Table of Contents

Programming with Application Domains and Assemblies

5/2/2018 • 1 min to read • Edit Online

Hosts such as Microsoft Internet Explorer, ASP.NET, and the Windows shell load the common language runtime into a process, create an application domain in that process, and then load and execute user code in that application domain when running a .NET Framework application. In most cases, you do not have to worry about creating application domains and loading assemblies into them because the runtime host performs those tasks.

However, if you are creating an application that will host the common language runtime, creating tools or code you want to unload programmatically, or creating pluggable components that can be unloaded and reloaded on the fly, you will be creating your own application domains. Even if you are not creating a runtime host, this section provides important information on how to work with application domains and assemblies loaded in these application domains.

In This Section

Application Domains and Assemblies How-to Topics
Provides links to all How-to topics found in the conceptual documentation for programming with application domains and assemblies.

Using Application Domains
Provides examples of creating, configuring, and using application domains.

Programming with Assemblies
Describes how to create, sign, and set attributes on assemblies.

Related Sections

Emitting Dynamic Methods and Assemblies
Describes how to create dynamic assemblies.

Assemblies in the Common Language Runtime
Provides a conceptual overview of assemblies.

Application Domains
Provides a conceptual overview of application domains.

Reflection Overview
Describes how to use the **Reflection** class to obtain information about an assembly.

Assemblies in the Common Language Runtime

5/2/2018 • 3 min to read • Edit Online

Assemblies are the building blocks of .NET Framework applications; they form the fundamental unit of deployment, version control, reuse, activation scoping, and security permissions. An assembly is a collection of types and resources that are built to work together and form a logical unit of functionality. An assembly provides the common language runtime with the information it needs to be aware of type implementations. To the runtime, a type does not exist outside the context of an assembly.

An assembly performs the following functions:

- It contains code that the common language runtime executes. Microsoft intermediate language (MSIL) code in a portable executable (PE) file will not be executed if it does not have an associated assembly manifest. Note that each assembly can have only one entry point (that is, `DllMain`, `WinMain`, or `Main`).
- It forms a security boundary. An assembly is the unit at which permissions are requested and granted. For more information about security boundaries as they apply to assemblies, see Assembly Security Considerations.
- It forms a type boundary. Every type's identity includes the name of the assembly in which it resides. A type called `MyType` that is loaded in the scope of one assembly is not the same as a type called `MyType` that is loaded in the scope of another assembly.
- It forms a reference scope boundary. The assembly's manifest contains assembly metadata that is used for resolving types and satisfying resource requests. It specifies the types and resources that are exposed outside the assembly. The manifest also enumerates other assemblies on which it depends.
- It forms a version boundary. The assembly is the smallest versionable unit in the common language runtime; all types and resources in the same assembly are versioned as a unit. The assembly's manifest describes the version dependencies you specify for any dependent assemblies. For more information about versioning, see Assembly Versioning.
- It forms a deployment unit. When an application starts, only the assemblies that the application initially calls must be present. Other assemblies, such as localization resources or assemblies containing utility classes, can be retrieved on demand. This allows applications to be kept simple and thin when first downloaded. For more information about deploying assemblies, see Deploying Applications.
- It is the unit at which side-by-side execution is supported. For more information about running multiple versions of an assembly, see Assemblies and Side-by-Side Execution.

Assemblies can be static or dynamic. Static assemblies can include .NET Framework types (interfaces and classes), as well as resources for the assembly (bitmaps, JPEG files, resource files, and so on). Static assemblies are stored on disk in portable executable (PE) files. You can also use the .NET Framework to create dynamic assemblies, which are run directly from memory and are not saved to disk before execution. You can save dynamic assemblies to disk after they have executed.

There are several ways to create assemblies. You can use development tools, such as Visual Studio, that you have used in the past to create .dll or .exe files. You can use tools provided in the Windows Software Development Kit (SDK) to create assemblies with modules created in other development environments. You can also use common language runtime APIs, such as System.Reflection.Emit, to create dynamic assemblies.

Related Topics

TITLE	DESCRIPTION
Assembly Contents	Describes the elements that make up an assembly.
Assembly Manifest	Describes the data in the assembly manifest, and how it is stored in assemblies.
Global Assembly Cache	Describes the global assembly cache and how it is used with assemblies.
Strong-Named Assemblies	Describes the characteristics of strong-named assemblies.
Assembly Security Considerations	Discusses how security works with assemblies.
Assembly Versioning	Provides an overview of the .NET Framework versioning policy.
Assembly Placement	Discusses where to locate assemblies.
Assemblies and Side-by-Side Execution	Provides an overview of using multiple versions of the runtime or of an assembly simultaneously.
Programming with Assemblies	Describes how to create, sign, and set attributes on assemblies.
Emitting Dynamic Methods and Assemblies	Describes how to create dynamic assemblies.
How the Runtime Locates Assemblies	Describes how the .NET Framework resolves assembly references at run time.

Reference

System.Reflection.Assembly

Assembly Contents

5/2/2018 • 1 min to read • Edit Online

In general, a static assembly can consist of four elements:

- The assembly manifest, which contains assembly metadata.
- Type metadata.
- Microsoft intermediate language (MSIL) code that implements the types.
- A set of resources.

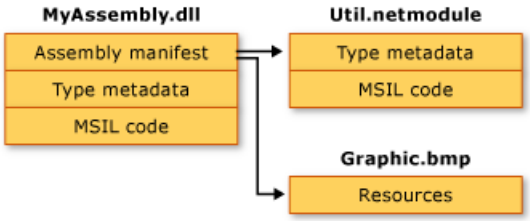
Only the assembly manifest is required, but either types or resources are needed to give the assembly any meaningful functionality.

There are several ways to group these elements in an assembly. You can group all elements in a single physical file, which is shown in the following illustration.

Single-file assembly

Alternatively, the elements of an assembly can be contained in several files. These files can be modules of compiled code (.netmodule), resources (such as .bmp or .jpg files), or other files required by the application. Create a multifile assembly when you want to combine modules written in different languages and to optimize downloading an application by putting seldom used types in a module that is downloaded only when needed.

In the following illustration, the developer of a hypothetical application has chosen to separate some utility code into a different module and to keep a large resource file (in this case a .bmp image) in its original file. The .NET Framework downloads a file only when it is referenced; keeping infrequently referenced code in a separate file from the application optimizes code download.



Multifile assembly

> **NOTE**
> The files that make up a multifile assembly are not physically linked by the file system. Rather, they are linked through the assembly manifest and the common language runtime manages them as a unit.

In this illustration, all three files belong to an assembly, as described in the assembly manifest contained in MyAssembly.dll. To the file system, they are three separate files. Note that the file Util.netmodule was compiled as a module because it contains no assembly information. When the assembly was created, the assembly manifest was added to MyAssembly.dll, indicating its relationship with Util.netmodule and Graphic.bmp.

As you currently design your source code, you make explicit decisions about how to partition the functionality of your application into one or more files. When designing .NET Framework code, you will make similar decisions about how to partition the functionality into one or more assemblies.

See Also

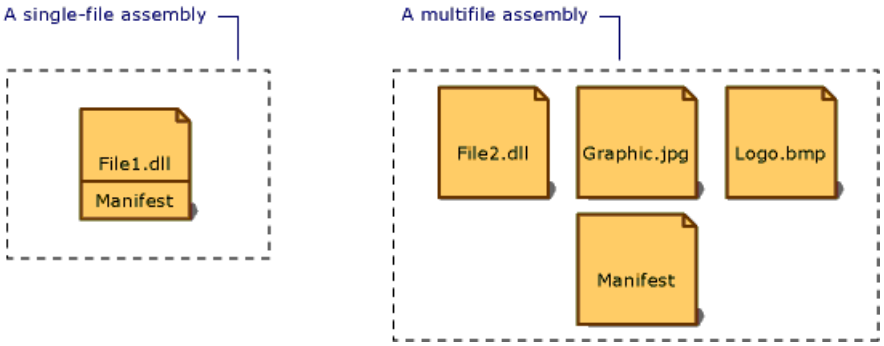
Assemblies in the Common Language Runtime
Assembly Manifest
Assembly Security Considerations

Assembly Manifest

5/2/2018 • 2 min to read • Edit Online

Every assembly, whether static or dynamic, contains a collection of data that describes how the elements in the assembly relate to each other. The assembly manifest contains this assembly metadata. An assembly manifest contains all the metadata needed to specify the assembly's version requirements and security identity, and all metadata needed to define the scope of the assembly and resolve references to resources and classes. The assembly manifest can be stored in either a PE file (an .exe or .dll) with Microsoft intermediate language (MSIL) code or in a standalone PE file that contains only assembly manifest information.

The following illustration shows the different ways the manifest can be stored.



Types of assemblies

For an assembly with one associated file, the manifest is incorporated into the PE file to form a single-file assembly. You can create a multifile assembly with a standalone manifest file or with the manifest incorporated into one of the PE files in the assembly.

Each assembly's manifest performs the following functions:

- Enumerates the files that make up the assembly.
- Governs how references to the assembly's types and resources map to the files that contain their declarations and implementations.
- Enumerates other assemblies on which the assembly depends.
- Provides a level of indirection between consumers of the assembly and the assembly's implementation details.
- Renders the assembly self-describing.

Assembly Manifest Contents

The following table shows the information contained in the assembly manifest. The first four items—the assembly name, version number, culture, and strong name information—make up the assembly's identity.

INFORMATION	DESCRIPTION
Assembly name	A text string specifying the assembly's name.
Version number	A major and minor version number, and a revision and build number. The common language runtime uses these numbers to enforce version policy.

INFORMATION	DESCRIPTION
Culture	Information on the culture or language the assembly supports. This information should be used only to designate an assembly as a satellite assembly containing culture- or language-specific information. (An assembly with culture information is automatically assumed to be a satellite assembly.)
Strong name information	The public key from the publisher if the assembly has been given a strong name.
List of all files in the assembly	A hash of each file contained in the assembly and a file name. Note that all files that make up the assembly must be in the same directory as the file containing the assembly manifest.
Type reference information	Information used by the runtime to map a type reference to the file that contains its declaration and implementation. This is used for types that are exported from the assembly.
Information on referenced assemblies	A list of other assemblies that are statically referenced by the assembly. Each reference includes the dependent assembly's name, assembly metadata (version, culture, operating system, and so on), and public key, if the assembly is strong named.

You can add or change some information in the assembly manifest by using assembly attributes in your code. You can change version information and informational attributes, including Trademark, Copyright, Product, Company, and Informational Version. For a complete list of assembly attributes, see Setting Assembly Attributes.

See Also

Assembly Contents
Assembly Versioning
Creating Satellite Assemblies
Strong-Named Assemblies

Global Assembly Cache

5/2/2018 • 2 min to read • Edit Online

Each computer where the Common Language Runtime is installed has a machine-wide code cache called the Global Assembly Cache. The Global Assembly Cache stores assemblies specifically designated to be shared by several applications on the computer.

You should share assemblies by installing them into the Global Assembly Cache only when you need to. As a general guideline, keep assembly dependencies private, and locate assemblies in the application directory unless sharing an assembly is explicitly required. In addition, it is not necessary to install assemblies into the Global Assembly Cache to make them accessible to COM interop or unmanaged code.

> **NOTE**
> There are scenarios where you explicitly do not want to install an assembly into the Global Assembly Cache. If you place one of the assemblies that make up an application in the Global Assembly Cache, you can no longer replicate or install the application by using the **xcopy** command to copy the application directory. You must move the assembly in the Global Assembly Cache as well.

There are two ways to deploy an assembly into the Global Assembly Cache:

- Use an installer designed to work with the Global Assembly Cache. This is the preferred option for installing assemblies into the Global Assembly Cache.
- Use a developer tool called the Global Assembly Cache tool (Gacutil.exe), provided by the Windows Software Development Kit (SDK).

  > **NOTE**
  > In deployment scenarios, use Windows Installer to install assemblies into the Global Assembly Cache. Use the Global Assembly Cache tool only in development scenarios, because it does not provide assembly reference counting and other features provided when using the Windows Installer.

Starting with the .NET Framework 4, the default location for the Global Assembly Cache is **%windir%\Microsoft.NET\assembly**. In earlier versions of the .NET Framework, the default location is **%windir%\assembly**.

Administrators often protect the systemroot directory using an access control list (ACL) to control write and execute access. Because the Global Assembly Cache is installed in a subdirectory of the systemroot directory, it inherits that directory's ACL. It is recommended that only users with Administrator privileges be allowed to delete files from the Global Assembly Cache.

Assemblies deployed in the Global Assembly Cache must have a strong name. When an assembly is added to the Global Assembly Cache, integrity checks are performed on all files that make up the assembly. The cache performs these integrity checks to ensure that an assembly has not been tampered with, for example, when a file has changed but the manifest does not reflect the change.

See Also

Assemblies in the Common Language Runtime
Working with Assemblies and the Global Assembly Cache

Strong-Named Assemblies

5/2/2018 • 2 min to read • Edit Online

Strong-naming an assembly creates a unique identity for the assembly, and can prevent assembly conflicts.

What makes a strong-named assembly?

A strong named assembly is generated by using the private key that corresponds to the public key distributed with the assembly, and the assembly itself. The assembly includes the assembly manifest, which contains the names and hashes of all the files that make up the assembly. Assemblies that have the same strong name should be identical.

You can strong-name assemblies by using Visual Studio or a command-line tool. For more information, see How to: Sign an Assembly with a Strong Name or Sn.exe (Strong Name Tool).

When a strong-named assembly is created, it contains the simple text name of the assembly, the version number, optional culture information, a digital signature, and the public key that corresponds to the private key used for signing.

> **WARNING**
> Do not rely on strong names for security. They provide a unique identity only.

Why strong-name your assemblies?

When you reference a strong-named assembly, you can expect certain benefits, such as versioning and naming protection. Strong-named assemblies can be installed in the Global Assembly Cache, which is required to enable some scenarios.

Strong-named assemblies are useful in the following scenarios:

- You want to enable your assemblies to be referenced by strong-named assemblies, or you want to give `friend` access to your assemblies from other strong-named assemblies.
- An app needs access to different versions of the same assembly. This means you need different versions of an assembly to load side by side in the same app domain without conflict. For example, if different extensions of an API exist in assemblies that have the same simple name, strong-naming provides a unique identity for each version of the assembly.
- You do not want to negatively affect performance of apps using your assembly, so you want the assembly to be domain neutral. This requires strong-naming because a domain-neutral assembly must be installed in the global assembly cache.
- When you want to centralize servicing for your app by applying publisher policy, which means the assembly must be installed in the global assembly cache.

If you are an open-source developer and you want the identity benefits of a strong-named assembly, consider checking in the private key associated with an assembly into your source control system.

See Also

Global Assembly Cache

How to: Sign an Assembly with a Strong Name
Sn.exe (Strong Name Tool)
Creating and Using Strong-Named Assemblies

Assembly Security Considerations

5/2/2018 • 3 min to read • Edit Online

When you build an assembly, you can specify a set of permissions that the assembly requires to run. Whether certain permissions are granted or not granted to an assembly is based on evidence.

There are two distinct ways evidence is used:

- The input evidence is merged with the evidence gathered by the loader to create a final set of evidence used for policy resolution. The methods that use this semantic include **Assembly.Load**, **Assembly.LoadFrom**, and **Activator.CreateInstance**.
- The input evidence is used unaltered as the final set of evidence used for policy resolution. The methods that use this semantic include **Assembly.Load(byte[])** and **AppDomain.DefineDynamicAssembly()**.

Optional permissions can be granted by the security policy set on the computer where the assembly will run. If you want your code to handle all potential security exceptions, you can do one of the following:

- Insert a permission request for all the permissions your code must have, and handle up front the load-time failure that occurs if the permissions are not granted.
- Do not use a permission request to obtain permissions your code might need, but be prepared to handle security exceptions if permissions are not granted.

> **NOTE**
> Security is a complex area, and you have many options to choose from. For more information, see Key Security Concepts.

At load time, the assembly's evidence is used as input to security policy. Security policy is established by the enterprise and the computer's administrator as well as by user policy settings, and determines the set of permissions that is granted to all managed code when executed. Security policy can be established for the publisher of the assembly (if it has a signing tool generated signature), for the Web site and zone (in Internet Explorer terms) the assembly was downloaded from, or for the assembly's strong name. For example, a computer's administrator can establish security policy that allows all code downloaded from a Web site and signed by a given software company to access a database on a computer, but does not grant access to write to the computer's disk.

Strong-Named Assemblies and Signing Tools

You can sign an assembly in two different but complementary ways: with a strong name or by using SignTool.exe (Sign Tool). Signing an assembly with a strong name adds public key encryption to the file containing the assembly manifest. Strong name signing helps to verify name uniqueness, prevent name spoofing, and provide callers with some identity when a reference is resolved.

However, no level of trust is associated with a strong name, which makes SignTool.exe (Sign Tool) important. The two signing tools require a publisher to prove its identity to a third-party authority and obtain a certificate. This certificate is then embedded in your file and can be used by an administrator to decide whether to trust the code's authenticity.

You can give both a strong name and a digital signature created using SignTool.exe (Sign Tool) to an assembly, or you can use either alone. The two signing tools can sign only one file at a time; for a multifile assembly, you sign

the file that contains the assembly manifest. A strong name is stored in the file containing the assembly manifest, but a signature created using SignTool.exe (Sign Tool) is stored in a reserved slot in the portable executable (PE) file containing the assembly manifest. Signing of an assembly using SignTool.exe (Sign Tool) can be used (with or without a strong name) when you already have a trust hierarchy that relies onSignTool.exe (Sign Tool) generated signatures, or when your policy uses only the key portion and does not check a chain of trust.

> **NOTE**
> When using both a strong name and a signing tool signature on an assembly, the strong name must be assigned first.

The common language runtime also performs a hash verification; the assembly manifest contains a list of all files that make up the assembly, including a hash of each file as it existed when the manifest was built. As each file is loaded, its contents are hashed and compared with the hash value stored in the manifest. If the two hashes do not match, the assembly fails to load.

Because strong naming and signing using SignTool.exe (Sign Tool) guarantee integrity, you can base code access security policy on these two forms of assembly evidence. Strong naming and signing using SignTool.exe (Sign Tool) guarantee integrity through digital signatures and certificates. All the technologies mentioned—hash verification, strong naming, and signing using SignTool.exe (Sign Tool)—work together to ensure that the assembly has not been altered in any way.

