors[i].Id->Equals(propertyName))

{

if (descriptors[i].Attributes & PropertyDeleted)

{

return None;

}

return (descriptors[i].Attributes & PropertyWritable) ? WritableData : Data;

}

}

return None;

}

template<size\_t size>

BOOL SimpleTypeHandler<size>::DeleteProperty(DynamicObject\* instance, PropertyId propertyId, PropertyOperationFlags propertyOperationFlags)

{

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

int index;

if (GetDescriptor(propertyId, &index))

{

if (descriptors[index].Attributes & PropertyDeleted)

{

// A locked type should not have deleted properties

Assert(!GetIsLocked());

return true;

}

if (!(descriptors[index].Attributes & PropertyConfigurable))

{

JavascriptError::ThrowCantDeleteIfStrictMode(propertyOperationFlags, scriptContext, scriptContext->GetPropertyName(propertyId)->GetBuffer());

return false;

}

if ((this->GetFlags() & IsPrototypeFlag)

|| JavascriptOperators::HasProxyOrPrototypeInlineCacheProperty(instance, propertyId))

{

// We don't evolve dictionary types when deleting a field, so we need to invalidate prototype caches.

// We only have to do this though if the current type is used as a prototype, or the current property

// is found on the prototype chain.)

scriptContext->InvalidateProtoCaches(propertyId);

}

instance->ChangeType();

CompileAssert(\_countof(descriptors) == size);

SetSlotUnchecked(instance, index, nullptr);

NullTypeHandlerBase\* nullTypeHandler = ((this->GetFlags() & IsPrototypeFlag) != 0) ?

(NullTypeHandlerBase\*)NullTypeHandler<true>::GetDefaultInstance() : (NullTypeHandlerBase\*)NullTypeHandler<false>::GetDefaultInstance();

if (instance->HasReadOnlyPropertiesInvisibleToTypeHandler())

{

nullTypeHandler->ClearHasOnlyWritableDataProperties();

}

SetInstanceTypeHandler(instance, nullTypeHandler, false);

return true;

}

// Check numeric propertyId only if objectArray available

uint32 indexVal;

if (instance->HasObjectArray() && scriptContext->IsNumericPropertyId(propertyId, &indexVal))

{

return SimpleTypeHandler<size>::DeleteItem(instance, indexVal, propertyOperationFlags);

}

return true;

}

template<size\_t size>

BOOL SimpleTypeHandler<size>::IsEnumerable(DynamicObject\* instance, PropertyId propertyId)

{

int index;

if (!GetDescriptor(propertyId, &index))

{

return true;

}

return descriptors[index].Attributes & PropertyEnumerable;

}

template<size\_t size>

BOOL SimpleTypeHandler<size>::IsWritable(DynamicObject\* instance, PropertyId propertyId)

{

int index;

if (!GetDescriptor(propertyId, &index))

{

return true;

}

return descriptors[index].Attributes & PropertyWritable;

}

template<size\_t size>

BOOL SimpleTypeHandler<size>::IsConfigurable(DynamicObject\* instance, PropertyId propertyId)

{

int index;

if (!GetDescriptor(propertyId, &index))

{

return true;

}

return descriptors[index].Attributes & PropertyConfigurable;

}

template<size\_t size>

BOOL SimpleTypeHandler<size>::SetEnumerable(DynamicObject\* instance, PropertyId propertyId, BOOL value)

{

int index;

if (!GetDescriptor(propertyId, &index))

{

// Upgrade type handler if set objectArray item attribute.

// Only check numeric propertyId if objectArray available.

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

uint32 indexVal;

if (instance->HasObjectArray() && scriptContext->IsNumericPropertyId(propertyId, &indexVal))

{

return SimpleTypeHandler<size>::ConvertToTypeWithItemAttributes(instance)

->SetEnumerable(instance, propertyId, value);

}

return true;

}

if (value)

{

if (SetAttribute(instance, index, PropertyEnumerable))

{

instance->SetHasNoEnumerableProperties(false);

}

}

else

{

ClearAttribute(instance, index, PropertyEnumerable);

}

return true;

}

template<size\_t size>

BOOL SimpleTypeHandler<size>::SetWritable(DynamicObject\* instance, PropertyId propertyId, BOOL value)

{

int index;

if (!GetDescriptor(propertyId, &index))

{

// Upgrade type handler if set objectArray item attribute.

// Only check numeric propertyId if objectArray available.

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

uint32 indexVal;

if (instance->HasObjectArray() && scriptContext->IsNumericPropertyId(propertyId, &indexVal))

{

return SimpleTypeHandler<size>::ConvertToTypeWithItemAttributes(instance)

->SetWritable(instance, propertyId, value);

}

return true;

}

const Type\* oldType = instance->GetType();

if (value)

{

if (SetAttribute(instance, index, PropertyWritable))

{

instance->ChangeTypeIf(oldType); // Ensure type change to invalidate caches

}

}

else

{

if (ClearAttribute(instance, index, PropertyWritable))

{

instance->ChangeTypeIf(oldType); // Ensure type change to invalidate caches

// Clearing the attribute may have changed the type handler, so make sure

// we access the current one.

DynamicTypeHandler\* const typeHandler = GetCurrentTypeHandler(instance);

typeHandler->ClearHasOnlyWritableDataProperties();

if (typeHandler->GetFlags() & IsPrototypeFlag)

{

instance->GetScriptContext()->InvalidateStoreFieldCaches(propertyId);

instance->GetLibrary()->NoPrototypeChainsAreEnsuredToHaveOnlyWritableDataProperties();

}

}

}

return true;

}

template<size\_t size>

BOOL SimpleTypeHandler<size>::SetConfigurable(DynamicObject\* instance, PropertyId propertyId, BOOL value)

{

int index;

if (!GetDescriptor(propertyId, &index))

{

// Upgrade type handler if set objectArray item attribute.

// Only check numeric propertyId if objectArray available.

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

uint32 indexVal;

if (instance->HasObjectArray() && scriptContext->IsNumericPropertyId(propertyId, &indexVal))

{

return SimpleTypeHandler<size>::ConvertToTypeWithItemAttributes(instance)

->SetConfigurable(instance, propertyId, value);

}

return true;

}

if (value)

{

SetAttribute(instance, index, PropertyConfigurable);

}

else

{

ClearAttribute(instance, index, PropertyConfigurable);

}

return true;

}

template<size\_t size>

BOOL SimpleTypeHandler<size>::GetDescriptor(PropertyId propertyId, int \* index)

{

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

{

if (descriptors[i].Id->GetPropertyId() == propertyId)

{

\*index = i;

return true;

}

}

return false;

}

//

// Set an attribute bit. Return true if change is made.

//

template<size\_t size>

BOOL SimpleTypeHandler<size>::SetAttribute(DynamicObject\* instance, int index, PropertyAttributes attribute)

{

if (descriptors[index].Attributes & PropertyDeleted)

{

// A locked type should not have deleted properties

Assert(!GetIsLocked());

return false;

}

PropertyAttributes attributes = descriptors[index].Attributes;

attributes |= attribute;

if (attributes == descriptors[index].Attributes)

{

return false;

}

// If the type is locked we must force type transition to invalidate any potential property string

// caches used in snapshot enumeration.

if (GetIsLocked())

{

#if DBG

DynamicType\* oldType = instance->GetDynamicType();

#endif

// This changes TypeHandler, but non-necessarily Type.

this->ConvertToNonSharedSimpleType(instance)->descriptors[index].Attributes = attributes;

#if DBG

Assert(!oldType->GetIsLocked() || instance->GetDynamicType() != oldType);

#endif

}

else

{

descriptors[index].Attributes = attributes;

}

return true;

}

//

// Clear an attribute bit. Return true if change is made.

//

template<size\_t size>

BOOL SimpleTypeHandler<size>::ClearAttribute(DynamicObject\* instance, int index, PropertyAttributes attribute)

{

if (descriptors[index].Attributes & PropertyDeleted)

{

// A locked type should not have deleted properties

Assert(!GetIsLocked());

return false;

}

PropertyAttributes attributes = descriptors[index].Attributes;

attributes &= ~attribute;

if (attributes == descriptors[index].Attributes)

{

return false;

}

// If the type is locked we must force type transition to invalidate any potential property string

// caches used in snapshot enumeration.

if (GetIsLocked())

{

#if DBG

DynamicType\* oldType = instance->GetDynamicType();

#endif

// This changes TypeHandler, but non-necessarily Type.

this->ConvertToNonSharedSimpleType(instance)->descriptors[index].Attributes = attributes;

#if DBG

Assert(!oldType->GetIsLocked() || instance->GetDynamicType() != oldType);

#endif

}

else

{

descriptors[index].Attributes = attributes;

}

return true;

}

template<size\_t size>

BOOL SimpleTypeHandler<size>::SetAccessors(DynamicObject\* instance, PropertyId propertyId, Var getter, Var setter, PropertyOperationFlags flags)

{

return ConvertToDictionaryType(instance)->SetAccessors(instance, propertyId, getter, setter, flags);

}

template<size\_t size>

BOOL SimpleTypeHandler<size>::PreventExtensions(DynamicObject\* instance)

{

return ConvertToDictionaryType(instance)->PreventExtensions(instance);

}

template<size\_t size>

BOOL SimpleTypeHandler<size>::Seal(DynamicObject\* instance)

{

return ConvertToDictionaryType(instance)->Seal(instance);

}

template<size\_t size>

BOOL SimpleTypeHandler<size>::FreezeImpl(DynamicObject\* instance, bool isConvertedType)

{

return ConvertToDictionaryType(instance)->Freeze(instance, true);

}

template<size\_t size>

BOOL SimpleTypeHandler<size>::SetPropertyWithAttributes(DynamicObject\* instance, PropertyId propertyId, Var value, PropertyAttributes attributes, PropertyValueInfo\* info, PropertyOperationFlags flags, SideEffects possibleSideEffects)

{

int index;

if (GetDescriptor(propertyId, &index))

{

if (descriptors[index].Attributes != attributes)

{

SimpleTypeHandler \* typeHandler = this;

if (GetIsLocked())

{

#if DBG

DynamicType\* oldType = instance->GetDynamicType();

#endif

typeHandler = this->ConvertToNonSharedSimpleType(instance);

#if DBG

Assert(!oldType->GetIsLocked() || instance->GetDynamicType() != oldType);

#endif

}

typeHandler->descriptors[index].Attributes = attributes;

if (attributes & PropertyEnumerable)

{

instance->SetHasNoEnumerableProperties(false);

}

if (!(attributes & PropertyWritable))

{

typeHandler->ClearHasOnlyWritableDataProperties();

if (typeHandler->GetFlags() & IsPrototypeFlag)

{

instance->GetScriptContext()->InvalidateStoreFieldCaches(propertyId);

instance->GetLibrary()->NoPrototypeChainsAreEnsuredToHaveOnlyWritableDataProperties();

}

}

}

SetSlotUnchecked(instance, index, value);

PropertyValueInfo::Set(info, instance, static\_cast<PropertyIndex>(index), descriptors[index].Attributes);

SetPropertyUpdateSideEffect(instance, propertyId, value, possibleSideEffects);

return true;

}

// Always check numeric propertyId. May create objectArray.

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

uint32 indexVal;

if (scriptContext->IsNumericPropertyId(propertyId, &indexVal))

{

return SimpleTypeHandler<size>::SetItemWithAttributes(instance, indexVal, value, attributes);

}

return this->AddProperty(instance, propertyId, value, attributes, info, flags, possibleSideEffects);

}

template<size\_t size>

BOOL SimpleTypeHandler<size>::SetAttributes(DynamicObject\* instance, PropertyId propertyId, PropertyAttributes attributes)

{

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

{

if (descriptors[i].Id->GetPropertyId() == propertyId)

{

if (descriptors[i].Attributes & PropertyDeleted)

{

return true;

}

descriptors[i].Attributes = (descriptors[i].Attributes & ~PropertyDynamicTypeDefaults) | (attributes & PropertyDynamicTypeDefaults);

if (descriptors[i].Attributes & PropertyEnumerable)

{

instance->SetHasNoEnumerableProperties(false);

}

if (!(descriptors[i].Attributes & PropertyWritable))

{

this->ClearHasOnlyWritableDataProperties();

if (GetFlags() & IsPrototypeFlag)

{

instance->GetScriptContext()->InvalidateStoreFieldCaches(propertyId);

instance->GetLibrary()->NoPrototypeChainsAreEnsuredToHaveOnlyWritableDataProperties();

}

}

return true;

}

}

// Check numeric propertyId only if objectArray available

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

uint32 indexVal;

if (instance->HasObjectArray() && scriptContext->IsNumericPropertyId(propertyId, &indexVal))

{

return SimpleTypeHandler<size>::SetItemAttributes(instance, indexVal, attributes);

}

return true;

}

template<size\_t size>

BOOL SimpleTypeHandler<size>::GetAttributesWithPropertyIndex(DynamicObject \* instance, PropertyId propertyId, BigPropertyIndex index, PropertyAttributes \* attributes)

{

if (index >= propertyCount) { return false; }

Assert(descriptors[index].Id->GetPropertyId() == propertyId);

if (descriptors[index].Attributes & PropertyDeleted)

{

return false;

}

\*attributes = descriptors[index].Attributes & PropertyDynamicTypeDefaults;

return true;

}

template<size\_t size>

BOOL SimpleTypeHandler<size>::AddProperty(DynamicObject\* instance, PropertyId propertyId, Var value, PropertyAttributes attributes, PropertyValueInfo\* info, PropertyOperationFlags flags, SideEffects possibleSideEffects)

{

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

#if DBG

int index;

uint32 indexVal;

Assert(!GetDescriptor(propertyId, &index));

Assert(!scriptContext->IsNumericPropertyId(propertyId, &indexVal));

#endif

if (propertyCount >= sizeof(descriptors)/sizeof(SimplePropertyDescriptor))

{

Assert(propertyId != Constants::NoProperty);

PropertyRecord const\* propertyRecord = scriptContext->GetPropertyName(propertyId);

return ConvertToSimpleDictionaryType(instance)->AddProperty(instance, propertyRecord, value, attributes, info, flags, possibleSideEffects);

}

descriptors[propertyCount].Id = scriptContext->GetPropertyName(propertyId);

descriptors[propertyCount].Attributes = attributes;

if (attributes & PropertyEnumerable)

{

instance->SetHasNoEnumerableProperties(false);

}

if (!(attributes & PropertyWritable))

{

this->ClearHasOnlyWritableDataProperties();

if (GetFlags() & IsPrototypeFlag)

{

instance->GetScriptContext()->InvalidateStoreFieldCaches(propertyId);

instance->GetLibrary()->NoPrototypeChainsAreEnsuredToHaveOnlyWritableDataProperties();

}

}

SetSlotUnchecked(instance, propertyCount, value);

PropertyValueInfo::Set(info, instance, static\_cast<PropertyIndex>(propertyCount), attributes);

propertyCount++;

if ((this->GetFlags() && IsPrototypeFlag)

|| JavascriptOperators::HasProxyOrPrototypeInlineCacheProperty(instance, propertyId))

{

scriptContext->InvalidateProtoCaches(propertyId);

}

SetPropertyUpdateSideEffect(instance, propertyId, value, possibleSideEffects);

return true;

}

template<size\_t size>

void SimpleTypeHandler<size>::SetAllPropertiesToUndefined(DynamicObject\* instance, bool invalidateFixedFields)

{

// Note: This method is currently only called from ResetObject, which in turn only applies to external objects.

// Before using for other purposes, make sure the assumptions made here make sense in the new context. In particular,

// the invalidateFixedFields == false is only correct if a) the object is known not to have any, or b) the type of the

// object has changed and/or property guards have already been invalidated through some other means.

// We can ignore invalidateFixedFields, because SimpleTypeHandler doesn't support fixed fields at this point.

Js::RecyclableObject\* undefined = instance->GetLibrary()->GetUndefined();

for (int propertyIndex = 0; propertyIndex < this->propertyCount; propertyIndex++)

{

SetSlotUnchecked(instance, propertyIndex, undefined);

}

}

template<size\_t size>

void SimpleTypeHandler<size>::MarshalAllPropertiesToScriptContext(DynamicObject\* instance, ScriptContext\* targetScriptContext, bool invalidateFixedFields)

{

// Note: This method is currently only called from ResetObject, which in turn only applies to external objects.

// Before using for other purposes, make sure the assumptions made here make sense in the new context. In particular,

// the invalidateFixedFields == false is only correct if a) the object is known not to have any, or b) the type of the

// object has changed and/or property guards have already been invalidated through some other means.

// We can ignore invalidateFixedFields, because SimpleTypeHandler doesn't support fixed fields at this point.

for (int propertyIndex = 0; propertyIndex < this->propertyCount; propertyIndex++)

{

SetSlotUnchecked(instance, propertyIndex, CrossSite::MarshalVar(targetScriptContext, GetSlot(instance, propertyIndex)));

}

}

template<size\_t size>

DynamicTypeHandler\* SimpleTypeHandler<size>::ConvertToTypeWithItemAttributes(DynamicObject\* instance)

{

return JavascriptArray::Is(instance) ?

ConvertToES5ArrayType(instance) : ConvertToDictionaryType(instance);

}

template<size\_t size>

void SimpleTypeHandler<size>::SetIsPrototype(DynamicObject\* instance)

{

// Don't return if IsPrototypeFlag is set, because we may still need to do a type transition and

// set fixed bits. If this handler is shared, this instance may not even be a prototype yet.

