Contents

Common Language Runtime (CLR) Integration Programming Concepts

10/1/2018 • 2 minutes to read • Edit Online

APPLIES TO: ✓ SQL Server ✗ Azure SQL Database ✗ Azure SQL Data Warehouse ✗ Parallel Data Warehouse

Beginning with SQL Server 2005 (9.x), SQL Server features the integration of the common language runtime (CLR) component of the .NET Framework for Microsoft Windows. This means that you can now write stored procedures, triggers, user-defined types, user-defined functions, user-defined aggregates, and streaming table-valued functions, using any .NET Framework language, including Microsoft Visual Basic .NET and Microsoft Visual C#.

The Microsoft.SqlServer.Server namespace includes core functionality for CLR programming in SQL Server. However, the Microsoft.SqlServer.Server namespace is documented in the .NET Framework SDK. This documentation is not included in SQL Server Books Online.

IMPORTANT

By default, the .NET Framework is installed with SQL Server, but the .NET Framework SDK is not. Without the SDK installed on your computer and included in the Books Online collection, links to SDK content in this section do not work. Install the .NET Framework SDK. Once installed, add the SDK to the Books Online collection and table of contents by following the instructions in Installing the .NET Framework SDK.

NOTE

CLR functionality, such as CLR user functions, are *not* supported for Azure SQL Database.

The following table lists the topics in this section.

Common Language Runtime (CLR) Integration Overview
Provides a brief overview of the CLR, and describes how and why this technology has been used in SQL Server. Describes the benefits of using the CLR to create database objects.

Assemblies (Database Engine)
Describes how assemblies are used in SQL Server to deploy functions, stored procedures, triggers, user-defined aggregates, and user-defined types that are written in one of the managed code languages hosted by the Microsoft .NET Framework common language runtime (CLR), and not written in Transact-SQL.

Building Database Objects with Common Language Runtime (CLR) Integration
Describes the kinds of objects that can be built using the CLR, and reviews the requirements for building CLR database objects.

Data Access from CLR Database Objects
Describes how a CLR routine can access data stored in an instance of SQL Server.

CLR Integration Security
Describes the CLR integration security model.

Debugging CLR Database Objects
Describes limitations of and requirements for debugging CLR database objects.

Deploying CLR Database Objects
Describes deploying assemblies to production servers.

Managing CLR Integration Assemblies
Describes how to create and drop CLR integration assemblies.

Monitoring and Troubleshooting Managed Database Objects
Provides information about the tools that can be used to monitor and troubleshoot managed database objects and assemblies running in SQL Server.

Usage Scenarios and Examples for Common Language Runtime (CLR) Integration
Describes usage scenarios and code samples using CLR objects.

See Also

Assemblies (Database Engine)
Installing the .NET Framework SDK

Building Database Objects with Common Language Runtime (CLR) Integration

10/1/2018 • 2 minutes to read • Edit Online

APPLIES TO: ✓ SQL Server ⊗ Azure SQL Database ⊗ Azure SQL Data Warehouse ⊗ Parallel Data Warehouse

You can build database objects using the SQL Server integration with the .NET Framework common language runtime (CLR). Managed code that runs inside of Microsoft SQL Server is referred to as a "CLR routine." These routines include:

- Scalar-valued user-defined functions (scalar UDFs)
- Table-valued user-defined functions (TVFs)
- User-defined procedures (UDPs)
- User-defined triggers

  CLR routines have the same structure in managed code. They are mapped to public, static (shared in Microsoft Visual Basic .NET) methods of a class. In addition to routines, user-defined types (UDTs) and user-defined aggregate functions can also be defined using the .NET Framework. UDTs and user-defined aggregates are mapped to entire .NET Framework classes.

  Each type of .NET Framework routine has a Transact-SQL declaration and can be used anywhere in SQL Server that the Transact-SQL equivalent can be used. For instance, scalar UDFs can be used in any scalar expression. A TVF can be used in any FROM clause. A procedure can be invoked in an EXEC statement or invoked from a client application.

> **NOTE**
> Execution of a CLR object (user-defined function, user-defined type, or trigger) on the common language runtime can take place on multiple threads (parallel plan), if the query optimizer decides it is beneficial. However, if a user-defined function accesses data, execution will be on a serial plan. When executed on a server version prior to SQL Server 2008, if a user-defined function contains LOB parameters or return values, execution also must be on a serial plan.

The following table lists the topics covered in this section.

Getting Started with CLR Integration
Provides a brief overview of the libraries and namespaces required to compile object using CLR integration with SQL Server. Includes an example "Hello World" CLR stored procedure.

Supported .NET Framework Libraries
Provides information on the .NET Framework libraries supported by CLR integration.

CLR Integration Programming Model Restrictions
Provides information about CLR integration programming model restrictions.

SQL Server Data Types in the .NET Framework
An overview of SQL Server data types and their .NET Framework equivalents.

Overview of CLR Integration Custom Attributes
Provides information about CLR integration custom attributes.

CLR User-Defined Functions
Describes how to implement and use the various types of CLR functions: table-valued, scalar, and user-defined aggregate functions.

CLR User-Defined Types
Describes how to implement and use CLR user-defined types.

CLR Stored Procedures
Describes how to implement and use CLR stored procedures.

CLR Triggers
Describes how to implement and use CLR triggers.

See Also

Common Language Runtime (CLR) Integration Overview

Data Access from CLR Database Objects

10/1/2018 • 2 minutes to read • Edit Online

APPLIES TO: ✓ SQL Server ✗ Azure SQL Database ✗ Azure SQL Data Warehouse ✗ Parallel Data Warehouse

A common language runtime (CLR) routine may easily access data stored in the instance of Microsoft SQL Server in which it runs, as well as data stored in remote instances. Which particular data the routine can access is determined by the user context in which the code is running. Access data from within a CLR database object by using the .NET Framework Data Provider for SQL Server, also referred to as **SqlClient**. This is the same provider used by developers accessing SQL Server data from managed client and middle-tier applications. Because of this, you can leverage your knowledge of ADO.NET and **SqlClient** in client and middle-tier applications.

NOTE

User-defined type methods and user-defined functions are not allowed to perform data access by default. You must set the **DataAccess** property of **SqlMethodAttribute** or **SqlFunctionAttribute** to **DataAccessKind.Read** to enable read-only data access from user-defined type (UDT) methods or user-defined functions. Data modification operations are not allowed from UDTs or user-defined functions, and throws exceptions at execution time if attempted.

This section discusses only the specific functional and behavioral differences when accessing data from within a CLR database object. For more information about the features and functionality of ADO.NET, see the ADO.NET documentation included in the .NET Framework SDK.

The following table lists the topics in this section.

Context Connection
Describes the context connection to SQL Server.

Impersonation and Credentials for Connections
Describes impersonating connections and connection credentials.

SQL Server In-Process Specific Extensions to ADO.NET
Discusses the in-process specific **SqlPipe**, **SqlContext**, **SqlTriggerContext**, and **SqlDataRecord** objects.

CLR Integration and Transactions
Describes how the new transaction framework provided in the System.Transactions namespace integrates with ADO.NET and SQL Server CLR integration.

XML Serialization from CLR Database Objects
Explains how to enable XML serialization scenarios of CLR database objects inside SQL Server.

CLR Integration Code Access Security

10/1/2018 • 6 minutes to read • Edit Online

APPLIES TO: ✓ SQL Server ✗ Azure SQL Database ✗ Azure SQL Data Warehouse ✗ Parallel Data Warehouse

The common language runtime (CLR) supports a security model called code access security for managed code. In this model, permissions are granted to assemblies based on the identity of the code. For more information, see the "Code Access Security" section in the .NET Framework software development kit.

The security policy that determines the permissions granted to assemblies is defined in three different places:

- Machine policy: This is the policy in effect for all managed code running in the machine on which SQL Server is installed.
- User policy: This is the policy in effect for managed code hosted by a process. For SQL Server, the user policy is specific to the Windows account on which the SQL Server service is running.
- Host policy: This is the policy set up by the host of the CLR (in this case, SQL Server) that is in effect for managed code running in that host.

  The code access security mechanism supported by the CLR is based on the assumption that the runtime can host both fully trusted and partially trusted code. The resources that are protected by CLR code access security are typically wrapped by managed application programming interfaces that require the corresponding permission before allowing access to the resource. The demand for the permission is satisfied only if all the callers (at the assembly level) in the call stack have the corresponding resource permission.

  The set of code access security permissions that are granted to managed code when running inside SQL Server is the intersection of the set of permissions granted by the above three policy levels. Even if SQL Server grants a set of permissions to an assembly loaded in SQL Server, the eventual set of permissions given to user code may be restricted further by the user and machine-level policies.

SQL Server Host Policy Level Permission Sets

The set of code access security permissions granted to assemblies by the SQL Server host policy level is determined by the permission set specified when creating the assembly. There are three permission sets: **SAFE**, **EXTERNAL_ACCESS** and **UNSAFE** (specified using the **PERMISSION_SET** option of CREATE ASSEMBLY (Transact-SQL)).

SQL Server supplies a host-level security policy level to the CLR while hosting it; this policy is an additional policy level below the two policy levels that are always in effect. This policy is set for every application domain that is created by SQL Server. This policy is not meant for the default application domain that would be in effect when SQL Server creates an instance of the CLR.

The SQL Server host-level policy is a combination of SQL Server fixed policy for system assemblies and user-specified policy for user assemblies.

The fixed policy for CLR assemblies and SQL Server system assemblies grants them full trust.

The user-specified portion of the SQL Server host policy is based on the assembly owner specifying one of three permission buckets for each assembly. For more information about the security permissions listed below, see the .NET Framework SDK.

SAFE

Only internal computation and local data access are allowed. **SAFE** is the most restrictive permission set. Code executed by an assembly with **SAFE** permissions cannot access external system resources such as files, the network, environment variables, or the registry.