See Also

Strong-Named Assemblies
Assemblies in the Common Language Runtime
SignTool.exe (Sign Tool)

Assembly Versioning

5/2/2018 • 3 min to read • Edit Online

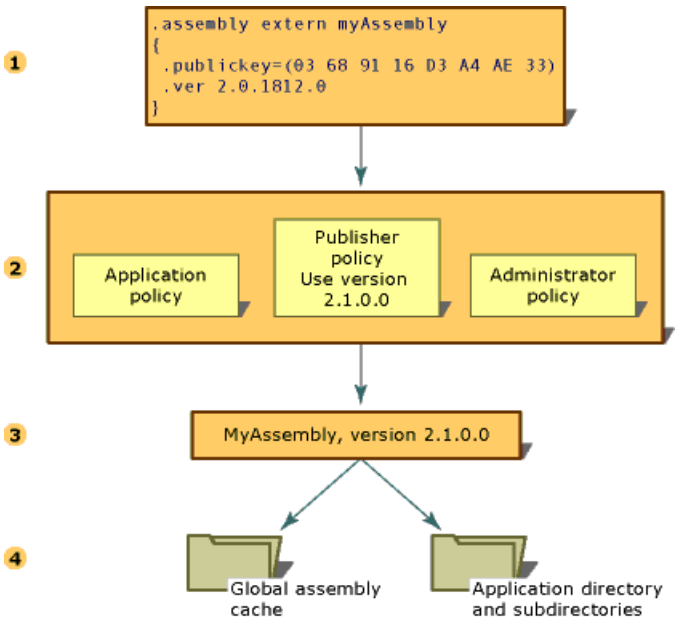
All versioning of assemblies that use the common language runtime is done at the assembly level. The specific version of an assembly and the versions of dependent assemblies are recorded in the assembly's manifest. The default version policy for the runtime is that applications run only with the versions they were built and tested with, unless overridden by explicit version policy in configuration files (the application configuration file, the publisher policy file, and the computer's administrator configuration file).

> **NOTE**
> Versioning is done only on assemblies with strong names.

The runtime performs several steps to resolve an assembly binding request:

1. Checks the original assembly reference to determine the version of the assembly to be bound.
2. Checks for all applicable configuration files to apply version policy.
3. Determines the correct assembly from the original assembly reference and any redirection specified in the configuration files, and determines the version that should be bound to the calling assembly.
4. Checks the global assembly cache, codebases specified in configuration files, and then checks the application's directory and subdirectories using the probing rules explained in How the Runtime Locates Assemblies.

The following illustration shows these steps.



Resolving an assembly binding request

For more information about configuring applications, see Configuring Apps. For more information about binding policy, see How the Runtime Locates Assemblies.

Version Information

Each assembly has two distinct ways of expressing version information:

- The assembly's version number, which, together with the assembly name and culture information, is part of the assembly's identity. This number is used by the runtime to enforce version policy and plays a key part in the type resolution process at run time.
- An informational version, which is a string that represents additional version information included for informational purposes only.

Assembly Version Number

Each assembly has a version number as part of its identity. As such, two assemblies that differ by version number are considered by the runtime to be completely different assemblies. This version number is physically represented as a four-part string with the following format:

<major version>.<minor version>.<build number>.<revision>

For example, version 1.5.1254.0 indicates 1 as the major version, 5 as the minor version, 1254 as the build number, and 0 as the revision number.

The version number is stored in the assembly manifest along with other identity information, including the assembly name and public key, as well as information on relationships and identities of other assemblies connected with the application.

When an assembly is built, the development tool records dependency information for each assembly that is referenced in the assembly manifest. The runtime uses these version numbers, in conjunction with configuration information set by an administrator, an application, or a publisher, to load the proper version of a referenced assembly.

The runtime distinguishes between regular and strong-named assemblies for the purposes of versioning. Version checking only occurs with strong-named assemblies.

For information about specifying version binding policies, see Configuring Apps. For information about how the runtime uses version information to find a particular assembly, see How the Runtime Locates Assemblies.

Assembly Informational Version

The informational version is a string that attaches additional version information to an assembly for informational purposes only; this information is not used at run time. The text-based informational version corresponds to the product's marketing literature, packaging, or product name and is not used by the runtime. For example, an informational version could be "Common Language Runtime version 1.0" or "NET Control SP 2". On the Version tab of the file properties dialog in Microsoft Windows, this information appears in the item "Product Version".

> **NOTE**
> Although you can specify any text, a warning message appears on compilation if the string is not in the format used by the assembly version number, or if it is in that format but contains wildcards. This warning is harmless.

The informational version is represented using the custom attribute System.Reflection.AssemblyInformationalVersionAttribute. For more information about the informational version attribute, see Setting Assembly Attributes.

See Also

How the Runtime Locates Assemblies
Configuring Apps
Setting Assembly Attributes
Assemblies in the Common Language Runtime

Assembly Placement

5/2/2018 • 1 min to read • Edit Online

For most .NET Framework applications, you locate assemblies that make up an application in the application's directory, in a subdirectory of the application's directory, or in the global assembly cache (if the assembly is shared). You can override where the common language runtime looks for an assembly by using the <codeBase> Element in a configuration file. If the assembly does not have a strong name, the location specified using the <codeBase> Element is restricted to the application directory or a subdirectory. If the assembly has a strong name, the <codeBase> Element can specify any location on the computer or on a network.

Similar rules apply to locating assemblies when working with unmanaged code or COM interop applications: if the assembly will be shared by multiple applications, it should be installed into the global assembly cache. Assemblies used with unmanaged code must be exported as a type library and registered. Assemblies used by COM interop must be registered in the catalog, although in some cases this registration occurs automatically.

See Also

How the Runtime Locates Assemblies
Configuring Apps
Advanced COM Interoperability
Assemblies in the Common Language Runtime

Assemblies and Side-by-Side Execution

5/2/2018 • 1 min to read • Edit Online

Side-by-side execution is the ability to store and execute multiple versions of an application or component on the same computer. This means that you can have multiple versions of the runtime, and multiple versions of applications and components that use a version of the runtime, on the same computer at the same time. Side-by-side execution gives you more control over what versions of a component an application binds to, and more control over what version of the runtime an application uses.

Support for side-by-side storage and execution of different versions of the same assembly is an integral part of strong naming and is built into the infrastructure of the runtime. Because the strong-named assembly's version number is part of its identity, the runtime can store multiple versions of the same assembly in the global assembly cache and load those assemblies at run time.

Although the runtime provides you with the ability to create side-by-side applications, side-by-side execution is not automatic. For more information on creating applications for side-by-side execution, see Guidelines for Creating Components for Side-by-Side Execution.

See Also

How the Runtime Locates Assemblies
Assemblies in the Common Language Runtime

Application Domains

5/2/2018 • 11 min to read • Edit Online

Operating systems and runtime environments typically provide some form of isolation between applications. For example, Windows uses processes to isolate applications. This isolation is necessary to ensure that code running in one application cannot adversely affect other, unrelated applications.

Application domains provide an isolation boundary for security, reliability, and versioning, and for unloading assemblies. Application domains are typically created by runtime hosts, which are responsible for bootstrapping the common language runtime before an application is run.

The topics in this section of the documentation explain how to use application domains to provide isolation between assemblies.

This overview contains the following sections:

The Benefits of Isolating Applications

Historically, process boundaries have been used to isolate applications running on the same computer. Each application is loaded into a separate process, which isolates the application from other applications running on the same computer.

The applications are isolated because memory addresses are process-relative; a memory pointer passed from one process to another cannot be used in any meaningful way in the target process. In addition, you cannot make direct calls between two processes. Instead, you must use proxies, which provide a level of indirection.

Managed code must be passed through a verification process before it can be run (unless the administrator has granted permission to skip the verification). The verification process determines whether the code can attempt to access invalid memory addresses or perform some other action that could cause the process in which it is running to fail to operate properly. Code that passes the verification test is said to be type-safe. The ability to verify code as type-safe enables the common language runtime to provide as great a level of isolation as the process boundary, at a much lower performance cost.

Application domains provide a more secure and versatile unit of processing that the common language runtime can use to provide isolation between applications. You can run several application domains in a single process with the same level of isolation that would exist in separate processes, but without incurring the additional overhead of making cross-process calls or switching between processes. The ability to run multiple applications within a single process dramatically increases server scalability.

Isolating applications is also important for application security. For example, you can run controls from several Web applications in a single browser process in such a way that the controls cannot access each other's data and resources.

The isolation provided by application domains has the following benefits:

- Faults in one application cannot affect other applications. Because type-safe code cannot cause memory faults, using application domains ensures that code running in one domain cannot affect other applications in the process.
- Individual applications can be stopped without stopping the entire process. Using application domains

enables you to unload the code running in a single application.

> **NOTE**
> You cannot unload individual assemblies or types. Only a complete domain can be unloaded.

- Code running in one application cannot directly access code or resources from another application. The common language runtime enforces this isolation by preventing direct calls between objects in different application domains. Objects that pass between domains are either copied or accessed by proxy. If the object is copied, the call to the object is local. That is, both the caller and the object being referenced are in the same application domain. If the object is accessed through a proxy, the call to the object is remote. In this case, the caller and the object being referenced are in different application domains. Cross-domain calls use the same remote call infrastructure as calls between two processes or between two machines. As such, the metadata for the object being referenced must be available to both application domains to allow the method call to be JIT-compiled properly. If the calling domain does not have access to the metadata for the object being called, the compilation might fail with an exception of type **System.IO.FileNotFound**. See Remote Objects for more details. The mechanism for determining how objects can be accessed across domains is determined by the object. For more information, see System.MarshalByRefObject.
- The behavior of code is scoped by the application in which it runs. In other words, the application domain provides configuration settings such as application version policies, the location of any remote assemblies it accesses, and information about where to locate assemblies that are loaded into the domain.
- Permissions granted to code can be controlled by the application domain in which the code is running.

Application Domains and Assemblies

This topic describes the relationship between application domains and assemblies. You must load an assembly into an application domain before you can execute the code it contains. Running a typical application causes several assemblies to be loaded into an application domain.

The way an assembly is loaded determines whether its just-in-time (JIT) compiled code can be shared by multiple application domains in the process, and whether the assembly can be unloaded from the process.

- If an assembly is loaded domain-neutral, all application domains that share the same security grant set can share the same JIT-compiled code, which reduces the memory required by the application. However, the assembly can never be unloaded from the process.
- If an assembly is not loaded domain-neutral, it must be JIT-compiled in every application domain in which it is loaded. However, the assembly can be unloaded from the process by unloading all the application domains in which it is loaded.

The runtime host determines whether to load assemblies as domain-neutral when it loads the runtime into a process. For managed applications, apply the LoaderOptimizationAttribute attribute to the entry-point method for the process, and specify a value from the associated LoaderOptimization enumeration. For unmanaged applications that host the common language runtime, specify the appropriate flag when you call the CorBindToRuntimeEx Function method.

There are three options for loading domain-neutral assemblies:

- LoaderOptimization loads no assemblies as domain-neutral, except Mscorlib, which is always loaded domain-neutral. This setting is called single domain because it is commonly used when the host is running only a single application in the process.
- LoaderOptimization loads all assemblies as domain-neutral. Use this setting when there are multiple application domains in the process, all of which run the same code.

- LoaderOptimization loads strong-named assemblies as domain-neutral, if they and all their dependencies have been installed in the global assembly cache. Other assemblies are loaded and JIT-compiled separately for each application domain in which they are loaded, and thus can be unloaded from the process. Use this setting when running more than one application in the same process, or if you have a mixture of assemblies that are shared by many application domains and assemblies that need to be unloaded from the process.

JIT-compiled code cannot be shared for assemblies loaded into the load-from context, using the LoadFrom method of the Assembly class, or loaded from images using overloads of the Load method that specify byte arrays.

Assemblies that have been compiled to native code by using the Ngen.exe (Native Image Generator) can be shared between application domains, if they are loaded domain-neutral the first time they are loaded into a process.

JIT-compiled code for the assembly that contains the application entry point is shared only if all its dependencies can be shared.

A domain-neutral assembly can be JIT-compiled more than once. For example, when the security grant sets of two application domains are different, they cannot share the same JIT-compiled code. However, each copy of the JIT-compiled assembly can be shared with other application domains that have the same grant set.

When you decide whether to load assemblies as domain-neutral, you must make a tradeoff between reducing memory use and other performance factors.

- Access to static data and methods is slower for domain-neutral assemblies because of the need to isolate assemblies. Each application domain that accesses the assembly must have a separate copy of the static data, to prevent references to objects in static fields from crossing domain boundaries. As a result, the runtime contains additional logic to direct a caller to the appropriate copy of the static data or method. This extra logic slows down the call.
- All the dependencies of an assembly must be located and loaded when the assembly is loaded domain-neutral, because a dependency that cannot be loaded domain-neutral prevents the assembly from being loaded domain-neutral.

Application Domains and Threads

An application domain forms an isolation boundary for security, versioning, reliability, and unloading of managed code. A thread is the operating system construct used by the common language runtime to execute code. At run time, all managed code is loaded into an application domain and is run by one or more managed threads.

There is not a one-to-one correlation between application domains and threads. Several threads can execute in a single application domain at any given time, and a particular thread is not confined to a single application domain. That is, threads are free to cross application domain boundaries; a new thread is not created for each application domain.

At any given time, every thread executes in an application domain. Zero, one, or multiple threads might be executing in any given application domain. The run time keeps track of which threads are running in which application domains. You can locate the domain in which a thread is executing at any time by calling the Thread.GetDomain method.

Application Domains and Cultures

Culture, which is represented by a CultureInfo object, is associated with threads. You can get the culture that is associated with the currently executing thread by using the CultureInfo.CurrentCulture property, and you can get or set the culture that is associated with the currently executing thread by using the Thread.CurrentCulture property. If the culture that is associated with a thread has been explicitly set by using the Thread.CurrentCulture property, it continues to be associated with that thread when the thread crosses application domain boundaries.

Otherwise, the culture that is associated with the thread at any given time is determined by the value of the CultureInfo.DefaultThreadCurrentCulture property in the application domain in which the thread is executing:

- If the value of the property is not `null`, the culture that is returned by the property is associated with the thread (and therefore returned by the Thread.CurrentCulture and CultureInfo.CurrentCulture properties).
- If the value of the property is `null`, the current system culture is associated with the thread.

Programming with Application Domains

Application domains are usually created and manipulated programmatically by runtime hosts. However, sometimes an application program might also want to work with application domains. For example, an application program could load an application component into a domain to be able to unload the domain (and the component) without having to stop the entire application.

The AppDomain is the programmatic interface to application domains. This class includes methods to create and unload domains, to create instances of types in domains, and to register for various notifications such as application domain unloading. The following table lists commonly used AppDomain methods.

APPDOMAIN METHOD	DESCRIPTION
CreateDomain	Creates a new application domain. It is recommended that you use an overload of this method that specifies an AppDomainSetup object. This is the preferred way to set the properties of a new domain, such as the application base, or root directory for the application; the location of the configuration file for the domain; and the search path that the common language runtime is to use to load assemblies into the domain.
ExecuteAssembly and ExecuteAssemblyByName	Executes an assembly in the application domain. This is an instance method, so it can be used to execute code in another application domain to which you have a reference.
CreateInstanceAndUnwrap	Creates an instance of a specified type in the application domain, and returns a proxy. Use this method to avoid loading the assembly containing the created type into the calling assembly.
Unload	Performs a graceful shutdown of the domain. The application domain is not unloaded until all threads running in the domain have either stopped or are no longer in the domain.

> **NOTE**
> The common language runtime does not support serialization of global methods, so delegates cannot be used to execute global methods in other application domains.

The unmanaged interfaces described in the common language runtime Hosting Interfaces Specification also provide access to application domains. Runtime hosts can use interfaces from unmanaged code to create and gain access to the application domains within a process.

COMPLUS_LoaderOptimization Environment Variable

An environment variable that sets the default loader optimization policy of an executable application.

Syntax

```
COMPLUS_LoaderOptimization = 1
```

Remarks

A typical application loads several assemblies into an application domain before the code they contain can be executed.

The way the assembly is loaded determines whether its just-in-time (JIT) compiled code can be shared by multiple application domains in the process.

- If an assembly is loaded domain-neutral, all application domains that share the same security grant set can share the same JIT-compiled code. This reduces the memory required by the application.
- If an assembly is not loaded domain-neutral, it must be JIT-compiled in every application domain in which it is loaded and the loader must not share internal resources across application domains.

When set to 1, the COMPLUS_LoaderOptimization environment flag forces the runtime host to load all assemblies in non-domain-neutral way known as SingleDomain. SingleDomain loads no assemblies as domain-neutral, except Mscorlib, which is always loaded domain-neutral. This setting is called single domain because it is commonly used when the host is running only a single application in the process.

Caution

The COMPLUS_LoaderOptimization environment flag was designed to be used in diagnostic and test scenarios. Having the flag turned on can cause severe slow-down and increase in memory usage.

Code Example

To force all assemblies not to be loaded as domain-neutral for the IISADMIN service can be achieved by appending `COMPLUS_LoaderOptimization=1` to the Environment's Multi-String Value in the HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\services\IISADMIN key.

```
Key = HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\services\IISADMIN
Name = Environment
Type = REG_MULTI_SZ
Value (to append) = COMPLUS_LoaderOptimization=1
```

Reference

System.MarshalByRefObject

Application Domains and Assemblies How-to Topics

5/2/2018 • 1 min to read • Edit Online

The following sections contain links to all How-to topics found in the conceptual documentation for programming with application domains and assemblies.

Application Domains

- How to: Create an Application Domain
- How to: Unload an Application Domain
- How to: Configure an Application Domain
- How to: Load Assemblies into an Application Domain
- How to: Obtain Type and Member Information from an Assembly

Assemblies

- How to: Determine an Assembly's Fully Qualified Name
- How to: Build a Single-File Assembly
- How to: Build a Multifile Assembly
- How to: Create a Public-Private Key Pair
- How to: Sign an Assembly with a Strong Name
- How to: Reference a Strong-Named Assembly
- How to: Disable the Strong-Name Bypass Feature
- How to: Install an Assembly into the Global Assembly Cache
- How to: View the Contents of the Global Assembly Cache
- How to: Remove an Assembly from the Global Assembly Cache
- How to: View Assembly Contents

See Also

Application Domains and Assemblies

Using Application Domains

5/2/2018 • 1 min to read • Edit Online

Application domains provide a unit of isolation for the common language runtime. They are created and run inside a process. Application domains are usually created by a runtime host, which is an application responsible for loading the runtime into a process and executing user code within an application domain. The runtime host creates a process and a default application domain, and runs managed code inside it. Runtime hosts include ASP.NET, Microsoft Internet Explorer, and the Windows shell.

For most applications, you do not need to create your own application domain; the runtime host creates any necessary application domains for you. However, you can create and configure additional application domains if your application needs to isolate code or to use and unload DLLs.

In This Section

How to: Create an Application Domain
Describes how to programmatically create an application domain.