// In this case we may need to convert to a non-shared type handler.

if (!ChangeTypeOnProto() && !(GetIsOrMayBecomeShared() && IsolatePrototypes()))

{

return;

}

ConvertToSimpleDictionaryType(instance)->SetIsPrototype(instance);

}

#if DBG

template<size\_t size>

bool SimpleTypeHandler<size>::CanStorePropertyValueDirectly(const DynamicObject\* instance, PropertyId propertyId, bool allowLetConst)

{

Assert(!allowLetConst);

int index;

if (GetDescriptor(propertyId, &index))

{

return true;

}

else

{

AssertMsg(false, "Asking about a property this type handler doesn't know about?");

return false;

}

}

#endif

template class SimpleTypeHandler<1>;

template class SimpleTypeHandler<2>;

}

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

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

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

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

#pragma once

namespace Js

{

template<size\_t size>

class SimpleTypeHandler sealed: public DynamicTypeHandler

{

friend class NullTypeHandlerBase;

private:

int propertyCount;

SimplePropertyDescriptor descriptors[size];

public:

DEFINE\_GETCPPNAME();

private:

SimpleTypeHandler(Recycler\*); // only used by NullTypeHandler

SimpleTypeHandler(SimpleTypeHandler<size> \* typeHandler);

DEFINE\_VTABLE\_CTOR\_NO\_REGISTER(SimpleTypeHandler, DynamicTypeHandler);

public:

SimpleTypeHandler(const PropertyRecord\* id, PropertyAttributes attributes = PropertyNone, PropertyTypes propertyTypes = PropertyTypesNone, uint16 inlineSlotCapacity = 0, uint16 offsetOfInlineSlots = 0);

// Constructor of a shared typed handler

SimpleTypeHandler(SimplePropertyDescriptor const (&SharedFunctionPropertyDescriptors)[size], PropertyTypes propertyTypes = PropertyTypesNone, uint16 inlineSlotCapacity = 0, uint16 offsetOfInlineSlots = 0);

virtual BOOL IsLockable() const override { return true; }

virtual BOOL IsSharable() const override { return true; }

virtual int GetPropertyCount() override;

virtual PropertyId GetPropertyId(ScriptContext\* scriptContext, PropertyIndex index) override;

virtual PropertyId GetPropertyId(ScriptContext\* scriptContext, BigPropertyIndex index) override;

virtual BOOL FindNextProperty(ScriptContext\* scriptContext, PropertyIndex& index, JavascriptString\*\* propertyString,

PropertyId\* propertyId, PropertyAttributes\* attributes, Type\* type, DynamicType \*typeToEnumerate, bool requireEnumerable, bool enumSymbols = false) override;

virtual PropertyIndex GetPropertyIndex(PropertyRecord const\* propertyRecord) override;

virtual bool GetPropertyEquivalenceInfo(PropertyRecord const\* propertyRecord, PropertyEquivalenceInfo& info) override;

virtual bool IsObjTypeSpecEquivalent(const Type\* type, const TypeEquivalenceRecord& record, uint& failedPropertyIndex) override;

virtual bool IsObjTypeSpecEquivalent(const Type\* type, const EquivalentPropertyEntry\* entry) override;

virtual BOOL HasProperty(DynamicObject\* instance, PropertyId propertyId, \_\_out\_opt bool \*noRedecl = nullptr) override;

virtual BOOL HasProperty(DynamicObject\* instance, JavascriptString\* propertyNameString) override;

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

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

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

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

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

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

virtual BOOL DeleteProperty(DynamicObject\* instance, PropertyId propertyId, PropertyOperationFlags flags) override;

virtual BOOL IsEnumerable(DynamicObject\* instance, PropertyId propertyId) override;

virtual BOOL IsWritable(DynamicObject\* instance, PropertyId propertyId) override;

virtual BOOL IsConfigurable(DynamicObject\* instance, PropertyId propertyId) override;

virtual BOOL SetEnumerable(DynamicObject\* instance, PropertyId propertyId, BOOL value) override;

virtual BOOL SetWritable(DynamicObject\* instance, PropertyId propertyId, BOOL value) override;

virtual BOOL SetConfigurable(DynamicObject\* instance, PropertyId propertyId, BOOL value) override;

virtual BOOL SetAccessors(DynamicObject\* instance, PropertyId propertyId, Var getter, Var setter, PropertyOperationFlags flags = PropertyOperation\_None) override;

virtual BOOL PreventExtensions(DynamicObject \*instance) override;

virtual BOOL Seal(DynamicObject\* instance) override;

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

virtual BOOL SetAttributes(DynamicObject\* instance, PropertyId propertyId, PropertyAttributes attributes) override;

virtual BOOL GetAttributesWithPropertyIndex(DynamicObject \* instance, PropertyId propertyId, BigPropertyIndex index, PropertyAttributes \* attributes) override;

virtual void SetAllPropertiesToUndefined(DynamicObject\* instance, bool invalidateFixedFields) override;

virtual void MarshalAllPropertiesToScriptContext(DynamicObject\* instance, ScriptContext\* targetScriptContext, bool invalidateFixedFields) override;

virtual DynamicTypeHandler\* ConvertToTypeWithItemAttributes(DynamicObject\* instance) override;

virtual void SetIsPrototype(DynamicObject\* instance) override;

#if DBG

virtual bool SupportsPrototypeInstances() const { return !ChangeTypeOnProto() && !(GetIsOrMayBecomeShared() && IsolatePrototypes()); }

virtual bool CanStorePropertyValueDirectly(const DynamicObject\* instance, PropertyId propertyId, bool allowLetConst) override;

#endif

#if DBG

bool HasSingletonInstanceOnlyIfNeeded() const

{

// If we add support for fixed fields to this type handler we will have to update this implementation.

return true;

}

#endif

private:

template <typename T>

T\* ConvertToTypeHandler(DynamicObject\* instance);

DictionaryTypeHandler\* ConvertToDictionaryType(DynamicObject\* instance);

SimpleDictionaryTypeHandler\* ConvertToSimpleDictionaryType(DynamicObject\* instance);

ES5ArrayTypeHandler\* ConvertToES5ArrayType(DynamicObject\* instance);

SimpleTypeHandler<size>\* ConvertToNonSharedSimpleType(DynamicObject \* instance);

BOOL GetDescriptor(PropertyId propertyId, int \* index);

BOOL SetAttribute(DynamicObject\* instance, int index, PropertyAttributes attribute);

BOOL ClearAttribute(DynamicObject\* instance, int index, PropertyAttributes attribute);

BOOL AddProperty(DynamicObject\* instance, PropertyId propertyId, Var value, PropertyAttributes attributes, PropertyValueInfo\* info, PropertyOperationFlags flags, SideEffects possibleSideEffects);

virtual BOOL FreezeImpl(DynamicObject\* instance, bool isConvertedType) override;

};

typedef SimpleTypeHandler<1> SimpleTypeHandlerSize1;

typedef SimpleTypeHandler<2> SimpleTypeHandlerSize2;

}

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

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

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

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

#include "RuntimeTypePch.h"

#include "Types\SpreadArgument.h"

namespace Js

{

bool SpreadArgument::Is(Var aValue)

{

return JavascriptOperators::GetTypeId(aValue) == TypeIds\_SpreadArgument;

}

SpreadArgument\* SpreadArgument::FromVar(Var aValue)

{

Assert(SpreadArgument::Is(aValue));

return static\_cast<SpreadArgument\*>(aValue);

}

SpreadArgument::SpreadArgument(Var iterable, RecyclableObject\* iterator, DynamicType \* type) : DynamicObject(type), iterable(iterable),

iterator(iterator), iteratorIndices(nullptr)

{

Var nextItem;

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

while (JavascriptOperators::IteratorStepAndValue(iterator, scriptContext, &nextItem))

{

if (iteratorIndices == nullptr)

{

iteratorIndices = RecyclerNew(scriptContext->GetRecycler(), VarList, scriptContext->GetRecycler());

}

iteratorIndices->Add(nextItem);

}

}

} // namespace Js

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

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

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

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

#pragma once

namespace Js

{

class SpreadArgument : public DynamicObject

{

private:

Var iterable;

RecyclableObject\* iterator;

typedef JsUtil::List<Var, Recycler> VarList;

VarList\* iteratorIndices;

void AssertAndFailFast() { AssertMsg(false, "This function should not be invoked"); Js::Throw::InternalError();}

protected:

DEFINE\_VTABLE\_CTOR(SpreadArgument, DynamicObject);

DEFINE\_MARSHAL\_OBJECT\_TO\_SCRIPT\_CONTEXT(SpreadArgument);

public:

static bool Is(Var aValue);

static SpreadArgument\* FromVar(Var value);

SpreadArgument(Var iterable, RecyclableObject\* iterator, DynamicType \* type);

Var GetArgument() const { return iterable; }

const Var\* GetArgumentSpread() const { return iteratorIndices ? iteratorIndices->GetBuffer() : nullptr; }

uint GetArgumentSpreadCount() const { return iteratorIndices ? iteratorIndices->Count() : 0; }

// A SpreadArgument should never call the Functions defined below this comment

virtual BOOL HasProperty(PropertyId propertyId) override { AssertAndFailFast(); return FALSE; };

virtual BOOL HasOwnProperty(PropertyId propertyId) override { AssertAndFailFast(); return FALSE; };

virtual BOOL SetProperty(PropertyId propertyId, Var value, PropertyOperationFlags flags, PropertyValueInfo\* info) override { AssertAndFailFast(); return FALSE; };

virtual BOOL GetProperty(Var originalInstance, PropertyId propertyId, Var\* value, PropertyValueInfo\* info, ScriptContext\* requestContext) override { AssertAndFailFast(); return FALSE; };

virtual BOOL DeleteProperty(PropertyId propertyId, PropertyOperationFlags flags) override{ AssertAndFailFast(); return FALSE;};

virtual BOOL GetPropertyReference(Var originalInstance, PropertyId propertyId, Var\* value, PropertyValueInfo\* info, ScriptContext\* requestContext) override { AssertAndFailFast(); return None; };

virtual BOOL SetProperty(JavascriptString\* propertyNameString, Var value, PropertyOperationFlags flags, PropertyValueInfo\* info) override { AssertAndFailFast(); return FALSE; };

virtual BOOL GetProperty(Var originalInstance, JavascriptString\* propertyNameString, Var\* value, PropertyValueInfo\* info, ScriptContext\* requestContext) override { AssertAndFailFast(); return FALSE; };

virtual DescriptorFlags GetSetter(PropertyId propertyId, Var \*setterValue, PropertyValueInfo\* info, ScriptContext\* requestContext) override { AssertAndFailFast(); return None; };

virtual DescriptorFlags GetSetter(JavascriptString\* propertyNameString, Var \*setterValue, PropertyValueInfo\* info, ScriptContext\* requestContext) override { AssertAndFailFast(); return None; };

virtual int GetPropertyCount() override { AssertAndFailFast(); return 0; };

virtual PropertyId GetPropertyId(PropertyIndex index) override { AssertAndFailFast(); return Constants::NoProperty; };

virtual PropertyId GetPropertyId(BigPropertyIndex index) override { AssertAndFailFast(); return Constants::NoProperty;; };

virtual BOOL SetInternalProperty(PropertyId internalPropertyId, Var value, PropertyOperationFlags flags, PropertyValueInfo\* info) override { AssertAndFailFast(); return FALSE; };

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

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

virtual BOOL IsFixedProperty(PropertyId propertyId) override { AssertAndFailFast(); return FALSE; };

virtual BOOL HasItem(uint32 index) override { AssertAndFailFast(); return FALSE; };

virtual BOOL HasOwnItem(uint32 index) override { AssertAndFailFast(); return FALSE; };

virtual BOOL GetItem(Var originalInstance, uint32 index, Var\* value, ScriptContext \* requestContext) override { AssertAndFailFast(); return FALSE; };

virtual BOOL GetItemReference(Var originalInstance, uint32 index, Var\* value, ScriptContext \* requestContext) override { AssertAndFailFast(); return FALSE; };

virtual DescriptorFlags GetItemSetter(uint32 index, Var\* setterValue, ScriptContext\* requestContext) override { AssertAndFailFast(); return None; };

virtual BOOL SetItem(uint32 index, Var value, PropertyOperationFlags flags) override { AssertAndFailFast(); return FALSE; };

virtual BOOL DeleteItem(uint32 index, PropertyOperationFlags flags) override { AssertAndFailFast(); return FALSE; };

virtual BOOL ToPrimitive(JavascriptHint hint, Var\* result, ScriptContext \* requestContext) override { AssertAndFailFast(); return FALSE; };

virtual BOOL GetEnumerator(BOOL enumNonEnumerable, Var\* enumerator, ScriptContext\* scriptContext, bool preferSnapshotSemantics = true, bool enumSymbols = false) override { AssertAndFailFast(); return FALSE; };

virtual BOOL SetAccessors(PropertyId propertyId, Var getter, Var setter, PropertyOperationFlags flags = PropertyOperation\_None) override { AssertAndFailFast(); return FALSE; };

virtual BOOL GetAccessors(PropertyId propertyId, Var \*getter, Var \*setter, ScriptContext \* requestContext) override { AssertAndFailFast(); return FALSE; };

virtual BOOL IsWritable(PropertyId propertyId) override { AssertAndFailFast(); return FALSE; };

virtual BOOL IsConfigurable(PropertyId propertyId) override { AssertAndFailFast(); return FALSE; };

virtual BOOL IsEnumerable(PropertyId propertyId) override { AssertAndFailFast(); return FALSE; };

virtual BOOL SetEnumerable(PropertyId propertyId, BOOL value) override { AssertAndFailFast(); return FALSE; };

virtual BOOL SetWritable(PropertyId propertyId, BOOL value) override { AssertAndFailFast(); return FALSE; };

virtual BOOL SetConfigurable(PropertyId propertyId, BOOL value) override { AssertAndFailFast(); return FALSE; };

virtual BOOL SetAttributes(PropertyId propertyId, PropertyAttributes attributes) override { AssertAndFailFast(); return FALSE; };

virtual BOOL IsExtensible() override { AssertAndFailFast(); return FALSE; };

virtual BOOL PreventExtensions() override { AssertAndFailFast(); return FALSE; };

virtual BOOL Seal() override { AssertAndFailFast(); return FALSE; };

virtual BOOL Freeze() override { AssertAndFailFast(); return FALSE; };

virtual BOOL IsSealed() override { AssertAndFailFast(); return FALSE; };

virtual BOOL IsFrozen() override { AssertAndFailFast(); return FALSE; };

virtual BOOL GetDiagValueString(StringBuilder<ArenaAllocator>\* stringBuilder, ScriptContext\* requestContext) override { AssertAndFailFast(); return FALSE; };

virtual Var GetTypeOfString(ScriptContext \* requestContext) override { AssertAndFailFast(); return RecyclableObject::GetTypeOfString(requestContext); };

};

}

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

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

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

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

#include "RuntimeTypePch.h"

namespace Js

{

StaticType \*

StaticType::New(ScriptContext\* scriptContext, TypeId typeId, RecyclableObject\* prototype, JavascriptMethod entryPoint)

{

return RecyclerNew(scriptContext->GetRecycler(), StaticType, scriptContext, typeId, prototype, entryPoint);

}

bool

StaticType::Is(TypeId typeId)

{

return typeId <= TypeIds\_LastStaticType;

}

BOOL RecyclableObject::GetDiagValueString(StringBuilder<ArenaAllocator>\* stringBuilder, ScriptContext\* requestContext)

{

ENTER\_PINNED\_SCOPE(JavascriptString, valueStr);

ScriptContext \*scriptContext = GetScriptContext();

switch(GetTypeId())

{

case TypeIds\_Undefined:

valueStr = GetLibrary()->GetUndefinedDisplayString();

break;

case TypeIds\_Null:

valueStr = GetLibrary()->GetNullDisplayString();

break;

case TypeIds\_Integer:

valueStr = scriptContext->GetIntegerString(this);

break;

case TypeIds\_Boolean:

valueStr = JavascriptBoolean::FromVar(this)->GetValue() ?

GetLibrary()->GetTrueDisplayString()

: GetLibrary()->GetFalseDisplayString();

break;

case TypeIds\_Number:

valueStr = JavascriptNumber::ToStringRadix10(JavascriptNumber::GetValue(this), scriptContext);

break;

case TypeIds\_String:

valueStr = JavascriptString::FromVar(this);

break;

default:

valueStr = GetLibrary()->GetUndefinedDisplayString();

}

stringBuilder->Append(valueStr->GetString(), valueStr->GetLength());

LEAVE\_PINNED\_SCOPE();

return TRUE;

}

BOOL RecyclableObject::GetDiagTypeString(StringBuilder<ArenaAllocator>\* stringBuilder, ScriptContext\* requestContext)

{

switch(GetTypeId())

{

case TypeIds\_Undefined:

stringBuilder->AppendCppLiteral(L"Undefined");

break;

case TypeIds\_Null:

stringBuilder->AppendCppLiteral(L"Null");

break;

case TypeIds\_Integer:

case TypeIds\_Number:

stringBuilder->AppendCppLiteral(L"Number");

break;

case TypeIds\_Boolean:

stringBuilder->AppendCppLiteral(L"Boolean");

break;

case TypeIds\_String:

stringBuilder->AppendCppLiteral(L"String");

break;

default:

stringBuilder->AppendCppLiteral(L"Object, (Static Type)");

break;

}

return TRUE;

}

}

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

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

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

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

#pragma once

namespace Js

{

class StaticType : public Type

{

public:

StaticType(StaticType \* type) : Type(type) {}

StaticType(ScriptContext\* scriptContext, TypeId typeId, RecyclableObject\* prototype, JavascriptMethod entryPoint) :

Type(scriptContext, typeId, prototype, entryPoint)

{

Assert(StaticType::Is(typeId));

#ifdef HEAP\_ENUMERATION\_VALIDATION

if (prototype) prototype->SetHeapEnumValidationCookie(HEAP\_ENUMERATION\_LIBRARY\_OBJECT\_COOKIE);

#endif

}

void SetDispatchInvoke(JavascriptMethod method) { Assert(typeId == TypeIds\_HostDispatch); entryPoint = method; }

public:

static bool Is(TypeId typeId);

static StaticType \* New(ScriptContext\* scriptContext, TypeId typeId, RecyclableObject\* prototype, JavascriptMethod entryPoint);

};

};

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

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

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

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

#include "RuntimeTypePch.h"

namespace Js

{

DEFINE\_RECYCLER\_TRACKER\_WEAKREF\_PERF\_COUNTER(Type);

InternalString Type::ObjectTypeNameString = InternalString(L"object", 6);

InternalString Type::UndefinedTypeNameString = InternalString(L"undefined", 9);

InternalString Type::BooleanTypeNameString = InternalString(L"boolean", 7);

InternalString Type::StringTypeNameString = InternalString(L"string", 6);

InternalString Type::NumberTypeNameString = InternalString(L"number", 6);

InternalString Type::FunctionTypeNameString = InternalString(L"function", 8);

Type::Type(ScriptContext\* scriptContext, TypeId typeId, RecyclableObject\* prototype, JavascriptMethod entryPoint) :

javascriptLibrary(scriptContext->GetLibrary()),

typeId(typeId),

prototype(prototype),

propertyCache(nullptr),

flags(TypeFlagMask\_None)

{

#ifdef PROFILE\_TYPES

if (typeId < sizeof(scriptContext->typeCount)/sizeof(int))

{

scriptContext->typeCount[typeId]++;

}

#endif

this->entryPoint = entryPoint != nullptr ? entryPoint : RecyclableObject::DefaultEntryPoint;

if (prototype)

{

Assert(! CrossSite::NeedMarshalVar(prototype,scriptContext));

prototype->SetIsPrototype();

}

}

Type::Type(Type \* type) :

typeId(type->typeId),

javascriptLibrary(type->javascriptLibrary),

prototype(type->prototype),

entryPoint(type->entryPoint),

flags(type->flags),

propertyCache(nullptr)

{

#ifdef PROFILE\_TYPES

if (typeId < sizeof(javascriptLibrary->GetScriptContext()->typeCount)/sizeof(int))

{

javascriptLibrary->GetScriptContext()->typeCount[typeId]++;

}

#endif

Assert(! (prototype && CrossSite::NeedMarshalVar(prototype, javascriptLibrary->GetScriptContext())));

// If the type property cache is copied over to this new type, then if a property ID caused the type to be changed for

// the purpose of invalidating caches due to the property being deleted or its attributes being changed, then the cache

// for that property ID must be cleared on this new type after the type property cache is copied. Also, types are not

// changed consistently to use this copy constructor, so those would need to be fixed as well.

if(type->AreThisAndPrototypesEnsuredToHaveOnlyWritableDataProperties())

{

SetAreThisAndPrototypesEnsuredToHaveOnlyWritableDataProperties(true);