SAFE assemblies have the following permissions and values:

PERMISSION	VALUE(S)/DESCRIPTION
SecurityPermission	**Execution:** Permission to execute managed code.
SqlClientPermission	**Context connection = true**, **context connection = yes**: Only the context-connection can be used and the connection string can only specify a value of "context connection=true" or "context connection=yes". **AllowBlankPassword = false:** Blank passwords are not permitted.

EXTERNAL_ACCESS

EXTERNAL_ACCESS assemblies have the same permissions as **SAFE** assemblies, with the additional ability to access external system resources such as files, networks, environmental variables, and the registry.

EXTERNAL_ACCESS assemblies also have the following permissions and values:

PERMISSION	VALUE(S)/DESCRIPTION
DistributedTransactionPermission	**Unrestricted:** Distributed transactions are allowed.
DNSPermission	**Unrestricted:** Permission to request information from Domain Name Servers.
EnvironmentPermission	**Unrestricted:** Full access to system and user environment variables is allowed.
EventLogPermission	**Administer:** The following actions are allowed: creating an event source, reading existing logs, deleting event sources or logs, responding to entries, clearing an event log, listening to events, and accessing a collection of all event logs.
FileIOPermission	**Unrestricted:** Full access to files and folders is allowed.
KeyContainerPermission	**Unrestricted:** Full access to key containers is allowed.
NetworkInformationPermission	**Access:** Pinging is permitted.
RegistryPermission	Allows read rights to **HKEY_CLASSES_ROOT**, **HKEY_LOCAL_MACHINE**, **HKEY_CURRENT_USER**, **HKEY_CURRENT_CONFIG**, and **HKEY_USERS.**

PERMISSION	VALUE(S)/DESCRIPTION
SecurityPermission	**Assertion:** Ability to assert that all the callers of this code have the requisite permission for the operation. **ControlPrincipal:** Ability to manipulate the principal object. **Execution:** Permission to execute managed code. **SerializationFormatter:** Ability to provide serialization services.
SmtpPermission	**Access:** Outbound connections to SMTP host port 25 are allowed.
SocketPermission	**Connect:** Outbound connections (all ports, all protocols) on a transport address are allowed.
SqlClientPermission	**Unrestricted:** Full access to the datasource is allowed.
StorePermission	**Unrestricted:** Full access to X.509 certificate stores is allowed.
WebPermission	**Connect:** Outbound connections to web resources are allowed.

UNSAFE

UNSAFE allows assemblies unrestricted access to resources, both within and outside SQL Server. Code executing from within an **UNSAFE** assembly can also call unmanaged code.

UNSAFE assemblies are given **FullTrust**.

IMPORTANT

SAFE is the recommended permission setting for assemblies that perform computation and data management tasks without accessing resources outside SQL Server. **EXTERNAL_ACCESS** is recommended for assemblies that access resources outside SQL Server. **EXTERNAL_ACCESS** assemblies by default execute as the SQL Server service account. It is possible for **EXTERNAL_ACCESS** code to explicitly impersonate the caller's Windows Authentication security context. Since the default is to execute as the SQL Server service account, permission to execute **EXTERNAL_ACCESS** should only be given to logins trusted to run as the service account. From a security perspective, **EXTERNAL_ACCESS** and **UNSAFE** assemblies are identical. However, **EXTERNAL_ACCESS** assemblies provide various reliability and robustness protections that are not in **UNSAFE** assemblies. Specifying **UNSAFE** allows the code in the assembly to perform illegal operations against the SQL Server process space, and hence can potentially compromise the robustness and scalability of SQL Server. For more information about creating CLR assemblies in SQL Server, see Managing CLR Integration Assemblies.

Accessing External Resources

If a user-defined type (UDT), stored procedure, or other type of construct assembly is registered with the **SAFE** permission set, then managed code executing in the construct is unable to access external resources. However, if either the **EXTERNAL_ACCESS** or **UNSAFE** permission sets are specified, and managed code attempts to access external resources, SQL Server applies the following rules:

IF	THEN
The execution context corresponds to a SQL Server login.	Attempts to access external resources are denied and a security exception is raised.

IF	THEN
The execution context corresponds to a Windows login and the execution context is the original caller.	The external resource is accessed under the security context of the SQL Server service account.
The caller is not the original caller.	Access is denied and a security exception is raised.
The execution context corresponds to a Windows login and the execution context is the original caller and the caller has been impersonated.	Access uses the caller security context; not the service account.

Permission Set Summary

The following chart summarizes the restrictions and permissions granted to the **SAFE**, **EXTERNAL_ACCESS**, and **UNSAFE** permission sets.

	SAFE	**EXTERNAL_ACCESS**	**UNSAFE**
Code Access Security Permissions	Execute only	Execute + access to external resources	Unrestricted (including P/Invoke)
Programming model restrictions	Yes	Yes	No restrictions
Verifiability requirement	Yes	Yes	No
Local data access	Yes	Yes	Yes
Ability to call native code	No	No	Yes

See Also

CLR Integration Security
Host Protection Attributes and CLR Integration Programming
CLR Integration Programming Model Restrictions
CLR Hosted Environment

Managing CLR Integration Assemblies

10/1/2018 • 2 minutes to read • Edit Online

APPLIES TO: ✓ SQL Server ✗ Azure SQL Database ✗ Azure SQL Data Warehouse ✗ Parallel Data Warehouse

Managed code is compiled and then deployed in units called an assembly. An assembly is packaged as a DLL or executable (.exe) file. While an executable file can run on its own, a DLL must be hosted in an existing application. Managed DLL assemblies can be loaded into and hosted by Microsoft SQL Server. SQL Server requires you to register the assembly in a SQL Server database using the CREATE ASSEMBLY statement, before it can be loaded in the process and used. Assemblies can also be updated from a more recent version using the ALTER ASSEMBLY statement, or removed from SQL Server using the DROP ASSEMBLY statement.

Assembly information is stored in the **sys.assembly_files** table in the database where the assembly has been installed. The **sys.assembly_files** table contains the following columns.

COLUMN	DESCRIPTION
assembly_id	The identifier defined for the assembly. This number is assigned to all objects relating to the same assembly.
name	The name of the object.
file_id	A number identifying each object, with the first object associated with a given **assembly_id** being given the value of 1. If multiple objects are associated with the same **assembly_id**, then each subsequent **file_id** value is incremented by 1.
content	The hexadecimal representation of the assembly or file.

In This Section

Creating an Assembly
Discusses creating SAFE, EXTERNAL_ACCESS, and UNSAFE CLR assemblies in SQL Server.

Altering an Assembly
Describes updating CLR assemblies in SQL Server.

Dropping an Assembly
Discusses dropping CLR assemblies from SQL Server.

See Also

CLR Integration Security
CLR Integration Code Access Security

Assemblies - Designing

10/1/2018 • 3 minutes to read • Edit Online

APPLIES TO: ✓ SQL Server ✗ Azure SQL Database ✗ Azure SQL Data Warehouse ✗ Parallel Data Warehouse

This topic describes the following factors you should consider when you design assemblies:

- Packaging assemblies
- Managing assembly security
- Restrictions on assemblies

Packaging Assemblies

An assembly can contain functionality for more than one SQL Server routine or type in its classes and methods. Most of the time, it makes sense to package the functionality of routines that perform related functions within the same assembly, especially if these routines share classes whose methods call one another. For example, classes that perform data entry management tasks for common language runtime (CLR) triggers and CLR stored procedures can be packaged in the same assembly. This is because the methods for these classes are more likely to call each other than those of less related tasks.

When you are packaging code into assembly, you should consider the following:

- CLR user-defined types and indexes that depend on CLR user-defined functions can cause persisted data to be in the database that depends on the assembly. Modifying the code of an assembly is frequently more complex when there is persisted data that depends on the assembly in the database. Therefore, it is generally better to separate code on which persisted data dependencies rely (such as user-defined types and indexes using user-defined functions) from code that does not have such persisted data dependencies. For more information, see Implementing Assemblies and ALTER ASSEMBLY (Transact-SQL).

- If a piece of managed code requires higher permission, it is better to separate that code into a separate assembly from code that does not require higher permission.

Managing Assembly Security

You can control how much an assembly can access resources protected by .NET Code Access Security when it runs managed code. You do this by specifying one of three permission sets when you create or modify an assembly: SAFE, EXTERNAL_ACCESS, or UNSAFE.

SAFE

SAFE is the default permission set and it is the most restrictive. Code run by an assembly with SAFE permissions cannot access external system resources such as files, the network, environment variables, or the registry. SAFE code can access data from the local SQL Server databases or perform computations and business logic that do not involve accessing resources outside the local databases.

Most assemblies perform computation and data management tasks without having to access resources outside SQL Server. Therefore, we recommend SAFE as the assembly permission set.

EXTERNAL_ACCESS

EXTERNAL_ACCESS allows for assemblies to access certain external system resources such as files, networks, Web services, environmental variables, and the registry. Only SQL Server logins with EXTERNAL ACCESS permissions can create EXTERNAL_ACCESS assemblies.

SAFE and EXTERNAL_ACCESS assemblies can contain only code that is verifiably type-safe. This means that these assemblies can only access classes through well-defined entry points that are valid for the type definition. Therefore, they cannot arbitrarily access memory buffers not owned by the code. Additionally, they cannot perform operations that might have an adverse effect on the robustness of the SQL Server process.

UNSAFE

UNSAFE gives assemblies unrestricted access to resources, both within and outside SQL Server. Code that is running from within an UNSAFE assembly can call unmanaged code.

Also, specifying UNSAFE allows for the code in the assembly to perform operations that are considered type-unsafe by the CLR verifier. These operations can potentially access memory buffers in the SQL Server process space in an uncontrolled manner. UNSAFE assemblies can also potentially subvert the security system of either SQL Server or the common language runtime. UNSAFE permissions should be granted only to highly trusted assemblies by experienced developers or administrators. Only members of the **sysadmin** fixed server role can create UNSAFE assemblies.