How to: Unload an Application Domain
Describes how to programmatically unload an application domain.

How to: Configure an Application Domain
Provides an introduction to configuring an application domain.

Retrieving Setup Information from an Application Domain
Describes how to retrieve setup information from an application domain.

How to: Load Assemblies into an Application Domain
Describes how to load an assembly into an application domain.

How to: Obtain Type and Member Information from an Assembly
Describes how to retrieve information about an assembly.

Shadow Copying Assemblies
Describes how shadow copying allows updates to assemblies while they are in use, and how to configure shadow copying.

How to: Receive First-Chance Exception Notifications
Explains how you can receive a notification that an exception has been thrown, before the common language runtime has begun searching for exception handlers.

Resolving Assembly Loads
Provides guidance on using the AppDomain.AssemblyResolve event to resolve assembly load failures.

Reference

AppDomain
Represents an application domain. Provides methods for creating and controlling application domains.

Related Sections

Assemblies in the Common Language Runtime
Provides an overview of the functions performed by assemblies.

Programming with Assemblies

Describes how to create, sign, and set attributes on assemblies.

Emitting Dynamic Methods and Assemblies

Describes how to create dynamic assemblies.

Application Domains

Provides a conceptual overview of application domains.

Reflection Overview

Describes how to use the **Reflection** class to obtain information about an assembly.

How to: Create an Application Domain

5/2/2018 • 1 min to read • Edit Online

A common language runtime host creates application domains automatically when they are needed. However, you can create your own application domains and load into them those assemblies that you want to manage personally. You can also create application domains from which you execute code.

You create a new application domain using one of the overloaded **CreateDomain** methods in the System.AppDomain class. You can give the application domain a name and reference it by that name.

The following example creates a new application domain, assigns it the name `MyDomain`, and then prints the name of the host domain and the newly created child application domain to the console.

Example

```
using namespace System;
using namespace System::Reflection;

ref class AppDomain1
{
public:
    static void Main()
    {
        Console::WriteLine("Creating new AppDomain.");
        AppDomain^ domain = AppDomain::CreateDomain("MyDomain");

        Console::WriteLine("Host domain: " + AppDomain::CurrentDomain->FriendlyName);
        Console::WriteLine("child domain: " + domain->FriendlyName);
    }
};

int main()
{
    AppDomain1::Main();
}
```

```
using System;
using System.Reflection;

class AppDomain1
{
    public static void Main()
    {
        Console.WriteLine("Creating new AppDomain.");
        AppDomain domain = AppDomain.CreateDomain("MyDomain");

        Console.WriteLine("Host domain: " + AppDomain.CurrentDomain.FriendlyName);
        Console.WriteLine("child domain: " + domain.FriendlyName);
    }
}
```

```
Imports System
Imports System.Reflection

Class AppDomain1
    Public Shared Sub Main()
        Console.WriteLine("Creating new AppDomain.")
        Dim domain As AppDomain = AppDomain.CreateDomain("MyDomain")

        Console.WriteLine("Host domain: " + AppDomain.CurrentDomain.FriendlyName)
        Console.WriteLine("child domain: " + domain.FriendlyName)
    End Sub
End Class
```

See Also

Programming with Application Domains
Using Application Domains

How to: Unload an Application Domain

5/2/2018 • 2 min to read • Edit Online

When you have finished using an application domain, unload it using the AppDomain.Unload method. The **Unload** method gracefully shuts down the specified application domain. During the unloading process, no new threads can access the application domain, and all application domain–specific data structures are freed.

Assemblies loaded into the application domain are removed and are no longer available. If an assembly in the application domain is domain-neutral, data for the assembly remains in memory until the entire process is shut down. There is no mechanism to unload a domain-neutral assembly other than shutting down the entire process. There are situations where the request to unload an application domain does not work and results in a CannotUnloadAppDomainException.

The following example creates a new application domain called `MyDomain`, prints some information to the console, and then unloads the application domain. Note that the code then attempts to print the friendly name of the unloaded application domain to the console. This action generates an exception that is handled by the try/catch statements at the end of the program.

Example

```
using namespace System;
using namespace System::Reflection;

ref class AppDomain2
{
public:
    static void Main()
    {
        Console::WriteLine("Creating new AppDomain.");
        AppDomain^ domain = AppDomain::CreateDomain("MyDomain", nullptr);

        Console::WriteLine("Host domain: " + AppDomain::CurrentDomain->FriendlyName);
        Console::WriteLine("child domain: " + domain->FriendlyName);
        AppDomain::Unload(domain);
        try
        {
            Console::WriteLine();
            Console::WriteLine("Host domain: " + AppDomain::CurrentDomain->FriendlyName);
            // The following statement creates an exception because the domain no longer exists.
            Console::WriteLine("child domain: " + domain->FriendlyName);
        }
        catch (AppDomainUnloadedException^ e)
        {
            Console::WriteLine(e->GetType()->FullName);
            Console::WriteLine("The appdomain MyDomain does not exist.");
        }
    }
};

int main()
{
    AppDomain2::Main();
}
```

```
using System;
using System.Reflection;

class AppDomain2
{
    public static void Main()
    {
        Console.WriteLine("Creating new AppDomain.");
        AppDomain domain = AppDomain.CreateDomain("MyDomain", null);

        Console.WriteLine("Host domain: " + AppDomain.CurrentDomain.FriendlyName);
        Console.WriteLine("child domain: " + domain.FriendlyName);
        AppDomain.Unload(domain);
        try
        {
            Console.WriteLine();
            Console.WriteLine("Host domain: " + AppDomain.CurrentDomain.FriendlyName);
            // The following statement creates an exception because the domain no longer exists.
            Console.WriteLine("child domain: " + domain.FriendlyName);
        }
        catch (AppDomainUnloadedException e)
        {
            Console.WriteLine(e.GetType().FullName);
            Console.WriteLine("The appdomain MyDomain does not exist.");
        }
    }
}
```

```
Imports System
Imports System.Reflection

Class AppDomain2
    Public Shared Sub Main()
        Console.WriteLine("Creating new AppDomain.")
        Dim domain As AppDomain = AppDomain.CreateDomain("MyDomain", Nothing)

        Console.WriteLine("Host domain: " + AppDomain.CurrentDomain.FriendlyName)
        Console.WriteLine("child domain: " + domain.FriendlyName)
        AppDomain.Unload(domain)
        Try
            Console.WriteLine()
            Console.WriteLine("Host domain: " + AppDomain.CurrentDomain.FriendlyName)
            ' The following statement creates an exception because the domain no longer exists.
            Console.WriteLine("child domain: " + domain.FriendlyName)
        Catch e As AppDomainUnloadedException
            Console.WriteLine(e.GetType().FullName)
            Console.WriteLine("The appdomain MyDomain does not exist.")
        End Try
    End Sub
End Class
```

See Also

Programming with Application Domains
How to: Create an Application Domain
Using Application Domains

How to: Configure an Application Domain

5/2/2018 • 1 min to read • Edit Online

You can provide the common language runtime with configuration information for a new application domain using the AppDomainSetup class. When creating your own application domains, the most important property is ApplicationBase. The other **AppDomainSetup** properties are used mainly by runtime hosts to configure a particular application domain.

The **ApplicationBase** property defines the root directory of the application. When the runtime needs to satisfy a type request, it probes for the assembly containing the type in the directory specified by the **ApplicationBase** property.

> **NOTE**
> A new application domain inherits only the **ApplicationBase** property of the creator.

The following example creates an instance of the **AppDomainSetup** class, uses this class to create a new application domain, writes the information to console, and then unloads the application domain.

Example

```
using namespace System;
using namespace System::Reflection;

ref class AppDomain4
{
public:
    static void Main()
    {
        // Create application domain setup information.
        AppDomainSetup^ domaininfo = gcnew AppDomainSetup();
        domaininfo->ApplicationBase = "f:\\work\\development\\latest";

        // Create the application domain.
        AppDomain^ domain = AppDomain::CreateDomain("MyDomain", nullptr, domaininfo);

        // Write application domain information to the console.
        Console::WriteLine("Host domain: " + AppDomain::CurrentDomain->FriendlyName);
        Console::WriteLine("child domain: " + domain->FriendlyName);
        Console::WriteLine("Application base is: " + domain->SetupInformation->ApplicationBase);

        // Unload the application domain.
        AppDomain::Unload(domain);
    }
};

int main()
{
    AppDomain4::Main();
}
```

```
using System;
using System.Reflection;

class AppDomain4
{
    public static void Main()
    {
        // Create application domain setup information.
        AppDomainSetup domaininfo = new AppDomainSetup();
        domaininfo.ApplicationBase = "f:\\work\\development\\latest";

        // Create the application domain.
        AppDomain domain = AppDomain.CreateDomain("MyDomain", null, domaininfo);

        // Write application domain information to the console.
        Console.WriteLine("Host domain: " + AppDomain.CurrentDomain.FriendlyName);
        Console.WriteLine("child domain: " + domain.FriendlyName);
        Console.WriteLine("Application base is: " + domain.SetupInformation.ApplicationBase);

        // Unload the application domain.
        AppDomain.Unload(domain);
    }
}
```

```
Imports System
Imports System.Reflection

Class AppDomain4
    Public Shared Sub Main()
        ' Create application domain setup information.
        Dim domaininfo As new AppDomainSetup()
        domaininfo.ApplicationBase = "f:\work\development\latest"

        ' Create the application domain.
        Dim domain As AppDomain = AppDomain.CreateDomain("MyDomain", Nothing, domaininfo)

        ' Write application domain information to the console.
        Console.WriteLine("Host domain: " + AppDomain.CurrentDomain.FriendlyName)
        Console.WriteLine("child domain: " + domain.FriendlyName)
        Console.WriteLine("Application base is: " + domain.SetupInformation.ApplicationBase)

        ' Unload the application domain.
        AppDomain.Unload(domain)
    End Sub
End Class
```

See Also

Programming with Application Domains
Using Application Domains

Retrieving Setup Information from an Application Domain

5/2/2018 • 2 min to read • Edit Online

Each instance of an application domain consists of both properties and AppDomainSetup information. You can retrieve setup information from an application domain using the System.AppDomain class. This class provides several members that retrieve configuration information about an application domain.

You can also query the **AppDomainSetup** object for the application domain to obtain setup information that was passed to the domain when it was created.

The following example creates a new application domain and then prints several member values to the console.

```
using namespace System;
using namespace System::Reflection;

ref class AppDomain3
{
public:
    static void Main()
    {
        // Create the new application domain.
        AppDomain^ domain = AppDomain::CreateDomain("MyDomain", nullptr);

        // Output to the console.
        Console::WriteLine("Host domain: " + AppDomain::CurrentDomain->FriendlyName);
        Console::WriteLine("New domain: " + domain->FriendlyName);
        Console::WriteLine("Application base is: " + domain->BaseDirectory);
        Console::WriteLine("Relative search path is: " + domain->RelativeSearchPath);
        Console::WriteLine("Shadow copy files is set to: " + domain->ShadowCopyFiles);
        AppDomain::Unload(domain);
    }
};

int main()
{
    AppDomain3::Main();
}
```

```
using System;
using System.Reflection;

class AppDomain3
{
    public static void Main()
    {
        // Create the new application domain.
        AppDomain domain = AppDomain.CreateDomain("MyDomain", null);

        // Output to the console.
        Console.WriteLine("Host domain: " + AppDomain.CurrentDomain.FriendlyName);
        Console.WriteLine("New domain: " + domain.FriendlyName);
        Console.WriteLine("Application base is: " + domain.BaseDirectory);
        Console.WriteLine("Relative search path is: " + domain.RelativeSearchPath);
        Console.WriteLine("Shadow copy files is set to: " + domain.ShadowCopyFiles);
        AppDomain.Unload(domain);
    }
}
```

```
Imports System
Imports System.Reflection

Class AppDomain3
    Public Shared Sub Main()
        ' Create the new application domain.
        Dim domain As AppDomain = AppDomain.CreateDomain("MyDomain", Nothing)

        ' Output to the console.
        Console.WriteLine("Host domain: " + AppDomain.CurrentDomain.FriendlyName)
        Console.WriteLine("New domain: " + domain.FriendlyName)
        Console.WriteLine("Application base is: " + domain.BaseDirectory)
        Console.WriteLine("Relative search path is: " + domain.RelativeSearchPath)
        Console.WriteLine("Shadow copy files is set to: " + domain.ShadowCopyFiles)
        AppDomain.Unload(domain)
    End Sub
End Class
```

The following example sets, and then retrieves, setup information for an application domain. Note that `AppDomain.SetupInformation.ApplicationBase` gets the configuration information.

```
using namespace System;
using namespace System::Reflection;

ref class AppDomain5
{
public:
    static void Main()
    {
        // Application domain setup information.
        AppDomainSetup^ domaininfo = gcnew AppDomainSetup();
        domaininfo->ApplicationBase = "f:\\work\\development\\latest";
        domaininfo->ConfigurationFile = "f:\\work\\development\\latest\\appdomain5.exe.config";

        // Creates the application domain.
        AppDomain^ domain = AppDomain::CreateDomain("MyDomain", nullptr, domaininfo);

        // Write the application domain information to the console.
        Console::WriteLine("Host domain: " + AppDomain::CurrentDomain->FriendlyName);
        Console::WriteLine("Child domain: " + domain->FriendlyName);
        Console::WriteLine();
        Console::WriteLine("Application base is: " + domain->SetupInformation->ApplicationBase);
        Console::WriteLine("Configuration file is: " + domain->SetupInformation->ConfigurationFile);

        // Unloads the application domain.
        AppDomain::Unload(domain);
    }
};

int main()
{
    AppDomain5::Main();
}
```

```
using System;
using System.Reflection;

class AppDomain5
{
    public static void Main()
    {
        // Application domain setup information.
        AppDomainSetup domaininfo = new AppDomainSetup();
        domaininfo.ApplicationBase = "f:\\work\\development\\latest";
        domaininfo.ConfigurationFile = "f:\\work\\development\\latest\\appdomain5.exe.config";

        // Creates the application domain.
        AppDomain domain = AppDomain.CreateDomain("MyDomain", null, domaininfo);

        // Write the application domain information to the console.
        Console.WriteLine("Host domain: " + AppDomain.CurrentDomain.FriendlyName);
        Console.WriteLine("Child domain: " + domain.FriendlyName);
        Console.WriteLine();
        Console.WriteLine("Application base is: " + domain.SetupInformation.ApplicationBase);
        Console.WriteLine("Configuration file is: " + domain.SetupInformation.ConfigurationFile);

        // Unloads the application domain.
        AppDomain.Unload(domain);
    }
}
```

```
Imports System
Imports System.Reflection

Class AppDomain5
    Public Shared Sub Main()
        ' Application domain setup information.
        Dim domaininfo As New AppDomainSetup()
        domaininfo.ApplicationBase = "f:\work\development\latest"
        domaininfo.ConfigurationFile = "f:\work\development\latest\appdomain5.exe.config"

        ' Creates the application domain.
        Dim domain As AppDomain = AppDomain.CreateDomain("MyDomain", Nothing, domaininfo)

        ' Write the application domain information to the console.
        Console.WriteLine("Host domain: " + AppDomain.CurrentDomain.FriendlyName)
        Console.WriteLine("Child domain: " + domain.FriendlyName)
        Console.WriteLine()
        Console.WriteLine("Application base is: " + domain.SetupInformation.ApplicationBase)
        Console.WriteLine("Configuration file is: " + domain.SetupInformation.ConfigurationFile)

        ' Unloads the application domain.
        AppDomain.Unload(domain)
    End Sub
End Class
```

See Also

Programming with Application Domains
Using Application Domains

How to: Load Assemblies into an Application Domain

5/2/2018 • 2 min to read • Edit Online

There are several ways to load an assembly into an application domain. The recommended way is to use the `static` (`Shared` in Visual Basic) Load method of the System.Reflection.Assembly class. Other ways assemblies can be loaded include:

- The LoadFrom method of the Assembly class loads an assembly given its file location. Loading assemblies with this method uses a different load context.
- The ReflectionOnlyLoad and ReflectionOnlyLoadFrom methods load an assembly into the reflection-only context. Assemblies loaded into this context can be examined but not executed, allowing the examination of assemblies that target other platforms. See How to: Load Assemblies into the Reflection-Only Context.

> **NOTE**
> The reflection-only context is new in the .NET Framework version 2.0.

- Methods such as CreateInstance and CreateInstanceAndUnwrap of the AppDomain class can load assemblies into an application domain.
- The GetType method of the Type class can load assemblies.
- The Load method of the System.AppDomain class can load assemblies, but is primarily used for COM interoperability. It should not be used to load assemblies into an application domain other than the application domain from which it is called.

> **NOTE**
> Starting with the .NET Framework version 2.0, the runtime will not load an assembly that was compiled with a version of the .NET Framework that has a higher version number than the currently loaded runtime. This applies to the combination of the major and minor components of the version number.

You can specify the way the just-in-time (JIT) compiled code from loaded assemblies is shared between application domains. For more information, see Application Domains and Assemblies.

Example

The following code loads an assembly named "example.exe" or "example.dll" into the current application domain, gets a type named `Example` from the assembly, gets a parameterless method named `MethodA` for that type, and executes the method. For a complete discussion on obtaining information from a loaded assembly, see Dynamically Loading and Using Types.

```
using namespace System;
using namespace System::Reflection;

public ref class Asmload0
{
public:
    static void Main()
    {
        // Use the file name to load the assembly into the current
        // application domain.
        Assembly^ a = Assembly::Load("example");
        // Get the type to use.
        Type^ myType = a->GetType("Example");
        // Get the method to call.
        MethodInfo^ myMethod = myType->GetMethod("MethodA");
        // Create an instance.
        Object^ obj = Activator::CreateInstance(myType);
        // Execute the method.
        myMethod->Invoke(obj, nullptr);
    }
};

int main()
{
    Asmload0::Main();
}
```

```
using System;
using System.Reflection;

public class Asmload0
{
    public static void Main()
    {
        // Use the file name to load the assembly into the current
        // application domain.
        Assembly a = Assembly.Load("example");
        // Get the type to use.
        Type myType = a.GetType("Example");
        // Get the method to call.
        MethodInfo myMethod = myType.GetMethod("MethodA");
        // Create an instance.
        object obj = Activator.CreateInstance(myType);
        // Execute the method.
        myMethod.Invoke(obj, null);
    }
}
```

```
Imports System
Imports System.Reflection

Public Class Asmload0
    Public Shared Sub Main()
        ' Use the file name to load the assembly into the current
        ' application domain.
        Dim a As Assembly = Assembly.Load("example")
        ' Get the type to use.
        Dim myType As Type = a.GetType("Example")
        ' Get the method to call.
        Dim myMethod As MethodInfo = myType.GetMethod("MethodA")
        ' Create an instance.
        Dim obj As Object = Activator.CreateInstance(myType)
        ' Execute the method.
        myMethod.Invoke(obj, Nothing)
    End Sub
End Class
```

See Also

ReflectionOnlyLoad
Programming with Application Domains
Reflection
Using Application Domains
How to: Load Assemblies into the Reflection-Only Context
Application Domains and Assemblies

How to: Obtain Type and Member Information from an Assembly

5/2/2018 • 1 min to read • Edit Online

The System.Reflection namespace contains many methods for obtaining information from an assembly. This section demonstrates one of these methods. For additional information, see Reflection Overview.