}

if(type->IsFalsy())

{

SetIsFalsy(true);

}

}

ScriptContext \*

Type::GetScriptContext() const

{

return GetLibrary()->GetScriptContext();

}

Recycler \*

Type::GetRecycler() const

{

return GetLibrary()->GetRecycler();

}

TypePropertyCache \*Type::GetPropertyCache()

{

return propertyCache;

}

TypePropertyCache \*Type::CreatePropertyCache()

{

Assert(!propertyCache);

propertyCache = RecyclerNew(GetRecycler(), TypePropertyCache);

return propertyCache;

}

void Type::SetAreThisAndPrototypesEnsuredToHaveOnlyWritableDataProperties(const bool truth)

{

if (truth)

{

if (GetScriptContext()->IsClosed())

{

// The cache is disabled after the script context is closed, to avoid issues between being closed and being deleted,

// where the cache of these types in JavascriptLibrary may be reclaimed at any point

return;

}

flags |= TypeFlagMask\_AreThisAndPrototypesEnsuredToHaveOnlyWritableDataProperties;

javascriptLibrary->TypeAndPrototypesAreEnsuredToHaveOnlyWritableDataProperties(this);

}

else

{

flags &= ~TypeFlagMask\_AreThisAndPrototypesEnsuredToHaveOnlyWritableDataProperties;

}

}

BOOL Type::AreThisAndPrototypesEnsuredToHaveOnlyWritableDataProperties() const

{

return flags & TypeFlagMask\_AreThisAndPrototypesEnsuredToHaveOnlyWritableDataProperties;

}

BOOL Type::IsFalsy() const

{

return flags & TypeFlagMask\_IsFalsy;

}

void Type::SetIsFalsy(const bool truth)

{

if (truth)

{

Assert(this->GetScriptContext()->GetThreadContext()->CanBeFalsy(this->GetTypeId()));

flags |= TypeFlagMask\_IsFalsy;

}

else

{

flags &= ~TypeFlagMask\_IsFalsy;

}

}

void Type::SetHasSpecialPrototype(const bool truth)

{

if (truth)

{

flags |= TypeFlagMask\_HasSpecialPrototype;

}

else

{

flags &= ~TypeFlagMask\_HasSpecialPrototype;

}

}

uint32 Type::GetOffsetOfTypeId()

{

return offsetof(Type, typeId);

}

uint32 Type::GetOffsetOfFlags()

{

return offsetof(Type, flags);

}

uint32 Type::GetOffsetOfEntryPoint()

{

return offsetof(Type, entryPoint);

}

uint32 Type::GetOffsetOfPrototype()

{

return offsetof(Type, prototype);

}

#if defined(PROFILE\_RECYCLER\_ALLOC) && defined(RECYCLER\_DUMP\_OBJECT\_GRAPH)

bool Type::DumpObjectFunction(type\_info const \* typeinfo, bool isArray, void \* objectAddress)

{

if (isArray)

{

// Don't deal with array

return false;

}

Output::Print(L"%S{%x} %p", typeinfo->name(), ((Type \*)objectAddress)->GetTypeId(), objectAddress);

return true;

}

#endif

}

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

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

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

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

#pragma once

enum TypeFlagMask : uint8

{

TypeFlagMask\_None = 0x00,

TypeFlagMask\_AreThisAndPrototypesEnsuredToHaveOnlyWritableDataProperties = 0x01,

TypeFlagMask\_IsFalsy = 0x02,

TypeFlagMask\_HasSpecialPrototype = 0x04,

TypeFlagMask\_External = 0x08,

TypeFlagMask\_SkipsPrototype = 0x10,

TypeFlagMask\_CanHaveInterceptors = 0x20,

};

ENUM\_CLASS\_HELPERS(TypeFlagMask, uint8);

namespace Js

{

class TypePropertyCache;

class Type

{

friend class DynamicObject;

friend class GlobalObject;

friend class ScriptEngineBase;

protected:

TypeId typeId;

TypeFlagMask flags;

JavascriptLibrary\* javascriptLibrary;

RecyclableObject\* prototype;

JavascriptMethod entryPoint;

private:

TypePropertyCache \*propertyCache;

protected:

Type(Type \* type);

Type(ScriptContext\* scriptContext, TypeId typeId, RecyclableObject\* prototype, JavascriptMethod entryPoint);

public:

static DWORD GetJavascriptLibraryOffset() { return offsetof(Type, javascriptLibrary); }

\_\_inline TypeId GetTypeId() const { return typeId; }

void SetTypeId(TypeId typeId) { this->typeId = typeId; }

RecyclableObject\* GetPrototype() const { return prototype; }

JavascriptMethod GetEntryPoint() const { return entryPoint; }

JavascriptLibrary\* GetLibrary() const { return javascriptLibrary; }

ScriptContext \* GetScriptContext() const;

Recycler \* GetRecycler() const;

TypePropertyCache \*GetPropertyCache();

TypePropertyCache \*CreatePropertyCache();

BOOL HasSpecialPrototype() const { return (flags & TypeFlagMask\_HasSpecialPrototype) == TypeFlagMask\_HasSpecialPrototype; }

// This function has a different meaning from RecyclableObject::HasOnlyWritableDataProperties. If this function returns

// true, then it's implied that RecyclableObject::HasOnlyWritableDataProperties would return true for an object of this

// type and all of its prototypes. However, if this function returns false, it does not imply the converse.

BOOL AreThisAndPrototypesEnsuredToHaveOnlyWritableDataProperties() const;

void SetAreThisAndPrototypesEnsuredToHaveOnlyWritableDataProperties(const bool truth);

\_\_inline BOOL IsExternal() const { return (this->flags & TypeFlagMask\_External) != 0; }

\_\_inline BOOL SkipsPrototype() const { return (this->flags & TypeFlagMask\_SkipsPrototype) != 0 ; }

\_\_inline BOOL CanHaveInterceptors() const { return (this->flags & TypeFlagMask\_CanHaveInterceptors) != 0; }

\_\_inline BOOL IsFalsy() const;

void SetIsFalsy(const bool truth);

void SetHasSpecialPrototype(const bool hasSpecialPrototype);

// This is for static lib verification use only.

static DWORD GetTypeIdFieldOffset() { return offsetof(Type, typeId); }

static size\_t OffsetOfWritablePropertiesFlag()

{

return offsetof(Type, flags);

}

static uint32 GetOffsetOfTypeId();

static uint32 GetOffsetOfFlags();

static uint32 GetOffsetOfEntryPoint();

static uint32 GetOffsetOfPrototype();

static InternalString UndefinedTypeNameString;

static InternalString ObjectTypeNameString;

static InternalString BooleanTypeNameString;

static InternalString NumberTypeNameString;

static InternalString StringTypeNameString;

static InternalString FunctionTypeNameString;

#if defined(PROFILE\_RECYCLER\_ALLOC) && defined(RECYCLER\_DUMP\_OBJECT\_GRAPH)

static bool DumpObjectFunction(type\_info const \* typeinfo, bool isArray, void \* objectAddress);

#endif

};

};

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

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

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

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

#include "RuntimeTypePch.h"

namespace Js

{

BigPropertyIndex

DynamicTypeHandler::GetPropertyIndexFromInlineSlotIndex(uint inlineSlot)

{

return inlineSlot - (offsetOfInlineSlots / sizeof(Var \*));

}

BigPropertyIndex

DynamicTypeHandler::GetPropertyIndexFromAuxSlotIndex(uint auxIndex)

{

return auxIndex + this->GetInlineSlotCapacity();

}

PropertyIndex DynamicTypeHandler::RoundUpObjectHeaderInlinedInlineSlotCapacity(const PropertyIndex slotCapacity)

{

const PropertyIndex objectHeaderInlinableSlotCapacity = GetObjectHeaderInlinableSlotCapacity();

if(slotCapacity <= objectHeaderInlinableSlotCapacity)

{

return objectHeaderInlinableSlotCapacity;

}

// Align the slot capacity for slots that are outside the object header, and add to that the slot capacity for slots

// that are inside the object header

return RoundUpInlineSlotCapacity(slotCapacity - objectHeaderInlinableSlotCapacity) + objectHeaderInlinableSlotCapacity;

}

PropertyIndex DynamicTypeHandler::RoundUpInlineSlotCapacity(const PropertyIndex slotCapacity)

{

return ::Math::Align<PropertyIndex>(slotCapacity, HeapConstants::ObjectGranularity / sizeof(Var));

}

int DynamicTypeHandler::RoundUpAuxSlotCapacity(const int slotCapacity)

{

CompileAssert(4 \* sizeof(Var) % HeapConstants::ObjectGranularity == 0);

return ::Math::Align<int>(slotCapacity, 4);

}

int DynamicTypeHandler::RoundUpSlotCapacity(const int slotCapacity, const PropertyIndex inlineSlotCapacity)

{

Assert(slotCapacity >= 0);

if(slotCapacity <= inlineSlotCapacity)

{

return inlineSlotCapacity;

}

const int auxSlotCapacity = RoundUpAuxSlotCapacity(slotCapacity - inlineSlotCapacity);

Assert(auxSlotCapacity + inlineSlotCapacity >= auxSlotCapacity);

const int maxSlotCapacity =

slotCapacity <= PropertyIndexRanges<PropertyIndex>::MaxValue

? PropertyIndexRanges<PropertyIndex>::MaxValue

: PropertyIndexRanges<BigPropertyIndex>::MaxValue;

return min(maxSlotCapacity, inlineSlotCapacity + auxSlotCapacity);

}

DynamicTypeHandler::DynamicTypeHandler(int slotCapacity, uint16 inlineSlotCapacity, uint16 offsetOfInlineSlots, BYTE flags) :

flags(flags),

propertyTypes(PropertyTypesWritableDataOnly | PropertyTypesReserved),

offsetOfInlineSlots(offsetOfInlineSlots),

unusedBytes(Js::AtomTag)

{

Assert(!GetIsOrMayBecomeShared() || GetIsLocked());

Assert(offsetOfInlineSlots != 0 || inlineSlotCapacity == 0);

Assert(!IsObjectHeaderInlined(offsetOfInlineSlots) || inlineSlotCapacity != 0);

// Align the slot capacities and set the total slot capacity

this->inlineSlotCapacity = inlineSlotCapacity =

IsObjectHeaderInlined(offsetOfInlineSlots)

? RoundUpObjectHeaderInlinedInlineSlotCapacity(inlineSlotCapacity)

: RoundUpInlineSlotCapacity(inlineSlotCapacity);

this->slotCapacity = RoundUpSlotCapacity(slotCapacity, inlineSlotCapacity);

this->isNotPathTypeHandlerOrHasUserDefinedCtor = true;

Assert(IsObjectHeaderInlinedTypeHandler() == IsObjectHeaderInlined(offsetOfInlineSlots));

}

Var DynamicTypeHandler::GetSlot(DynamicObject \* instance, int index)

{

if (index < inlineSlotCapacity)

{

Var \* slots = reinterpret\_cast<Var\*>(reinterpret\_cast<size\_t>(instance) + offsetOfInlineSlots);

Var value = slots[index];

Assert(ThreadContext::IsOnStack(instance) || !ThreadContext::IsOnStack(value) || TaggedNumber::Is(value));

return value;

}

else

{

Var value = instance->auxSlots[index - inlineSlotCapacity];

Assert(ThreadContext::IsOnStack(instance) || !ThreadContext::IsOnStack(value) || TaggedNumber::Is(value));

return value;

}

}

Var DynamicTypeHandler::GetInlineSlot(DynamicObject \* instance, int index)

{

AssertMsg(index >= (int)(offsetOfInlineSlots / sizeof(Var)), "index should be relative to the address of the object");

Assert(index - (int)(offsetOfInlineSlots / sizeof(Var)) < this->GetInlineSlotCapacity());

Var \* slots = reinterpret\_cast<Var\*>(instance);

Var value = slots[index];

Assert(ThreadContext::IsOnStack(instance) || !ThreadContext::IsOnStack(value) || TaggedNumber::Is(value));

return value;

}

Var DynamicTypeHandler::GetAuxSlot(DynamicObject \* instance, int index)

{

// We should only assign a stack value only to an stack object (current mark temp number in mark temp object)

Assert(index < GetSlotCapacity() - GetInlineSlotCapacity());

Var value = instance->auxSlots[index];

Assert(ThreadContext::IsOnStack(instance) || !ThreadContext::IsOnStack(value) || TaggedNumber::Is(value));

return value;

}

#if DBG

void DynamicTypeHandler::SetSlot(DynamicObject\* instance, PropertyId propertyId, bool allowLetConst, int index, Var value)

#else

void DynamicTypeHandler::SetSlot(DynamicObject\* instance, int index, Var value)

#endif

{

Assert(index < GetSlotCapacity());

Assert(propertyId == Constants::NoProperty || CanStorePropertyValueDirectly(instance, propertyId, allowLetConst));

SetSlotUnchecked(instance, index, value);

}

void DynamicTypeHandler::SetSlotUnchecked(DynamicObject \* instance, int index, Var value)

{

// We should only assign a stack value only to an stack object (current mark temp number in mark temp object)

Assert(ThreadContext::IsOnStack(instance) || !ThreadContext::IsOnStack(value) || TaggedNumber::Is(value));

uint16 inlineSlotCapacity = instance->GetTypeHandler()->GetInlineSlotCapacity();

uint16 offsetOfInlineSlots = instance->GetTypeHandler()->GetOffsetOfInlineSlots();

int slotCapacity = instance->GetTypeHandler()->GetSlotCapacity();

if (index < inlineSlotCapacity)

{

Var \* slots = reinterpret\_cast<Var\*>(reinterpret\_cast<size\_t>(instance) + offsetOfInlineSlots);

slots[index] = value;

}

else

{

Assert((index - inlineSlotCapacity) < (slotCapacity - inlineSlotCapacity));

instance->auxSlots[index - inlineSlotCapacity] = value;

}

}

#if DBG

void DynamicTypeHandler::SetInlineSlot(DynamicObject\* instance, PropertyId propertyId, bool allowLetConst, int index, Var value)

#else

void DynamicTypeHandler::SetInlineSlot(DynamicObject\* instance, int index, Var value)

#endif

{

// We should only assign a stack value only to an stack object (current mark temp number in mark temp object)

Assert(ThreadContext::IsOnStack(instance) || !ThreadContext::IsOnStack(value) || TaggedNumber::Is(value));

AssertMsg(index >= (int)(offsetOfInlineSlots / sizeof(Var)), "index should be relative to the address of the object");

Assert(index - (int)(offsetOfInlineSlots / sizeof(Var)) < this->GetInlineSlotCapacity());

Assert(propertyId == Constants::NoProperty || CanStorePropertyValueDirectly(instance, propertyId, allowLetConst));

Var \* slots = reinterpret\_cast<Var\*>(instance);

slots[index] = value;

}

#if DBG

void DynamicTypeHandler::SetAuxSlot(DynamicObject\* instance, PropertyId propertyId, bool allowLetConst, int index, Var value)

#else

void DynamicTypeHandler::SetAuxSlot(DynamicObject\* instance, int index, Var value)

#endif

{

// We should only assign a stack value only to an stack object (current mark temp number in mark temp object)

Assert(ThreadContext::IsOnStack(instance) || !ThreadContext::IsOnStack(value) || TaggedNumber::Is(value));

Assert(index < GetSlotCapacity() - GetInlineSlotCapacity());

Assert(propertyId == Constants::NoProperty || CanStorePropertyValueDirectly(instance, propertyId, allowLetConst));

instance->auxSlots[index] = value;

}

void

DynamicTypeHandler::SetInstanceTypeHandler(DynamicObject \* instance, bool hasChanged)

{

SetInstanceTypeHandler(instance, this, hasChanged);

}

bool DynamicTypeHandler::IsObjectHeaderInlined(const uint16 offsetOfInlineSlots)

{

return offsetOfInlineSlots == GetOffsetOfObjectHeaderInlineSlots();

}

bool DynamicTypeHandler::IsObjectHeaderInlinedTypeHandlerUnchecked() const

{

return IsObjectHeaderInlined(GetOffsetOfInlineSlots());

}

bool DynamicTypeHandler::IsObjectHeaderInlinedTypeHandler() const

{

const bool isObjectHeaderInlined = IsObjectHeaderInlinedTypeHandlerUnchecked();

if(isObjectHeaderInlined)

{

VerifyObjectHeaderInlinedTypeHandler();

}

return isObjectHeaderInlined;

}

void DynamicTypeHandler::VerifyObjectHeaderInlinedTypeHandler() const

{

Assert(IsObjectHeaderInlined(GetOffsetOfInlineSlots()));

Assert(GetInlineSlotCapacity() >= GetObjectHeaderInlinableSlotCapacity());

Assert(GetInlineSlotCapacity() == GetSlotCapacity());

}

uint16 DynamicTypeHandler::GetOffsetOfObjectHeaderInlineSlots()

{

return offsetof(DynamicObject, auxSlots);

}

PropertyIndex DynamicTypeHandler::GetObjectHeaderInlinableSlotCapacity()

{

const PropertyIndex maxAllowedSlotCapacity = (sizeof(DynamicObject) - DynamicTypeHandler::GetOffsetOfObjectHeaderInlineSlots()) / sizeof(Var);

AssertMsg(maxAllowedSlotCapacity == 2, "Today we should be getting 2 with the math here. Change this Assert, if we are changing this logic in the future");

return maxAllowedSlotCapacity;

}

void

DynamicTypeHandler::SetInstanceTypeHandler(DynamicObject \* instance, DynamicTypeHandler \* typeHandler, bool hasChanged)

{

instance->SetTypeHandler(typeHandler, hasChanged);

}

DynamicTypeHandler \*

DynamicTypeHandler::GetCurrentTypeHandler(DynamicObject \* instance)

{

return instance->GetTypeHandler();

}

void

DynamicTypeHandler::ReplaceInstanceType(DynamicObject \* instance, DynamicType \* type)

{

instance->ReplaceType(type);

}

void

DynamicTypeHandler::ResetTypeHandler(DynamicObject \* instance)

{

// just reuse the current type handler.

this->SetInstanceTypeHandler(instance);

}

BOOL

DynamicTypeHandler::FindNextProperty(ScriptContext\* scriptContext, BigPropertyIndex& index, JavascriptString\*\* propertyString,

PropertyId\* propertyId, PropertyAttributes\* attributes, Type\* type, DynamicType \*typeToEnumerate, bool requireEnumerable, bool enumSymbols)

{

// Type handlers that support big property indexes override this function, so if we're here then this type handler does

// not support big property indexes. Forward the call to the small property index version.

Assert(GetSlotCapacity() <= PropertyIndexRanges<PropertyIndex>::MaxValue);

PropertyIndex smallIndex = static\_cast<PropertyIndex>(index);

Assert(static\_cast<BigPropertyIndex>(smallIndex) == index);

const BOOL found = FindNextProperty(scriptContext, smallIndex, propertyString, propertyId, attributes, type, typeToEnumerate, requireEnumerable, enumSymbols);

index = smallIndex;

return found;

}

template<bool isStoreField>

void DynamicTypeHandler::InvalidateInlineCachesForAllProperties(ScriptContext\* requestContext)

{

int count = GetPropertyCount();

if (count < 128) // Invalidate a propertyId involves dictionary lookups. Only do this when the number is relatively small.