Restrictions on Assemblies

SQL Server puts certain restrictions on managed code in assemblies to make sure that they can run in a reliable and scalable manner. This means that certain operations that can compromise the robustness of the server are not permitted in SAFE and EXTERNAL_ACCESS assemblies.

Disallowed Custom Attributes

Assemblies cannot be annotated with the following custom attributes:

```
System.ContextStaticAttribute
System.MTAThreadAttribute
System.Runtime.CompilerServices.MethodImplAttribute
System.Runtime.CompilerServices.CompilationRelaxationsAttribute
System.Runtime.Remoting.Contexts.ContextAttribute
System.Runtime.Remoting.Contexts.SynchronizationAttribute
System.Runtime.InteropServices.DllImportAttribute
System.Security.Permissions.CodeAccessSecurityAttribute
System.STAThreadAttribute
System.ThreadStaticAttribute
```

Additionally, SAFE and EXTERNAL_ACCESS assemblies cannot be annotated with the following custom attributes:

```
System.Security.SuppressUnmanagedCodeSecurityAttribute
System.Security.UnverifiableCodeAttribute
```

Disallowed .NET Framework APIs

Any Microsoft .NET Framework API that is annotated with one of the disallowed **HostProtectionAttributes** cannot be called from SAFE and EXTERNAL_ACCESS assemblies.

```
eSelfAffectingProcessMgmt
eSelfAffectingThreading
eSynchronization
eSharedState
eExternalProcessMgmt
eExternalThreading
eSecurityInfrastructure
eMayLeakOnAbort
eUI
```

Supported .NET Framework Assemblies

Any assembly that is referenced by your custom assembly must be loaded into SQL Server by using CREATE ASSEMBLY. The following .NET Framework assemblies are already loaded into SQL Server and, therefore, can be referenced by custom assemblies without having to use CREATE ASSEMBLY.

```
custommarshallers.dll
Microsoft.visualbasic.dll
Microsoft.visualc.dll
mscorlib.dll
system.data.dll
System.Data.SqlXml.dll
system.dll
system.security.dll
system.web.services.dll
system.xml.dll
System.Transactions
System.Data.OracleClient
System.Configuration
```

See Also

Assemblies (Database Engine)
CLR Integration Security

Assemblies - Getting Information

10/1/2018 • 2 minutes to read • Edit Online

APPLIES TO: ✓ SQL Server ✗ Azure SQL Database ✗ Azure SQL Data Warehouse ✗ Parallel Data Warehouse

The following catalog views and functions can be queried for metadata about assemblies.

To get information about individual assemblies

- ASSEMBLYPROPERTY (Transact-SQL)

  To get information about all assemblies in the database

- sys.assemblies (Transact-SQL)

  To get information about assembly files, including assembly binaries, source files, and debug files

- sys.assembly_files (Transact-SQL)

  To get information about cross-assembly references

- sys.assembly_references (Transact-SQL)

  To get assembly information about user-defined types

- sys.assembly_types (Transact-SQL)
- sys.types (Transact-SQL)

  To get assembly information about common language runtime (CLR) stored procedures, triggers, and functions

- sys.assembly_modules (Transact-SQL)

  To get information about non-CLR objects

- sys.sql_modules (Transact-SQL)

See Also

Assemblies (Database Engine)
Designing Assemblies
Implementing Assemblies

Assemblies - Implementing

10/1/2018 • 4 minutes to read • Edit Online

APPLIES TO: ✓ SQL Server ✗ Azure SQL Database ✗ Azure SQL Data Warehouse ✗ Parallel Data Warehouse

This topic provides information about the following areas to help you implement and work with assemblies in the database:

- Creating assemblies
- Modifying assemblies
- Dropping, disabling, and enabling assemblies
- Managing assembly versions

Creating Assemblies

Assemblies are created in SQL Server by using the Transact-SQL CREATE ASSEMBLY statement, or in the SQL Server Management Studio by using the Assembly Assisted Editor. Additionally, deploying a SQL Server Project in Microsoft Visual Studio registers an assembly in the database that was specified for the project. For more information, see Deploying CLR Database Objects.

To create an assembly by using Transact-SQL

- CREATE ASSEMBLY (Transact-SQL)

  To create an assembly by using SQL Server Management Studio

- Assembly Properties (General Page)

Modifying Assemblies

Assemblies are modified in SQL Server by using the Transact-SQL ALTER ASSEMBLY statement or in SQL Server Management Studio by using the Assembly Assisted Editor. You can modify an assembly when you want to do the following:

- Change the implementation of the assembly by uploading a newer version of the binaries of the assembly. For more information, see Managing Assembly Versions later in this topic.
- Change the permission set of the assembly. For more information, see Designing Assemblies.
- Change the visibility of the assembly. Visible assemblies are available for referencing in SQL Server. Nonvisible assemblies are not available, even if they have been uploaded in the database. By default, assemblies uploaded to an instance of SQL Server are visible.
- Add or drop a debug or source file associated with the assembly.

  To modify an assembly by using Transact-SQL

- ALTER ASSEMBLY (Transact-SQL)

  To modify an assembly by using SQL Server Management Studio

- Assembly Properties (General Page)

Dropping, Disabling, and Enabling Assemblies

Assemblies are dropped by using the Transact-SQL DROP ASSEMBLY statement or SQL Server Management Studio.

To drop an assembly by using Transact-SQL

- DROP ASSEMBLY (Transact-SQL)

  To drop an assembly by using SQL Server Management Studio

- Delete Objects

  By default, all assemblies that are created in SQL Server are disabled from executing. You can use the **clr enabled** option of the **sp_configure** system stored procedure to disable or enable the execution of all assemblies that are uploaded in SQL Server. Disabling assembly execution prevents common language runtime (CLR) functions, stored procedures, triggers, aggregates, and user-defined types from executing, and stops those that are currently executing. Disabling assembly execution does not disable the ability to create, alter, or drop assemblies. For more information, see clr enabled Server Configuration Option.

  To disable and enable assembly execution

- sp_configure (Transact-SQL)

Managing Assembly Versions

When an assembly is uploaded to an instance SQL Server, the assembly is stored and managed within the database system catalogs. Any changes made to the definition of the assembly in the Microsoft .NET Framework should be propagated to the assembly that is stored in the database catalog.

When you have to modify an assembly, you must issue an ALTER ASSEMBLY statement to update the assembly in the database. This will update the assembly to the latest copy of .NET Framework modules holding its implementation.

The WITH UNCHECKED DATA clause of the ALTER ASSEMBLY statement instructs SQL Server to refresh even those assemblies upon which persisted data in the database is dependent. Specifically, you must specify WITH UNCHECKED DATA if any of the following exist:

- Persisted computed columns that reference methods in the assembly, either directly, or indirectly, through Transact-SQL functions or methods.

- Columns of a CLR user-defined type that depend on the assembly, and the type implements a **UserDefined** (non-**Native**) serialization format.

Caution

If WITH UNCHECKED DATA is not specified, SQL Server tries to prevent ALTER ASSEMBLY from executing if the new assembly version affects existing data in tables, indexes, or other persistent sites. However, SQL Server does not guarantee that computed columns, indexes, indexed views, or expressions will be consistent with the underlying routines and types when the CLR assembly is updated. Be careful when you execute ALTER ASSEMBLY to make sure that there is no mismatch between the result of an expression and a value that is based on that expression stored in the assembly.

Only members of the **db_owner** and **db_ddlowner** fixed database role can execute run ALTER ASSEMBLY by using the WITH UNCHECKED DATA clause.

SQL Server posts a message to the Windows application event log that the assembly has been modified with unchecked data in the tables. SQL Server then marks any tables that contain data dependent on the assembly as having unchecked data. The **has_unchecked_assembly_data** column of the **sys.tables** catalog view contains the value 1 for tables that contain unchecked data, and 0 for tables without unchecked data.

To resolve the integrity of unchecked data, run DBCC CHECKDB WITH EXTENDED_LOGICAL_CHECKS against each table that has unchecked data. If DBCC CHECKDB WITH EXTENDED_LOGICAL_CHECKS fails, you must either delete the table rows that are not valid or modify the assembly code to address problems, and then issue additional ALTER ASSEMBLY statements.

ALTER ASSEMBLY changes the assembly version. The culture and public key token of the assembly remain the same.SQL Server does not allow registering different versions of an assembly with the same name, culture and public key.

Interactions with Computer-wide Policy for Version Binding

If references to assemblies stored in SQL Server are redirected to specific versions by using publisher policy or computer-wide administrator policy, you must do either of the following:

- Make sure the new version to which this redirection is made is in the database.
- Modify any statements to the external policy files of the computer or publisher policy to make sure that they reference the specific version that is in the database.

  Otherwise, an attempt to load a new assembly version to the instance of SQL Server will fail.

  To update the version of an assembly

- ALTER ASSEMBLY (Transact-SQL)

See Also

Assemblies (Database Engine)
Getting Information About Assemblies

Assemblies - Properties

10/1/2018 • 2 minutes to read • Edit Online

APPLIES TO: ✓ SQL Server ✗ Azure SQL Database ✗ Azure SQL Data Warehouse ✗ Parallel Data Warehouse

Use this page to view or modify properties for the assembly.

Options

Assembly name
Displays the assembly name, which always matches the name of the CLR assembly.

Assembly owner
Type the owner name or schema name or select from the list.

Permission set
Set the security level for the assembly. Three levels of security are provided: **Safe**, **External access**, and **Unsafe** access.

Path to assembly
Type the path to the assembly file.

Browse
Navigate to the assembly you want to add. Click **Browse** if you do not want to type the path to the assembly file.

Additional Properties Grid

Creation Date
Displays the date the assembly was created/registered.

Strong Name
Displays **True** if the assembly has been digitally signed, **False** if it has not been digitally signed.

Version
Displays the version number of the assembly.