The following example obtains type and member information from an assembly.

Example

```
using namespace System;
using namespace System::Reflection;

ref class Asminfo1
{
public:
    static void Main()
    {
        Console::WriteLine ("\nReflection.MemberInfo");

        // Get the Type and MemberInfo.
        // Insert the fully qualified class name inside the quotation marks in the
        // following statement.
        Type^ MyType = Type::GetType("System.IO.BinaryReader");
        array<MemberInfo^>^ Mymemberinfoarray = MyType->GetMembers(BindingFlags::Public |
            BindingFlags::NonPublic | BindingFlags::Static |
            BindingFlags::Instance | BindingFlags::DeclaredOnly);

        // Get and display the DeclaringType method.
        Console::Write("\nThere are {0} documentable members in ", Mymemberinfoarray->Length);
        Console::Write("{0}.", MyType->FullName);

        for each (MemberInfo^ Mymemberinfo in Mymemberinfoarray)
        {
            Console::Write("\n" + Mymemberinfo->Name);
        }
    }
};

int main()
{
    Asminfo1::Main();
}
```

```
using System;
using System.Reflection;

class Asminfo1
{
    public static void Main()
    {
        Console.WriteLine ("\nReflection.MemberInfo");

        // Get the Type and MemberInfo.
        // Insert the fully qualified class name inside the quotation marks in the
        // following statement.
        Type MyType = Type.GetType("System.IO.BinaryReader");
        MemberInfo[] Mymemberinfoarray = MyType.GetMembers(BindingFlags.Public |
            BindingFlags.NonPublic | BindingFlags.Static |
            BindingFlags.Instance | BindingFlags.DeclaredOnly);

        // Get and display the DeclaringType method.
        Console.Write("\nThere are {0} documentable members in ", Mymemberinfoarray.Length);
        Console.Write("{0}.", MyType.FullName);

        foreach (MemberInfo Mymemberinfo in Mymemberinfoarray)
        {
            Console.Write("\n" + Mymemberinfo.Name);
        }
    }
}
```

```
Imports System
Imports System.Reflection

Class Asminfo1
    Public Shared Sub Main()
        Console.WriteLine ("\nReflection.MemberInfo")

        ' Get the Type and MemberInfo.
        ' Insert the fully qualified class name inside the quotation marks in the
        ' following statement.
        Dim MyType As Type = Type.GetType("System.IO.BinaryReader")
        Dim Mymemberinfoarray() As MemberInfo = MyType.GetMembers(BindingFlags.Public Or
            BindingFlags.NonPublic Or BindingFlags.Static Or
            BindingFlags.Instance Or BindingFlags.DeclaredOnly)

        ' Get and display the DeclaringType method.
        Console.Write("\nThere are {0} documentable members in ", Mymemberinfoarray.Length)
        Console.Write("{0}.", MyType.FullName)

        For Each Mymemberinfo As MemberInfo in Mymemberinfoarray
            Console.Write("\n" + Mymemberinfo.Name)
        Next
    End Sub
End Class
```

See Also

Programming with Application Domains Reflection
Using Application Domains

Shadow Copying Assemblies

5/2/2018 • 4 min to read • Edit Online

Shadow copying enables assemblies that are used in an application domain to be updated without unloading the application domain. This is particularly useful for applications that must be available continuously, such as ASP.NET sites.

> **IMPORTANT**
> Shadow copying is not supported in Windows 8.x Store apps.

The common language runtime locks an assembly file when the assembly is loaded, so the file cannot be updated until the assembly is unloaded. The only way to unload an assembly from an application domain is by unloading the application domain, so under normal circumstances, an assembly cannot be updated on disk until all the application domains that are using it have been unloaded.

When an application domain is configured to shadow copy files, assemblies from the application path are copied to another location and loaded from that location. The copy is locked, but the original assembly file is unlocked and can be updated.

> **IMPORTANT**
> The only assemblies that can be shadow copied are those stored in the application directory or its subdirectories, specified by the ApplicationBase and PrivateBinPath properties when the application domain is configured. Assemblies stored in the global assembly cache are not shadow copied.

This article contains the following sections:

- Enabling and Using Shadow Copying describes the basic use and the options that are available for shadow copying.
- Startup Performance describes the changes that are made to shadow copying in the .NET Framework 4 to improve startup performance, and how to revert to the behavior of earlier versions.
- Obsolete Methods describes the changes that were made to the properties and methods that control shadow copying in the .NET Framework 2.0.

Enabling and Using Shadow Copying

You can use the properties of the AppDomainSetup class as follows to configure an application domain for shadow copying:

- Enable shadow copying by setting the ShadowCopyFiles property to the string value `"true"`.

  By default, this setting causes all assemblies in the application path to be copied to a download cache before they are loaded. This is the same cache maintained by the common language runtime to store files downloaded from other computers, and the common language runtime automatically deletes the files when they are no longer needed.

- Optionally set a custom location for shadow copied files by using the CachePath property and the ApplicationName property.

The base path for the location is formed by concatenating the ApplicationName property to the CachePath property as a subdirectory. Assemblies are shadow copied to subdirectories of this path, not to the base path itself.

> **NOTE**
> If the ApplicationName property is not set, the CachePath property is ignored and the download cache is used. No exception is thrown.

If you specify a custom location, you are responsible for cleaning up the directories and copied files when they are no longer needed. They are not deleted automatically.

There are a few reasons why you might want to set a custom location for shadow copied files. You might want to set a custom location for shadow copied files if your application generates a large number of copies. The download cache is limited by size, not by lifetime, so it is possible that the common language runtime will attempt to delete a file that is still in use. Another reason to set a custom location is when users running your application do not have write access to the directory location the common language runtime uses for the download cache.

- Optionally limit the assemblies that are shadow copied by using the ShadowCopyDirectories property.

  When you enable shadow copying for an application domain, the default is to copy all assemblies in the application path — that is, in the directories specified by the ApplicationBase and PrivateBinPath properties. You can limit the copying to selected directories by creating a string that contains only those directories you want to shadow copy, and assigning the string to the ShadowCopyDirectories property. Separate the directories with semicolons. The only assemblies that are shadow copied are the ones in the selected directories.

  > **NOTE**
  > If you don't assign a string to the ShadowCopyDirectories property, or if you set this property to `null`, all assemblies in the directories specified by the ApplicationBase and PrivateBinPath properties are shadow copied.

  > **IMPORTANT**
  > Directory paths must not contain semicolons, because the semicolon is the delimiter character. There is no escape character for semicolons.

Startup Performance

When an application domain that uses shadow copying starts, there is a delay while assemblies in the application directory are copied to the shadow copy directory, or verified if they are already in that location. Before the .NET Framework 4, all assemblies were copied to a temporary directory. Each assembly was opened to verify the assembly name, and the strong name was validated. Each assembly was checked to see whether it had been updated more recently than the copy in the shadow copy directory. If so, it was copied to the shadow copy directory. Finally, the temporary copies were discarded.

Beginning with the .NET Framework 4, the default startup behavior is to directly compare the file date and time of each assembly in the application directory with the file date and time of the copy in the shadow copy directory. If the assembly has been updated, it is copied by using the same procedure as in earlier versions of the .NET Framework; otherwise, the copy in the shadow copy directory is loaded.

The resulting performance improvement is largest for applications in which assemblies do not change frequently

and changes usually occur in a small subset of assemblies. If a majority of assemblies in an application change frequently, the new default behavior might cause a performance regression. You can restore the startup behavior of previous versions of the .NET Framework by adding the <shadowCopyVerifyByTimestamp> element to the configuration file, with `enabled="false"`.

Obsolete Methods

The AppDomain class has several methods, such as SetShadowCopyFiles and ClearShadowCopyPath, that can be used to control shadow copying on an application domain, but these have been marked obsolete in the .NET Framework version 2.0. The recommended way to configure an application domain for shadow copying is to use the properties of the AppDomainSetup class.

See Also

AppDomainSetup.ShadowCopyFiles
AppDomainSetup.CachePath
AppDomainSetup.ApplicationName
AppDomainSetup.ShadowCopyDirectories
<shadowCopyVerifyByTimestamp> Element

How to: Receive First-Chance Exception Notifications

5/2/2018 • 8 min to read • Edit Online

The FirstChanceException event of the AppDomain class lets you receive a notification that an exception has been thrown, before the common language runtime has begun searching for exception handlers.

The event is raised at the application domain level. A thread of execution can pass through multiple application domains, so an exception that is unhandled in one application domain could be handled in another application domain. The notification occurs in each application domain that has added a handler for the event, until an application domain handles the exception.

The procedures and examples in this article show how to receive first-chance exception notifications in a simple program that has one application domain, and in an application domain that you create.

For a more complex example that spans several application domains, see the example for the FirstChanceException event.

Receiving First-Chance Exception Notifications in the Default Application Domain

In the following procedure, the entry point for the application, the `Main()` method, runs in the default application domain.

To demonstrate first-chance exception notifications in the default application domain

1. Define an event handler for the FirstChanceException event, using a lambda function, and attach it to the event. In this example, the event handler prints the name of the application domain where the event was handled and the exception's Message property.

```
using System;
using System.Runtime.ExceptionServices;

class Example
{
    static void Main()
    {
        AppDomain.CurrentDomain.FirstChanceException +=
            (object source, FirstChanceExceptionEventArgs e) =>
            {
                Console.WriteLine("FirstChanceException event raised in {0}: {1}",
                    AppDomain.CurrentDomain.FriendlyName, e.Exception.Message);
            };
```

```
Imports System.Runtime.ExceptionServices

Class Example

    Shared Sub Main()

        AddHandler AppDomain.CurrentDomain.FirstChanceException,
                   Sub(source As Object, e As FirstChanceExceptionEventArgs)
                       Console.WriteLine("FirstChanceException event raised in {0}: {1}",
                                         AppDomain.CurrentDomain.FriendlyName,
                                         e.Exception.Message)
                   End Sub
```

2. Throw an exception and catch it. Before the runtime locates the exception handler, the FirstChanceException event is raised and displays a message. This message is followed by the message that is displayed by the exception handler.

```
try
{
    throw new ArgumentException("Thrown in " + AppDomain.CurrentDomain.FriendlyName);
}
catch (ArgumentException ex)
{
    Console.WriteLine("ArgumentException caught in {0}: {1}",
        AppDomain.CurrentDomain.FriendlyName, ex.Message);
}
```

```
Try
    Throw New ArgumentException("Thrown in " & AppDomain.CurrentDomain.FriendlyName)

Catch ex As ArgumentException

    Console.WriteLine("ArgumentException caught in {0}: {1}",
        AppDomain.CurrentDomain.FriendlyName, ex.Message)
End Try
```

3. Throw an exception, but do not catch it. Before the runtime looks for an exception handler, the FirstChanceException event is raised and displays a message. There is no exception handler, so the application terminates.

```
        throw new ArgumentException("Thrown in " + AppDomain.CurrentDomain.FriendlyName);
    }
}
```

```
        Throw New ArgumentException("Thrown in " & AppDomain.CurrentDomain.FriendlyName)
    End Sub
End Class
```

The code that is shown in the first three steps of this procedure forms a complete console application. The output from the application varies, depending on the name of the .exe file, because the name of the default application domain consists of the name and extension of the .exe file. See the following for sample output.

```
/* This example produces output similar to the following:

FirstChanceException event raised in Example.exe: Thrown in Example.exe
ArgumentException caught in Example.exe: Thrown in Example.exe
FirstChanceException event raised in Example.exe: Thrown in Example.exe

Unhandled Exception: System.ArgumentException: Thrown in Example.exe
   at Example.Main()
 */
```

```
' This example produces output similar to the following:
'
'FirstChanceException event raised in Example.exe: Thrown in Example.exe
'ArgumentException caught in Example.exe: Thrown in Example.exe
'FirstChanceException event raised in Example.exe: Thrown in Example.exe
'
'Unhandled Exception: System.ArgumentException: Thrown in Example.exe
'   at Example.Main()
```

Receiving First-Chance Exception Notifications in Another Application Domain

If your program contains more than one application domain, you can choose which application domains receive notifications.

To receive first-chance exception notifications in an application domain that you create

1. Define an event handler for the FirstChanceException event. This example uses a `static` method (`Shared` method in Visual Basic) that prints the name of the application domain where the event was handled and the exception's Message property.

```
static void FirstChanceHandler(object source, FirstChanceExceptionEventArgs e)
{
    Console.WriteLine("FirstChanceException event raised in {0}: {1}",
        AppDomain.CurrentDomain.FriendlyName, e.Exception.Message);
}
```

```
Shared Sub FirstChanceHandler(ByVal source As Object,
                              ByVal e As FirstChanceExceptionEventArgs)

    Console.WriteLine("FirstChanceException event raised in {0}: {1}",
        AppDomain.CurrentDomain.FriendlyName, e.Exception.Message)
End Sub
```

2. Create an application domain and add the event handler to the FirstChanceException event for that application domain. In this example, the application domain is named `AD1`.

```
AppDomain ad = AppDomain.CreateDomain("AD1");
ad.FirstChanceException += FirstChanceHandler;
```

```
Dim ad As AppDomain = AppDomain.CreateDomain("AD1")
AddHandler ad.FirstChanceException, AddressOf FirstChanceHandler
```

   You can handle this event in the default application domain in the same way. Use the `static` (`Shared` in Visual Basic) AppDomain.CurrentDomain property in `Main()` to get a reference to the default application domain.

To demonstrate first-chance exception notifications in the application domain

1. Create a `Worker` object in the application domain that you created in the previous procedure. The `Worker` class must be public, and must derive from MarshalByRefObject, as shown in the complete example at the end of this article.

```
Worker w = (Worker) ad.CreateInstanceAndUnwrap(
                        typeof(Worker).Assembly.FullName, "Worker");
```

```
Dim w As Worker = CType(ad.CreateInstanceAndUnwrap(
                            GetType(Worker).Assembly.FullName, "Worker"),
                        Worker)
```

2. Call a method of the `Worker` object that throws an exception. In this example, the `Thrower` method is called twice. The first time, the method argument is `true`, which causes the method to catch its own exception. The second time, the argument is `false`, and the `Main()` method catches the exception in the default application domain.

```
// The worker throws an exception and catches it.
w.Thrower(true);

try
{
    // The worker throws an exception and doesn't catch it.
    w.Thrower(false);
}
catch (ArgumentException ex)
{
    Console.WriteLine("ArgumentException caught in {0}: {1}",
        AppDomain.CurrentDomain.FriendlyName, ex.Message);
}
```

```
' The worker throws an exception and catches it.
w.Thrower(true)

Try
    ' The worker throws an exception and doesn't catch it.
    w.Thrower(false)

Catch ex As ArgumentException

    Console.WriteLine("ArgumentException caught in {0}: {1}",
        AppDomain.CurrentDomain.FriendlyName, ex.Message)
End Try
```

3. Place code in the `Thrower` method to control whether the method handles its own exception.

```
if (catchException)
{
    try
    {
        throw new ArgumentException("Thrown in " + AppDomain.CurrentDomain.FriendlyName);
    }
    catch (ArgumentException ex)
    {
        Console.WriteLine("ArgumentException caught in {0}: {1}",
            AppDomain.CurrentDomain.FriendlyName, ex.Message);
    }
}
else
{
    throw new ArgumentException("Thrown in " + AppDomain.CurrentDomain.FriendlyName);
}
```

```
If catchException

    Try
        Throw New ArgumentException("Thrown in " & AppDomain.CurrentDomain.FriendlyName)

    Catch ex As ArgumentException

        Console.WriteLine("ArgumentException caught in {0}: {1}",
            AppDomain.CurrentDomain.FriendlyName, ex.Message)
    End Try
Else

    Throw New ArgumentException("Thrown in " & AppDomain.CurrentDomain.FriendlyName)
End If
```

Example

The following example creates an application domain named `AD1` and adds an event handler to the application domain's FirstChanceException event. The example creates an instance of the `Worker` class in the application domain, and calls a method named `Thrower` that throws an ArgumentException. Depending on the value of its argument, the method either catches the exception or fails to handle it.

Each time the `Thrower` method throws an exception in `AD1`, the FirstChanceException event is raised in `AD1`, and the event handler displays a message. The runtime then looks for an exception handler. In the first case, the exception handler is found in `AD1`. In the second case, the exception is unhandled in `AD1`, and instead is caught in the default application domain.

> **NOTE**
> The name of the default application domain is the same as the name of the executable.