{

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

{

PropertyId propertyId = GetPropertyId(requestContext, static\_cast<PropertyIndex>(i));

if (propertyId != Constants::NoProperty)

{

isStoreField ? requestContext->InvalidateStoreFieldCaches(propertyId) : requestContext->InvalidateProtoCaches(propertyId);

}

}

}

else

{

isStoreField ? requestContext->InvalidateAllStoreFieldCaches() : requestContext->InvalidateAllProtoCaches();

}

}

void DynamicTypeHandler::InvalidateProtoCachesForAllProperties(ScriptContext\* requestContext)

{

InvalidateInlineCachesForAllProperties<false>(requestContext);

}

void DynamicTypeHandler::InvalidateStoreFieldCachesForAllProperties(ScriptContext\* requestContext)

{

InvalidateInlineCachesForAllProperties<true>(requestContext);

}

void DynamicTypeHandler::RemoveFromPrototype(DynamicObject\* instance, ScriptContext \* requestContext)

{

InvalidateProtoCachesForAllProperties(requestContext);

}

void DynamicTypeHandler::AddToPrototype(DynamicObject\* instance, ScriptContext \* requestContext)

{

InvalidateStoreFieldCachesForAllProperties(requestContext);

}

void DynamicTypeHandler::SetPrototype(DynamicObject\* instance, RecyclableObject\* newPrototype)

{

// Force a type transition on the instance to invalidate its inline caches

DynamicTypeHandler::ResetTypeHandler(instance);

// Put new prototype in place

instance->GetDynamicType()->SetPrototype(newPrototype);

}

bool DynamicTypeHandler::TryUseFixedProperty(PropertyRecord const\* propertyRecord, Var \* pProperty, FixedPropertyKind propertyType, ScriptContext \* requestContext)

{

if (PHASE\_VERBOSE\_TRACE1(Js::FixedMethodsPhase) || PHASE\_VERBOSE\_TESTTRACE1(Js::FixedMethodsPhase) ||

PHASE\_VERBOSE\_TRACE1(Js::UseFixedDataPropsPhase) || PHASE\_VERBOSE\_TESTTRACE1(Js::UseFixedDataPropsPhase))

{

Output::Print(L"FixedFields: attempt to use fixed property %s from DynamicTypeHandler returned false.\n", propertyRecord->GetBuffer());

if (this->HasSingletonInstance() && this->GetSingletonInstance()->Get()->GetScriptContext() != requestContext)

{

Output::Print(L"FixedFields: Cross Site Script Context is used for property %s. \n", propertyRecord->GetBuffer());

}

Output::Flush();

}

return false;

}

bool DynamicTypeHandler::TryUseFixedAccessor(PropertyRecord const\* propertyRecord, Var \* pAccessor, FixedPropertyKind propertyType, bool getter, ScriptContext \* requestContext)

{

if (PHASE\_VERBOSE\_TRACE1(Js::FixedMethodsPhase) || PHASE\_VERBOSE\_TESTTRACE1(Js::FixedMethodsPhase) ||

PHASE\_VERBOSE\_TRACE1(Js::UseFixedDataPropsPhase) || PHASE\_VERBOSE\_TESTTRACE1(Js::UseFixedDataPropsPhase))

{

Output::Print(L"FixedFields: attempt to use fixed accessor %s from DynamicTypeHandler returned false.\n", propertyRecord->GetBuffer());

if (this->HasSingletonInstance() && this->GetSingletonInstance()->Get()->GetScriptContext() != requestContext)

{

Output::Print(L"FixedFields: Cross Site Script Context is used for property %s. \n", propertyRecord->GetBuffer());

}

Output::Flush();

}

return false;

}

bool DynamicTypeHandler::IsFixedMethodProperty(FixedPropertyKind fixedPropKind)

{

return (fixedPropKind & Js::FixedPropertyKind::FixedMethodProperty) == Js::FixedPropertyKind::FixedMethodProperty;

}

bool DynamicTypeHandler::IsFixedDataProperty(FixedPropertyKind fixedPropKind)

{

return ((fixedPropKind & Js::FixedPropertyKind::FixedDataProperty) == Js::FixedPropertyKind::FixedDataProperty) &&

!PHASE\_OFF1(UseFixedDataPropsPhase);

}

bool DynamicTypeHandler::IsFixedAccessorProperty(FixedPropertyKind fixedPropKind)

{

return (fixedPropKind & Js::FixedPropertyKind::FixedAccessorProperty) == Js::FixedPropertyKind::FixedAccessorProperty;

}

bool DynamicTypeHandler::CheckHeuristicsForFixedDataProps(DynamicObject\* instance, const PropertyRecord \* propertyRecord, Var value)

{

if (PHASE\_FORCE1(Js::FixDataPropsPhase))

{

return true;

}

if (Js::TaggedInt::Is(value) &&

((instance->GetTypeId() == TypeIds\_GlobalObject && instance->GetScriptContext()->IsIntConstPropertyOnGlobalObject(propertyRecord->GetPropertyId())) ||

(instance->GetTypeId() == TypeIds\_Object && instance->GetScriptContext()->IsIntConstPropertyOnGlobalUserObject(propertyRecord->GetPropertyId()))))

{

return true;

}

// Disabled by default

if (PHASE\_ON1(Js::FixDataVarPropsPhase))

{

if (instance->GetTypeHandler()->GetFlags() & IsPrototypeFlag)

{

return true;

}

if (instance->GetType()->GetTypeId() == TypeIds\_GlobalObject)

{

// if we have statically seen multiple stores - we should not do this optimization

RootObjectInlineCache\* cache = (static\_cast<Js::RootObjectBase\*>(instance))->GetRootInlineCache(propertyRecord, /\*isLoadMethod\*/ false, /\*isStore\*/ true);

uint refCount = cache->Release();

return refCount <= 1;

}

}

return false;

}

bool DynamicTypeHandler::CheckHeuristicsForFixedDataProps(DynamicObject\* instance, PropertyId propertyId, Var value)

{

return CheckHeuristicsForFixedDataProps(instance, instance->GetScriptContext()->GetPropertyName(propertyId), value);

}

bool DynamicTypeHandler::CheckHeuristicsForFixedDataProps(DynamicObject\* instance, JavascriptString \* propertyKey, Var value)

{

return false;

}

bool DynamicTypeHandler::CheckHeuristicsForFixedDataProps(DynamicObject\* instance, const PropertyRecord \* propertyRecord, PropertyId propertyId, Var value)

{

if(propertyRecord)

{

return CheckHeuristicsForFixedDataProps(instance, propertyRecord, value);

}

else

{

return CheckHeuristicsForFixedDataProps(instance,propertyId,value);

}

}

void DynamicTypeHandler::TraceUseFixedProperty(PropertyRecord const \* propertyRecord, Var \* pProperty, bool result, LPCWSTR typeHandlerName, ScriptContext \* requestContext)

{

LPCWSTR fixedPropertyResultType = nullptr;

bool log = false;

if (pProperty && \*pProperty && ((Js::JavascriptFunction::Is(\*pProperty) && (PHASE\_VERBOSE\_TRACE1(Js::FixedMethodsPhase) || PHASE\_VERBOSE\_TESTTRACE1(Js::FixedMethodsPhase))) ||

((PHASE\_VERBOSE\_TRACE1(Js::UseFixedDataPropsPhase) || PHASE\_VERBOSE\_TESTTRACE1(Js::UseFixedDataPropsPhase))) ))

{

if(\*pProperty == nullptr)

{

fixedPropertyResultType = L"null";

}

else if (Js::JavascriptFunction::Is(\*pProperty))

{

fixedPropertyResultType = L"function";

}

else if (TaggedInt::Is(\*pProperty))

{

fixedPropertyResultType = L"int constant";

}

else

{

fixedPropertyResultType = L"Var";

}

log = true;

}

if(log)

{

Output::Print(L"FixedFields: attempt to use fixed property %s, which is a %s, from %s returned %s.\n",

propertyRecord->GetBuffer(), fixedPropertyResultType, typeHandlerName, IsTrueOrFalse(result));

if (this->HasSingletonInstance() && this->GetSingletonInstance()->Get()->GetScriptContext() != requestContext)

{

Output::Print(L"FixedFields: Cross Site Script Context is used for property %s. \n", propertyRecord->GetBuffer());

}

Output::Flush();

}

}

BOOL DynamicTypeHandler::GetInternalProperty(DynamicObject\* instance, Var originalInstance, PropertyId propertyId, Var\* value)

{

// Type handlers that store internal properties differently from normal properties

// override this method to provide access to them. Otherwise, by default, simply

// defer to GetProperty()

return this->GetProperty(instance, originalInstance, propertyId, value, nullptr, nullptr);

}

BOOL DynamicTypeHandler::InitProperty(DynamicObject\* instance, PropertyId propertyId, Var value, PropertyOperationFlags flags, PropertyValueInfo\* info)

{

// By default just call the SetProperty method

return this->SetProperty(instance, propertyId, value, flags, info);

}

BOOL DynamicTypeHandler::SetInternalProperty(DynamicObject\* instance, PropertyId propertyId, Var value, PropertyOperationFlags flags)

{

// Type handlers that store internal properties differently from normal properties

// override this method to provide access to them. Otherwise, by default, simply

// defer to SetProperty()

return this->SetProperty(instance, propertyId, value, flags, nullptr);

}

//

// Default implementations delegate to instance objectArray

//

BOOL DynamicTypeHandler::HasItem(DynamicObject\* instance, uint32 index)

{

return instance->HasObjectArrayItem(index);

}

BOOL DynamicTypeHandler::SetItem(DynamicObject\* instance, uint32 index, Var value, PropertyOperationFlags flags)

{

return instance->SetObjectArrayItem(index, value, flags);

}

BOOL DynamicTypeHandler::DeleteItem(DynamicObject\* instance, uint32 index, PropertyOperationFlags flags)

{

return instance->DeleteObjectArrayItem(index, flags);

}

BOOL DynamicTypeHandler::GetItem(DynamicObject\* instance, Var originalInstance, uint32 index, Var\* value, ScriptContext \* requestContext)

{

return instance->GetObjectArrayItem(originalInstance, index, value, requestContext);

}

DescriptorFlags DynamicTypeHandler::GetSetter(DynamicObject\* instance, PropertyId propertyId, Var\* setterValue, PropertyValueInfo\* info, ScriptContext\* requestContext)

{

PropertyValueInfo::SetNoCache(info, instance);

return this->HasProperty(instance, propertyId) ? WritableData : None;

}

DescriptorFlags DynamicTypeHandler::GetSetter(DynamicObject\* instance, JavascriptString\* propertyNameString, Var\* setterValue, PropertyValueInfo\* info, ScriptContext\* requestContext)

{

PropertyValueInfo::SetNoCache(info, instance);

return this->HasProperty(instance, propertyNameString) ? WritableData : None;

}

DescriptorFlags DynamicTypeHandler::GetItemSetter(DynamicObject\* instance, uint32 index, Var\* setterValue, ScriptContext\* requestContext)

{

return this->HasItem(instance, index) ? WritableData : None;

}

//

// Default implementations upgrades type handler with item attribute/getter/setter support

//

BOOL DynamicTypeHandler::SetItemWithAttributes(DynamicObject\* instance, uint32 index, Var value, PropertyAttributes attributes)

{

return ConvertToTypeWithItemAttributes(instance)->SetItemWithAttributes(instance, index, value, attributes);

}

BOOL DynamicTypeHandler::SetItemAttributes(DynamicObject\* instance, uint32 index, PropertyAttributes attributes)

{

return ConvertToTypeWithItemAttributes(instance)->SetItemAttributes(instance, index, attributes);

}

BOOL DynamicTypeHandler::SetItemAccessors(DynamicObject\* instance, uint32 index, Var getter, Var setter)

{

return ConvertToTypeWithItemAttributes(instance)->SetItemAccessors(instance, index, getter, setter);

}

void DynamicTypeHandler::SetPropertyUpdateSideEffect(DynamicObject\* instance, PropertyId propertyId, Var value, SideEffects possibleSideEffects)

{

if (possibleSideEffects && propertyId < PropertyIds::\_countJSOnlyProperty)

{

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

if (scriptContext->GetConfig()->IsES6ToPrimitiveEnabled() && propertyId == PropertyIds::\_symbolToPrimitive)

{

scriptContext->optimizationOverrides.SetSideEffects((SideEffects)(SideEffects\_ValueOf & possibleSideEffects));

scriptContext->optimizationOverrides.SetSideEffects((SideEffects)(SideEffects\_ToString & possibleSideEffects));

}

else if (propertyId == PropertyIds::valueOf)

{

scriptContext->optimizationOverrides.SetSideEffects((SideEffects)(SideEffects\_ValueOf & possibleSideEffects));

}

else if (propertyId == PropertyIds::toString)

{

scriptContext->optimizationOverrides.SetSideEffects((SideEffects)(SideEffects\_ToString & possibleSideEffects));

}

else if (IsMathLibraryId(propertyId))

{

if (instance == scriptContext->GetLibrary()->GetMathObject())

{

scriptContext->optimizationOverrides.SetSideEffects((SideEffects)(SideEffects\_MathFunc & possibleSideEffects));

}

}

else if (propertyId == PropertyIds::Math)

{

if (instance == scriptContext->GetLibrary()->GetGlobalObject())

{

scriptContext->optimizationOverrides.SetSideEffects((SideEffects)(SideEffects\_MathFunc & possibleSideEffects));

}

}

}

}

bool DynamicTypeHandler::VerifyIsExtensible(ScriptContext\* scriptContext, bool alwaysThrow)

{

if (!(this->GetFlags() & IsExtensibleFlag))

{

if (alwaysThrow)

{

if (scriptContext && scriptContext->GetThreadContext()->RecordImplicitException())

{

JavascriptError::ThrowTypeError(scriptContext, JSERR\_NonExtensibleObject);

}

}

return false;

}

return true;

}

void DynamicTypeHandler::EnsureSlots(DynamicObject\* instance, int oldCount, int newCount, ScriptContext \* scriptContext, DynamicTypeHandler \* newTypeHandler)

{

Assert(oldCount == instance->GetTypeHandler()->GetSlotCapacity());

AssertMsg(oldCount <= newCount, "Old count should be less than or equal to new count");

if (oldCount < newCount && newCount > GetInlineSlotCapacity())

{

const PropertyIndex newInlineSlotCapacity = newTypeHandler->GetInlineSlotCapacity();

Assert(newCount > newInlineSlotCapacity);

AdjustSlots(instance, newInlineSlotCapacity, newCount - newInlineSlotCapacity);

}

}

void DynamicTypeHandler::AdjustSlots\_Jit(

DynamicObject \*const object,

const PropertyIndex newInlineSlotCapacity,

const int newAuxSlotCapacity)

{

Assert(object);

// The JIT may call AdjustSlots multiple times on the same object, even after changing its type to the new type. Check

// if anything needs to be done.

DynamicTypeHandler \*const oldTypeHandler = object->GetTypeHandler();

const PropertyIndex oldInlineSlotCapacity = oldTypeHandler->GetInlineSlotCapacity();

if(oldInlineSlotCapacity == newInlineSlotCapacity &&

oldTypeHandler->GetSlotCapacity() - oldInlineSlotCapacity == newAuxSlotCapacity)

{

return;

}

AdjustSlots(object, newInlineSlotCapacity, newAuxSlotCapacity);

}

void DynamicTypeHandler::AdjustSlots(

DynamicObject \*const object,

const PropertyIndex newInlineSlotCapacity,

const int newAuxSlotCapacity)

{

Assert(object);

// Allocate new aux slot array

Recycler \*const recycler = object->GetRecycler();

TRACK\_ALLOC\_INFO(recycler, Var, Recycler, 0, newAuxSlotCapacity);

Var \*const newAuxSlots = reinterpret\_cast<Var \*>(recycler->AllocZero(newAuxSlotCapacity \* sizeof(Var)));

DynamicTypeHandler \*const oldTypeHandler = object->GetTypeHandler();

const PropertyIndex oldInlineSlotCapacity = oldTypeHandler->GetInlineSlotCapacity();

if(oldInlineSlotCapacity == newInlineSlotCapacity)

{

const int oldAuxSlotCapacity = oldTypeHandler->GetSlotCapacity() - oldInlineSlotCapacity;

Assert(oldAuxSlotCapacity < newAuxSlotCapacity);

if(oldAuxSlotCapacity > 0)

{

// Copy aux slots to the new array

Var \*const oldAuxSlots = object->auxSlots;

Assert(oldAuxSlots);

int i = 0;

do

{

newAuxSlots[i] = oldAuxSlots[i];

} while(++i < oldAuxSlotCapacity);

#ifdef EXPLICIT\_FREE\_SLOTS

recycler->ExplicitFreeNonLeaf(oldAuxSlots, oldAuxSlotCapacity \* sizeof(Var));

#endif

}

object->auxSlots = newAuxSlots;

return;

}

// An object header-inlined type handler is transitioning into one that is not. Some inline slots need to move, and

// there are no old aux slots that need to be copied.

Assert(oldTypeHandler->IsObjectHeaderInlinedTypeHandler());

Assert(oldInlineSlotCapacity > newInlineSlotCapacity);

Assert(oldInlineSlotCapacity - newInlineSlotCapacity == DynamicTypeHandler::GetObjectHeaderInlinableSlotCapacity());

Assert(newAuxSlotCapacity >= DynamicTypeHandler::GetObjectHeaderInlinableSlotCapacity());

// Move the last few inline slots into the aux slots

if(PHASE\_TRACE1(Js::ObjectHeaderInliningPhase))

{

Output::Print(L"ObjectHeaderInlining: Moving inlined properties to aux slots.\n");

Output::Flush();

}

Var \*const oldInlineSlots =

reinterpret\_cast<Var \*>(

reinterpret\_cast<uintptr\_t>(object) + DynamicTypeHandler::GetOffsetOfObjectHeaderInlineSlots());

Assert(DynamicTypeHandler::GetObjectHeaderInlinableSlotCapacity() == 2);

newAuxSlots[0] = oldInlineSlots[oldInlineSlotCapacity - 2];

newAuxSlots[1] = oldInlineSlots[oldInlineSlotCapacity - 1];

if(newInlineSlotCapacity > 0)

{

// Move the remaining inline slots such that none are object header-inlined. Copy backwards, as the two buffers may

// overlap, with the new inline slot array starting beyond the start of the old inline slot array.

if(PHASE\_TRACE1(Js::ObjectHeaderInliningPhase))

{

Output::Print(L"ObjectHeaderInlining: Moving inlined properties out of the object header.\n");

Output::Flush();

}

Var \*const newInlineSlots = reinterpret\_cast<Var \*>(object + 1);

PropertyIndex i = newInlineSlotCapacity;

do

{

--i;

newInlineSlots[i] = oldInlineSlots[i];

} while(i > 0);

}

object->auxSlots = newAuxSlots;

object->objectArray = nullptr;

}

bool DynamicTypeHandler::CanBeSingletonInstance(DynamicObject \* instance)

{

return !ThreadContext::IsOnStack(instance);

}

}

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

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

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

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

#pragma once

namespace Js

{

enum DeferredInitializeMode

{

DeferredInitializeMode\_Default,

DeferredInitializeMode\_Extensions,

DeferredInitializeMode\_Set,

DeferredInitializeMode\_SetAccessors

};

enum FixedPropertyKind : CHAR

{

FixedDataProperty = 1 << 0,

FixedMethodProperty = 1 << 1,

FixedAccessorProperty = 1 << 2,

};

struct PropertyEquivalenceInfo

{

PropertyIndex slotIndex;

bool isAuxSlot;

bool isWritable;

PropertyEquivalenceInfo():

slotIndex(Constants::NoSlot), isAuxSlot(false), isWritable(false) {}

PropertyEquivalenceInfo(PropertyIndex slotIndex, bool isAuxSlot, bool isWritable):

slotIndex(slotIndex), isAuxSlot(isAuxSlot), isWritable(isWritable) {}

};

struct EquivalentPropertyEntry

{

Js::PropertyId propertyId;

Js::PropertyIndex slotIndex;

bool isAuxSlot;

bool mustBeWritable;

};

struct TypeEquivalenceRecord

{

uint propertyCount;

EquivalentPropertyEntry\* properties;

};

typedef void (\_\_cdecl \*DeferredTypeInitializer)(DynamicObject\* instance, DeferredTypeHandlerBase \* typeHandler, DeferredInitializeMode mode);

class DynamicTypeHandler

{

friend class DeferredTypeHandlerBase;

template <DeferredTypeInitializer initializer, typename DeferredTypeFilter, bool isPrototypeTemplate, uint16 \_inlineSlotCapacity, uint16 \_offsetOfInlineSlots>

friend class DeferredTypeHandler;

friend class PathTypeHandlerBase;

friend struct InlineCache;

friend class DynamicObject;

private:

// Holds flags that represent general information about the types of properties

// handled by this handler.