See Also

CREATE ASSEMBLY (Transact-SQL)

Assemblies (Database Engine)

10/1/2018 • 2 minutes to read • Edit Online

APPLIES TO: ✓ SQL Server ✗ Azure SQL Database ✗ Azure SQL Data Warehouse ✗ Parallel Data Warehouse

The topics in this section provide information to help you understand, design, and implement assemblies.

Assemblies are DLL files used in an instance of SQL Server to deploy functions, stored procedures, triggers, user-defined aggregates, and user-defined types that are written in one of the managed code languages hosted by the Microsoft .NET Framework common language runtime (CLR), instead of in Transact-SQL.

An assembly in SQL Server is an object that references a managed application module (.dll file) that was created in the .NET Framework common language runtime. An assembly contains class metadata and managed code. Uploading an assembly to an instance of SQL Server is the first step toward creating any of the following database objects:

- CLR functions. For more information, see Create CLR Functions.
- CLR stored procedures. For more information, see CLR Stored Procedures.
- CLR triggers. For more information, see Create CLR Triggers.
- User-defined aggregate functions. For more information, see Create User-defined Aggregates.
- User-defined types. For more information, see Using User-Defined Types.

  Assemblies perform the following functions in SQL Server:

- Contain the managed code that runs the functionality of one or more of the CLR database objects previously listed.
- Contain metadata that includes the version number and culture of the assembly, an optional public key that uniquely identifies the list of classes of the assembly, the methods that are defined in the assembly, and the processor architecture of the assembly.
- Manage the degree to which managed code can access outside resources by regulating code access permissions.
- Contain metadata about dependencies on other assemblies that are referenced by the assembly.

In This Section

TOPIC	DESCRIPTION
Designing Assemblies	Explains what you have to consider before creating an assembly. This includes packaging assemblies, code access permissions, and other restrictions.
Implementing Assemblies	Explains how to create and drop assemblies, how and when to modify assemblies, and how to retrieve metadata about assemblies.
Getting Information About Assemblies	Lists the catalog views and functions that can be queried for metadata about assemblies.

See Also

Common Language Runtime (CLR) Integration Programming Concepts

CLR Integration - Enabling

10/1/2018 • 2 minutes to read • Edit Online

APPLIES TO: ✓ SQL Server ✗ Azure SQL Database ✗ Azure SQL Data Warehouse ✗ Parallel Data Warehouse

The common language runtime (CLR) integration feature is off by default, and must be enabled in order to use objects that are implemented using CLR integration. To enable CLR integration, use the **clr enabled** option of the **sp_configure** stored procedure in SQL Server Management Studio:

```
sp_configure 'show advanced options', 1;
GO
RECONFIGURE;
GO
sp_configure 'clr enabled', 1;
GO
RECONFIGURE;
GO
```

You can disable CLR integration by setting the **clr enabled** option to 0. When you disable CLR integration, SQL Server stops executing all CLR routines and unloads all application domains.

NOTE

To enable CLR integration, you must have ALTER SETTINGS server level permission, which is implicitly held by members of the **sysadmin** and **serveradmin** fixed server roles.

NOTE

Computers configured with large amounts of memory and a large number of processors may fail to load the CLR integration feature of SQL Server when starting the server. To address this issue, start the server by using the **-gmemory_to_reserve** SQL Server service startup option, and specify a memory value large enough. For more information, see Database Engine Service Startup Options.

NOTE

Common language runtime (CLR) execution is not supported under lightweight pooling. Before enabling CLR integration, you must disable lightweight pooling. For more information, see lightweight pooling Server Configuration Option.

See Also

sp_configure (Transact-SQL)
clr enabled Server Configuration Option
RECONFIGURE (Transact-SQL)
GRANT (Transact-SQL)
Server-Level Roles

CLR Integration - Overview

10/1/2018 • 4 minutes to read • Edit Online

APPLIES TO: ✓ SQL Server ✗ Azure SQL Database ✗ Azure SQL Data Warehouse ✗ Parallel Data Warehouse

The common language runtime (CLR) is the heart of the Microsoft .NET Framework and provides the execution environment for all .NET Framework code. Code that runs within the CLR is referred to as managed code. The CLR provides various functions and services required for program execution, including just-in-time (JIT) compilation, allocating and managing memory, enforcing type safety, exception handling, thread management, and security. See the .NET Framework SDK for more information.

With the CLR hosted in Microsoft SQL Server (called CLR integration), you can author stored procedures, triggers, user-defined functions, user-defined types, and user-defined aggregates in managed code. Because managed code compiles to native code prior to execution, you can achieve significant performance increases in some scenarios.

Managed code uses Code Access Security (CAS) to prevent assemblies from performing certain operations. SQL Server uses CAS to help secure the managed code and prevent compromise of the operating system or database server.

Advantages of CLR Integration

Transact-SQL is specifically designed for direct data access and manipulation in the database. While Transact-SQL excels at data access and management, it is not a full-fledged programming language. For example, Transact-SQL does not support arrays, collections, for-each loops, bit shifting, or classes. While some of these constructs can be simulated in Transact-SQL, managed code has integrated support for these constructs. Depending on the scenario, these features can provide a compelling reason to implement certain database functionality in managed code.

Microsoft Visual Basic .NET and Microsoft Visual C# offer object-oriented capabilities such as encapsulation, inheritance, and polymorphism. Related code can now be easily organized into classes and namespaces. When you are working with large amounts of server code, this allows you to more easily organize and maintain your code.

Managed code is better suited than Transact-SQL for calculations and complicated execution logic, and features extensive support for many complex tasks, including string handling and regular expressions. With the functionality found in the .NET Framework Library, you have access to thousands of pre-built classes and routines. These can be easily accessed from any stored procedure, trigger or user defined function. The Base Class Library (BCL) includes classes that provide functionality for string manipulation, advanced math operations, file access, cryptography, and more.

NOTE

While many of these classes are available for use from within CLR code in SQL Server, those that are not appropriate for server-side use (for example, windowing classes), are not available. For more information, see Supported .NET Framework Libraries.

One of the benefits of managed code is type safety, or the assurance that code accesses types only in well-defined, permissible ways. Before managed code is executed, the CLR verifies that the code is safe. For example, the code is checked to ensure that no memory is read that has not previously been written. The CLR can also help ensure that code does not manipulate unmanaged memory.

CLR integration offers the potential for improved performance. For information, see Performance of CLR Integration.

WARNING

CLR uses Code Access Security (CAS) in the .NET Framework, which is no longer supported as a security boundary. A CLR assembly created with `PERMISSION_SET = SAFE` may be able to access external system resources, call unmanaged code, and acquire sysadmin privileges. Beginning with SQL Server 2017 (14.x), an `sp_configure` option called `clr strict security` is introduced to enhance the security of CLR assemblies. `clr strict security` is enabled by default, and treats `SAFE` and `EXTERNAL_ACCESS` assemblies as if they were marked `UNSAFE`. The `clr strict security` option can be disabled for backward compatibility, but this is not recommended. Microsoft recommends that all assemblies be signed by a certificate or asymmetric key with a corresponding login that has been granted `UNSAFE ASSEMBLY` permission in the master database. For more information, see CLR strict security.

Choosing Between Transact-SQL and Managed Code

When writing stored procedures, triggers, and user-defined functions, one decision you must make is whether to use traditional Transact-SQL, or a .NET Framework language such as Visual Basic .NET or Visual C#. Use Transact-SQL when the code will mostly perform data access with little or no procedural logic. Use managed code for CPU-intensive functions and procedures that feature complex logic, or when you want to make use of the BCL of the .NET Framework.

Choosing Between Execution in the Server and Execution in the Client

Another factor in your decision about whether to use Transact-SQL or managed code is where you would like your code to reside, the server computer or the client computer. Both Transact-SQL and managed code can be run on the server. This places code and data close together, and allows you to take advantage of the processing power of the server. On the other hand, you may wish to avoid placing processor intensive tasks on your database server. Most client computers today are very powerful, and you may wish to take advantage of this processing power by placing as much code as possible on the client. Managed code can run on a client computer, while Transact-SQL cannot.

Choosing Between Extended Stored Procedures and Managed Code

Extended stored procedures can be built to perform functionality not possible with Transact-SQL stored procedures. Extended stored procedures can, however, compromise the integrity of the SQL Server process, while managed code that is verified to be type-safe cannot. Further, memory management, scheduling of threads and fibers, and synchronization services are more deeply integrated between the managed code of the CLR and SQL Server. With CLR integration, you have a more secure way than extended stored procedures to write the stored procedures you need to perform tasks not possible in Transact-SQL. For more information about CLR integration and extended stored procedures, see Performance of CLR Integration.

See Also

Installing the .NET Framework
Architecture of CLR Integration
Data Access from CLR Database Objects
Getting Started with CLR Integration

CLR Integration - What's New

10/1/2018 • 2 minutes to read • Edit Online

APPLIES TO: ✓ SQL Server ✗ Azure SQL Database ✗ Azure SQL Data Warehouse ✗ Parallel Data Warehouse

The following are new features in CLR integration in SQL Server 2012 (11.x):

- In version 4 of the CLR, CLR database objects no longer catch corrupted state exceptions. These exceptions are now caught in the CLR integration hosting layer. These exceptions can still be caught by the CLR database components by setting a code attribute (<legacyCorruptedStateExceptionsPolicy> Element). However, this is not recommended because results are not reliable when a corrupted state exception occurs.

- Due to the strict security requirements of SQL Server 2012 (11.x), CLR database components will continue to use the Code Access Security model defined in CLR version 2.0.

- In CLR version 4, a format error in a **System.TimeSpan** value will generate a **System.FormatExceptions**. Prior to version 4 of the CLR, a format error in a **System.TimeSpan** value was ignored. Database applications that rely on the behavior prior to version 4 of the CLR should run with a database compatibility level (**ALTER DATABASE Compatibility Level**) of 100 or lower. For more information, see <TimeSpan_LegacyFormatMode> Element.