If you add a handler for the FirstChanceException event to the default application domain, the event is raised and handled before the default application domain handles the exception. To see this, add the C# code `AppDomain.CurrentDomain.FirstChanceException += FirstChanceException;` (in Visual Basic, `AddHandler AppDomain.CurrentDomain.FirstChanceException, FirstChanceException`) at the beginning of `Main()`.

```
using System;
using System.Reflection;
using System.Runtime.ExceptionServices;

class Example
{
    static void Main()
    {
        // To receive first chance notifications of exceptions in
        // an application domain, handle the FirstChanceException
        // event in that application domain.
        AppDomain ad = AppDomain.CreateDomain("AD1");
        ad.FirstChanceException += FirstChanceHandler;

        // Create a worker object in the application domain.
        Worker w = (Worker) ad.CreateInstanceAndUnwrap(
                                typeof(Worker).Assembly.FullName, "Worker");

        // The worker throws an exception and catches it.
        w.Thrower(true);

        try
        {
            // The worker throws an exception and doesn't catch it.
```

```
            // The worker throws an exception and doesn't catch it.
            w.Thrower(false);
        }
        catch (ArgumentException ex)
        {
            Console.WriteLine("ArgumentException caught in {0}: {1}",
                AppDomain.CurrentDomain.FriendlyName, ex.Message);
        }
    }

    static void FirstChanceHandler(object source, FirstChanceExceptionEventArgs e)
    {
        Console.WriteLine("FirstChanceException event raised in {0}: {1}",
            AppDomain.CurrentDomain.FriendlyName, e.Exception.Message);
    }
}

public class Worker : MarshalByRefObject
{
    public void Thrower(bool catchException)
    {
        if (catchException)
        {
            try
            {
                throw new ArgumentException("Thrown in " + AppDomain.CurrentDomain.FriendlyName);
            }
            catch (ArgumentException ex)
            {
                Console.WriteLine("ArgumentException caught in {0}: {1}",
                    AppDomain.CurrentDomain.FriendlyName, ex.Message);
            }
        }
        else
        {
            throw new ArgumentException("Thrown in " + AppDomain.CurrentDomain.FriendlyName);
        }
    }
}

/* This example produces output similar to the following:

FirstChanceException event raised in AD1: Thrown in AD1
ArgumentException caught in AD1: Thrown in AD1
FirstChanceException event raised in AD1: Thrown in AD1
ArgumentException caught in Example.exe: Thrown in AD1
 */
```

```
Imports System.Reflection
Imports System.Runtime.ExceptionServices

Class Example
    Shared Sub Main()

        ' To receive first chance notifications of exceptions in
        ' an application domain, handle the FirstChanceException
        ' event in that application domain.
        Dim ad As AppDomain = AppDomain.CreateDomain("AD1")
        AddHandler ad.FirstChanceException, AddressOf FirstChanceHandler


        ' Create a worker object in the application domain.
        Dim w As Worker = CType(ad.CreateInstanceAndUnwrap(
                                    GetType(Worker).Assembly.FullName, "Worker"),
                                Worker)

        ' The worker throws an exception and catches it.
        w.Thrower(true)
```

```
        w.InitializeLifetimeService()

        Try
            ' The worker throws an exception and doesn't catch it.
            w.Thrower(false)

        Catch ex As ArgumentException

            Console.WriteLine("ArgumentException caught in {0}: {1}",
                AppDomain.CurrentDomain.FriendlyName, ex.Message)
        End Try
    End Sub

    Shared Sub FirstChanceHandler(ByVal source As Object,
                                  ByVal e As FirstChanceExceptionEventArgs)

        Console.WriteLine("FirstChanceException event raised in {0}: {1}",
            AppDomain.CurrentDomain.FriendlyName, e.Exception.Message)
    End Sub
End Class

Public Class Worker
    Inherits MarshalByRefObject

    Public Sub Thrower(ByVal catchException As Boolean)

        If catchException

            Try
                Throw New ArgumentException("Thrown in " & AppDomain.CurrentDomain.FriendlyName)

            Catch ex As ArgumentException

                Console.WriteLine("ArgumentException caught in {0}: {1}",
                    AppDomain.CurrentDomain.FriendlyName, ex.Message)
            End Try
        Else

            Throw New ArgumentException("Thrown in " & AppDomain.CurrentDomain.FriendlyName)
        End If
    End Sub
End Class

' This example produces output similar to the following:
'
'FirstChanceException event raised in AD1: Thrown in AD1
'ArgumentException caught in AD1: Thrown in AD1
'FirstChanceException event raised in AD1: Thrown in AD1
'ArgumentException caught in Example.exe: Thrown in AD1
```

Compiling the Code

- This example is a command-line application. To compile and run this code in Visual Studio 2010, add the C# code `Console.ReadLine();` (in Visual Basic, `Console.ReadLine()`) at the end of `Main()`, to prevent the command window from closing before you can read the output.

See Also

FirstChanceException

Resolving Assembly Loads

5/2/2018 • 5 min to read • Edit Online

The .NET Framework provides the AppDomain.AssemblyResolve event for applications that require greater control over assembly loading. By handling this event, your application can load an assembly into the load context from outside the normal probing paths, select which of several assembly versions to load, emit a dynamic assembly and return it, and so on. This topic provides guidance for handling the AssemblyResolve event.

> **NOTE**
>
> For resolving assembly loads in the reflection-only context, use the AppDomain.ReflectionOnlyAssemblyResolve event instead.

How the AssemblyResolve Event Works

When you register a handler for the AssemblyResolve event, the handler is invoked whenever the runtime fails to bind to an assembly by name. For example, calling the following methods from user code can cause the AssemblyResolve event to be raised:

- An AppDomain.Load method overload or Assembly.Load method overload whose first argument is a string that represents the display name of the assembly to load (that is, the string returned by the Assembly.FullName property).
- An AppDomain.Load method overload or Assembly.Load method overload whose first argument is an AssemblyName object that identifies the assembly to load.
- An Assembly.LoadWithPartialName method overload.
- An AppDomain.CreateInstance or AppDomain.CreateInstanceAndUnwrap method overload that instantiates an object in another application domain.

What the Event Handler Does

The handler for the AssemblyResolve event receives the display name of the assembly to be loaded, in the ResolveEventArgs.Name property. If the handler does not recognize the assembly name, it returns null (`Nothing` in Visual Basic, `nullptr` in Visual C++).

If the handler recognizes the assembly name, it can load and return an assembly that satisfies the request. The following list describes some sample scenarios.

- If the handler knows the location of a version of the assembly, it can load the assembly by using the Assembly.LoadFrom or Assembly.LoadFile method, and can return the loaded assembly if successful.
- If the handler has access to a database of assemblies stored as byte arrays, it can load a byte array by using one of the Assembly.Load method overloads that take a byte array.
- The handler can generate a dynamic assembly and return it.

NOTE

The handler must load the assembly into the load-from context, into the load context, or without context.If the handler loads the assembly into the reflection-only context by using the Assembly.ReflectionOnlyLoad or the Assembly.ReflectionOnlyLoadFrom method, the load attempt that raised the AssemblyResolve event fails.

It is the responsibility of the event handler to return a suitable assembly. The handler can parse the display name of the requested assembly by passing the ResolveEventArgs.Name property value to the AssemblyName(String) constructor. Beginning with the .NET Framework 4, the handler can use the ResolveEventArgs.RequestingAssembly property to determine whether the current request is a dependency of another assembly. This information can help identify an assembly that will satisfy the dependency.

The event handler can return a different version of the assembly than the version that was requested.

In most cases, the assembly that is returned by the handler appears in the load context, regardless of the context the handler loads it into. For example, if the handler uses the Assembly.LoadFrom method to load an assembly into the load-from context, the assembly appears in the load context when the handler returns it. However, in the following case the assembly appears without context when the handler returns it:

- The handler loads an assembly without context.
- The ResolveEventArgs.RequestingAssembly property is not null.
- The requesting assembly (that is, the assembly that is returned by the ResolveEventArgs.RequestingAssembly property) was loaded without context.

For information about contexts, see the Assembly.LoadFrom(String) method overload.

Multiple versions of the same assembly can be loaded into the same application domain. This practice is not recommended, because it can lead to type assignment problems. See Best Practices for Assembly Loading.

What the Event Handler Should Not Do

The primary rule for handling the AssemblyResolve event is that you should not try to return an assembly you do not recognize. When you write the handler, you should know which assemblies might cause the event to be raised. Your handler should return null for other assemblies.

IMPORTANT

Beginning with the .NET Framework 4, the AssemblyResolve event is raised for satellite assemblies. This change affects an event handler that was written for an earlier version of the .NET Framework, if the handler tries to resolve all assembly load requests. Event handlers that ignore assemblies they do not recognize are not affected by this change: They return null, and normal fallback mechanisms are followed.

When loading an assembly, the event handler must not use any of the AppDomain.Load or Assembly.Load method overloads that can cause the AssemblyResolve event to be raised recursively, because this can lead to a stack overflow. (See the list provided earlier in this topic.) This happens even if you provide exception handling for the load request, because no exception is thrown until all event handlers have returned. Thus, the following code results in a stack overflow if `MyAssembly` is not found:

```
using namespace System;
using namespace System::Reflection;

ref class Example
{
internal:
    static Assembly^ MyHandler(Object^ source, ResolveEventArgs^ e)
    {
        Console::WriteLine("Resolving {0}", e->Name);
        return Assembly::Load(e->Name);
    }
};

void main()
{
    AppDomain^ ad = AppDomain::CreateDomain("Test");
    ad->AssemblyResolve += gcnew ResolveEventHandler(&Example::MyHandler);

    try
    {
        Object^ obj = ad->CreateInstanceAndUnwrap(
            "MyAssembly, version=1.2.3.4, culture=neutral, publicKeyToken=null",
            "MyType");
    }
    catch (Exception^ ex)
    {
        Console::WriteLine(ex->Message);
    }
}

/* This example produces output similar to the following:

Resolving MyAssembly, Version=1.2.3.4, Culture=neutral, PublicKeyToken=null
Resolving MyAssembly, Version=1.2.3.4, Culture=neutral, PublicKeyToken=null
...
Resolving MyAssembly, Version=1.2.3.4, Culture=neutral, PublicKeyToken=null
Resolving MyAssembly, Version=1.2.3.4, Culture=neutral, PublicKeyToken=null

Process is terminated due to StackOverflowException.
 */
```

```
using System;
using System.Reflection;

class BadExample
{
    static void Main()
    {
        AppDomain ad = AppDomain.CreateDomain("Test");
        ad.AssemblyResolve += MyHandler;

        try
        {
            object obj = ad.CreateInstanceAndUnwrap(
                "MyAssembly, version=1.2.3.4, culture=neutral, publicKeyToken=null",
                "MyType");
        }
        catch (Exception ex)
        {
            Console.WriteLine(ex.Message);
        }
    }

    static Assembly MyHandler(object source, ResolveEventArgs e)
    {
        Console.WriteLine("Resolving {0}", e.Name);
        return Assembly.Load(e.Name);
    }
}

/* This example produces output similar to the following:

Resolving MyAssembly, Version=1.2.3.4, Culture=neutral, PublicKeyToken=null
Resolving MyAssembly, Version=1.2.3.4, Culture=neutral, PublicKeyToken=null
...
Resolving MyAssembly, Version=1.2.3.4, Culture=neutral, PublicKeyToken=null
Resolving MyAssembly, Version=1.2.3.4, Culture=neutral, PublicKeyToken=null

Process is terminated due to StackOverflowException.
 */
```

```
Imports System
Imports System.Reflection

Class BadExample

    Shared Sub Main()

        Dim ad As AppDomain = AppDomain.CreateDomain("Test")
        AddHandler ad.AssemblyResolve, AddressOf MyHandler

        Try
            Dim obj As object = ad.CreateInstanceAndUnwrap(
                "MyAssembly, version=1.2.3.4, culture=neutral, publicKeyToken=null",
                "MyType")
        Catch ex As Exception
            Console.WriteLine(ex.Message)
        End Try
    End Sub

    Shared Function MyHandler(ByVal source As Object, _
                              ByVal e As ResolveEventArgs) As Assembly
        Console.WriteLine("Resolving {0}", e.Name)
        Return Assembly.Load(e.Name)
    End Function
End Class

' This example produces output similar to the following:
'
'Resolving MyAssembly, Version=1.2.3.4, Culture=neutral, PublicKeyToken=null
'Resolving MyAssembly, Version=1.2.3.4, Culture=neutral, PublicKeyToken=null
'...
'Resolving MyAssembly, Version=1.2.3.4, Culture=neutral, PublicKeyToken=null
'Resolving MyAssembly, Version=1.2.3.4, Culture=neutral, PublicKeyToken=null
'
'Process is terminated due to StackOverflowException.
```

See Also

Best Practices for Assembly Loading
Using Application Domains

Programming with Assemblies

5/2/2018 • 2 min to read • Edit Online

Assemblies are the building blocks of the .NET Framework; they form the fundamental unit of deployment, version control, reuse, activation scoping, and security permissions. An assembly provides the common language runtime with the information it needs to be aware of type implementations. It is a collection of types and resources that are built to work together and form a logical unit of functionality. To the runtime, a type does not exist outside the context of an assembly.

This section describes how to create modules, create assemblies from modules, create a key pair and sign an assembly with a strong name, and install an assembly into the global assembly cache. In addition, this section describes how to use the MSIL Disassembler (Ildasm.exe) to view assembly manifest information.

> **NOTE**
>
> Starting with the .NET Framework version 2.0, the runtime will not load an assembly that was compiled with a version of the .NET Framework that has a higher version number than the currently loaded runtime. This applies to the combination of the major and minor components of the version number.

In This Section

Creating Assemblies
Provides an overview of single-file and multifile assemblies.

Assembly Names
Provides an overview of assembly naming.

How to: Determine an Assembly's Fully Qualified Name
Describes how to determine the fully qualified name of an assembly.

Running Intranet Applications in Full Trust
Describes how to specify legacy security policy for full-trust assemblies on an intranet share.

Assembly Location
Provides an overview of where to locate assemblies.

How to: Build a Single-File Assembly
Describes how to create a single-file assembly.

Multifile Assemblies
Describes reasons for creating multifile assemblies.

How to: Build a Multifile Assembly
Describes how to create a multifile assembly.

Setting Assembly Attributes
Describes assembly attributes and how to set them.

Creating and Using Strong-Named Assemblies
Describes how and why you sign an assembly with a strong name, and includes how-to topics.

Delay Signing an Assembly
Describes how to delay-sign an assembly.

Working with Assemblies and the Global Assembly Cache
Describes how and why you add assemblies to the global assembly cache, and includes how-to topics.

How to: View Assembly Contents
Describes how to use the MSIL Disassembler (Ildasm.exe) to view assembly contents.

Type Forwarding in the Common Language Runtime
Describes how to use type forwarding to move a type into a different assembly without breaking existing applications.

Reference

Assembly
The .NET Framework class that represents an assembly.

Related Sections

How to: Obtain Type and Member Information from an Assembly
Describes how to programmatically obtain type and other information from an assembly.

Assemblies in the Common Language Runtime
Provides a conceptual overview of common language runtime assemblies.

Assembly Versioning
Provides an overview of assembly binding and of the AssemblyVersionAttribute and AssemblyInformationalVersionAttribute attributes.

How the Runtime Locates Assemblies
Describes how the runtime determines which assembly to use to fulfill a binding request.

Reflection
Describes how to use the **Reflection** class to obtain information about an assembly.

Creating Assemblies

5/2/2018 • 1 min to read • Edit Online

You can create single-file or multifile assemblies using an IDE, such as Visual Studio 2005, or the compilers and tools provided by the Windows Software Development Kit (SDK). The simplest assembly is a single file that has a simple name and is loaded into a single application domain. This assembly cannot be referenced by other assemblies outside the application directory and does not undergo version checking. To uninstall the application made up of the assembly, you simply delete the directory where it resides. For many developers, an assembly with these features is all that is needed to deploy an application.

You can create a multifile assembly from several code modules and resource files. You can also create an assembly that can be shared by multiple applications. A shared assembly must have a strong name and can be deployed in the global assembly cache.

You have several options when grouping code modules and resources into assemblies, depending on the following factors:

- Versioning

  Group modules that should have the same version information.

- Deployment

  Group code modules and resources that support your model of deployment.

- Reuse

  Group modules if they can be logically used together for some purpose. For example, an assembly consisting of types and classes used infrequently for program maintenance can be put in the same assembly. In addition, types that you intend to share with multiple applications should be grouped into an assembly and the assembly should be signed with a strong name.

- Security

  Group modules containing types that require the same security permissions.

- Scoping

  Group modules containing types whose visibility should be restricted to the same assembly.

Special considerations must be made when making common language runtime assemblies available to unmanaged COM applications. For more information about working with unmanaged code, see Exposing .NET Framework Components to COM.

See Also

Programming with Assemblies
Assembly Versioning
How to: Build a Single-File Assembly
How to: Build a Multifile Assembly
How the Runtime Locates Assemblies
Multifile Assemblies

Assembly Names

5/2/2018 • 3 min to read • Edit Online

An assembly's name is stored in metadata and has a significant impact on the assembly's scope and use by an application. A strong-named assembly has a fully qualified name that includes the assembly's name, culture, public key, and version number. This is frequently referred to as the display name, and for loaded assemblies can be obtained by using the FullName property.

The runtime uses this information to locate the assembly and differentiate it from other assemblies with the same name. For example, a strong-named assembly called `myTypes` could have the following fully qualified name:

```
myTypes, Version=1.0.1234.0, Culture=en-US, PublicKeyToken=b77a5c561934e089c, ProcessorArchitecture=msil
```

NOTE

Processor architecture is added to the assembly identity in the .NET Framework version 2.0, to allow processor-specific versions of assemblies. You can create versions of an assembly whose identity differs only by processor architecture, for example 32-bit and 64-bit processor-specific versions. Processor architecture is not required for strong names. For more information, see AssemblyName.ProcessorArchitecture.

In this example, the fully qualified name indicates that the `myTypes` assembly has a strong name with a public key token, has the culture value for US English, and has a version number of 1.0.1234.0. Its processor architecture is "msil", which means that it will be just-in-time (JIT)-compiled to 32-bit code or 64-bit code depending on the operating system and processor.

Code that requests types in an assembly must use a fully qualified assembly name. This is called fully qualified binding. Partial binding, which specifies only an assembly name, is not permitted when referencing assemblies in the .NET Framework.

All assembly references to assemblies that make up the .NET Framework also must contain a fully qualified name of the assembly. For example, to reference the System.Data .NET Framework assembly for version 1.0 would include:

```
System.data, version=1.0.3300.0, Culture=neutral, PublicKeyToken=b77a5c561934e089
```

Note that the version corresponds to the version number of all .NET Framework assemblies that shipped with .NET Framework version 1.0. For .NET Framework assemblies, the culture value is always neutral, and the public key is the same as shown in the above example.

For example, to add an assembly reference in a configuration file to set up a trace listener, you would include the fully qualified name of the system .NET Framework assembly:

```
<add name="myListener" type="System.Diagnostics.TextWriterTraceListener, System, Version=1.0.3300.0,
Culture=neutral, PublicKeyToken=b77a5c561934e089" initializeData="c:\myListener.log" />
```

NOTE

The runtime treats assembly names as case-insensitive when binding to an assembly, but preserves whatever case is used in an assembly name. Several tools in the Windows Software Development Kit (SDK) handle assembly names as case-sensitive. For best results, manage assembly names as though they were case-sensitive.

Naming Application Components

The runtime does not consider the file name when determining an assembly's identity. The assembly identity, which consists of the assembly name, version, culture, and strong name, must be clear to the runtime.

For example, if you have an assembly called myAssembly.exe that references an assembly called myAssembly.dll, binding occurs correctly if you execute myAssembly.exe. However, if another application executes myAssembly.exe using the method AppDomain.ExecuteAssembly, the runtime determines that "myAssembly" is already loaded when myAssembly.exe requests binding to "myAssembly." In this case, myAssembly.dll is never loaded. Because myAssembly.exe does not contain the requested type , a TypeLoadException occurs.

To avoid this problem, make sure the assemblies that make up your application do not have the same assembly name or place assemblies with the same name in different directories.

NOTE

If you put a strong-named assembly in the global assembly cache, the assembly's file name must match the assembly name (not including the file name extension, such as .exe or .dll). For example, if the file name of an assembly is myAssembly.dll, the assembly name must be myAssembly. Private assemblies deployed only in the root application directory can have an assembly name that is different from the file name.