// \* PropertyTypesWritableDataOnly - when true, the type being handled is known to have only writable data properties

// \* PropertyTypesWritableDataOnlyDetection - set each time SetHasOnlyWritableDataProperties is called.

// PropertyTypesReserved (0x1) is always on so that the DWORD formed with the following boolean doesn't look like

// a pointer.

PropertyTypes propertyTypes;

BYTE flags;

uint16 offsetOfInlineSlots;

int slotCapacity;

uint16 unusedBytes; // This always has it's lowest bit set to avoid false references

uint16 inlineSlotCapacity;

bool isNotPathTypeHandlerOrHasUserDefinedCtor;

public:

DEFINE\_GETCPPNAME\_ABSTRACT();

DynamicTypeHandler(DynamicTypeHandler \* typeHandler) :

flags(typeHandler->flags),

propertyTypes(typeHandler->propertyTypes),

slotCapacity(typeHandler->slotCapacity),

offsetOfInlineSlots(typeHandler->offsetOfInlineSlots),

isNotPathTypeHandlerOrHasUserDefinedCtor(typeHandler->isNotPathTypeHandlerOrHasUserDefinedCtor),

unusedBytes(typeHandler->unusedBytes)

{

}

public:

DEFINE\_VTABLE\_CTOR\_NOBASE\_ABSTRACT(DynamicTypeHandler);

DynamicTypeHandler(int slotCapacity, uint16 inlineSlotCapacity = 0, uint16 offsetOfInlineSlots = 0, BYTE flags = DefaultFlags);

void SetInstanceTypeHandler(DynamicObject \* instance, bool hasChanged = true);

static DynamicTypeHandler \* GetCurrentTypeHandler(DynamicObject \* instance);

static void SetInstanceTypeHandler(DynamicObject \* instance, DynamicTypeHandler \* typeHandler, bool hasChanged = true);

static void ReplaceInstanceType(DynamicObject \* instance, DynamicType \* type);

private:

static bool IsObjectHeaderInlined(const uint16 offsetOfInlineSlots);

bool IsObjectHeaderInlinedTypeHandlerUnchecked() const;

public:

bool IsObjectHeaderInlinedTypeHandler() const;

private:

void VerifyObjectHeaderInlinedTypeHandler() const;

public:

static uint16 GetOffsetOfObjectHeaderInlineSlots();

static PropertyIndex GetObjectHeaderInlinableSlotCapacity();

// UnusedBytes is a tagged value to prevent GC false references

uint16 GetUnusedBytesValue() const

{

return unusedBytes >> 1;

}

void SetUnusedBytesValue(uint16 value)

{

// Tagging to prevent a GC false reference

this->unusedBytes = ((value << 1) | 1);

}

public:

static const BYTE IsExtensibleFlag = 0x01;

static const BYTE HasKnownSlot0Flag = 0x02;

static const BYTE IsLockedFlag = 0x04;

static const BYTE MayBecomeSharedFlag = 0x08;

static const BYTE IsSharedFlag = 0x10;

static const BYTE IsPrototypeFlag = 0x20;

static const BYTE IsSealedOnceFlag = 0x40; // Set state means the object is sealed, clear state means nothing (can be sealed, or not).

static const BYTE IsFrozenOnceFlag = 0x80; // Set state means the object is frozen, clear state means nothing (can be frozen, or not).

static const BYTE DefaultFlags = IsExtensibleFlag;

public:

static PropertyIndex RoundUpObjectHeaderInlinedInlineSlotCapacity(const PropertyIndex slotCapacity);

static PropertyIndex RoundUpInlineSlotCapacity(const PropertyIndex slotCapacity);

private:

static int RoundUpAuxSlotCapacity(const int slotCapacity);

public:

static int RoundUpSlotCapacity(const int slotCapacity, const PropertyIndex inlineSlotCapacity);

uint16 GetInlineSlotCapacity() const { return this->inlineSlotCapacity; }

int GetSlotCapacity() const { return this->slotCapacity; }

size\_t GetInlineSlotsSize() const

{

PropertyIndex inlineSlotsToAllocate = GetInlineSlotCapacity();

if(IsObjectHeaderInlinedTypeHandler())

{

inlineSlotsToAllocate -= GetObjectHeaderInlinableSlotCapacity();

}

return inlineSlotsToAllocate \* sizeof(Var);

}

uint16 GetOffsetOfInlineSlots() const { return this->offsetOfInlineSlots; }

void EnsureSlots(DynamicObject \* instance, int oldCount, int newCount, ScriptContext \* scriptContext, DynamicTypeHandler \* newTypeHandler = nullptr);

Var GetSlot(DynamicObject \* instance, int index);

Var GetInlineSlot(DynamicObject \* instance, int index);

Var GetAuxSlot(DynamicObject \* instance, int index);

void TraceUseFixedProperty(PropertyRecord const \* propertyRecord, Var \* pProperty, bool result, LPCWSTR typeHandlerName, ScriptContext \* requestContext);

bool IsFixedMethodProperty(FixedPropertyKind fixedPropKind);

bool IsFixedDataProperty(FixedPropertyKind fixedPropKind);

bool IsFixedAccessorProperty(FixedPropertyKind fixedPropKind);

static bool CheckHeuristicsForFixedDataProps(DynamicObject\* instance, const PropertyRecord \* propertyRecord, PropertyId propertyId, Var value);

static bool CheckHeuristicsForFixedDataProps(DynamicObject\* instance, const PropertyRecord \* propertyRecord, Var value);

static bool CheckHeuristicsForFixedDataProps(DynamicObject\* instance, PropertyId propertyId, Var value);

static bool CheckHeuristicsForFixedDataProps(DynamicObject\* instance, JavascriptString \* propertyKey, Var value);

#if DBG

void SetSlot(DynamicObject \* instance, PropertyId propertyId, bool allowLetConst, int index, Var value);

void SetInlineSlot(DynamicObject \* instance, PropertyId propertyId, bool allowLetConst, int index, Var value);

void SetAuxSlot(DynamicObject \* instance, PropertyId propertyId, bool allowLetConst, int index, Var value);

#else

void SetSlot(DynamicObject \* instance, int index, Var value);

void SetInlineSlot(DynamicObject \* instance, int index, Var value);

void SetAuxSlot(DynamicObject \* instance, int index, Var value);

#endif

protected:

void SetSlotUnchecked(DynamicObject \* instance, int index, Var value);

public:

\_\_inline PropertyIndex AdjustSlotIndexForInlineSlots(PropertyIndex slotIndex)

{

return slotIndex != Constants::NoSlot ? AdjustValidSlotIndexForInlineSlots(slotIndex) : Constants::NoSlot;

}

\_\_inline PropertyIndex AdjustValidSlotIndexForInlineSlots(PropertyIndex slotIndex)

{

Assert(slotIndex != Constants::NoSlot);

return slotIndex < inlineSlotCapacity ?

slotIndex + (offsetOfInlineSlots / sizeof(Var)) : slotIndex - (PropertyIndex)inlineSlotCapacity;

}

\_\_inline void PropertyIndexToInlineOrAuxSlotIndex(PropertyIndex propertyIndex, PropertyIndex \* inlineOrAuxSlotIndex, bool \* isInlineSlot) const

{

if (propertyIndex < inlineSlotCapacity)

{

\*inlineOrAuxSlotIndex = propertyIndex + (offsetOfInlineSlots / sizeof(Var));

\*isInlineSlot = true;

}

else

{

\*inlineOrAuxSlotIndex = propertyIndex - (PropertyIndex)inlineSlotCapacity;

\*isInlineSlot = false;

}

}

PropertyIndex InlineOrAuxSlotIndexToPropertyIndex(PropertyIndex inlineOrAuxSlotIndex, bool isInlineSlot) const

{

if (isInlineSlot)

{

return inlineOrAuxSlotIndex - (offsetOfInlineSlots / sizeof(Var));

}

else

{

return inlineOrAuxSlotIndex + (PropertyIndex)inlineSlotCapacity;

}

}

protected:

void SetFlags(BYTE values)

{

// Don't set a shared flag if the type handler isn't locked.

Assert((this->flags & IsLockedFlag) != 0 || (values & IsLockedFlag) != 0 || (values & IsSharedFlag) == 0);

// Don't set a shared flag if the type handler isn't expecting to become shared.

Assert((this->flags & MayBecomeSharedFlag) != 0 || (values & MayBecomeSharedFlag) != 0 || (values & IsSharedFlag) == 0);

// It's ok to set up a shared prototype type handler through a constructor (see NullTypeHandler and DeferredTypeHandler),

// but it's not ok to change these after the fact.

// If we isolate prototypes, don't set a prototype flag on a type handler that is shared or may become shared

Assert((this->flags & IsPrototypeFlag) != 0 || !IsolatePrototypes() || (this->flags & (MayBecomeSharedFlag | IsSharedFlag)) == 0 || (values & IsPrototypeFlag) == 0);

// If we isolate prototypes, don't set a shared or may become shared flag on a prototype type handler.

Assert((this->flags & IsSharedFlag) != 0 || !IsolatePrototypes() || (this->flags & IsPrototypeFlag) == 0 || (values & (MayBecomeSharedFlag | IsSharedFlag)) == 0);

// Don't set a shared flag if this type handler has a singleton instance.

Assert(!this->HasSingletonInstance() || (values & IsSharedFlag) == 0);

this->flags |= values;

}

void ClearFlags(BYTE values)

{

// Don't clear the locked, shared or prototype flags.

Assert((values & IsLockedFlag) == 0 && (values & IsSharedFlag) == 0 && (values & IsPrototypeFlag) == 0);

this->flags &= ~values;

}

void SetFlags(BYTE selector, BYTE values)

{

SetFlags(selector & values);

}

void ChangeFlags(BYTE selector, BYTE values)

{

// Don't clear the locked, shared or prototype flags.

Assert((this->flags & IsLockedFlag) == 0 || (selector & IsLockedFlag) == 0 || (values & IsLockedFlag) != 0);

Assert((this->flags & IsSharedFlag) == 0 || (selector & IsSharedFlag) == 0 || (values & IsSharedFlag) != 0);

Assert((this->flags & IsPrototypeFlag) == 0 || (selector & IsPrototypeFlag) == 0 || (values & IsPrototypeFlag) != 0);

// Don't set a shared flag if the type handler isn't locked.

Assert((this->flags & IsLockedFlag) != 0 || ((selector & values) & IsLockedFlag) != 0 || ((selector & values) & IsSharedFlag) == 0);

// Don't set a shared flag if the type handler isn't locked.

Assert((this->flags & MayBecomeSharedFlag) != 0 || (values & MayBecomeSharedFlag) != 0 || (values & IsSharedFlag) == 0);

// It's ok to set up a shared prototype type handler through a constructor (see NullTypeHandler and DeferredTypeHandler),

// but it's not ok to change these after the fact.

// If we isolate prototypes, don't set a prototype flag on a shared type handler.

Assert((this->flags & IsPrototypeFlag) != 0 || !IsolatePrototypes() || (this->flags & (MayBecomeSharedFlag | IsSharedFlag)) == 0 || ((selector & values) & IsPrototypeFlag) == 0);

// If we isolate prototypes, don't set a shared flag on a prototype type handler.

Assert((this->flags & IsSharedFlag) != 0 || !IsolatePrototypes() || (this->flags & IsPrototypeFlag) == 0 || ((selector & values) & (MayBecomeSharedFlag | IsSharedFlag)) == 0);

// Don't set a shared flag if this type handler has a singleton instance.

Assert(!this->HasSingletonInstance() || ((selector & values) & IsSharedFlag) == 0);

this->flags = (selector & values) | (~selector & this->flags);

}

void SetPropertyTypes(PropertyTypes selector, PropertyTypes values)

{

Assert((selector & PropertyTypesReserved) == 0);

Assert((this->propertyTypes & PropertyTypesReserved) != 0);

this->propertyTypes |= (selector & values);

Assert((this->propertyTypes & PropertyTypesReserved) != 0);

}

void ClearPropertyTypes(PropertyTypes selector, PropertyTypes values)

{

Assert((selector & PropertyTypesReserved) == 0);

Assert((this->propertyTypes & PropertyTypesReserved) != 0);

this->propertyTypes |= (selector & ~values);

Assert((this->propertyTypes & PropertyTypesReserved) != 0);

}

void CopyPropertyTypes(PropertyTypes selector, PropertyTypes values)

{

Assert((selector & PropertyTypesReserved) == 0);

Assert((this->propertyTypes & PropertyTypesReserved) != 0);

this->propertyTypes = (selector & values) | (~selector & this->propertyTypes);

Assert((this->propertyTypes & PropertyTypesReserved) != 0);

}

void CopyClearedPropertyTypes(PropertyTypes selector, PropertyTypes values)

{

Assert((selector & PropertyTypesReserved) == 0);

Assert((this->propertyTypes & PropertyTypesReserved) != 0);

this->propertyTypes = (selector & (values & this->propertyTypes)) | (~selector & this->propertyTypes);

Assert((this->propertyTypes & PropertyTypesReserved) != 0);

}

static bool CanBeSingletonInstance(DynamicObject \* instance);

public:

BYTE GetFlags() const { return this->flags; }

static int GetOffsetOfFlags() { return offsetof(DynamicTypeHandler, flags); }

static int GetOffsetOfOffsetOfInlineSlots() { return offsetof(DynamicTypeHandler, offsetOfInlineSlots); }

bool GetIsLocked() const { return (this->flags & IsLockedFlag) != 0; }

bool GetIsShared() const { return (this->flags & IsSharedFlag) != 0; }

bool GetMayBecomeShared() const { return (this->flags & MayBecomeSharedFlag) != 0; }

bool GetIsOrMayBecomeShared() const { return (this->flags & (MayBecomeSharedFlag | IsSharedFlag)) != 0; }

bool GetHasKnownSlot0() const { return (this->flags & HasKnownSlot0Flag) != 0; }

bool GetIsPrototype() const { return (this->flags & IsPrototypeFlag) != 0; }

bool GetIsInlineSlotCapacityLocked() const { return (this->propertyTypes & PropertyTypesInlineSlotCapacityLocked) != 0; }

void LockTypeHandler() { Assert(IsLockable()); SetFlags(IsLockedFlag); }

void ShareTypeHandler(ScriptContext\* scriptContext)

{

Assert(IsSharable());

Assert(GetMayBecomeShared());

LockTypeHandler();

if ((GetFlags() & IsSharedFlag) == 0)

{

DoShareTypeHandler(scriptContext);

}

SetFlags(IsSharedFlag);

}

void SetMayBecomeShared()

{

SetFlags(MayBecomeSharedFlag);

}

void SetHasKnownSlot0()

{

SetFlags(HasKnownSlot0Flag);

}

void SetIsInlineSlotCapacityLocked()

{

Assert(!GetIsInlineSlotCapacityLocked());

SetPropertyTypes(PropertyTypesInlineSlotCapacityLocked, PropertyTypesInlineSlotCapacityLocked);

}

PropertyTypes GetPropertyTypes() { Assert((propertyTypes & PropertyTypesReserved) != 0); return propertyTypes; }

bool GetHasOnlyWritableDataProperties() { return (GetPropertyTypes() & PropertyTypesWritableDataOnly) == PropertyTypesWritableDataOnly; }

// Do not use this method. It's here only for the \_\_proto\_\_ performance workaround.

void SetHasOnlyWritableDataProperties() { SetHasOnlyWritableDataProperties(true); }

void ClearHasOnlyWritableDataProperties() { SetHasOnlyWritableDataProperties(false); };

private:

void SetHasOnlyWritableDataProperties(bool value)

{

if (value != GetHasOnlyWritableDataProperties())

{

propertyTypes ^= PropertyTypesWritableDataOnly;

}

// Turn on the detection bit.

propertyTypes |= PropertyTypesWritableDataOnlyDetection;

Assert((propertyTypes & PropertyTypesReserved) != 0);

}

public:

void ClearWritableDataOnlyDetectionBit() { Assert((propertyTypes & PropertyTypesReserved) != 0); propertyTypes &= ~PropertyTypesWritableDataOnlyDetection; }

bool IsWritableDataOnlyDetectionBitSet()

{

return (GetPropertyTypes() & PropertyTypesWritableDataOnlyDetection) == PropertyTypesWritableDataOnlyDetection;

}

BOOL Freeze(DynamicObject \*instance, bool isConvertedType = false) { return FreezeImpl(instance, isConvertedType); }

bool GetIsNotPathTypeHandlerOrHasUserDefinedCtor() const { return this->isNotPathTypeHandlerOrHasUserDefinedCtor; }

virtual BOOL IsStringTypeHandler() const { return false; }

virtual BOOL AllPropertiesAreEnumerable() { return false; }

virtual BOOL IsLockable() const = 0;

virtual BOOL IsSharable() const = 0;

virtual void DoShareTypeHandler(ScriptContext\* scriptContext) {};

virtual int GetPropertyCount() = 0;

virtual PropertyId GetPropertyId(ScriptContext\* scriptContext, PropertyIndex index) = 0;

virtual PropertyId GetPropertyId(ScriptContext\* scriptContext, BigPropertyIndex index) = 0;

virtual BOOL FindNextProperty(ScriptContext\* scriptContext, PropertyIndex& index, JavascriptString\*\* propertyString,

PropertyId\* propertyId, PropertyAttributes\* attributes, Type\* type, DynamicType \*typeToEnumerate, bool requireEnumerable, bool enumSymbols = false) = 0;

virtual BOOL FindNextProperty(ScriptContext\* scriptContext, BigPropertyIndex& index, JavascriptString\*\* propertyString,

PropertyId\* propertyId, PropertyAttributes\* attributes, Type\* type, DynamicType \*typeToEnumerate, bool requireEnumerable, bool enumSymbols = false);

virtual PropertyIndex GetPropertyIndex(PropertyRecord const\* propertyRecord) = 0;

virtual bool GetPropertyEquivalenceInfo(PropertyRecord const\* propertyRecord, PropertyEquivalenceInfo& info) = 0;

virtual bool IsObjTypeSpecEquivalent(const Type\* type, const Js::TypeEquivalenceRecord& record, uint& failedPropertyIndex) = 0;

virtual bool IsObjTypeSpecEquivalent(const Type\* type, const EquivalentPropertyEntry\* entry) = 0;

virtual bool EnsureObjectReady(DynamicObject\* instance) { return true; }

virtual BOOL HasProperty(DynamicObject\* instance, PropertyId propertyId, \_\_out\_opt bool \*pNoRedecl = nullptr) = 0;

virtual BOOL HasProperty(DynamicObject\* instance, JavascriptString\* propertyNameString) = 0;

virtual BOOL GetProperty(DynamicObject\* instance, Var originalInstance, PropertyId propertyId, Var\* value, PropertyValueInfo\* info, ScriptContext\* requestContext) = 0;

virtual BOOL GetProperty(DynamicObject\* instance, Var originalInstance, JavascriptString\* propertyNameString, Var\* value, PropertyValueInfo\* info, ScriptContext\* requestContext) = 0;

virtual BOOL GetInternalProperty(DynamicObject\* instance, Var originalInstance, PropertyId propertyId, Var\* value);

virtual BOOL InitProperty(DynamicObject\* instance, PropertyId propertyId, Var value, PropertyOperationFlags flags, PropertyValueInfo\* info);

virtual BOOL SetProperty(DynamicObject\* instance, PropertyId propertyId, Var value, PropertyOperationFlags flags, PropertyValueInfo\* info) = 0;

virtual BOOL SetProperty(DynamicObject\* instance, JavascriptString\* propertyNameString, Var value, PropertyOperationFlags flags, PropertyValueInfo\* info) = 0;

virtual BOOL SetInternalProperty(DynamicObject\* instance, PropertyId propertyId, Var value, PropertyOperationFlags flags);

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

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

virtual BOOL DeleteProperty(DynamicObject\* instance, PropertyId propertyId, PropertyOperationFlags flags) = 0;

// ===================================================================================================================

// Special versions of the various \*Property methods that recognize PropertyLetConstGlobal properties.