- Version 4 of the CLR supports Unicode 5.1. Sort operations involving some accent marks and symbols will be improved. Compatibility problems may occur if your application relies on legacy sorting behavior. To enable legacy sorting, the database compatibility level (**ALTER DATABASE Compatibility Level**) must be set to 100 or lower. To support this, SQL Server 2012 (11.x) will install sort00001000.dll in the .NET Framework 4 directory (C:\Windows\Microsoft.NET\Framework\v4.0.30319). For more information, see <CompatSortNLSVersion> Element.

- The following columns have been added to sys.dm_clr_appdomains: **total_processor_time_ms**, **total_allocated_memory_kb**, and **survived_memory_kb**.

CLR Integration Architecture - Performance

10/1/2018 • 6 minutes to read • Edit Online

APPLIES TO: ✓ SQL Server ✗ Azure SQL Database ✗ Azure SQL Data Warehouse ✗ Parallel Data Warehouse

This topic discusses some of the design choices that enhance the performance of Microsoft SQL Server integration with the Microsoft .NET Framework common language runtime (CLR).

The Compilation Process

During compilation of SQL expressions, when a reference to a managed routine is encountered, a Microsoft intermediate language (MSIL) stub is generated. This stub includes code to marshal the routine parameters from SQL Server to the CLR, invoke the function, and return the result. This "glue" code is based on the type of parameter and on parameter direction (in, out, or reference).

The glue code enables type-specific optimizations and ensures efficient enforcement of SQL Server semantics, such as nullability, constraining facets, by-value, and standard exception handling. By generating code for the exact types of the arguments, you avoid type coercion or wrapper object creation costs (called "boxing") across the invocation boundary.

The generated stub is then compiled to native code and optimized for the particular hardware architecture on which SQL Server executes, using the JIT (just-in-time) compilation services of the CLR. The JIT services are invoked at the method level and allow the SQL Server hosting environment to create a single compilation unit that spans both SQL Server and CLR execution. Once the stub is compiled, the resulting function pointer becomes the run-time implementation of the function. This code generation approach ensures that there are no additional invocation costs related to reflection or metadata access at run time.

Fast Transitions Between SQL Server and CLR

The compilation process yields a function pointer that can be called at run time from native code. In the case of scalar-valued user-defined functions, this function invocation happens on a per-row basis. To minimize the cost of transitioning between SQL Server and the CLR, statements that contain any managed invocation have a startup step to identify the target application domain. This identification step reduces the cost of transitioning for each row.

Performance Considerations

The following summarizes performance considerations specific to CLR integration in SQL Server. More detailed information can be found in "Using CLR Integration in SQL Server 2005" on the MSDN Web site. General information regarding managed code performance can be found in "Improving .NET Application Performance and Scalability" on the MSDN Web site.

User-Defined Functions

CLR functions benefit from a quicker invocation path than that of Transact-SQL user-defined functions. Additionally, managed code has a decisive performance advantage over Transact-SQL in terms of procedural code, computation, and string manipulation. CLR functions that are computing-intensive and that do not perform data access are better written in managed code. Transact-SQL functions do, however, perform data access more efficiently than CLR integration.

User-Defined Aggregates

Managed code can significantly outperform cursor-based aggregation. Managed code generally performs slightly slower than built-in SQL Server aggregate functions. We recommend that if a native built-in aggregate function exists, you should use it. In cases in which the needed aggregation is not natively supported, consider a CLR user-defined aggregate over a cursor-based implementation for performance reasons.

Streaming Table-Valued Functions

Applications often need to return a table as a result of invoking a function. Examples include reading tabular data from a file as part of an import operation, and converting comma-separated-values to a relational representation. Typically, you can accomplish this by materializing and populating the result table before it can be consumed by the caller. The integration of the CLR into SQL Server introduces a new extensibility mechanism called a streaming table-valued function (STVF). Managed STVFs perform better than comparable extended stored procedure implementations.

STVFs are managed functions that return an **IEnumerable** interface. **IEnumerable** has methods to navigate the result set returned by the STVF. When the STVF is invoked, the returned **IEnumerable** is directly connected to the query plan. The query plan calls **IEnumerable** methods when it needs to fetch rows. This iteration model allows results to be consumed immediately after the first row is produced, instead of waiting until the entire table is populated. It also significantly reduces the memory consumed by invoking the function.

Arrays vs. Cursors

When Transact-SQL cursors must traverse data that is more easily expressed as an array, managed code can be used with significant performance gains.

String Data

SQL Server character data, such as **varchar**, can be of the type SqlString or SqlChars in managed functions. SqlString variables create an instance of the entire value into memory. SqlChars variables provide a streaming interface that can be used to achieve better performance and scalability by not creating an instance of the entire value into memory. This becomes particularly important for large object (LOB) data. Additionally, server XML data can be accessed through a streaming interface returned by **SqlXml.CreateReader()**.

CLR vs. Extended Stored Procedures

The Microsoft.SqlServer.Server application programming interfaces (APIs) that allow managed procedures to send result sets back to the client perform better than the Open Data Services (ODS) APIs used by extended stored procedures. Furthermore, the System.Data.SqlServer APIs support data types such as **xml**, **varchar(max)**, **nvarchar(max)**, and **varbinary(max)**, introduced in SQL Server 2005 (9.x), while the ODS APIs have not been extended to support the new data types.

With managed code, SQL Server manages use of resources such as memory, threads, and synchronization. This is because the managed APIs that expose these resources are implemented on top of the SQL Server resource manager. Conversely, SQL Server has no view or control over the resource usage of the extended stored procedure. For example, if an extended stored procedure consumes too much CPU or memory resources, there is no way to detect or control this with SQL Server. With managed code, however, SQL Server can detect that a given thread has not yielded for a long period of time, and then force the task to yield so that other work can be scheduled. Consequently, using managed code provides for better scalability and system resource usage.

Managed code may incur additional overhead necessary to maintain the execution environment and perform security checks. This is the case, for example, when running inside SQL Server and numerous transitions from managed to native code are required (because SQL Server needs to do additional maintenance on thread-specific settings when moving out to native code and back). Consequently, extended stored procedures can significantly outperform managed code running inside SQL Server for cases in which there are frequent transitions between managed and native code.

> **NOTE**
>
> It is recommended that you do not develop new extended stored procedures, because this feature has been deprecated.

Native Serialization for User-Defined Types

User-defined types (UDTs) are designed as an extensibility mechanism for the scalar type system. SQL Server implements a serialization format for UDTs called **Format.Native**. During compilation, the structure of the type is examined to generate MSIL that is customized for that particular type definition.

Native serialization is the default implementation for SQL Server. User-defined serialization invokes a method defined by the type author to do the serialization. **Format.Native** serialization should be used when possible for best performance.

Normalization of Comparable UDTs

Relational operations, such as sorting and comparing UDTs, operate directly on the binary representation of the value. This is accomplished by storing a normalized (binary ordered) representation of the state of the UDT on disk.

Normalization has two benefits: it makes the comparison operation considerably less expensive by avoiding the construction of the type instance and the method invocation overhead; and it creates a binary domain for the UDT, enabling the construction of histograms, indexes, and histograms for values of the type. Consequently, normalized UDTs have a very similar performance profile to the native built-in types for operations that do not involve method invocation.

Scalable Memory Usage

In order for managed garbage collection to perform and scale well in SQL Server, avoid large, single allocation. Allocations greater than 88 kilobytes (KB) in size will be placed on the Large Object Heap, which will cause garbage collection to perform and scale much worse than many smaller allocations. For example, if you need to allocate a large multi-dimensional array, it is better to allocate a jagged (scattered) array.

See Also

CLR User-Defined Types

CLR Integration Architecture - CLR Hosted Environment

10/1/2018 • 12 minutes to read • Edit Online

APPLIES TO: ✓ SQL Server ✗ Azure SQL Database ✗ Azure SQL Data Warehouse ✗ Parallel Data Warehouse

SQL Server integration with the .NET Framework common language runtime (CLR) enables database programmers to use languages such as Visual C#, Visual Basic .NET, and Visual C++. Functions, stored procedures, triggers, data types, and aggregates are among the kinds of business logic that programmers can write with these languages.

The CLR features garbage-collected memory, pre-emptive threading, metadata services (type reflection), code verifiability, and code access security. The CLR uses metadata to locate and load classes, lay out instances in memory, resolve method invocations, generate native code, enforce security, and set run-time context boundaries.

The CLR and SQL Server differ as run-time environments in the way they handle memory, threads, and synchronization. This topic describes the way in which these two run times are integrated so that all system resources are managed uniformly. This topic also covers the way in which CLR code access security (CAS) and SQL Server security are integrated to provide a reliable and secure execution environment for user code.

Basic Concepts of CLR Architecture

In the .NET Framework, a programmer writes in a high-level language that implements a class defining its structure (for example, the fields or properties of the class) and methods. Some of these methods can be static functions. The compilation of the program produces a file called an assembly that contains the compiled code in the Microsoft intermediate language (MSIL), and a manifest that contains all references to dependent assemblies.

NOTE

Assemblies are a vital element in the architecture of the CLR. They are the units of packaging, deployment, and versioning of application code in the .NET Framework. Using assemblies, you can deploy application code inside the database and provide a uniform way to administer, back up, and restore complete database applications.

The assembly manifest contains metadata about the assembly, describing all of the structures, fields, properties, classes, inheritance relationships, functions, and methods defined in the program. The manifest establishes the assembly identity, specifies the files that make up the assembly implementation, specifies the types and resources that make up the assembly, itemizes the compile-time dependencies on other assemblies, and specifies the set of permissions required for the assembly to run properly. This information is used at run time to resolve references, enforce version binding policy, and validate the integrity of loaded assemblies.

The .NET Framework supports custom attributes for annotating classes, properties, functions, and methods with additional information the application may capture in metadata. All .NET Framework compilers consume these annotations without interpretation and store them as assembly metadata. These annotations can be examined in the same way as any other metadata.