See Also

How to: Determine an Assembly's Fully Qualified Name
Creating Assemblies
Strong-Named Assemblies
Global Assembly Cache
How the Runtime Locates Assemblies
Programming with Assemblies

How to: Determine an Assembly's Fully Qualified Name

5/2/2018 • 2 min to read • Edit Online

To discover the fully qualified name of an assembly in the global assembly cache, use the Global Assembly Cache Tool (Gacutil.exe). See How to: View the Contents of the Global Assembly Cache.

For assemblies that are not in the global assembly cache, you can get the fully qualified assembly name in a number of ways: can use code to output the information to the console or to a variable, or you can use the Ildasm.exe (IL Disassembler) to examine the assembly's metadata, which contains the fully qualified name.

- If the assembly is already loaded by the application, you can retrieve the value of the Assembly.FullName property to get the fully qualified name. You can use this approach whether or not the assembly is in the GAC. The example provides an illustration.
- If you know the assembly's file system path, you can call the static (`Shared` in Visual Basic) AssemblyName.GetAssemblyName method to get the fully qualified assembly name. The following is a simple example.

```
using System;
using System.Reflection;

public class Example
{
   public static void Main()
   {
      Console.WriteLine(AssemblyName.GetAssemblyName(@".\UtilityLibrary.dll"));
   }
}
// The example displays output like the following:
//   UtilityLibrary, Version=1.1.0.0, Culture=neutral, PublicKeyToken=null
```

```
Imports System.Reflection

Public Module Example
   Public Sub Main
      Console.WriteLine(AssemblyName.GetAssemblyName(".\UtilityLibrary.dll"))
   End Sub
End Module
' The example displays output like the following:
'   UtilityLibrary, Version=1.1.0.0, Culture=neutral, PublicKeyToken=null
```

- You can use the Ildasm.exe (IL Disassembler) to examine the assembly's metadata, which contains the fully qualified name.

For more information about setting assembly attributes such as version, culture, and assembly name, see Setting Assembly Attributes. For more information about giving an assembly a strong name, see Creating and Using Strong-Named Assemblies.

Example

The following code example shows how to display the fully qualified name of an assembly containing a specified class to the console. Because it retrieves the name of an assembly that the app has already loaded, it does not

matter whether the assembly is in the GAC.

```
#using <System.dll>
#using <System.Data.dll>

using namespace System;
using namespace System::Reflection;

ref class asmname
{
public:
    static void Main()
    {
        Type^ t = System::Data::DataSet::typeid;
        String^ s = t->Assembly->FullName->ToString();
        Console::WriteLine("The fully qualified assembly name " +
            "containing the specified class is {0}.", s);
    }
};

int main()
{
    asmname::Main();
}
```

```
using System;
using System.Reflection;

class asmname
{
    public static void Main()
    {
        Type t = typeof(System.Data.DataSet);
        string s = t.Assembly.FullName.ToString();
        Console.WriteLine("The fully qualified assembly name " +
            "containing the specified class is {0}.", s);
    }
}
```

```
Imports System
Imports System.Reflection

Class asmname
    Public Shared Sub Main()
        Dim t As Type = GetType(System.Data.DataSet)
        Dim s As String = t.Assembly.FullName.ToString()
        Console.WriteLine("The fully qualified assembly name " +
            "containing the specified class is {0}.", s)
    End Sub
End Class
```

See Also

Assembly Names
Creating Assemblies
Creating and Using Strong-Named Assemblies
Global Assembly Cache
How the Runtime Locates Assemblies
Programming with Assemblies

Running Intranet Applications in Full Trust

5/2/2018 • 1 min to read • Edit Online

Starting with the .NET Framework version 3.5 Service Pack 1 (SP1), applications and their library assemblies can be run as full-trust assemblies from a network share. MyComputer zone evidence is automatically added to assemblies that are loaded from a share on the intranet. This evidence gives those assemblies the same grant set (which is typically full trust) as the assemblies that reside on the computer. This functionality does not apply to ClickOnce applications or to applications that are designed to run on a host.

Rules for Library Assemblies

The following rules apply to assemblies that are loaded by an executable on a network share:

- Library assemblies must reside in the same folder as the executable assembly. Assemblies that reside in a subfolder or are referenced on a different path are not given the full-trust grant set.
- If the executable delay-loads an assembly, it must use the same path that was used to start the executable. For example, if the share *network-computer**share* is mapped to a drive letter and the executable is run from that path, assemblies that are loaded by the executable by using the network path will not be granted full trust. To delay-load an assembly in the MyComputer zone, the executable must use the drive letter path.

Restoring the Former Intranet Policy

In earlier versions of the .NET Framework, shared assemblies were granted Intranet zone evidence. You had to specify code access security policy to grant full trust to an assembly on a share.

This new behavior is the default for intranet assemblies. You can return to the earlier behavior of providing Intranet evidence by setting a registry key that applies to all applications on the computer. This process is different for 32-bit and 64-bit computers:

- On 32-bit computers, create a subkey under the HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\.NETFramework key in the system registry. Use the key name LegacyMyComputerZone with a DWORD value of 1.
- On 64-bit computers, create a subkey under the HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\.NETFramework key in the system registry. Use the key name LegacyMyComputerZone with a DWORD value of 1. Create the same subkey under the HKEY_LOCAL_MACHINE\SOFTWARE\Wow6432Node\Microsoft\.NETFramework key.

See Also

Programming with Assemblies

Assembly Location

5/2/2018 • 1 min to read • Edit Online

An assembly's location determines whether the common language runtime can locate it when referenced, and can also determine whether the assembly can be shared with other assemblies. You can deploy an assembly in the following locations:

- The application's directory or subdirectories.

  This is the most common location for deploying an assembly. The subdirectories of an application's root directory can be based on language or culture. If an assembly has information in the culture attribute, it must be in a subdirectory under the application directory with that culture's name.

- The global assembly cache.

  This is a machine-wide code cache that is installed wherever the common language runtime is installed. In most cases, if you intend to share an assembly with multiple applications, you should deploy it into the global assembly cache.

- On an HTTP server.

  An assembly deployed on an HTTP server must have a strong name; you point to the assembly in the codebase section of the application's configuration file.

See Also

Creating Assemblies
Global Assembly Cache
How the Runtime Locates Assemblies
Programming with Assemblies

How to: Build a Single-File Assembly

5/2/2018 • 1 min to read • Edit Online

A single-file assembly, which is the simplest type of assembly, contains type information and implementation, as well as the assembly manifest. You can use command-line compilers or Visual Studio 2005 to create a single-file assembly. By default, the compiler creates an assembly file with an .exe extension.

> **NOTE**
>
> Visual Studio 2005 for C# and Visual Basic can be used only to create single-file assemblies. If you want to create multifile assemblies, you must use command-line compilers or Visual Studio 2005 for Visual C++.

The following procedures show how to create single-file assemblies using command-line compilers.

To create an assembly with an .exe extension

1. At the command prompt, type the following command:

   <compiler command> <module name>

   In this command, *compiler command* is the compiler command for the language used in your code module, and *module name* is the name of the code module to compile into the assembly.

The following example creates an assembly named `myCode.exe` from a code module called `myCode`.

```
csc myCode.cs
```

```
vbc myCode.vb
```

To create an assembly with an .exe extension and specify the output file name

1. At the command prompt, type the following command:

   <compiler command> **/out:***<file name> <module name>*

   In this command, *compiler command* is the compiler command for the language used in your code module, *file name* is the output file name, and *module name* is the name of the code module to compile into the assembly.

The following example creates an assembly named `myAssembly.exe` from a code module called `myCode`.

```
csc -out:myAssembly.exe myCode.cs
```

```
vbc -out:myAssembly.exe myCode.vb
```

Creating Library Assemblies

A library assembly is similar to a class library. It contains types that will be referenced by other assemblies, but it has no entry point to begin execution.

To create a library assembly

1. At the command prompt, type the following command:

   <compiler command> **-t:library** *<module name>*

   In this command, *compiler command* is the compiler command for the language used in your code module, and *module name* is the name of the code module to compile into the assembly. You can also use other compiler options, such as the **-out:** option.

The following example creates a library assembly named `myCodeAssembly.dll` from a code module called `myCode`.

```
csc -out:myCodeLibrary.dll -t:library myCode.cs
```

```
vbc -out:myCodeLibrary.dll -t:library myCode.vb
```

See Also

Creating Assemblies
Multifile Assemblies
How to: Build a Multifile Assembly
Programming with Assemblies

Multifile Assemblies

5/2/2018 • 1 min to read • Edit Online

You can create multifile assemblies using command-line compilers or Visual Studio 2005 with Visual C++. One file in the assembly must contain the assembly manifest. An assembly that starts an application must also contain an entry point, such as a Main or WinMain method.

For example, suppose you have an application that contains two code modules, Client.cs and Stringer.cs. Stringer.cs creates the `myStringer` namespace that is referenced by the code in Client.cs. Client.cs contains the `Main` method, which is the application's entry point. In this example, you compile the two code modules, and then create a third file that contains the assembly manifest, which launches the application. The assembly manifest references both the `Client` and `Stringer` modules.

> **NOTE**
> Multifile assemblies can have only one entry point, even if the assembly has multiple code modules.

There are several reasons you might want to create a multifile assembly:

- To combine modules written in different languages. This is the most common reason for creating a multifile assembly.
- To optimize downloading an application by putting seldom-used types in a module that is downloaded only when needed.

  > **NOTE**
  > If you are creating applications that will be downloaded using the `<object>` tag with Microsoft Internet Explorer, it is important that you create multifile assemblies. In this scenario, you create a file separate from your code modules that contains only the assembly manifest. Internet Explorer downloads the assembly manifest first, and then creates worker threads to download any additional modules or assemblies required. While the file containing the assembly manifest is being downloaded, Internet Explorer will be unresponsive to user input. The smaller the file containing the assembly manifest, the less time Internet Explorer will be unresponsive.

- To combine code modules written by several developers. Although each developer can compile each code module into an assembly, this can force some types to be exposed publicly that are not exposed if all modules are put into a multifile assembly.

Once you create the assembly, you can sign the file that contains the assembly manifest (and hence the assembly), or you can give the file (and the assembly) a strong name and put it in the global assembly cache.

See Also

How to: Build a Multifile Assembly
Programming with Assemblies

How to: Build a Multifile Assembly

5/2/2018 • 4 min to read • Edit Online

This article explains how to create a multifile assembly and provides code that illustrates each step in the procedure.

> **NOTE**
> The Visual Studio IDE for C# and Visual Basic can only be used to create single-file assemblies. If you want to create multifile assemblies, you must use the command-line compilers or Visual Studio with Visual C++.

To create a multifile assembly

1. Compile all files that contain namespaces referenced by other modules in the assembly into code modules. The default extension for code modules is .netmodule.

   For example, let's say the `Stringer` file has a namespace called `myStringer`, which includes a class called `Stringer`. The `Stringer` class contains a method called `StringerMethod` that writes a single line to the console.

```
// Assembly building example in the .NET Framework.
using namespace System;

namespace myStringer
{
    public ref class Stringer
    {
    public:
        void StringerMethod()
        {
            System::Console::WriteLine("This is a line from StringerMethod.");
        }
    };
}
```

```
// Assembly building example in the .NET Framework.
using System;

namespace myStringer
{
    public class Stringer
    {
        public void StringerMethod()
        {
            System.Console.WriteLine("This is a line from StringerMethod.");
        }
    }
}
```

```
' Assembly building example in the .NET Framework.
Imports System

Namespace myStringer
    Public Class Stringer
        Public Sub StringerMethod()
            System.Console.WriteLine("This is a line from StringerMethod.")
        End Sub
    End Class
End Namespace
```

Use the following command to compile this code:

```
cl /clr:pure /LN Stringer.cpp
```

```
csc /t:module Stringer.cs
```

```
vbc /t:module Stringer.vb
```

Specifying the *module* parameter with the **/t:** compiler option indicates that the file should be compiled as a module rather than as an assembly. The compiler produces a module called `Stringer.netmodule`, which can be added to an assembly.

2. Compile all other modules, using the necessary compiler options to indicate the other modules that are referenced in the code. This step uses the **/addmodule** compiler option.

   In the following example, a code module called `Client` has an entry point `Main` method that references a method in the `Stringer.dll` module created in step 1.

```
#using "Stringer.netmodule"

using namespace System;
using namespace myStringer; //The namespace created in Stringer.netmodule.

ref class MainClientApp
{
    // Static method Main is the entry point method.
public:
    static void Main()
    {
        Stringer^ myStringInstance = gcnew Stringer();
        Console::WriteLine("Client code executes");
        myStringInstance->StringerMethod();
    }
};

int main()
{
    MainClientApp::Main();
}
```

```
using System;
using myStringer;

class MainClientApp
{
    // Static method Main is the entry point method.
    public static void Main()
    {
        Stringer myStringInstance = new Stringer();
        Console.WriteLine("Client code executes");
        myStringInstance.StringerMethod();
    }
}
```

```
Imports System
Imports myStringer

Class MainClientApp
    ' Static method Main is the entry point method.
    Public Shared Sub Main()
        Dim myStringInstance As New Stringer()
        Console.WriteLine("Client code executes")
        myStringInstance.StringerMethod()
    End Sub
End Class
```

Use the following command to compile this code:

```
cl /clr:pure /FUStringer.netmodule /LN Client.cpp
```

```
csc /addmodule:Stringer.netmodule /t:module Client.cs
```

```
vbc /addmodule:Stringer.netmodule /t:module Client.vb
```

Specify the **/t:module** option because this module will be added to an assembly in a future step. Specify the **/addmodule** option because the code in `Client` references a namespace created by the code in `Stringer.netmodule`. The compiler produces a module called `Client.netmodule` that contains a reference to another module, `Stringer.netmodule`.

NOTE

The C# and Visual Basic compilers support directly creating multifile assemblies using the following two different syntaxes.

- Two compilations create a two-file assembly:

```
cl /clr:pure /LN Stringer.cpp
cl /clr:pure Client.cpp /link /ASSEMBLYMODULE:Stringer.netmodule
```

```
csc /t:module Stringer.cs
csc Client.cs /addmodule:Stringer.netmodule
```

```
vbc /t:module Stringer.vb
vbc Client.vb /addmodule:Stringer.netmodule
```

  - One compilation creates a two-file assembly:

```
cl /clr:pure /LN Stringer.cpp
cl /clr:pure Client.cpp /link /ASSEMBLYMODULE:Stringer.netmodule
```

```
csc /out:Client.exe Client.cs /out:Stringer.netmodule Stringer.cs
```

```
vbc /out:Client.exe Client.vb /out:Stringer.netmodule Stringer.vb
```

3. Use the Assembly Linker (Al.exe) to create the output file that contains the assembly manifest. This file contains reference information for all modules or resources that are part of the assembly.

   At the command prompt, type the following command:

   al <*module name*> <*module name*> ... **/main:**<*method name*> **/out:**<*file name*> **/target:**<*assembly file type*>

   In this command, the *module name* arguments specify the name of each module to include in the assembly. The **/main:** option specifies the method name that is the assembly's entry point. The **/out:** option specifies the name of the output file, which contains assembly metadata. The **/target:** option specifies that the assembly is a console application executable (.exe) file, a Windows executable (.win) file, or a library (.lib) file.

   In the following example, Al.exe creates an assembly that is a console application executable called `myAssembly.exe`. The application consists of two modules called `Client.netmodule` and `Stringer.netmodule`, and the executable file called `myAssembly.exe,` which contains only assembly metadata . The entry point of the assembly is the `Main` method in the class `MainClientApp`, which is located in `Client.dll`.

```
al Client.netmodule Stringer.netmodule /main:MainClientApp.Main /out:myAssembly.exe /target:exe
```

   You can use the MSIL Disassembler (Ildasm.exe) to examine the contents of an assembly, or determine whether a file is an assembly or a module.

See Also

Creating Assemblies

How to: View Assembly Contents
How the Runtime Locates Assemblies
Multifile Assemblies

Setting Assembly Attributes

5/2/2018 • 3 min to read • Edit Online

Assembly attributes are values that provide information about an assembly. The attributes are divided into the following sets of information:

- Assembly identity attributes.
- Informational attributes.
- Assembly manifest attributes.
- Strong name attributes.

Assembly Identity Attributes

Three attributes, together with a strong name (if applicable), determine the identity of an assembly: name, version, and culture. These attributes form the full name of the assembly and are required when referencing the assembly in code. You can use attributes to set an assembly's version and culture. The compiler or the Assembly Linker (Al.exe) sets the name value when the assembly is created, based on the file containing the assembly manifest.

The following table describes the version and culture attributes.

ASSEMBLY IDENTITY ATTRIBUTE	DESCRIPTION
AssemblyCultureAttribute	Enumerated field indicating the culture that the assembly supports. An assembly can also specify culture independence, indicating that it contains the resources for the default culture. **Note:** The runtime treats any assembly that does not have the culture attribute set to null as a satellite assembly. Such assemblies are subject to satellite assembly binding rules. For more information, see How the Runtime Locates Assemblies.
AssemblyFlagsAttribute	Value that sets assembly attributes, such as whether the assembly can be run side by side.
AssemblyVersionAttribute	Numeric value in the format *major.minor.build.revision* (for example, 2.4.0.0). The common language runtime uses this value to perform binding operations in strong-named assemblies. **Note:** If the AssemblyInformationalVersionAttribute attribute is not applied to an assembly, the version number specified by the AssemblyVersionAttribute attribute is used by the Application.ProductVersion, Application.UserAppDataPath, and Application.UserAppDataRegistry properties.

The following code example shows how to apply the version and culture attributes to an assembly.

```
// Set version number for the assembly.
[assembly:AssemblyVersionAttribute("4.3.2.1")];
// Set culture as German.
[assembly:AssemblyCultureAttribute("de")];
```

```
// Set version number for the assembly.
[assembly:AssemblyVersionAttribute("4.3.2.1")]
// Set culture as German.
[assembly:AssemblyCultureAttribute("de")]
```

```
' Set version number for the assembly.
<Assembly:AssemblyVersionAttribute("4.3.2.1")>
' Set culture as German.
<Assembly:AssemblyCultureAttribute("de")>
```

Informational Attributes

You can use informational attributes to provide additional company or product information for an assembly. The following table describes the informational attributes you can apply to an assembly.