// Only used for GlobalObject and ModuleRoot and so only recognized by SimpleDictionary and Dictionary type handlers.

//

// "Root" here means via root access, i.e. without an object.

//

// Each of these will throw InternalFatalError because they should not be called on type handlers other than

// SimpleDictionary and Dictionary, both of which provide overrides.

//

virtual PropertyIndex GetRootPropertyIndex(PropertyRecord const\* propertyRecord) { Throw::FatalInternalError(); }

virtual BOOL HasRootProperty(DynamicObject\* instance, PropertyId propertyId, \_\_out\_opt bool \*pNoRedecl = nullptr, \_\_out\_opt bool \*pDeclaredProperty = nullptr) { Throw::FatalInternalError(); }

virtual BOOL GetRootProperty(DynamicObject\* instance, Var originalInstance, PropertyId propertyId, Var\* value, PropertyValueInfo\* info, ScriptContext\* requestContext) { Throw::FatalInternalError(); }

virtual BOOL SetRootProperty(DynamicObject\* instance, PropertyId propertyId, Var value, PropertyOperationFlags flags, PropertyValueInfo\* info) { Throw::FatalInternalError(); }

virtual DescriptorFlags GetRootSetter(DynamicObject\* instance, PropertyId propertyId, Var\* setterValue, PropertyValueInfo\* info, ScriptContext\* requestContext) { Throw::FatalInternalError(); }

virtual BOOL DeleteRootProperty(DynamicObject\* instance, PropertyId propertyId, PropertyOperationFlags flags) { Throw::FatalInternalError(); }

#if DBG

virtual bool IsLetConstGlobal(DynamicObject\* instance, PropertyId propertyId) { Throw::FatalInternalError(); }

#endif

// Would be nicer to simply pass in lambda callbacks to a Map function here, but virtual methods

// cannot be templatized and we do not have std::function<> so cannot specify a parameter that will

// accept lambdas.

virtual bool NextLetConstGlobal(int& index, RootObjectBase\* instance, const PropertyRecord\*\* propertyRecord, Var\* value, bool\* isConst) { Throw::FatalInternalError(); }

// ===================================================================================================================

virtual BOOL IsFixedProperty(const DynamicObject\* instance, PropertyId propertyId) { return false; };

virtual BOOL IsEnumerable(DynamicObject\* instance, PropertyId propertyId) = 0;

virtual BOOL IsWritable(DynamicObject\* instance, PropertyId propertyId) = 0;

virtual BOOL IsConfigurable(DynamicObject\* instance, PropertyId propertyId) = 0;

virtual BOOL SetEnumerable(DynamicObject\* instance, PropertyId propertyId, BOOL value) = 0;

virtual BOOL SetWritable(DynamicObject\* instance, PropertyId propertyId, BOOL value) = 0;

virtual BOOL SetConfigurable(DynamicObject\* instance, PropertyId propertyId, BOOL value) = 0;

virtual BOOL HasItem(DynamicObject\* instance, uint32 index);

virtual BOOL SetItem(DynamicObject\* instance, uint32 index, Var value, PropertyOperationFlags flags);

virtual BOOL SetItemWithAttributes(DynamicObject\* instance, uint32 index, Var value, PropertyAttributes attributes);

virtual BOOL SetItemAttributes(DynamicObject\* instance, uint32 index, PropertyAttributes attributes);

virtual BOOL SetItemAccessors(DynamicObject\* instance, uint32 index, Var getter, Var setter);

virtual BOOL DeleteItem(DynamicObject\* instance, uint32 index, PropertyOperationFlags flags);

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

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

virtual BOOL SetAccessors(DynamicObject\* instance, PropertyId propertyId, Var getter, Var setter, PropertyOperationFlags flags = PropertyOperation\_None) = 0;

virtual BOOL GetAccessors(DynamicObject\* instance, PropertyId propertyId, Var\* getter, Var\* setter) { return false; };

virtual BOOL PreventExtensions(DynamicObject \*instance) = 0;

virtual BOOL Seal(DynamicObject \*instance) = 0;

virtual BOOL IsSealed(DynamicObject \*instance) { return false; }

virtual BOOL IsFrozen(DynamicObject \*instance) { return false; }

virtual BOOL SetPropertyWithAttributes(DynamicObject\* instance, PropertyId propertyId, Var value, PropertyAttributes attributes, PropertyValueInfo\* info, PropertyOperationFlags flags = PropertyOperation\_None, SideEffects possibleSideEffects = SideEffects\_Any) = 0;

virtual BOOL SetAttributes(DynamicObject\* instance, PropertyId propertyId, PropertyAttributes attributes) = 0;

virtual BOOL GetAttributesWithPropertyIndex(DynamicObject \* instance, PropertyId propertyId, BigPropertyIndex index, PropertyAttributes \* attributes) = 0;

virtual void ShrinkSlotAndInlineSlotCapacity() { VerifyInlineSlotCapacityIsLocked(); };

virtual void LockInlineSlotCapacity() { VerifyInlineSlotCapacityIsLocked(); }

virtual void EnsureInlineSlotCapacityIsLocked() { VerifyInlineSlotCapacityIsLocked(); }

virtual void VerifyInlineSlotCapacityIsLocked() { Assert(GetIsInlineSlotCapacityLocked()); }

// ES5Array type handler specific methods. Only implemented by ES5ArrayTypeHandlers.

virtual bool IsLengthWritable() const { Assert(false); return false; }

virtual void SetLength(ES5Array\* arr, uint32 newLen, PropertyOperationFlags propertyOperationFlags) { Assert(false); }

virtual BOOL IsObjectArrayFrozen(ES5Array\* arr) { Assert(false); return FALSE; }

virtual BOOL IsItemEnumerable(ES5Array\* arr, uint32 index) { Assert(false); return FALSE; }

virtual BOOL IsValidDescriptorToken(void \* descriptorValidationToken) const { Assert(false); return FALSE; }

virtual uint32 GetNextDescriptor(uint32 key, IndexPropertyDescriptor\*\* descriptor, void \*\* descriptorValidationToken) { Assert(false); return 0; }

virtual BOOL GetDescriptor(uint32 index, IndexPropertyDescriptor \*\*ppDescriptor) { Assert(false); return FALSE; }

// Convert instance type/typeHandler to support SetItem with attribute/getter/setter

virtual DynamicTypeHandler\* ConvertToTypeWithItemAttributes(DynamicObject\* instance) = 0;

private:

template<bool isStoreField>

void InvalidateInlineCachesForAllProperties(ScriptContext\* requestContext);

public:

void InvalidateProtoCachesForAllProperties(ScriptContext\* requestContext);

void InvalidateStoreFieldCachesForAllProperties(ScriptContext\* requestContext);

// For changing \_\_proto\_\_

void RemoveFromPrototype(DynamicObject\* instance, ScriptContext \* requestContext);

void AddToPrototype(DynamicObject\* instance, ScriptContext \* requestContext);

virtual void SetPrototype(DynamicObject\* instance, RecyclableObject\* newPrototype);

virtual void ResetTypeHandler(DynamicObject \* instance);

virtual void SetAllPropertiesToUndefined(DynamicObject\* instance, bool invalidateFixedFields) = 0;

virtual void MarshalAllPropertiesToScriptContext(DynamicObject\* instance, ScriptContext\* targetScriptContext, bool invalidateFixedFields) = 0;

virtual BOOL IsDeferredTypeHandler() const { return FALSE; }

virtual BOOL IsPathTypeHandler() const { return FALSE; }

virtual BOOL IsSimpleDictionaryTypeHandler() const {return FALSE; }

virtual BOOL IsDictionaryTypeHandler() const {return FALSE;}

static bool IsolatePrototypes() { return CONFIG\_FLAG(IsolatePrototypes); }

static bool ChangeTypeOnProto() { return CONFIG\_FLAG(ChangeTypeOnProto); }

static bool ShouldFixMethodProperties() { return !PHASE\_OFF1(FixMethodPropsPhase); }

static bool ShouldFixDataProperties() { return !PHASE\_OFF1(FixDataPropsPhase); }

static bool ShouldFixAccessorProperties() { return !PHASE\_OFF1(FixAccessorPropsPhase); }

static bool ShouldFixAnyProperties() { return ShouldFixDataProperties() || ShouldFixMethodProperties() || ShouldFixAccessorProperties(); }

static bool AreSingletonInstancesNeeded() { return ShouldFixAnyProperties(); }

virtual void SetIsPrototype(DynamicObject\* instance) = 0;

#if DBG

virtual bool SupportsPrototypeInstances() const { return false; }

virtual bool RespectsIsolatePrototypes() const { return true; }

virtual bool RespectsChangeTypeOnProto() const { return true; }

#endif

virtual bool HasSingletonInstance() const { return false; }

virtual bool TryUseFixedProperty(PropertyRecord const\* propertyRecord, Var\* pProperty, FixedPropertyKind propertyType, ScriptContext \* requestContext);

virtual bool TryUseFixedAccessor(PropertyRecord const\* propertyRecord, Var\* pAccessor, FixedPropertyKind propertyType, bool getter, ScriptContext \* requestContext);

#if DBG

virtual bool CanStorePropertyValueDirectly(const DynamicObject\* instance, PropertyId propertyId, bool allowLetConst) { return false; }

virtual bool CheckFixedProperty(PropertyRecord const \* propertyRecord, Var \* pProperty, ScriptContext \* requestContext) { return false; };

virtual bool HasAnyFixedProperties() const { return false; }

#endif

#ifdef ENABLE\_DEBUG\_CONFIG\_OPTIONS

virtual void DumpFixedFields() const {};

#endif

public:

virtual RecyclerWeakReference<DynamicObject>\* GetSingletonInstance() const { return nullptr; }

bool SetSingletonInstanceIfNeeded(DynamicObject\* instance)

{

if (AreSingletonInstancesNeeded() && CanBeSingletonInstance(instance))

{

SetSingletonInstance(instance->CreateWeakReferenceToSelf());

return true;

}

return false;

}

void SetSingletonInstanceIfNeeded(RecyclerWeakReference<DynamicObject>\* instance)

{

if (AreSingletonInstancesNeeded())

{

SetSingletonInstance(instance);

}

}

void SetSingletonInstance(RecyclerWeakReference<DynamicObject>\* instance)

{

Assert(AreSingletonInstancesNeeded());

SetSingletonInstanceUnchecked(instance);

}

virtual void SetSingletonInstanceUnchecked(RecyclerWeakReference<DynamicObject>\* instance) { Assert(false); }

virtual void ClearSingletonInstance() { Assert(false); }

public:

static void AdjustSlots\_Jit(DynamicObject \*const object, const PropertyIndex newInlineSlotCapacity, const int newAuxSlotCapacity);

static void AdjustSlots(DynamicObject \*const object, const PropertyIndex newInlineSlotCapacity, const int newAuxSlotCapacity);

BigPropertyIndex GetPropertyIndexFromInlineSlotIndex(uint inlininlineSlotIndexeSlot);

BigPropertyIndex GetPropertyIndexFromAuxSlotIndex(uint auxIndex);

protected:

void SetPropertyUpdateSideEffect(DynamicObject\* instance, PropertyId propertyId, Var value, SideEffects possibleSideEffects);

bool VerifyIsExtensible(ScriptContext\* scriptContext, bool alwaysThrow);

void SetOffsetOfInlineSlots(const uint16 offsetOfInlineSlots) { this->offsetOfInlineSlots = offsetOfInlineSlots; }

void SetInlineSlotCapacity(int16 newInlineSlotCapacity) { this->inlineSlotCapacity = newInlineSlotCapacity; }

void SetSlotCapacity(int newSlotCapacity) { this->slotCapacity = newSlotCapacity; }

private:

virtual BOOL FreezeImpl(DynamicObject \*instance, bool isConvertedType) = 0;

};

}

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

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

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

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

namespace Js

{

#include "EdgeJavascriptTypeId.h"

// All WinRT dates are regular Javascript dates too

inline bool IsDateTypeId(TypeId typeId) { return (typeId == TypeIds\_Date || typeId == TypeIds\_WinRTDate); }

}

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

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

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

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

#include "RuntimeTypePch.h"

namespace Js {

TypePath\* TypePath::New(Recycler\* recycler, uint size)

{

size = max(size, InitialTypePathSize);

size = PowerOf2Policy::GetSize(size);

if (PHASE\_OFF1(Js::TypePathDynamicSizePhase))

{

size = MaxPathTypeHandlerLength;

}

Assert(size <= MaxPathTypeHandlerLength);

TypePath \* newTypePath = RecyclerNewPlusZ(recycler, sizeof(PropertyRecord \*) \* size, TypePath);

newTypePath->pathSize = (uint16)size;

return newTypePath;

}

PropertyIndex TypePath::Lookup(PropertyId propId,int typePathLength)

{

return LookupInline(propId,typePathLength);

}

\_\_inline PropertyIndex TypePath::LookupInline(PropertyId propId,int typePathLength)

{

if (propId == Constants::NoProperty) {

return Constants::NoSlot;

}

PropertyIndex propIndex = Constants::NoSlot;

if (map.TryGetValue(propId, &propIndex, assignments)) {

if (propIndex<typePathLength) {

return propIndex;

}

}

return Constants::NoSlot;

}

TypePath \* TypePath::Branch(Recycler \* recycler, int pathLength, bool couldSeeProto)

{

AssertMsg(pathLength < this->pathLength, "Why are we branching at the tip of the type path?");

// Ensure there is at least one free entry in the new path, so we can extend it.

// TypePath::New will take care of aligning this appropriately.

TypePath \* branchedPath = TypePath::New(recycler, pathLength + 1);

for (PropertyIndex i = 0; i < pathLength; i++)

{

branchedPath->AddInternal(assignments[i]);

#ifdef SUPPORT\_FIXED\_FIELDS\_ON\_PATH\_TYPES

if (couldSeeProto)

{

if (this->usedFixedFields.Test(i))

{

// We must conservatively copy all used as fixed bits if some prototype instance could also take

// this transition. See comment in PathTypeHandlerBase::ConvertToSimpleDictionaryType.

// Yes, we could devise a more efficient way of copying bits 1 through pathLength, if performance of this

// code path proves important enough.

branchedPath->usedFixedFields.Set(i);

}

else if (this->fixedFields.Test(i))

{

// We must clear any fixed fields that are not also used as fixed if some prototype instance could also take

// this transition. See comment in PathTypeHandlerBase::ConvertToSimpleDictionaryType.

this->fixedFields.Clear(i);

}

}

#endif

}

#ifdef SUPPORT\_FIXED\_FIELDS\_ON\_PATH\_TYPES

// When branching, we must ensure that fixed field values on the prefix shared by the two branches are always

// consistent. Hence, we can't leave any of them uninitialized, because they could later get initialized to

// different values, by two different instances (one on the old branch and one on the new branch). If that happened

// and the instance from the old branch later switched to the new branch, it would magically gain a different set

// of fixed properties!

if (this->maxInitializedLength < pathLength)

{

this->maxInitializedLength = pathLength;

}

branchedPath->maxInitializedLength = pathLength;

#endif

#ifdef SUPPORT\_FIXED\_FIELDS\_ON\_PATH\_TYPES

if (PHASE\_VERBOSE\_TRACE1(FixMethodPropsPhase))

{

Output::Print(L"FixedFields: TypePath::Branch: singleton: 0x%p(0x%p)\n", this->singletonInstance, this->singletonInstance->Get());

Output::Print(L" fixed fields:");

for (PropertyIndex i = 0; i < GetPathLength(); i++)

{

Output::Print(L" %s %d%d%d,", GetPropertyId(i)->GetBuffer(),

i < GetMaxInitializedLength() ? 1 : 0,

GetIsFixedFieldAt(i, GetPathLength()) ? 1 : 0,

GetIsUsedFixedFieldAt(i, GetPathLength()) ? 1 : 0);

}

Output::Print(L"\n");

}

#endif

return branchedPath;

}

TypePath \* TypePath::Grow(Recycler \* recycler)

{

AssertMsg(this->pathSize == this->pathLength, "Why are we growing the type path?");

// Ensure there is at least one free entry in the new path, so we can extend it.

// TypePath::New will take care of aligning this appropriately.