Managed code is MSIL executed in the CLR, rather than directly by the operating system. Managed code applications acquire CLR services, such as automatic garbage collection, run-time type checking, and security support. These services help provide uniform platform- and language-independent behavior of managed code applications.

Design Goals of CLR Integration

When user code runs inside the CLR-hosted environment in SQL Server (called CLR integration), the following design goals apply:

Reliability (Safety)

User code should not be allowed to perform operations that compromise the integrity of the Database Engine process, such as popping a message box requesting a user response or exiting the process. User code should not be able to overwrite Database Engine memory buffers or internal data structures.

Scalability

SQL Server and the CLR have different internal models for scheduling and memory management. SQL Server supports a cooperative, non-preemptive threading model in which the threads voluntarily yield execution periodically, or when they are waiting on locks or I/O. The CLR supports a preemptive threading model. If user code running inside SQL Server can directly call the operating system threading primitives, then it does not integrate well into the SQL Server task scheduler and can degrade the scalability of the system. The CLR does not distinguish between virtual and physical memory, but SQL Server directly manages physical memory and is required to use physical memory within a configurable limit.

The different models for threading, scheduling, and memory management present an integration challenge for a relational database management system (RDBMS) that scales to support thousands of concurrent user sessions. The architecture should ensure that the scalability of the system is not compromised by user code calling application programming interfaces (APIs) for threading, memory, and synchronization primitives directly.

Security

User code running in the database must follow SQL Server authentication and authorization rules when accessing database objects such as tables and columns. In addition, database administrators should be able to control access to operating system resources, such as files and network access, from user code running in the database. This becomes important as managed programming languages (unlike non-managed languages such as Transact-SQL) provide APIs to access such resources. The system must provide a secure way for user code to access machine resources outside the Database Engine process. For more information, see CLR Integration Security.

Performance

Managed user code running in the Database Engine should have computational performance comparable to the same code run outside the server. Database access from managed user code is not as fast as native Transact-SQL. For more information, see Performance of CLR Integration.

CLR Services

The CLR provides a number of services to help achieve the design goals of CLR integration with SQL Server.

Type safety verification

Type-safe code is code that accesses memory structures only in well-defined ways. For example, given a valid object reference, type-safe code can access memory at fixed offsets corresponding to actual field members. However, if the code accesses memory at arbitrary offsets inside or outside the range of memory that belongs to the object, then it is not type-safe. When assemblies are loaded in the CLR, prior to the MSIL being compiled using just-in-time (JIT) compilation, the runtime performs a verification phase that examines code to determine its type-safety. Code that successfully passes this verification is called verifiably type-safe code.

Application domains

The CLR supports the notion of application domains as execution zones within a host process where managed code assemblies can be loaded and executed. The application domain boundary provides isolation between assemblies. The assemblies are isolated in terms of visibility of static variables and data members and the ability to call code dynamically. Application domains are also the mechanism for loading and unloading code. Code can be unloaded from memory only by unloading the application domain. For more information, see Application Domains and CLR Integration Security.

Code Access Security (CAS)

The CLR security system provides a way to control what kinds of operations managed code can perform by assigning permissions to code. Code access permissions are assigned based on code identity (for example, the signature of the assembly or the origin of the code).

The CLR provides for a computer-wide policy that can be set by the computer administrator. This policy defines the permission grants for any managed code running on the machine. In addition, there is a host-level security policy that can be used by hosts such as SQL Server to specify additional restrictions on managed code.

If a managed API in the .NET Framework exposes operations on resources that are protected by a code access permission, the API will demand that permission before accessing the resource. This demand causes the CLR security system to trigger a comprehensive check of every unit of code (assembly) in the call stack. Only if the entire call chain has permission will access to the resource be granted.

Note that the ability to generate managed code dynamically, using the Reflection.Emit API, is not supported inside the CLR-hosted environment in SQL Server. Such code would not have the CAS permissions to run and would therefore fail at run time. For more information, see CLR Integration Code Access Security.

Host Protection Attributes (HPAs)

The CLR provides a mechanism to annotate managed APIs that are part of the .NET Framework with certain attributes that may be of interest to a host of the CLR. Examples of such attributes include:

- SharedState, which indicates whether the API exposes the ability to create or manage shared state (for example, static class fields).

- Synchronization, which indicates whether the API exposes the ability to perform synchronization between threads.

- ExternalProcessMgmt, which indicates whether the API exposes a way to control the host process.

  Given these attributes, the host can specify a list of HPAs, such as the SharedState attribute, that should be disallowed in the hosted environment. In this case, the CLR denies attempts by user code to call APIs that are annotated by the HPAs in the prohibited list. For more information, see Host Protection Attributes and CLR Integration Programming.

How SQL Server and the CLR Work Together

This section discusses how SQL Server integrates the threading, scheduling, synchronization, and memory management models of SQL Server and the CLR. In particular, this section examines the integration in light of scalability, reliability, and security goals. SQL Server essentially acts as the operating system for the CLR when it is hosted inside SQL Server. The CLR calls low-level routines implemented by SQL Server for threading, scheduling, synchronization, and memory management. These are the same primitives that the rest of the SQL Server engine uses. This approach provides several scalability, reliability, and security benefits.

Scalability: Common threading, scheduling, and synchronization

CLR calls SQL Server APIs for creating threads, both for running user code and for its own internal use. In order to synchronize between multiple threads, the CLR calls SQL Server synchronization objects. This allows the SQL Server scheduler to schedule other tasks when a thread is waiting on a synchronization object. For example, when the CLR initiates garbage collection, all of its threads wait for garbage collection to finish. Because the CLR threads and the synchronization objects they are waiting on are known to the SQL Server scheduler, SQL Server can schedule threads that are running other database tasks not involving the CLR. This also enables SQL Server to detect deadlocks that involve locks taken by CLR synchronization objects and employ traditional techniques for deadlock removal.

Managed code runs preemptively in SQL Server. The SQL Server scheduler has the ability to detect and stop threads that have not yielded for a significant amount of time. The ability to hook CLR threads to SQL Server threads implies that the SQL Server scheduler can identify "runaway" threads in the CLR and manage their priority. Such runaway threads are suspended and put back in the queue. Threads that are repeatedly identified as runaway threads are not allowed to run for a given period of time so that other executing workers can run.

NOTE

Long-running managed code that accesses data or allocates enough memory to trigger garbage collection will yield automatically. Long-running managed code that does not access data or allocate enough managed memory to trigger garbage collection should explicitly yield by calling the System.Thread.Sleep() function of the .NET Framework.

Scalability: Common memory management

The CLR calls SQL Server primitives for allocating and de-allocating its memory. Because the memory used by the CLR is accounted for in the total memory usage of the system, SQL Server can stay within its configured memory limits and ensure the CLR and SQL Server are not competing with each other for memory. SQL Server can also reject CLR memory requests when system memory is constrained, and ask CLR to reduce its memory use when other tasks need memory.

Reliability: Application domains and unrecoverable exceptions

When managed code in the .NET Framework APIs encounters critical exceptions, such as out-of-memory or stack overflow, it is not always possible to recover from such failures and ensure consistent and correct semantics for their implementation. These APIs raise a thread abort exception in response to these failures.

When hosted in SQL Server, such thread aborts are handled as follows: the CLR detects any shared state in the application domain in which the thread abort occurs. The CLR does this by checking for the presence of synchronization objects. If there is shared state in the application domain, then the application domain itself is unloaded. The unloading of the application domain stops database transactions that are currently running in that application domain. Because the presence of shared state can widen the impact of such critical exceptions to user sessions other than the one triggering the exception, SQL Server and the CLR have taken steps to reduce the likelihood of shared state. For more information, see the .NET Framework documentation.

Security: Permission sets

SQL Server allows users to specify the reliability and security requirements for code deployed into the database. When assemblies are uploaded into the database, the author of the assembly can specify one of three permission sets for that assembly: SAFE, EXTERNAL_ACCESS and UNSAFE.

Permission set	SAFE	EXTERNAL_ACCESS	UNSAFE
Code Access Security	Execute only	Execute + access to external resources	Unrestricted
Programming model restrictions	Yes	Yes	No restrictions
Verifiability requirement	Yes	Yes	No
Ability to call native code	No	No	Yes

SAFE is the most reliable and secure mode with associated restrictions in terms of the allowed programming model. SAFE assemblies are given enough permission to run, perform computations, and have access to the local database. SAFE assemblies need to be verifiably type safe and are not allowed to call unmanaged code.

UNSAFE is for highly trusted code that can only be created by database administrators. This trusted code has no code access security restrictions, and it can call unmanaged (native) code.

EXTERNAL_ACCESS provides an intermediate security option, allowing code to access resources external to the database but still having the reliability guarantees of SAFE.

SQL Server uses the host-level CAS policy layer to set up a host policy that grants one of the three sets of permissions based on the permission set stored in SQL Server catalogs. Managed code running inside the database always gets one of these code access permission sets.

Programming Model Restrictions

The programming model for managed code in SQL Server involves writing functions, procedures, and types which typically do not require the use of state held across multiple invocations or the sharing of state across multiple user sessions. Further, as described earlier, the presence of shared state can cause critical exceptions that impact the scalability and the reliability of the application.

Given these considerations, we discourage the use of static variables and static data members of classes used in SQL Server. For SAFE and EXTERNAL_ACCESS assemblies, SQL Server examines the metadata of the assembly at CREATE ASSEMBLY time and fails the creation of such assemblies if it finds the use of static data members and variables.

SQL Server also disallows calls to .NET Framework APIs that are annotated with the **SharedState**, **Synchronization** and **ExternalProcessMgmt** host protection attributes. This prevents SAFE and EXTERNAL_ACCESS assemblies from calling any APIs that enable sharing state, performing synchronization, and affecting the integrity of the SQL Server process. For more information, see CLR Integration Programming Model Restrictions.