INFORMATIONAL ATTRIBUTE	DESCRIPTION
AssemblyCompanyAttribute	String value specifying a company name.
AssemblyCopyrightAttribute	String value specifying copyright information.
AssemblyFileVersionAttribute	String value specifying the Win32 file version number. This normally defaults to the assembly version.
AssemblyInformationalVersionAttribute	String value specifying version information that is not used by the common language runtime, such as a full product version number. **Note:** If this attribute is applied to an assembly, the string it specifies can be obtained at run time by using the Application.ProductVersion property. The string is also used in the path and registry key provided by the Application.UserAppDataPath and Application.UserAppDataRegistry properties.
AssemblyProductAttribute	String value specifying product information.
AssemblyTrademarkAttribute	String value specifying trademark information.

These attributes can appear on the Windows Properties page of the assembly, or they can be overridden using the **/win32res** compiler option to specify your own Win32 resource file.

Assembly Manifest Attributes

You can use assembly manifest attributes to provide information in the assembly manifest, including title, description, the default alias, and configuration. The following table describes the assembly manifest attributes.

ASSEMBLY MANIFEST ATTRIBUTE	DESCRIPTION
AssemblyConfigurationAttribute	String value indicating the configuration of the assembly, such as Retail or Debug. The runtime does not use this value.

ASSEMBLY MANIFEST ATTRIBUTE	DESCRIPTION
AssemblyDefaultAliasAttribute	String value specifying a default alias to be used by referencing assemblies. This value provides a friendly name when the name of the assembly itself is not friendly (such as a GUID value). This value can also be used as a short form of the full assembly name.
AssemblyDescriptionAttribute	String value specifying a short description that summarizes the nature and purpose of the assembly.
AssemblyTitleAttribute	String value specifying a friendly name for the assembly. For example, an assembly named comdlg might have the title Microsoft Common Dialog Control.

Strong Name Attributes

You can use strong name attributes to set a strong name for an assembly. The following table describes the strong name attributes.

STRONG NAME ATTRIBUTES	DESCRIPTION
AssemblyDelaySignAttribute	Boolean value indicating that delay signing is being used.
AssemblyKeyFileAttribute	String value indicating the name of the file that contains either the public key (if using delay signing) or both the public and private keys passed as a parameter to the constructor of this attribute. Note that the file name is relative to the output file path (the .exe or .dll), not the source file path.
AssemblyKeyNameAttribute	Indicates the key container that contains the key pair passed as a parameter to the constructor of this attribute.

The following code example shows the attributes to apply when using delay signing to create a strong-named assembly with a public key file called `myKey.snk`.

```
[assembly:AssemblyKeyFileAttribute("myKey.snk")];
[assembly:AssemblyDelaySignAttribute(true)];
```

```
[assembly:AssemblyKeyFileAttribute("myKey.snk")]
[assembly:AssemblyDelaySignAttribute(true)]
```

```
<Assembly:AssemblyKeyFileAttribute("myKey.snk")>
<Assembly:AssemblyDelaySignAttribute(True)>
```

See Also

Creating Assemblies
Programming with Assemblies

Creating and Using Strong-Named Assemblies

5/2/2018 • 3 min to read • Edit Online

A strong name consists of the assembly's identity—its simple text name, version number, and culture information (if provided)—plus a public key and a digital signature. It is generated from an assembly file using the corresponding private key. (The assembly file contains the assembly manifest, which contains the names and hashes of all the files that make up the assembly.)

A strong-named assembly can only use types from other strong-named assemblies. Otherwise, the integrity of the strong-named assembly would be compromised.

This overview contains the following sections:

Strong Name Scenario

The following scenario outlines the process of signing an assembly with a strong name and later referencing it by that name.

1. Assembly A is created with a strong name using one of the following methods:
   - Using a development environment that supports creating strong names, such as Visual Studio 2005.
   - Creating a cryptographic key pair using the Strong Name tool (Sn.exe) and assigning that key pair to the assembly using either a command-line compiler or the Assembly Linker (Al.exe). The Windows Software Development Kit (SDK) provides both Sn.exe and Al.exe.
2. The development environment or tool signs the hash of the file containing the assembly's manifest with the developer's private key. This digital signature is stored in the portable executable (PE) file that contains Assembly A's manifest.
3. Assembly B is a consumer of Assembly A. The reference section of Assembly B's manifest includes a token that represents Assembly A's public key. A token is a portion of the full public key and is used rather than the key itself to save space.
4. The common language runtime verifies the strong name signature when the assembly is placed in the global assembly cache. When binding by strong name at run time, the common language runtime compares the key stored in Assembly B's manifest with the key used to generate the strong name for Assembly A. If the .NET Framework security checks pass and the bind succeeds, Assembly B has a guarantee that Assembly A's bits have not been tampered with and that these bits actually come from the developers of Assembly A.

> **NOTE**
>
> This scenario doesn't address trust issues. Assemblies can carry full Microsoft Authenticode signatures in addition to a strong name. Authenticode signatures include a certificate that establishes trust. It's important to note that strong names don't require code to be signed in this way. Strong names only provide a unique identity.

Back to top

Bypassing Signature Verification of Trusted Assemblies

Starting with the .NET Framework 3.5 Service Pack 1, strong-name signatures are not validated when an assembly is loaded into a full-trust application domain, such as the default application domain for the `MyComputer` zone. This is referred to as the strong-name bypass feature. In a full-trust environment, demands for StrongNameIdentityPermission always succeed for signed, full-trust assemblies, regardless of their signature. The strong-name bypass feature avoids the unnecessary overhead of strong-name signature verification of full-trust assemblies in this situation, allowing the assemblies to load faster.

The bypass feature applies to any assembly that is signed with a strong name and that has the following characteristics:

- Fully trusted without StrongName evidence (for example, has `MyComputer` zone evidence).
- Loaded into a fully trusted AppDomain.
- Loaded from a location under the ApplicationBase property of that AppDomain.
- Not delay-signed.

This feature can be disabled for individual applications or for a computer. See How to: Disable the Strong-Name Bypass Feature.

Back to top

Related Topics

TITLE	DESCRIPTION
How to: Create a Public-Private Key Pair	Describes how to create a cryptographic key pair for signing an assembly.
How to: Sign an Assembly with a Strong Name	Describes how to create a strong-named assembly.
Enhanced Strong Naming	Describes enhancements to strong-names in the .NET Framework 4.5.
How to: Reference a Strong-Named Assembly	Describes how to reference types or resources in a strong-named assembly at compile time or run time.
How to: Disable the Strong-Name Bypass Feature	Describes how to disable the feature that bypasses the validation of strong-name signatures. This feature can be disabled for all or for specific applications.
Creating Assemblies	Provides an overview of single-file and multifile assemblies.
How to Delay Sign an Assembly in Visual Studio	Explains how to sign an assembly with a strong name after the assembly has been created.
Sn.exe (Strong Name Tool)	Describes the tool included in the .NET Framework that helps create assemblies with strong names. This tool provides options for key management, signature generation, and signature verification.

TITLE	DESCRIPTION
Al.exe (Assembly Linker)	Describes the tool included in the .NET Framework that generates a file that has an assembly manifest from modules or resource files.

How to: Create a Public-Private Key Pair

5/2/2018 • 1 min to read • Edit Online

To sign an assembly with a strong name, you must have a public/private key pair. This public and private cryptographic key pair is used during compilation to create a strong-named assembly. You can create a key pair using the Strong Name tool (Sn.exe). Key pair files usually have an .snk extension.

> **NOTE**
>
> In Visual Studio, the C# and Visual Basic project property pages include a **Signing** tab that enables you to select existing key files or to generate new key files without using Sn.exe. In Visual C++, you can specify the location of an existing key file in the **Advanced** property page in the **Linker** section of the **Configuration Properties** section of the **Property Pages** window. The use of the AssemblyKeyFileAttribute attribute to identify key file pairs has been made obsolete beginning with Visual Studio 2005.

To create a key pair

1. At the command prompt, type the following command:

   sn –k *<file name>*

   In this command, *file name* is the name of the output file containing the key pair.

The following example creates a key pair called `sgKey.snk`.

```
sn -k sgKey.snk
```

If you intend to delay sign an assembly and you control the whole key pair (which is unlikely outside test scenarios), you can use the following commands to generate a key pair and then extract the public key from it into a separate file. First, create the key pair:

```
sn -k keypair.snk
```

Next, extract the public key from the key pair and copy it to a separate file:

```
sn -p keypair.snk public.snk
```

Once you create the key pair, you must put the file where the strong name signing tools can find it.

When signing an assembly with a strong name, the Assembly Linker (Al.exe) looks for the key file relative to the current directory and to the output directory. When using command-line compilers, you can simply copy the key to the current directory containing your code modules.

If you are using an earlier version of Visual Studio that does not have a **Signing** tab in the project properties, the recommended key file location is the project directory with the file attribute specified as follows:

```
[assembly:AssemblyKeyFileAttribute("keyfile.snk")];
```

```
[assembly:AssemblyKeyFileAttribute("keyfile.snk")]
```

```
<Assembly:AssemblyKeyFileAttribute("keyfile.snk")>
```

See Also

Creating and Using Strong-Named Assemblies

How to: Sign an Assembly with a Strong Name

5/2/2018 • 3 min to read • Edit Online

There are a number of ways to sign an assembly with a strong name:

- By using the **Signing** tab in a project's **Properties** dialog box in Visual Studio. This is the easiest and most convenient way to sign an assembly with a strong name.
- By using the Assembly Linker (Al.exe) to link a .NET Framework code module (a .netmodule file) with a key file.
- By using assembly attributes to insert the strong name information into your code. You can use either the AssemblyKeyFileAttribute or the AssemblyKeyNameAttribute attribute, depending on where the key file to be used is located.
- By using compiler options.

You must have a cryptographic key pair to sign an assembly with a strong name. For more information about creating a key pair, see How to: Create a Public-Private Key Pair.

To create and sign an assembly with a strong name by using Visual Studio

1. In **Solution Explorer**, open the shortcut menu for the project, and then choose **Properties**.
2. Choose the **Signing** tab.
3. Select the **Sign the assembly** box.
4. In the **Choose a strong name key file** box, choose **<Browse...>**, and then navigate to the key file. To create a new key file, choose **<New...>** and enter its name in the **Create Strong Name Key** dialog box.

To create and sign an assembly with a strong name by using the Assembly Linker

- At the Visual Studio Command Prompt, type the following command:

  al /out:<*assemblyName*> <*moduleName*> **/keyfile:**<*keyfileName*>

  where:

  assemblyName
  The name of the strongly signed assembly (a .dll or .exe file) that Assembly Linker will emit.

  moduleName
  The name of a .NET Framework code module (a .netmodule file) that includes one or more types. You can create a .netmodule file by compiling your code with the `/target:module` switch in C# or Visual Basic.

  keyfileName
  The name of the container or file that contains the key pair. Assembly Linker interprets a relative path in relationship to the current directory.

The following example signs the assembly `MyAssembly.dll` with a strong name by using the key file `sgKey.snk`.

```
al /out:MyAssembly.dll MyModule.netmodule /keyfile:sgKey.snk
```

For more information about this tool, see Assembly Linker.

To sign an assembly with a strong name by using attributes

1. Add the System.Reflection.AssemblyKeyFileAttribute or AssemblyKeyNameAttribute attribute to your source code file, and specify the name of the file or container that contains the key pair to use when signing the assembly with a strong name.

2. Compile the source code file normally.

> **NOTE**
> The C# and Visual Basic compilers issue compiler warnings (CS1699 and BC41008, respectively) when they encounter the AssemblyKeyFileAttribute or AssemblyKeyNameAttribute attribute in source code. You can ignore the warnings.

The following example uses the AssemblyKeyFileAttribute attribute with a key file called `keyfile.snk`, which is located in the directory where the assembly is compiled.

```
[assembly:AssemblyKeyFileAttribute("keyfile.snk")];
```

```
[assembly:AssemblyKeyFileAttribute("keyfile.snk")]
```

```
<Assembly:AssemblyKeyFileAttribute("keyfile.snk")>
```

You can also delay sign an assembly when compiling your source file. For more information, see Delay Signing an Assembly.

To sign an assembly with a strong name by using the compiler

- Compile your source code file or files with the `/keyfile` or `/delaysign` compiler option in C# and Visual Basic, or the `/KEYFILE` or `/DELAYSIGN` linker option in C++. After the option name, add a colon and the name of the key file. When using command-line compilers, you can copy the key file to the directory that contains your source code files.

  For information on delay signing, see Delay Signing an Assembly.

  The following example uses the C# compiler and signs the assembly `UtilityLibrary.dll` with a strong name by using the key file `sgKey.snk`.

  ```
  csc /t:library UtilityLibrary.cs /keyfile:sgKey.snk
  ```

See Also

Creating and Using Strong-Named Assemblies
How to: Create a Public-Private Key Pair
Al.exe (Assembly Linker)
Delay Signing an Assembly
Managing Assembly and Manifest Signing
Signing Page, Project Designer

Enhanced Strong Naming

5/2/2018 • 3 min to read • Edit Online

A strong name signature is an identity mechanism in the .NET Framework for identifying assemblies. It is a public-key digital signature that is typically used to verify the integrity of data being passed from an originator (signer) to a recipient (verifier). This signature is used as a unique identity for an assembly and ensures that references to the assembly are not ambiguous. The assembly is signed as part of the build process and then verified when it is loaded.

Strong name signatures help prevent malicious parties from tampering with an assembly and then re-signing the assembly with the original signer's key. However, strong name keys don't contain any reliable information about the publisher, nor do they contain a certificate hierarchy. A strong name signature does not guarantee the trustworthiness of the person who signed the assembly or indicate whether that person was a legitimate owner of the key; it indicates only that the owner of the key signed the assembly. Therefore, we do not recommend using a strong name signature as a security validator for trusting third-party code. Microsoft Authenticode is the recommended way to authenticate code.

Limitations of Conventional Strong Names

The strong naming technology used in versions before the .NET Framework 4.5 has the following shortcomings:

- Keys are constantly under attack, and improved techniques and hardware make it easier to infer a private key from a public key. To guard against attacks, larger keys are necessary. .NET Framework versions before the .NET Framework 4.5 provide the ability to sign with any size key (the default size is 1024 bits), but signing an assembly with a new key breaks all binaries that reference the older identity of the assembly. Therefore, it is extremely difficult to upgrade the size of a signing key if you want to maintain compatibility.
- Strong name signing supports only the SHA-1 algorithm. SHA-1 has recently been found to be inadequate for secure hashing applications. Therefore, a stronger algorithm (SHA-256 or greater) is necessary. It is possible that SHA-1 will lose its FIPS-compliant standing, which would present problems for those who choose to use only FIPS-compliant software and algorithms.

Advantages of Enhanced Strong Names

The main advantages of enhanced strong names are compatibility with pre-existing strong names and the ability to claim that one identity is equivalent to another:

- Developers who have pre-existing signed assemblies can migrate their identities to the SHA-2 algorithms while maintaining compatibility with assemblies that reference the old identities.
- Developers who create new assemblies and are not concerned with pre-existing strong name signatures can use the more secure SHA-2 algorithms and sign the assemblies as they always have.

Using Enhanced Strong Names

Strong name keys consist of a signature key and an identity key. The assembly is signed with the signature key and is identified by the identity key. Prior to the .NET Framework 4.5, these two keys were identical. Starting with the .NET Framework 4.5, the identity key remains the same as in earlier .NET Framework versions, but the signature key is enhanced with a stronger hash algorithm. In addition, the signature key is signed with the identity key to create a counter-signature.

The AssemblySignatureKeyAttribute attribute enables the assembly metadata to use the pre-existing public key for

assembly identity, which allows old assembly references to continue to work. The AssemblySignatureKeyAttribute attribute uses the counter-signature to ensure that the owner of the new signature key is also the owner of the old identity key.

Signing with SHA-2, Without Key Migration

Run the following commands from a Command Prompt window to sign an assembly without migrating a strong name signature:

1. Generate the new identity key (if necessary).

```
sn -k IdentityKey.snk
```

2. Extract the identity public key, and specify that a SHA-2 algorithm should be used when signing with this key.

```
sn -p IdentityKey.snk IdentityPubKey.snk sha256
```

3. Delay-sign the assembly with the identity public key file.

```
csc MyAssembly.cs /keyfile:IdentityPubKey.snk /delaySign+
```

4. Re-sign the assembly with the full identity key pair.

```
sn -Ra MyAssembly.exe IdentityKey.snk
```

Signing with SHA-2, with Key Migration

Run the following commands from a Command Prompt window to sign an assembly with a migrated strong name signature.

1. Generate an identity and signature key pair (if necessary).

```
sn -k IdentityKey.snk
sn -k SignatureKey.snk
```

2. Extract the signature public key, and specify that a SHA-2 algorithm should be used when signing with this key.

```
sn -p SignatureKey.snk SignaturePubKey.snk sha256
```

3. Extract the identity public key, which determines the hash algorithm that generates a counter-signature.

```
sn -p IdentityKey.snk IdentityPubKey.snk
```

4. Generate the parameters for a AssemblySignatureKeyAttribute attribute, and attach the attribute to the assembly.

```
sn -ac IdentityPubKey.snk IdentityKey.snk SignaturePubKey.snk
```

5. Delay-sign the assembly with the identity public key.

```
csc MyAssembly.cs /keyfile:IdentityPubKey.snk /delaySign+
```

6. Fully sign the assembly with the signature key pair.

```
sn -Ra MyAssembly.exe SignatureKey.snk
```

See Also

Creating and Using Strong-Named Assemblies

How to: Reference a Strong-Named Assembly

5/2/2018 • 1 min to read • Edit Online

The process for referencing types or resources in a strong-named assembly is usually transparent. You can make the reference either at compile time (early binding) or at run time.

A compile-time reference occurs when you indicate to the compiler that your assembly explicitly references another assembly. When you use compile-time referencing, the compiler automatically gets the public key of the targeted strong-named assembly and places it in the assembly reference of the assembly being compiled.

> **NOTE**
> A strong-named assembly can only use types from other strong-named assemblies. Otherwise, the security of the strong-named assembly would be compromised.

To make a compile-time reference to a strong-named assembly

1. At the command prompt, type the following command:

   <compiler command> **/reference:***<assembly name>*

   In this command, *compiler command* is the compiler command for the language you are using, and *assembly name* is the name of the strong-named assembly being referenced. You can also use other compiler options, such as the **/t:library** option for creating a library assembly.

The following example creates an assembly called `myAssembly.dll` that references a strong-named assembly called `myLibAssembly.dll` from a code module called `myAssembly.cs`.

```
csc /t:library myAssembly.cs /reference:myLibAssembly.dll
```

To make a run-time reference to a strong-named assembly

1. When you make a run-time reference to a strong-named assembly (for example, by using the Assembly.Load or Assembly.GetType method), you must use the display name of the referenced strong-named assembly. The syntax of a display name is as follows:

   <assembly name>, *<version number>*, *<culture>*, *<public key token>*

   For example:

   ```
   myDll, Version=1.1.0.0, Culture=en, PublicKeyToken=03689116d3a4ae33
   ```

   In this example, `PublicKeyToken` is the hexadecimal form of the public key token. If there is no culture value, use `Culture=neutral`.