TypePath \* clonedPath = TypePath::New(recycler, this->pathLength + 1);

for (PropertyIndex i = 0; i < pathLength; i++)

{

clonedPath->AddInternal(assignments[i]);

}

#ifdef SUPPORT\_FIXED\_FIELDS\_ON\_PATH\_TYPES

// Copy fixed field info

clonedPath->maxInitializedLength = this->maxInitializedLength;

clonedPath->singletonInstance = this->singletonInstance;

clonedPath->fixedFields = this->fixedFields;

clonedPath->usedFixedFields = this->usedFixedFields;

#endif

return clonedPath;

}

#if DBG

bool TypePath::HasSingletonInstanceOnlyIfNeeded()

{

#ifdef SUPPORT\_FIXED\_FIELDS\_ON\_PATH\_TYPES

return DynamicTypeHandler::AreSingletonInstancesNeeded() || this->singletonInstance == nullptr;

#else

return true;

#endif

}

#endif

Var TypePath::GetSingletonFixedFieldAt(PropertyIndex index, int typePathLength, ScriptContext \* requestContext)

{

#ifdef SUPPORT\_FIXED\_FIELDS\_ON\_PATH\_TYPES

Assert(index < this->pathLength);

Assert(index < typePathLength);

Assert(typePathLength <= this->pathLength);

if (!CanHaveFixedFields(typePathLength))

{

return nullptr;

}

DynamicObject\* localSingletonInstance = this->singletonInstance->Get();

return localSingletonInstance != nullptr && localSingletonInstance->GetScriptContext() == requestContext && this->fixedFields.Test(index) ? localSingletonInstance->GetSlot(index) : nullptr;

#else

return nullptr;

#endif

}

int TypePath::AddInternal(const PropertyRecord\* propId)

{

Assert(pathLength < this->pathSize);

if (pathLength >= this->pathSize)

{

Throw::InternalError();

}

#if DBG

PropertyIndex temp;

if (map.TryGetValue(propId->GetPropertyId(), &temp, assignments))

{

AssertMsg(false, "Adding a duplicate to the type path");

}

#endif

map.Add((unsigned int)propId->GetPropertyId(), (byte)pathLength);

#ifdef SUPPORT\_FIXED\_FIELDS\_ON\_PATH\_TYPES

if (PHASE\_VERBOSE\_TRACE1(FixMethodPropsPhase))

{

Output::Print(L"FixedFields: TypePath::AddInternal: singleton = 0x%p(0x%p)\n",

this->singletonInstance, this->singletonInstance != nullptr ? this->singletonInstance->Get() : nullptr);

Output::Print(L" fixed fields:");

for (PropertyIndex i = 0; i < GetPathLength(); i++)

{

Output::Print(L" %s %d%d%d,", GetPropertyId(i)->GetBuffer(),

i < GetMaxInitializedLength() ? 1 : 0,

GetIsFixedFieldAt(i, GetPathLength()) ? 1 : 0,

GetIsUsedFixedFieldAt(i, GetPathLength()) ? 1 : 0);

}

Output::Print(L"\n");

}

#endif

assignments[pathLength] = propId;

pathLength++;

return (pathLength - 1);

}

void TypePath::AddBlankFieldAt(PropertyIndex index, int typePathLength)

{

Assert(index >= this->maxInitializedLength);

this->maxInitializedLength = index + 1;

#ifdef SUPPORT\_FIXED\_FIELDS\_ON\_PATH\_TYPES

if (PHASE\_VERBOSE\_TRACE1(FixMethodPropsPhase))

{

Output::Print(L"FixedFields: TypePath::AddBlankFieldAt: singleton = 0x%p(0x%p)\n",

this->singletonInstance, this->singletonInstance != nullptr ? this->singletonInstance->Get() : nullptr);

Output::Print(L" fixed fields:");

for (PropertyIndex i = 0; i < GetPathLength(); i++)

{

Output::Print(L" %s %d%d%d,", GetPropertyId(i)->GetBuffer(),

i < GetMaxInitializedLength() ? 1 : 0,

GetIsFixedFieldAt(i, GetPathLength()) ? 1 : 0,

GetIsUsedFixedFieldAt(i, GetPathLength()) ? 1 : 0);

}

Output::Print(L"\n");

}

#endif

}

void TypePath::AddSingletonInstanceFieldAt(DynamicObject\* instance, PropertyIndex index, bool isFixed, int typePathLength)

{

Assert(index < this->pathLength);

Assert(typePathLength >= this->maxInitializedLength);

Assert(index >= this->maxInitializedLength);

// This invariant is predicated on the properties getting initialized in the order of indexes in the type handler.

Assert(instance != nullptr);

Assert(this->singletonInstance == nullptr || this->singletonInstance->Get() == instance);

Assert(!fixedFields.Test(index) && !usedFixedFields.Test(index));

if (this->singletonInstance == nullptr)

{

this->singletonInstance = instance->CreateWeakReferenceToSelf();

}

this->maxInitializedLength = index + 1;

if (isFixed)

{

this->fixedFields.Set(index);

}

#ifdef SUPPORT\_FIXED\_FIELDS\_ON\_PATH\_TYPES

if (PHASE\_VERBOSE\_TRACE1(FixMethodPropsPhase))

{

Output::Print(L"FixedFields: TypePath::AddSingletonInstanceFieldAt: singleton = 0x%p(0x%p)\n",

this->singletonInstance, this->singletonInstance != nullptr ? this->singletonInstance->Get() : nullptr);

Output::Print(L" fixed fields:");

for (PropertyIndex i = 0; i < GetPathLength(); i++)

{

Output::Print(L" %s %d%d%d,", GetPropertyId(i)->GetBuffer(),

i < GetMaxInitializedLength() ? 1 : 0,

GetIsFixedFieldAt(i, GetPathLength()) ? 1 : 0,

GetIsUsedFixedFieldAt(i, GetPathLength()) ? 1 : 0);

}

Output::Print(L"\n");

}

#endif

}

void TypePath::AddSingletonInstanceFieldAt(PropertyIndex index, int typePathLength)

{

Assert(index < this->pathLength);

Assert(typePathLength >= this->maxInitializedLength);

Assert(index >= this->maxInitializedLength);

Assert(!fixedFields.Test(index) && !usedFixedFields.Test(index));

this->maxInitializedLength = index + 1;

#ifdef SUPPORT\_FIXED\_FIELDS\_ON\_PATH\_TYPES

if (PHASE\_VERBOSE\_TRACE1(FixMethodPropsPhase))

{

Output::Print(L"FixedFields: TypePath::AddSingletonInstanceFieldAt: singleton = 0x%p(0x%p)\n",

this->singletonInstance, this->singletonInstance != nullptr ? this->singletonInstance->Get() : nullptr);

Output::Print(L" fixed fields:");

for (PropertyIndex i = 0; i < GetPathLength(); i++)

{

Output::Print(L" %s %d%d%d,", GetPropertyId(i)->GetBuffer(),

i < GetMaxInitializedLength() ? 1 : 0,

GetIsFixedFieldAt(i, GetPathLength()) ? 1 : 0,

GetIsUsedFixedFieldAt(i, GetPathLength()) ? 1 : 0);

}

Output::Print(L"\n");

}

#endif

}

}

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

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

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

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

#pragma once

#define MAX\_SIZE\_PATH\_LENGTH (128)

namespace Js

{

class TinyDictionary

{

static const int PowerOf2\_BUCKETS = 8;

static const byte NIL = 0xff;

static const int NEXTPTRCOUNT = MAX\_SIZE\_PATH\_LENGTH;

byte buckets[PowerOf2\_BUCKETS];

byte next[NEXTPTRCOUNT];

public:

TinyDictionary()

{

DWORD\* init = (DWORD\*)buckets;

init[0] = init[1] = 0xffffffff;

}

void Add(PropertyId key, byte value)

{

Assert(value < NEXTPTRCOUNT);

\_\_analysis\_assume(value < NEXTPTRCOUNT);

uint32 bucketIndex = key&(PowerOf2\_BUCKETS-1);

byte i = buckets[bucketIndex];

buckets[bucketIndex] = value;

next[value] = i;

}

// Template shared with diagnostics

template <class Data>

\_\_inline bool TryGetValue(PropertyId key, PropertyIndex\* index, const Data& data)

{

uint32 bucketIndex = key&(PowerOf2\_BUCKETS-1);

for (byte i = buckets[bucketIndex] ; i != NIL ; i = next[i])

{

Assert(i < NEXTPTRCOUNT);

\_\_analysis\_assume(i < NEXTPTRCOUNT);

if (data[i]->GetPropertyId()== key)

{

\*index = i;

return true;

}

Assert(i != next[i]);

}

return false;

}

};

class TypePath

{

friend class PathTypeHandlerBase;

friend class DynamicObject;

friend class SimplePathTypeHandler;

friend class PathTypeHandler;

public:

static const uint MaxPathTypeHandlerLength = MAX\_SIZE\_PATH\_LENGTH;

static const uint InitialTypePathSize = 16;

private:

TinyDictionary map;

uint16 pathLength; // Entries in use

uint16 pathSize; // Allocated entries

#ifdef SUPPORT\_FIXED\_FIELDS\_ON\_PATH\_TYPES

// We sometimes set up PathTypeHandlers and associate TypePaths before we create any instances

// that populate the corresponding slots, e.g. for object literals or constructors with only

// this statements. This field keeps track of the longest instance associated with the given

// TypePath.

int maxInitializedLength;

RecyclerWeakReference<DynamicObject>\* singletonInstance;

BVStatic<MAX\_SIZE\_PATH\_LENGTH> fixedFields;

BVStatic<MAX\_SIZE\_PATH\_LENGTH> usedFixedFields;

#endif

// PropertyRecord assignments are allocated off the end of the structure

const PropertyRecord \* assignments[0];

#ifdef SUPPORT\_FIXED\_FIELDS\_ON\_PATH\_TYPES

TypePath()

: pathLength(0), maxInitializedLength(0), singletonInstance(nullptr)

{

}

#else

TypePath()

: pathLength(0)

{

}

#endif

public:

static TypePath\* New(Recycler\* recycler, uint size = InitialTypePathSize);

TypePath \* Branch(Recycler \* alloc, int pathLength, bool couldSeeProto);

TypePath \* Grow(Recycler \* alloc);

const PropertyRecord\* GetPropertyIdUnchecked(int index)

{

Assert(((uint)index) < ((uint)pathLength));

return assignments[index];

}

const PropertyRecord\* GetPropertyId(int index)

{

if (((uint)index) < ((uint)pathLength))

return GetPropertyIdUnchecked(index);

else

return nullptr;

}

const PropertyRecord \*\* GetPropertyAssignments()

{

return assignments;

}

int Add(const PropertyRecord \* propertyRecord)

{

#ifdef SUPPORT\_FIXED\_FIELDS\_ON\_PATH\_TYPES

Assert(this->pathLength == this->maxInitializedLength);

this->maxInitializedLength++;

#endif

return AddInternal(propertyRecord);

}

uint16 GetPathLength() { return this->pathLength; }

uint16 GetPathSize() const { return this->pathSize; }

PropertyIndex Lookup(PropertyId propId,int typePathLength);

PropertyIndex LookupInline(PropertyId propId,int typePathLength);

private:

int AddInternal(const PropertyRecord\* propId);

#ifdef SUPPORT\_FIXED\_FIELDS\_ON\_PATH\_TYPES

int GetMaxInitializedLength() { return this->maxInitializedLength; }

void SetMaxInitializedLength(int newMaxInitializedLength)

{

Assert(this->maxInitializedLength <= newMaxInitializedLength);

this->maxInitializedLength = newMaxInitializedLength;

}

Var GetSingletonFixedFieldAt(PropertyIndex index, int typePathLength, ScriptContext \* requestContext);

bool HasSingletonInstance() const

{

return this->singletonInstance != nullptr;

}

RecyclerWeakReference<DynamicObject>\* GetSingletonInstance() const

{

return this->singletonInstance;

}

void SetSingletonInstance(RecyclerWeakReference<DynamicObject>\* instance, int typePathLength)

{

Assert(this->singletonInstance == nullptr && instance != nullptr);

Assert(typePathLength >= this->maxInitializedLength);

this->singletonInstance = instance;

}

void ClearSingletonInstance()

{

this->singletonInstance = nullptr;

}

void ClearSingletonInstanceIfSame(DynamicObject\* instance)

{

if (this->singletonInstance != nullptr && this->singletonInstance->Get() == instance)

{

ClearSingletonInstance();

}

}

void ClearSingletonInstanceIfDifferent(DynamicObject\* instance)

{

if (this->singletonInstance != nullptr && this->singletonInstance->Get() != instance)

{

ClearSingletonInstance();

}

}

bool GetIsFixedFieldAt(PropertyIndex index, int typePathLength)

{

Assert(index < this->pathLength);

Assert(index < typePathLength);

Assert(typePathLength <= this->pathLength);

return this->fixedFields.Test(index) != 0;

}

bool GetIsUsedFixedFieldAt(PropertyIndex index, int typePathLength)

{

Assert(index < this->pathLength);

Assert(index < typePathLength);

Assert(typePathLength <= this->pathLength);

return this->usedFixedFields.Test(index) != 0;

}

void SetIsUsedFixedFieldAt(PropertyIndex index, int typePathLength)

{

Assert(index < this->maxInitializedLength);

Assert(CanHaveFixedFields(typePathLength));

this->usedFixedFields.Set(index);

}

void ClearIsFixedFieldAt(PropertyIndex index, int typePathLength)

{

Assert(index < this->maxInitializedLength);

Assert(index < typePathLength);

Assert(typePathLength <= this->pathLength);

this->fixedFields.Clear(index);

this->usedFixedFields.Clear(index);

}

bool CanHaveFixedFields(int typePathLength)

{

// We only support fixed fields on singleton instances.

// If the instance in question is a singleton, it must be the tip of the type path.

return this->singletonInstance != nullptr && typePathLength >= this->maxInitializedLength;

}

void AddBlankFieldAt(PropertyIndex index, int typePathLength);

void AddSingletonInstanceFieldAt(DynamicObject\* instance, PropertyIndex index, bool isFixed, int typePathLength);

void AddSingletonInstanceFieldAt(PropertyIndex index, int typePathLength);

#if DBG

bool HasSingletonInstanceOnlyIfNeeded();

#endif

#else

int GetMaxInitializedLength() { Assert(false); return this->pathLength; }

Var GetSingletonFixedFieldAt(PropertyIndex index, int typePathLength, ScriptContext \* requestContext);

bool HasSingletonInstance() const { Assert(false); return false; }

RecyclerWeakReference<DynamicObject>\* GetSingletonInstance() const { Assert(false); return nullptr; }

void SetSingletonInstance(RecyclerWeakReference<DynamicObject>\* instance, int typePathLength) { Assert(false); }

void ClearSingletonInstance() { Assert(false); }

void ClearSingletonInstanceIfSame(RecyclerWeakReference<DynamicObject>\* instance) { Assert(false); }

void ClearSingletonInstanceIfDifferent(RecyclerWeakReference<DynamicObject>\* instance) { Assert(false); }

bool GetIsFixedFieldAt(PropertyIndex index, int typePathLength) { Assert(false); return false; }

bool GetIsUsedFixedFieldAt(PropertyIndex index, int typePathLength) { Assert(false); return false; }

void SetIsUsedFixedFieldAt(PropertyIndex index, int typePathLength) { Assert(false); }

void ClearIsFixedFieldAt(PropertyIndex index, int typePathLength) { Assert(false); }

bool CanHaveFixedFields(int typePathLength) { Assert(false); return false; }

void AddBlankFieldAt(PropertyIndex index, int typePathLength) { Assert(false); }

void AddSingletonInstanceFieldAt(DynamicObject\* instance, PropertyIndex index, bool isFixed, int typePathLength) { Assert(false); }

void AddSingletonInstanceFieldAt(PropertyIndex index, int typePathLength) { Assert(false); }

#if DBG

bool HasSingletonInstanceOnlyIfNeeded();

#endif

#endif

};

}

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

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

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

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

#include "RuntimeTypePch.h"

namespace Js

{

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

// TypePropertyCacheElement

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

TypePropertyCacheElement::TypePropertyCacheElement() : id(Constants::NoProperty), index(0), prototypeObjectWithProperty(0)

{

}

PropertyId TypePropertyCacheElement::Id() const

{

return id;

}

PropertyIndex TypePropertyCacheElement::Index() const

{

return index;

}

bool TypePropertyCacheElement::IsInlineSlot() const

{

return isInlineSlot;

}

bool TypePropertyCacheElement::IsSetPropertyAllowed() const

{

return isSetPropertyAllowed;

}

bool TypePropertyCacheElement::IsMissing() const

{

return isMissing;

}

DynamicObject \*TypePropertyCacheElement::PrototypeObjectWithProperty() const

{

return prototypeObjectWithProperty;

}

void TypePropertyCacheElement::Cache(

const PropertyId id,

const PropertyIndex index,

const bool isInlineSlot,

const bool isSetPropertyAllowed)

{

Assert(id != Constants::NoProperty);

Assert(index != Constants::NoSlot);

this->id = id;

this->index = index;

this->isInlineSlot = isInlineSlot;

this->isSetPropertyAllowed = isSetPropertyAllowed;

this->isMissing = false;

this->prototypeObjectWithProperty = 0;

}

void TypePropertyCacheElement::Cache(

const PropertyId id,

const PropertyIndex index,

const bool isInlineSlot,

const bool isSetPropertyAllowed,

const bool isMissing,

DynamicObject \*const prototypeObjectWithProperty,

Type \*const myParentType)

{

Assert(id != Constants::NoProperty);

Assert(index != Constants::NoSlot);

Assert(prototypeObjectWithProperty);

Assert(myParentType);

if(this->id != id || !this->prototypeObjectWithProperty)

myParentType->GetScriptContext()->GetThreadContext()->RegisterTypeWithProtoPropertyCache(id, myParentType);

this->id = id;

this->index = index;

this->isInlineSlot = isInlineSlot;

this->isSetPropertyAllowed = isSetPropertyAllowed;

this->isMissing = isMissing;

this->prototypeObjectWithProperty = prototypeObjectWithProperty;

Assert(this->isMissing == (uint16)(this->prototypeObjectWithProperty == this->prototypeObjectWithProperty->GetLibrary()->GetMissingPropertyHolder()));

}

void TypePropertyCacheElement::Clear()

{

id = Constants::NoProperty;

}

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

// TypePropertyCache

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

size\_t TypePropertyCache::ElementIndex(const PropertyId id)

{

Assert(id != Constants::NoProperty);

Assert((TypePropertyCache\_NumElements & TypePropertyCache\_NumElements - 1) == 0);

return id & TypePropertyCache\_NumElements - 1;

}

\_\_inline bool TypePropertyCache::TryGetIndexForLoad(

const bool checkMissing,

const PropertyId id,

PropertyIndex \*const index,

bool \*const isInlineSlot,

bool \*const isMissing,

DynamicObject \* \*const prototypeObjectWithProperty) const

{

Assert(index);

Assert(isInlineSlot);

Assert(isMissing);

Assert(prototypeObjectWithProperty);

const TypePropertyCacheElement &element = elements[ElementIndex(id)];

if(element.Id() != id || !checkMissing && element.IsMissing())

return false;

\*index = element.Index();

\*isInlineSlot = element.IsInlineSlot();

\*isMissing = checkMissing ? element.IsMissing() : false;

\*prototypeObjectWithProperty = element.PrototypeObjectWithProperty();

return true;