See Also

CLR Integration Security
Performance of CLR Integration

Common Language Runtime Integration Overview

10/1/2018 • 2 minutes to read • Edit Online

APPLIES TO: ✓ SQL Server ✗ Azure SQL Database ✗ Azure SQL Data Warehouse ✗ Parallel Data Warehouse

Microsoft SQL Server now features the integration of the common language runtime (CLR) component of the .NET Framework for Microsoft Windows. The CLR supplies managed code with services such as cross-language integration, code access security, object lifetime management, and debugging and profiling support. For SQL Server users and application developers, CLR integration means that you can now write stored procedures, triggers, user-defined types, user-defined functions (scalar and table-valued), and user-defined aggregate functions using any .NET Framework language, including Microsoft Visual Basic .NET and Microsoft Visual C#. SQL Server includes the .NET Framework version 4 pre-installed.

WARNING

CLR uses Code Access Security (CAS) in the .NET Framework, which is no longer supported as a security boundary. A CLR assembly created with `PERMISSION_SET = SAFE` may be able to access external system resources, call unmanaged code, and acquire sysadmin privileges. Beginning with SQL Server 2017 (14.x), an `sp_configure` option called `clr strict security` is introduced to enhance the security of CLR assemblies. `clr strict security` is enabled by default, and treats `SAFE` and `EXTERNAL_ACCESS` assemblies as if they were marked `UNSAFE`. The `clr strict security` option can be disabled for backward compatibility, but this is not recommended. Microsoft recommends that all assemblies be signed by a certificate or asymmetric key with a corresponding login that has been granted `UNSAFE ASSEMBLY` permission in the master database. For more information, see CLR strict security. SQL Server administrators can also add assemblies to a list of assemblies, which the Database Engine should trust. For more information, see sys.sp_add_trusted_assembly.

Among the major benefits of this integration are:

- **A better programming model.** The .NET Framework languages are in many respects richer than Transact-SQL, offering constructs and capabilities previously not available to SQL Server developers. Developers may also leverage the power of the .NET Framework Library, which provides an extensive set of classes that can be used to quickly and efficiently solve programming problems.
- **Improved safety and security.** Managed code runs in a common language run-time environment, hosted by the Database Engine. SQL Server leverages this to provide a safer and more secure alternative to the extended stored procedures available in earlier versions of SQL Server.
- **Ability to define data types and aggregate functions.** User defined types and user defined aggregates are two new managed database objects which expand the storage and querying capabilities of SQL Server.
- **Streamlined development through a standardized environment.** Database development is integrated into future releases of the Microsoft Visual Studio .NET development environment. Developers use the same tools for developing and debugging database objects and scripts as they use to write middle-tier or client-tier .NET Framework components and services.
- **Potential for improved performance and scalability.** In many situations, the .NET Framework language compilation and execution models deliver improved performance over Transact-SQL.

  This following table lists the topics in this section.

  Overview of CLR Integration
  Describes the kinds of objects that can be built using CLR integration, and reviews the requirements for building database objects using CLR integration.

What's New in CLR Integration
Describes the new features in this release.

Architecture of CLR Integration
Describes the design goals of CLR integration.

Enabling CLR Integration
Describes how to enable CLR integration.

See Also

Installing the .NET Framework
Performance of CLR Integration

Debugging CLR Database Objects

10/1/2018 • 4 minutes to read • Edit Online

APPLIES TO: ✓ SQL Server ✗ Azure SQL Database ✗ Azure SQL Data Warehouse ✗ Parallel Data Warehouse

SQL Server provides support for debugging Transact-SQL and common language runtime (CLR) objects in the database. The key aspects of debugging in SQL Server are the ease of setup and use, and the integration of the SQL Server debugger with the Microsoft Visual Studio debugger. Furthermore, debugging works across languages. Users can step seamlessly into CLR objects from Transact-SQL, and vice versa. The Transact-SQL debugger in SQL Server Management Studio cannot be used to debug managed database objects, but you can debug the objects by using the debuggers in Visual Studio. Managed database object debugging in Visual Studio supports all common debugging features, such as "step into" and "step over" statements within routines executing on the server. Debuggers can set breakpoints, inspect the call stack, inspect variables, and modify variable values while debugging. Note that Visual Studio .NET 2003 cannot be used for CLR integration programming or debugging. SQL Server includes the .NET Framework pre-installed, and Visual Studio .NET 2003 cannot use the .NET Framework 2.0 assemblies.

For more information about debugging managed code using Visual Studio, see the "Debugging Managed Code" topic in the Visual Studio documentation.

Debugging Permissions and Restrictions

Debugging is a highly privileged operation, and therefore only members of the **sysadmin** fixed server role are allowed to do so in SQL Server.

The following restrictions apply while debugging:

- Debugging CLR routines is restricted to one debugger instance at a time. This limitation applies because all CLR code execution freezes when a break point is hit and execution does not continue until the debugger advances from the break point. However, you can continue debugging Transact-SQL in other connections. Although Transact-SQL debugging does not freeze other executions on the server, it could cause other connections to wait by holding a lock.
- Existing connections cannot be debugged, only new connections, as SQL Server requires information about the client and debugger environment before the connection can be made.

  Due to the above restrictions, we recommend that Transact-SQL and CLR code be debugged on a test server, and not on a production server.

Overview of Debugging Managed Database Objects

Debugging in SQL Server follows a per-connection model. A debugger can detect and debug activities only to the client connection to which it is attached. Because the functionality of the debugger is not limited by the type of connection, both tabular data stream (TDS) and HTTP connections can be debugged. However, SQL Server does not allow debugging existing connections. Debugging supports all common debugging features within routines executing on the server. The interaction between a debugger and SQL Server happens through distributed Component Object Model (COM).

For more information and scenarios about debugging managed stored procedures, functions, triggers, user-defined types, and aggregates, see the "SQL Server CLR Integration Database Debugging" topic in the Visual Studio documentation.

The TCP/IP network protocol must be enabled on the SQL Server instance in order to use Visual Studio for remote development, debugging, and development. For more information about enabling TCP/IP protocol on the server, see Configure Client Protocols.

To debug a managed database object

1. Open Microsoft Visual Studio, create a new SQL Server project, and establish a connection to a database on an instance of SQL Server.

2. Create a new type. In **Solution Explorer**, right-click the project, select **Add** and **New Item...** From the **Add New Item** window, select **Stored Procedure**, **User-Defined Function**, **User-Defined Type**, **Trigger**, **Aggregate**, or **Class**. Specify a name for the source file of the new type and click **Add**.

3. Add code for the new type to the text editor. For sample code for an example stored procedure, see the section later in this topic.

4. Add a script that tests the type. In **Solution Explorer**, expand the **TestScripts** directory double-click **Test.sql** to open the default test script source file. Add the test script, one that invokes the code to be debugged, to the text editor. See below for a sample script.

5. Place one or more breakpoints in the source code. Right-click on a line of code in the text editor, within the function or routine you want to debug, and select **Breakpoint** and **Insert Breakpoint**. The breakpoint is added, highlighting the line of code in red.

6. In the **Debug** menu, select **Start Debugging** to compile, deploy, and test the project. The test script in Test.sql will be run and the managed database object will be invoked.

7. When the yellow arrow designating the instruction pointer appears at the breakpoint code execution pauses and you can begin debugging your managed database object. You can **Step Over** from the **Debug** menu to advance the instruction pointer to the next line of code. The **Locals** window is used to observe the state of the objects currently highlighted by the instruction pointer. Variables can be added to the **Watch** window. The state of watched variables can be observed throughout the entire debugging session, not only when the variable is in the line of code currently highlighted by instruction pointer. Select Continue from the Debug menu to advance the instruction pointer to the next breakpoint or to complete execution of the routine if there are no more breakpoints.

Example

The following example returns the SQL Server version to the caller.

C#

```
using System;
using System.Data;
using System.Data.SqlTypes;
using System.Data.SqlClient;
using Microsoft.SqlServer.Server;

public class StoredProcedures
{
   [Microsoft.SqlServer.Server.SqlProcedure]
   public static void GetVersion()
   {
   using(SqlConnection connection = new SqlConnection("context connection=true"))
   {
      connection.Open();
      SqlCommand command = new SqlCommand("select @@version",
                                          connection);
      SqlContext.Pipe.ExecuteAndSend(command);
      }
   }
}
```

Visual Basic

```
Imports System
Imports System.Data
Imports System.Data.Sql
Imports System.Data.SqlTypes
Imports Microsoft.SqlServer.Server
Imports System.Data.SqlClient

Partial Public Class StoredProcedures
    <Microsoft.SqlServer.Server.SqlProcedure> _
    Public Shared Sub GetVersion()
        Using connection As New SqlConnection("context connection=true")
            connection.Open()
            Dim command As New SqlCommand("SELECT @@VERSION", connection)
            SqlContext.Pipe.ExecuteAndSend(command)
        End Using
    End Sub
End Class
```

The following is a test script that invokes the GetVersion stored procedure defined above.

```
EXEC GetVersion
```

See Also

Common Language Runtime (CLR) Integration Programming Concepts

Deploying CLR Database Objects

10/1/2018 • 5 minutes to read • Edit Online

APPLIES TO: ✓ SQL Server ✗ Azure SQL Database ✗ Azure SQL Data Warehouse ✗ Parallel Data Warehouse

Deployment is the process by which you distribute a finished application or module to be installed and run on another computer. Using Microsoft Visual Studio, you can develop common language runtime (CLR) database objects and deploy them to a test server. Alternatively, the managed database objects can also be compiled with the Microsoft .NET Framework redistribution files, instead of Visual Studio. Once compiled, the assemblies containing the CLR database objects can then be deployed to a test server using Visual Studio or Transact-SQL statements. Note that Visual Studio .NET 2003 cannot be used for CLR integration programming or deployment. SQL Server includes the .NET Framework pre-installed, and Visual Studio .NET 2003 cannot use the .NET Framework 2.0 assemblies.