}

\_\_inline bool TypePropertyCache::TryGetIndexForStore(

const PropertyId id,

PropertyIndex \*const index,

bool \*const isInlineSlot) const

{

Assert(index);

Assert(isInlineSlot);

const TypePropertyCacheElement &element = elements[ElementIndex(id)];

if(element.Id() != id ||

!element.IsSetPropertyAllowed() ||

element.PrototypeObjectWithProperty())

{

return false;

}

Assert(!element.IsMissing());

\*index = element.Index();

\*isInlineSlot = element.IsInlineSlot();

return true;

}

bool TypePropertyCache::TryGetProperty(

const bool checkMissing,

RecyclableObject \*const propertyObject,

const PropertyId propertyId,

Var \*const propertyValue,

ScriptContext \*const requestContext,

PropertyCacheOperationInfo \*const operationInfo,

PropertyValueInfo \*const propertyValueInfo)

{

Assert(propertyValueInfo);

Assert(propertyValueInfo->GetInlineCache() || propertyValueInfo->GetPolymorphicInlineCache());

PropertyIndex propertyIndex;

DynamicObject \*prototypeObjectWithProperty;

bool isInlineSlot, isMissing;

if(!TryGetIndexForLoad(

checkMissing,

propertyId,

&propertyIndex,

&isInlineSlot,

&isMissing,

&prototypeObjectWithProperty))

{

#if DBG\_DUMP

if(PHASE\_TRACE1(TypePropertyCachePhase))

{

CacheOperators::TraceCache(

static\_cast<InlineCache \*>(nullptr),

L"TypePropertyCache get miss",

propertyId,

requestContext,

propertyObject);

}

#endif

return false;

}

if(!prototypeObjectWithProperty)

{

#if DBG\_DUMP

if(PHASE\_TRACE1(TypePropertyCachePhase))

{

CacheOperators::TraceCache(

static\_cast<InlineCache \*>(nullptr),

L"TypePropertyCache get hit",

propertyId,

requestContext,

propertyObject);

}

#endif

#if DBG

const PropertyIndex typeHandlerPropertyIndex =

DynamicObject

::FromVar(propertyObject)

->GetDynamicType()

->GetTypeHandler()

->InlineOrAuxSlotIndexToPropertyIndex(propertyIndex, isInlineSlot);

Assert(typeHandlerPropertyIndex == propertyObject->GetPropertyIndex(propertyId));

#endif

\*propertyValue =

isInlineSlot

? DynamicObject::FromVar(propertyObject)->GetInlineSlot(propertyIndex)

: DynamicObject::FromVar(propertyObject)->GetAuxSlot(propertyIndex);

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

{

Assert(\*propertyValue == JavascriptOperators::GetProperty(propertyObject, propertyId, requestContext));

CacheOperators::Cache<false, true, false>(

false,

DynamicObject::FromVar(propertyObject),

false,

propertyObject->GetType(),

nullptr,

propertyId,

propertyIndex,

isInlineSlot,

false,

0,

propertyValueInfo,

requestContext);

return true;

}

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

// Cannot use GetProperty and compare results since they may not compare equal when they're marshaled

if(operationInfo)

{

operationInfo->cacheType = CacheType\_TypeProperty;

operationInfo->slotType = isInlineSlot ? SlotType\_Inline : SlotType\_Aux;

}

return true;

}

#if DBG\_DUMP

if(PHASE\_TRACE1(TypePropertyCachePhase))

{

CacheOperators::TraceCache(

static\_cast<InlineCache \*>(nullptr),

L"TypePropertyCache get hit prototype",

propertyId,

requestContext,

propertyObject);

}

#endif

#if DBG

const PropertyIndex typeHandlerPropertyIndex =

prototypeObjectWithProperty

->GetDynamicType()

->GetTypeHandler()

->InlineOrAuxSlotIndexToPropertyIndex(propertyIndex, isInlineSlot);

Assert(typeHandlerPropertyIndex == prototypeObjectWithProperty->GetPropertyIndex(propertyId));

#endif

\*propertyValue =

isInlineSlot

? prototypeObjectWithProperty->GetInlineSlot(propertyIndex)

: prototypeObjectWithProperty->GetAuxSlot(propertyIndex);

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

{

Assert(\*propertyValue == JavascriptOperators::GetProperty(propertyObject, propertyId, requestContext));

if(propertyObject->GetScriptContext() != requestContext)

{

return true;

}

CacheOperators::Cache<false, true, false>(

true,

prototypeObjectWithProperty,

false,

propertyObject->GetType(),

nullptr,

propertyId,

propertyIndex,

isInlineSlot,

isMissing,

0,

propertyValueInfo,

requestContext);

return true;

}

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

// Cannot use GetProperty and compare results since they may not compare equal when they're marshaled

if(operationInfo)

{

operationInfo->cacheType = CacheType\_TypeProperty;

operationInfo->slotType = isInlineSlot ? SlotType\_Inline : SlotType\_Aux;

}

return true;

}

bool TypePropertyCache::TrySetProperty(

RecyclableObject \*const object,

const PropertyId propertyId,

Var propertyValue,

ScriptContext \*const requestContext,

PropertyCacheOperationInfo \*const operationInfo,

PropertyValueInfo \*const propertyValueInfo)

{

Assert(propertyValueInfo);

Assert(propertyValueInfo->GetInlineCache() || propertyValueInfo->GetPolymorphicInlineCache());

PropertyIndex propertyIndex;

bool isInlineSlot;

if(!TryGetIndexForStore(propertyId, &propertyIndex, &isInlineSlot))

{

#if DBG\_DUMP

if(PHASE\_TRACE1(TypePropertyCachePhase))

{

CacheOperators::TraceCache(

static\_cast<InlineCache \*>(nullptr),

L"TypePropertyCache set miss",

propertyId,

requestContext,

object);

}

#endif

return false;

}

#if DBG\_DUMP

if(PHASE\_TRACE1(TypePropertyCachePhase))

{

CacheOperators::TraceCache(

static\_cast<InlineCache \*>(nullptr),

L"TypePropertyCache set hit",

propertyId,

requestContext,

object);

}

#endif

Assert(!object->IsFixedProperty(propertyId));

Assert(

(

DynamicObject

::FromVar(object)

->GetDynamicType()

->GetTypeHandler()

->InlineOrAuxSlotIndexToPropertyIndex(propertyIndex, isInlineSlot)

) ==

object->GetPropertyIndex(propertyId));

Assert(object->CanStorePropertyValueDirectly(propertyId, false));

ScriptContext \*const objectScriptContext = object->GetScriptContext();

if(objectScriptContext != requestContext)

{

propertyValue = CrossSite::MarshalVar(objectScriptContext, propertyValue);

}

if(isInlineSlot)

{

DynamicObject::FromVar(object)->SetInlineSlot(SetSlotArguments(propertyId, propertyIndex, propertyValue));

}

else

{

DynamicObject::FromVar(object)->SetAuxSlot(SetSlotArguments(propertyId, propertyIndex, propertyValue));

}

if(objectScriptContext == requestContext)

{

CacheOperators::Cache<false, false, false>(

false,

DynamicObject::FromVar(object),

false,

object->GetType(),

nullptr,

propertyId,

propertyIndex,

isInlineSlot,

false,

0,

propertyValueInfo,

requestContext);

return true;

}

if(operationInfo)

{

operationInfo->cacheType = CacheType\_TypeProperty;

operationInfo->slotType = isInlineSlot ? SlotType\_Inline : SlotType\_Aux;

}

return true;

}

void TypePropertyCache::Cache(

const PropertyId id,

const PropertyIndex index,

const bool isInlineSlot,

const bool isSetPropertyAllowed)

{

elements[ElementIndex(id)].Cache(id, index, isInlineSlot, isSetPropertyAllowed);

}

void TypePropertyCache::Cache(

const PropertyId id,

const PropertyIndex index,

const bool isInlineSlot,

const bool isSetPropertyAllowed,

const bool isMissing,

DynamicObject \*const prototypeObjectWithProperty,

Type \*const myParentType)

{

Assert(myParentType);

Assert(myParentType->GetPropertyCache() == this);

elements[ElementIndex(id)].Cache(

id,

index,

isInlineSlot,

isSetPropertyAllowed,

isMissing,

prototypeObjectWithProperty,

myParentType);

}

void TypePropertyCache::ClearIfPropertyIsOnAPrototype(const PropertyId id)

{

TypePropertyCacheElement &element = elements[ElementIndex(id)];

if(element.Id() == id && element.PrototypeObjectWithProperty())

element.Clear();

}

void TypePropertyCache::Clear(const PropertyId id)

{

TypePropertyCacheElement &element = elements[ElementIndex(id)];

if(element.Id() == id)

element.Clear();

}

}

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

// 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

// Must be a power of 2

#define TypePropertyCache\_NumElements 16

namespace Js

{

struct PropertyCacheOperationInfo;

class TypePropertyCacheElement

{

private:

DynamicObject \*prototypeObjectWithProperty;

PropertyId id;

PropertyIndex index;

bool isInlineSlot : 1;

bool isSetPropertyAllowed : 1;

bool isMissing : 1;

public:

TypePropertyCacheElement();

public:

PropertyId Id() const;

PropertyIndex Index() const;

bool IsInlineSlot() const;

bool IsSetPropertyAllowed() const;

bool IsMissing() const;

DynamicObject \*PrototypeObjectWithProperty() const;

void Cache(const PropertyId id, const PropertyIndex index, const bool isInlineSlot, const bool isSetPropertyAllowed);

void Cache(

const PropertyId id,

const PropertyIndex index,

const bool isInlineSlot,

const bool isSetPropertyAllowed,

const bool isMissing,

DynamicObject \*const prototypeObjectWithProperty,

Type \*const myParentType);

void Clear();

};

class TypePropertyCache

{

private:

TypePropertyCacheElement elements[TypePropertyCache\_NumElements];

private:

static size\_t ElementIndex(const PropertyId id);

bool TryGetIndexForLoad(const bool checkMissing, const PropertyId id, PropertyIndex \*const index, bool \*const isInlineSlot, bool \*const isMissing, DynamicObject \* \*const prototypeObjectWithProperty) const;

bool TryGetIndexForStore(const PropertyId id, PropertyIndex \*const index, bool \*const isInlineSlot) const;

public:

bool TryGetProperty(const bool checkMissing, RecyclableObject \*const propertyObject, const PropertyId propertyId, Var \*const propertyValue, ScriptContext \*const requestContext, PropertyCacheOperationInfo \*const operationInfo, PropertyValueInfo \*const propertyValueInfo);

bool TrySetProperty(RecyclableObject \*const object, const PropertyId propertyId, Var propertyValue, ScriptContext \*const requestContext, PropertyCacheOperationInfo \*const operationInfo, PropertyValueInfo \*const propertyValueInfo);

public:

void Cache(const PropertyId id, const PropertyIndex index, const bool isInlineSlot, const bool isSetPropertyAllowed);

void Cache(const PropertyId id, const PropertyIndex index, const bool isInlineSlot, const bool isSetPropertyAllowed, const bool isMissing, DynamicObject \*const prototypeObjectWithProperty, Type \*const myParentType);

void ClearIfPropertyIsOnAPrototype(const PropertyId id);

void Clear(const PropertyId id);

};

}

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

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

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

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

#include "RuntimeTypePch.h"

namespace Js

{

bool WithScopeObject::Is(Var aValue)

{

return JavascriptOperators::GetTypeId(aValue) == TypeIds\_WithScopeObject;

}

WithScopeObject\* WithScopeObject::FromVar(Var aValue)

{

Assert(WithScopeObject::Is(aValue));

return static\_cast<WithScopeObject\*>(aValue);

}

BOOL WithScopeObject::HasProperty(PropertyId propertyId)

{

return JavascriptOperators::HasPropertyUnscopables(wrappedObject, propertyId);

}

BOOL WithScopeObject::HasOwnProperty(PropertyId propertyId)

{

Assert(!Js::IsInternalPropertyId(propertyId));

return HasProperty(propertyId);

}

BOOL WithScopeObject::SetProperty(PropertyId propertyId, Var value, PropertyOperationFlags flags, PropertyValueInfo\* info)

{

return JavascriptOperators::SetPropertyUnscopable(wrappedObject, wrappedObject, propertyId, value, info, wrappedObject->GetScriptContext());

}

BOOL WithScopeObject::GetProperty(Var originalInstance, PropertyId propertyId, Var\* value, PropertyValueInfo\* info, ScriptContext\* requestContext)

{

return JavascriptOperators::GetPropertyUnscopable(wrappedObject, wrappedObject, propertyId, value, requestContext, info);

}

BOOL WithScopeObject::DeleteProperty(PropertyId propertyId, PropertyOperationFlags flags)

{

return JavascriptOperators::DeletePropertyUnscopables(wrappedObject, propertyId, flags);

}

DescriptorFlags WithScopeObject::GetSetter(PropertyId propertyId, Var \*setterValue, PropertyValueInfo\* info, ScriptContext\* requestContext)

{

return JavascriptOperators::GetterSetterUnscopable(wrappedObject, propertyId, setterValue, info, requestContext);

}

BOOL WithScopeObject::GetPropertyReference(Var originalInstance, PropertyId propertyId, Var\* value, PropertyValueInfo\* info, ScriptContext\* requestContext)

{

RecyclableObject\* copyState = wrappedObject;

return JavascriptOperators::PropertyReferenceWalkUnscopable(wrappedObject, &copyState, propertyId, value, info, requestContext);

}

} // namespace Js

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

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

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

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

#pragma once

namespace Js

{

class WithScopeObject : public RecyclableObject

{

private:

RecyclableObject \*wrappedObject;

void AssertAndFailFast() { AssertMsg(false, "This function should not be invoked"); Js::Throw::InternalError(); }

protected:

DEFINE\_VTABLE\_CTOR(WithScopeObject, RecyclableObject);

public:

WithScopeObject(RecyclableObject \*wrappedObject, StaticType \* type) : RecyclableObject(type), wrappedObject(wrappedObject) {}

static bool Is(Var aValue);

static WithScopeObject\* FromVar(Var value);

RecyclableObject \*GetWrappedObject() { return wrappedObject; }

virtual BOOL HasProperty(PropertyId propertyId) override;

virtual BOOL HasOwnProperty(PropertyId propertyId) override;

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

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

virtual BOOL DeleteProperty(PropertyId propertyId, PropertyOperationFlags flags) override;

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

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

// A WithScopeObject should never call the Functions defined below this comment

virtual BOOL SetProperty(JavascriptString\* propertyNameString, Var value, PropertyOperationFlags flags, PropertyValueInfo\* info) override { AssertAndFailFast(); return FALSE; };

virtual BOOL GetProperty(Var originalInstance, JavascriptString\* propertyNameString, Var\* value, PropertyValueInfo\* info, ScriptContext\* requestContext) override { AssertAndFailFast(); return FALSE; };

virtual DescriptorFlags GetSetter(JavascriptString\* propertyNameString, Var \*setterValue, PropertyValueInfo\* info, ScriptContext\* requestContext) override { AssertAndFailFast(); return None; };

virtual int GetPropertyCount() override { AssertAndFailFast(); return 0; };

virtual PropertyId GetPropertyId(PropertyIndex index) override { AssertAndFailFast(); return Constants::NoProperty; };

virtual PropertyId GetPropertyId(BigPropertyIndex index) override { AssertAndFailFast(); return Constants::NoProperty;; };

virtual BOOL SetInternalProperty(PropertyId internalPropertyId, Var value, PropertyOperationFlags flags, PropertyValueInfo\* info) override { AssertAndFailFast(); return FALSE; };

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

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

virtual BOOL IsFixedProperty(PropertyId propertyId) override { AssertAndFailFast(); return FALSE; };

virtual BOOL HasItem(uint32 index) override { AssertAndFailFast(); return FALSE; };

virtual BOOL HasOwnItem(uint32 index) override { AssertAndFailFast(); return FALSE; };

virtual BOOL GetItem(Var originalInstance, uint32 index, Var\* value, ScriptContext \* requestContext) override { AssertAndFailFast(); return FALSE; };

virtual BOOL GetItemReference(Var originalInstance, uint32 index, Var\* value, ScriptContext \* requestContext) override { AssertAndFailFast(); return FALSE; };

virtual DescriptorFlags GetItemSetter(uint32 index, Var\* setterValue, ScriptContext\* requestContext) override { AssertAndFailFast(); return None; };

virtual BOOL SetItem(uint32 index, Var value, PropertyOperationFlags flags) override { AssertAndFailFast(); return FALSE; };

virtual BOOL DeleteItem(uint32 index, PropertyOperationFlags flags) override { AssertAndFailFast(); return FALSE; };

virtual BOOL ToPrimitive(JavascriptHint hint, Var\* result, ScriptContext \* requestContext) override { AssertAndFailFast(); return FALSE; };

virtual BOOL GetEnumerator(BOOL enumNonEnumerable, Var\* enumerator, ScriptContext\* scriptContext, bool preferSnapshotSemantics = true, bool enumSymbols = false) override { AssertAndFailFast(); return FALSE; };

virtual BOOL SetAccessors(PropertyId propertyId, Var getter, Var setter, PropertyOperationFlags flags = PropertyOperation\_None) override { AssertAndFailFast(); return FALSE; };

virtual BOOL GetAccessors(PropertyId propertyId, Var \*getter, Var \*setter, ScriptContext \* requestContext) override { AssertAndFailFast(); return FALSE; };

virtual BOOL IsWritable(PropertyId propertyId) override { AssertAndFailFast(); return FALSE; };

virtual BOOL IsConfigurable(PropertyId propertyId) override { AssertAndFailFast(); return FALSE; };

virtual BOOL IsEnumerable(PropertyId propertyId) override { AssertAndFailFast(); return FALSE; };

virtual BOOL SetEnumerable(PropertyId propertyId, BOOL value) override { AssertAndFailFast(); return FALSE; };

virtual BOOL SetWritable(PropertyId propertyId, BOOL value) override { AssertAndFailFast(); return FALSE; };

virtual BOOL SetConfigurable(PropertyId propertyId, BOOL value) override { AssertAndFailFast(); return FALSE; };

virtual BOOL SetAttributes(PropertyId propertyId, PropertyAttributes attributes) override { AssertAndFailFast(); return FALSE; };

virtual BOOL IsExtensible() override { AssertAndFailFast(); return FALSE; };

virtual BOOL PreventExtensions() override { AssertAndFailFast(); return FALSE; };

virtual BOOL Seal() override { AssertAndFailFast(); return FALSE; };

virtual BOOL Freeze() override { AssertAndFailFast(); return FALSE; };

virtual BOOL IsSealed() override { AssertAndFailFast(); return FALSE; };

virtual BOOL IsFrozen() override { AssertAndFailFast(); return FALSE; };

virtual BOOL GetDiagValueString(StringBuilder<ArenaAllocator>\* stringBuilder, ScriptContext\* requestContext) override { AssertAndFailFast(); return FALSE; };

virtual Var GetTypeOfString(ScriptContext \* requestContext) override { AssertAndFailFast(); return RecyclableObject::GetTypeOfString(requestContext); };

};

} // namespace Js