Once the CLR methods have been tested and verified on the test server, they can be distributed to production servers using a deployment script. The deployment script can be generated manually, or by using SQL Server Management Studio (see the procedure later in this topic).

The CLR integration feature is turned off by default in SQL Server and must be enabled in order to use CLR assemblies. For more information, see Enabling CLR Integration.

Deploying the Assembly to the Test Server

Using Visual Studio, you can develop CLR functions, procedures, triggers, user-defined types (UDTs), or user-defined aggregates (UDAs), and deploy them to a test server. These managed database objects can also be compiled with the command line compilers, such as csc.exe and vbc.exe, included with the .NET Framework redistribution files. The Visual Studio Integrated Development Environment is not required to develop managed database objects for SQL Server.

Make sure that all compiler errors and warnings are resolved. The assemblies containing the CLR routines can then be registered in a SQL Server database using Visual Studio or Transact-SQL statements.

NOTE

The TCP/IP network protocol must be enabled on the SQL Server instance in order to use Microsoft Visual Studio for remote development, debugging, and development. For more information about enabling TCP/IP protocol on the server, see Configure Client Protocols.

To deploy the assembly using Visual Studio

1. Build the project by selecting **Build** <project name> from the **Build** menu.

2. Resolve all build errors and warnings before deploying the assembly to the test server.

3. Select **Deploy** from the **Build** menu. The assembly will then be registered in the SQL Server instance and database specified when the SQL Server project was first created in Visual Studio.

To deploy the assembly using Transact-SQL

1. Compile the assembly from the source file using the command line compilers included with the .NET Framework.

2. For Microsoft Visual C# source files:

3.
   ```
   csc /target:library C:\helloworld.cs
   ```

4. For Microsoft Visual Basic source files:

   ```
   vbc /target:library C:\helloworld.vb
   ```

   These commands launch the Visual C# or Visual Basic compiler using the **/target** option to specify building a library DLL.

5. Resolve all build errors and warnings before deploying the assembly to the test server.

6. Open SQL Server Management Studio on the test server. Create a new query, connected to a suitable test database (such as AdventureWorks).

7. Create the assembly in the server by adding the following Transact-SQL to the query.

   ```
   CREATE ASSEMBLY HelloWorld from 'c:\helloworld.dll' WITH PERMISSION_SET = SAFE;
   ```

8. The procedure, function, aggregate, user-defined type, or trigger must then be created in the instance of SQL Server. If the **HelloWorld** assembly contains a method named **HelloWorld** in the **Procedures** class, the following Transact-SQL can be added to the query to create a procedure called **hello** in SQL Server.

   ```
   CREATE PROCEDURE hello
   ```

   ```
   AS
   ```

   ```
   EXTERNAL NAME HelloWorld.Procedures.HelloWorld
   ```

   For more information about creating the different types of managed database objects in SQL Server, see CLR User-Defined Functions, CLR User-Defined Aggregates, CLR User-Defined Types, CLR Stored Procedures, and CLR Triggers.

Deploying the Assembly to Production Servers

Once the CLR database objects have been tested and verified on the test server, they can be distributed to production servers. For more information about debugging managed database objects, see Debugging CLR Database Objects.

The deployment of managed database objects is similar to that of regular database objects (tables, Transact-SQL routines, and so on). The assemblies containing the CLR database objects can be deployed to other servers using a deployment script. The deployment script can be built by using the "Generate Scripts" functionality of Management Studio. The deployment script can also be built manually, or built using "Generate Scripts" and manually altered. Once the deployment script has been built, it can be run on other instances of SQL Server to deploy the managed database objects.

To generate a deployment script using generate scripts

1. Open Management Studio and connect to the SQL Server instance where the managed assembly or database object to be deployed is registered.

2. In the **Object Explorer**, expand the **<server name>** and **Databases** trees. Right-click the database where the managed database object is registered, select **Tasks**, and then select **Generate Scripts**. The Script Wizard opens.

3. Select the database from the list box and click **Next**.

4. In the **Choose Script Options** pane, click **Next**, or change the options and then click **Next**.

5. In the **Choose Object Types** pane, choose the type of database object to be deployed. Click **Next**.

6. For every object type selected in the **Choose Object Types** pane, a **Choose <type>** pane is presented. In

this pane, you can choose from all the instances of that database object type registered in the specified database. Select one or more objects and click **Next**.

7. The **Output Options** pane comes up when all of the desired database object types have been selected. Select **Script to file** and specify a file path for the script. Select **Next**. Review your selections and click **Finish**. The deployment script is saved to the specified file path.

Post Deployment Scripts

You can run a post deployment script.

To add a post deployment script, add a file called postdeployscript.sql in your Visual Studio project directory. For example, right click your project in **Solution Explorer** and select **Add Existing Item**. Add the file in the root of the project, rather than in the Test Scripts folder.

When you click deploy, Visual Studio will run this script after the deployment of your project.

See Also

Common Language Runtime (CLR) Integration Programming Concepts

Monitoring and Troubleshooting Managed Database Objects

10/1/2018 • 3 minutes to read • Edit Online

APPLIES TO: ✓ SQL Server ✗ Azure SQL Database ✗ Azure SQL Data Warehouse ✗ Parallel Data Warehouse

This topic provides information about the tools that can be used to monitor and troubleshoot managed database objects and assemblies running in SQL Server.

Profiler Trace Events

SQL Server provides SQL Trace and event notifications to monitor events that occur in the Database Engine. By recording specified events, SQL Trace helps you troubleshoot performance, audit database activity, gather sample data for a test environment, debug Transact-SQL statements and stored procedures, and gather data for performance analysis tools. For more information, see SQL Trace and Extended Events.

EVENT	DESCRIPTION
Assembly Load Event Class	Used to monitor assembly load requests (success and failures).
SQL:BatchStarting Event Class, SQL:BatchCompleted Event Class	Provides information about Transact-SQL batches that have started or completed.
SP:Starting Event Class, SP:Completed Event Class	Used to monitor the execution of Transact-SQL stored procedures.
SQL:StmtStarting Event Class, SQL:StmtCompleted Event Class	Used to monitor the execution of CLR and Transact-SQL routines.

Performance Counters

SQL Server provides objects and counters that can be used by System Monitor to monitor activity in computers running an instance of SQL Server. An object is any SQL Server resource, such as a SQL Server lock or a Windows process. Each object contains one or more counters that determine various aspects of the objects to monitor. For more information, see Use SQL Server Objects.

OBJECT	DESCRIPTION
SQL Server, CLR Object	Total time spent in CLR execution.

Windows System Monitor (PERFMON.EXE) Counters

The Windows System Monitor (PERFMON.EXE) tool has several performance counters that can be used to monitor CLR integration applications. The .NET CLR performance counters can be filtered by the "sqlservr" process name to track CLR integration applications that are currently running.

PERFORMANCE OBJECT	DESCRIPTION
SqlServer:CLR	Provides CPU statistics for the server.
.NET CLR Exceptions	Tracks the number of exceptions per second.
.NET CLR Loading	Provides information about the AppDomains and assemblies loaded in the server.
.NET CLR Memory	Provides information about CLR memory usage. This object can be used to flag alerts if memory usage gets too large.
.NET Data Provider for SQL Server	Tracks the number of connects and disconnects per second. This object can be used for monitoring the level of database activity.

Catalog Views

Catalog views return information that is used by the SQL Server Database Engine. We recommend that you use catalog views because they are the most general interface to the catalog metadata and provide the most efficient way to obtain, transform, and present customized forms of this information. All user-available catalog metadata is exposed through catalog views. For more information, see Catalog Views (Transact-SQL).

CATALOG VIEW	DESCRIPTION
sys.assemblies (Transact-SQL)	Returns information about the assemblies registered in a database.
sys.assembly_references (Transact-SQL)	Identifies assemblies that reference other assemblies.
sys.assembly_modules (Transact-SQL)	Returns information about each function, stored procedure, and trigger defined in an assembly.
sys.assembly_files (Transact-SQL)	Returns information about the assembly files registered in the database.
sys.assembly_types (Transact-SQL)	Identifies the user-defined types (UDTs) defined by an assembly.
sys.module_assembly_usages (Transact-SQL)	Identifies the assemblies that CLR modules are defined in.
sys.parameter_type_usages (Transact-SQL)	Returns information about parameters that are user-defined types.
sys.server_assembly_modules (Transact-SQL)	Identifies the assembly that a CLR trigger is defined in.
sys.server_triggers (Transact-SQL)	Identifies the server-level DDL triggers on a server, including CLR triggers.
sys.type_assembly_usages (Transact-SQL)	Identifies the assemblies that user-defined types are defined in.
sys.types (Transact-SQL)	Returns the system and user-defined types registered in the database.

Dynamic Management Views

Dynamic management views and functions return server state information that can be used to monitor the health of a server instance, diagnose problems, and tune performance. For more information, see Dynamic Management Views and Functions (Transact-SQL).

DMV	DESCRIPTION
sys.dm_clr_appdomains (Transact-SQL)	Provides information about each application domain in the server.
sys.dm_clr_loaded_assemblies (Transact-SQL)	Identifies each managed assembly registered on the server.
sys.dm_clr_properties (Transact-SQL)	Returns information about the hosted CLR.
sys.dm_clr_tasks (Transact-SQL)	Identifies all the CLR tasks that are currently running.
sys.dm_exec_cached_plans (Transact-SQL)	Returns information about the query execution plans that are cached by SQL Server for faster query execution.
sys.dm_exec_query_stats (Transact-SQL)	Returns aggregate performance statistics for cached query plans.
sys.dm_exec_requests (Transact-SQL)	Returns information about each request that is executing within SQL Server.
sys.dm_os_memory_clerks (Transact-SQL)	Returns all the memory clerks currently active in the SQL Server instance, including CLR memory clerks.

See Also

Common Language Runtime (CLR) Integration Programming Concepts