The following code example shows how to use this information with the Assembly.Load method.

```
Assembly^ myDll =
    Assembly::Load("myDll, Version=1.0.0.1, Culture=neutral, PublicKeyToken=9b35aa32c18d4fb1");
```

```
Assembly myDll =
    Assembly.Load("myDll, Version=1.0.0.1, Culture=neutral, PublicKeyToken=9b35aa32c18d4fb1");
```

```
Dim myDll As Assembly = _
    Assembly.Load("myDll, Version=1.0.0.1, Culture=neutral, PublicKeyToken=9b35aa32c18d4fb1")
```

You can print the hexadecimal format of the public key and public key token for a specific assembly by using the following Strong Name (Sn.exe) command:

sn -Tp < *assembly* **>**

If you have a public key file, you can use the following command instead (note the difference in case on the command-line option):

sn -tp < *assembly* **>**

See Also

Creating and Using Strong-Named Assemblies

How to: Disable the Strong-Name Bypass Feature

5/2/2018 • 2 min to read • Edit Online

Starting with the .NET Framework version 3.5 Service Pack 1 (SP1), strong-name signatures are not validated when an assembly is loaded into a full-trust AppDomain object, such as the default AppDomain for the `MyComputer` zone. This is referred to as the strong-name bypass feature. In a full-trust environment, demands for StrongNameIdentityPermission always succeed for signed, full-trust assemblies regardless of their signature. The only restriction is that the assembly must be fully trusted because its zone is fully trusted. Because the strong name is not a determining factor under these conditions, there is no reason for it to be validated. Bypassing the validation of strong-name signatures provides significant performance improvements.

The bypass feature applies to any full-trust assembly that is not delay-signed and that is loaded into any full-trust AppDomain from the directory specified by its ApplicationBase property.

You can override the bypass feature for all applications on a computer by setting a registry key value. You can override the setting for a single application by using an application configuration file. You cannot reinstate the bypass feature for a single application if it has been disabled by the registry key.

When you override the bypass feature, the strong name is validated only for correctness; it is not checked for a StrongNameIdentityPermission. If you want to confirm a specific strong name, you have to perform that check separately.

> **IMPORTANT**
>
> The ability to force strong-name validation depends on a registry key, as described in the following procedure. If an application is running under an account that does not have access control list (ACL) permission to access that registry key, the setting is ineffective. You must ensure that ACL rights are configured for this key so that it can be read for all assemblies.

To disable the strong-name bypass feature for all applications

- On 32-bit computers, in the system registry, create a DWORD entry with a value of 0 named `AllowStrongNameBypass` under the HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\.NETFramework key.
- On 64-bit computers, in the system registry, create a DWORD entry with a value of 0 named `AllowStrongNameBypass` under the HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\.NETFramework and HKEY_LOCAL_MACHINE\SOFTWARE\Wow6432Node\Microsoft\.NETFramework keys.

To disable the strong-name bypass feature for a single application

1. Open or create the application configuration file.

   For more information about this file, see the Application Configuration Files section in Configuring Apps.

2. Add the following entry:

```
<configuration>
  <runtime>
    <bypassTrustedAppStrongNames enabled="false" />
  </runtime>
</configuration>
```

You can restore the bypass feature for the application by removing the configuration file setting or by setting the attribute to "true".

NOTE

You can turn strong-name validation on and off for an application only if the bypass feature is enabled for the computer. If the bypass feature has been turned off for the computer, strong names are validated for all applications and you cannot bypass validation for a single application.

See Also

Sn.exe (Strong Name Tool)
<bypassTrustedAppStrongNames> Element
Creating and Using Strong-Named Assemblies

Delay Signing an Assembly

5/2/2018 • 3 min to read • Edit Online

An organization can have a closely guarded key pair that developers do not have access to on a daily basis. The public key is often available, but access to the private key is restricted to only a few individuals. When developing assemblies with strong names, each assembly that references the strong-named target assembly contains the token of the public key used to give the target assembly a strong name. This requires that the public key be available during the development process.

You can use delayed or partial signing at build time to reserve space in the portable executable (PE) file for the strong name signature, but defer the actual signing until some later stage (typically just before shipping the assembly).

The following steps outline the process to delay sign an assembly:

1. Obtain the public key portion of the key pair from the organization that will do the eventual signing. Typically this key is in the form of an .snk file, which can be created using the Strong Name tool (Sn.exe) provided by the Windows Software Development Kit (SDK).
2. Annotate the source code for the assembly with two custom attributes from System.Reflection:
   - AssemblyKeyFileAttribute, which passes the name of the file containing the public key as a parameter to its constructor.
   - AssemblyDelaySignAttribute, which indicates that delay signing is being used by passing **true** as a parameter to its constructor. For example:

   ```
   [assembly:AssemblyKeyFileAttribute("myKey.snk")];
   [assembly:AssemblyDelaySignAttribute(true)];
   ```

   ```
   [assembly:AssemblyKeyFileAttribute("myKey.snk")]
   [assembly:AssemblyDelaySignAttribute(true)]
   ```

   ```
   <Assembly:AssemblyKeyFileAttribute("myKey.snk")>
   <Assembly:AssemblyDelaySignAttribute(True)>
   ```
3. The compiler inserts the public key into the assembly manifest and reserves space in the PE file for the full strong name signature. The real public key must be stored while the assembly is built so that other assemblies that reference this assembly can obtain the key to store in their own assembly reference.
4. Because the assembly does not have a valid strong name signature, the verification of that signature must be turned off. You can do this by using the **–Vr** option with the Strong Name tool.

   The following example turns off verification for an assembly called `myAssembly.dll`.

   ```
   sn –Vr myAssembly.dll
   ```

   To turn off verification on platforms where you can't run the Strong Name tool, such as Advanced RISC Machine (ARM) microprocessors, use the **–Vk** option to create a registry file. Import the registry file into the registry on the computer where you want to turn off verification. The following example creates a

registry file for `myAssembly.dll`.

```
sn –Vk myRegFile.reg myAssembly.dll
```

With either the **–Vr** or **–Vk** option, you can optionally include an .snk file for test key signing.

> **WARNING**
> Do not rely on strong names for security. They provide a unique identity only.

> **NOTE**
> If you use delay signing during development with Visual Studio on a 64-bit computer, and you compile an assembly for **Any CPU**, you might have to apply the **-Vr** option twice. (In Visual Studio, **Any CPU** is a value of the **Platform Target** build property; when you compile from the command line, it is the default.) To run your application from the command line or from File Explorer, use the 64-bit version of the Sn.exe (Strong Name Tool) to apply the **-Vr** option to the assembly. To load the assembly into Visual Studio at design time (for example, if the assembly contains components that are used by other assemblies in your application), use the 32-bit version of the strong-name tool. This is because the just-in-time (JIT) compiler compiles the assembly to 64-bit native code when the assembly is run from the command line, and to 32-bit native code when the assembly is loaded into the design-time environment.

5. Later, usually just before shipping, you submit the assembly to your organization's signing authority for the actual strong name signing using the **–R** option with the Strong Name tool.

   The following example signs an assembly called `myAssembly.dll` with a strong name using the `sgKey.snk` key pair.

   ```
   sn -R myAssembly.dll sgKey.snk
   ```

See Also

Creating Assemblies
How to: Create a Public-Private Key Pair
Sn.exe (Strong Name Tool)
Programming with Assemblies

Working with Assemblies and the Global Assembly Cache

5/2/2018 • 2 min to read • Edit Online

If you intend to share an assembly among several applications, you can install it into the global assembly cache. Each computer where the common language runtime is installed has this machine-wide code cache. The global assembly cache stores assemblies specifically designated to be shared by several applications on the computer. An assembly must have a strong name to be installed in the global assembly cache.

> **NOTE**
>
> Assemblies placed in the global assembly cache must have the same assembly name and file name (not including the file name extension). For example, an assembly with the assembly name of myAssembly must have a file name of either myAssembly.exe or myAssembly.dll.

You should share assemblies by installing them into the global assembly cache only when necessary. As a general guideline, keep assembly dependencies private and locate assemblies in the application directory unless sharing an assembly is explicitly required. In addition, you do not have to install assemblies into the global assembly cache to make them accessible to COM interop or unmanaged code.

There are several reasons why you might want to install an assembly into the global assembly cache:

- Shared location.

  Assemblies that should be used by applications can be put in the global assembly cache. For example, if all applications should use an assembly located in the global assembly cache, a version policy statement can be added to the Machine.config file that redirects references to the assembly.

- File security.

  Administrators often protect the systemroot directory using an Access Control List (ACL) to control write and execute access. Because the global assembly cache is installed in the systemroot directory, it inherits that directory's ACL. It is recommended that only users with Administrator privileges be allowed to delete files from the global assembly cache.

- Side-by-side versioning.

  Multiple copies of assemblies with the same name but different version information can be maintained in the global assembly cache.

- Additional search location.

  The common language runtime checks the global assembly cache for an assembly that matches the assembly request before probing or using the codebase information in a configuration file.

Note that there are scenarios where you explicitly do not want to install an assembly into the global assembly cache. If you place one of the assemblies that make up an application into the global assembly cache, you can no longer replicate or install the application by using XCOPY to copy the application directory. In this case, you must also move the assembly into the global assembly cache.

In This Section

How to: Install an Assembly into the Global Assembly Cache
Describes the ways to install an assembly into the global assembly cache.

How to: View the Contents of the Global Assembly Cache
Explains how to use the Gacutil.exe (Global Assembly Cache Tool) to view the contents of the global assembly cache.

How to: Remove an Assembly from the Global Assembly Cache
Explains how to use the Gacutil.exe (Global Assembly Cache Tool) to remove an assembly from the global assembly cache.

Using Serviced Components with the Global Assembly Cache
Explains why serviced components (managed COM+ components) should be placed in the global assembly cache.

Related Sections

Creating Assemblies
Provides an overview of creating assemblies.

Global Assembly Cache
Describes the global assembly cache.

How to: View Assembly Contents
Explains how to use the Ildasm.exe (IL Disassembler) to view Microsoft intermediate language (MSIL) information in an assembly.

How the Runtime Locates Assemblies
Describes how the common language runtime locates and loads the assemblies that make up your application.

Programming with Assemblies
Describes assemblies, the building blocks of managed applications.

How to: Install an Assembly into the Global Assembly Cache

5/2/2018 • 2 min to read • Edit Online

There are two ways to install a strong-named assembly into the global assembly cache (GAC):

> **IMPORTANT**
>
> Only strong-named assemblies can be installed into the GAC. For information about how to create a strong-named assembly, see How to: Sign an Assembly with a Strong Name.

- Using Windows Installer.

  You do this in Visual Studio 2012 and Visual Studio 2013 by creating an InstallShield Limited Edition Project.

  This is the recommended and most common way to add assemblies to the global assembly cache. The installer provides reference counting of assemblies in the global assembly cache, plus other benefits.

- Using the Global Assembly Cache tool (Gacutil.exe).

  You can use Gacutil.exe to add strong-named assemblies to the global assembly cache and to view the contents of the global assembly cache.

  > **NOTE**
  >
  > Gacutil.exe is only for development purposes and should not be used to install production assemblies into the global assembly cache.

> **NOTE**
>
> In earlier versions of the .NET Framework, the Shfusion.dll Windows shell extension enabled you to install assemblies by dragging them in File Explorer. Beginning with the .NET Framework 4, Shfusion.dll is obsolete.

To use the Global Assembly Cache tool (Gacutil.exe)

1. At the command prompt, type the following command:

   gacutil -i *<assembly name>*

   In this command, *assembly name* is the name of the assembly to install in the global assembly cache.

The following example installs an assembly with the file name `hello.dll` into the global assembly cache.

```
gacutil -i hello.dll
```

For more information, see Global Assembly Cache tool (Gacutil.exe).

To use an InstallShield Limited Edition Project

1. Add a setup and deployment package to your solution by opening the shortcut menu for your solution, and then choosing **Add**, **New Project**.

2. In the **Add New Project** dialog box, in the **Installed** folder, choose **Other Project Types**, **Setup and Deployment**, **InstallShield Limited Edition**, and give your project a name. (If prompted, download, install, and activate InstallShield.)

3. Perform the general configuration of your setup and deployment project either by using the Project Assistant in **Solution Explorer**, or by choosing the substeps of the numbered steps in the **Solution Explorer**.Configure your setup as you would if you were not adding assemblies to the GAC.

4. To begin the process of adding an assembly to the GAC, choose **Files**, which is under the **Specify Application Data** step in **Solution Explorer**.

5. In the **Destination computer's folders** pane, open the shortcut menu for **Destination Computer**, and then choose **Show Predefined Folder**, **[GlobalAssemblyCache]**.

6. For each project in the solution that contains an assembly that you want to install in the global assembly cache:

   a. In the **Source computer's folders** pane, choose the project.

   b. In the **Destination computer's folders** pane, choose **[GlobalAssemblyCache]**.

   c. In the **Source computer's files** pane, choose **Primary output from** <*project_name*>.

   d. Drag the file in step c to the **Destination computer's files** pane (or use the **Copy** and **Paste** commands from the file's shortcut menu).

See Also

Working with Assemblies and the Global Assembly Cache
How to: Remove an Assembly from the Global Assembly Cache
Gacutil.exe (Global Assembly Cache Tool)
How to: Sign an Assembly with a Strong Name
Windows Installer Deployment

How to: View the Contents of the Global Assembly Cache

5/2/2018 • 1 min to read • Edit Online

Use the Global Assembly Cache tool (Gacutil.exe) to view the contents of the global assembly cache.

To view a list of the assemblies in the global assembly cache

1. At the Visual Studio command prompt, type the following command:

   gacutil -l
   -or-
   gacutil /l

In earlier versions of the .NET Framework, the Shfusion.dll Windows shell extension enabled you to view the global assembly cache in File Explorer. Beginning with the .NET Framework 4, Shfusion.dll is obsolete.

See Also

Working with Assemblies and the Global Assembly Cache
Gacutil.exe (Global Assembly Cache Tool)

How to: Remove an Assembly from the Global Assembly Cache

5/2/2018 • 1 min to read • Edit Online

There are two ways to remove an assembly from the global assembly cache (GAC):

- By using the Global Assembly Cache tool (Gacutil.exe). You can use this option to uninstall assemblies that you've placed in the GAC during development and testing.
- By using Windows Installer. You should use this option to uninstall assemblies when testing installation packages and for production systems.

Removing an assembly with Gacutil.exe

1. At the command prompt, type the following command:

   gacutil –u *<assembly name>*

   In this command, *assembly name* is the name of the assembly to remove from the global assembly cache.

   > **WARNING**
   > You should not use Gacutil.exe to remove assemblies on production systems because of the possibility that the assembly may still be required by some application. Instead, you should use the Windows Installer, which maintains a reference count for each assembly it installs in the GAC.

The following example removes an assembly named `hello.dll` from the global assembly cache.

```
gacutil -u hello
```

Removing an assembly with Windows Installer

1. From the **Programs and Features** app in **Control Panel**, select the app that you want to uninstall. If the installation package placed assemblies in the GAC, Windows Installer will remove them if they are not used by another application.

   > **NOTE**
   > Windows Installer maintains a reference count for assemblies installed in the GAC. An assembly is removed from the GAC only when its reference count reaches zero, which indicates that it is not used by any application installed by a Windows Installer package.

See Also

Working with Assemblies and the Global Assembly Cache
How to: Install an Assembly into the Global Assembly Cache
Gacutil.exe (Global Assembly Cache Tool)

Using Serviced Components with the Global Assembly Cache

5/2/2018 • 1 min to read • Edit Online

Serviced components (managed code COM+ components) should be put in the Global Assembly Cache. In some scenarios, the Common Language Runtime and COM+ Services can handle serviced components that are not in the Global Assembly Cache; in other scenarios, they cannot. The following scenarios illustrate this:

- For serviced components in a COM+ Server application, the assembly containing the components must be in the Global Assembly Cache, because Dllhost.exe does not run in the same directory as the one that contains the serviced components.
- For serviced components in a COM+ Library application, the runtime and COM+ Services can resolve the reference to the assembly containing the components by searching in the current directory. In this case, the assembly does not have to be in the global assembly cache.
- For serviced components in an ASP.NET application, the situation is different. If you place the assembly containing the serviced components in the bin directory of the application base and use on-demand registration, the assembly will be shadow-copied into the download cache because ASP.NET leverages the shadow capabilities of the runtime.

See Also

Working with Assemblies and the Global Assembly Cache
Gacutil.exe (Global Assembly Cache Tool)

How to: View Assembly Contents

5/2/2018 • 2 min to read • Edit Online

You can use the Ildasm.exe (IL Disassembler) to view Microsoft intermediate language (MSIL) information in a file. If the file being examined is an assembly, this information can include the assembly's attributes, as well as references to other modules and assemblies. This information can be helpful in determining whether a file is an assembly or part of an assembly, and whether the file has references to other modules or assemblies.

To display the contents of an assembly using Ildasm.exe

1. Type **ildasm** *<assembly name>* at the command prompt. For example, the following command disassembles the `Hello.exe` assembly.

```
ildasm Hello.exe
```

To view assembly manifest information

1. Double-click the MANIFEST icon in the MSIL Disassembler window.

Example

The following example starts with a basic "Hello, World" program. After compiling the program, use Ildasm.exe to disassemble the Hello.exe assembly and view the assembly manifest.

```
using namespace System;

class MainApp
{
public:
    static void Main()
    {
        Console::WriteLine("Hello World using C++/CLI!");
    }
};

int main()
{
    MainApp::Main();
}
```

```
using System;

class MainApp
{
    public static void Main()
    {
        Console.WriteLine("Hello World using C#!");
    }
}
```

```
Imports System

Class MainApp
    Public Shared Sub Main()
        Console.WriteLine("Hello World using Visual Basic!")
    End Sub
End Class
```

Running the command ildasm.exe on the Hello.exe assembly and double-clicking the MANIFEST icon in the IL DASM window produces the following output:

```
// Metadata version: v4.0.30319
.assembly extern mscorlib
{
  .publickeytoken = (B7 7A 5C 56 19 34 E0 89 )                         // .z\V.4..
  .ver 4:0:0:0
}
.assembly Hello
{
  .custom instance void
[mscorlib]System.Runtime.CompilerServices.CompilationRelaxationsAttribute::.ctor(int32) = ( 01 00 08 00 00 00
00 00 )
  .custom instance void [mscorlib]System.Runtime.CompilerServices.RuntimeCompatibilityAttribute::.ctor() = (
01 00 01 00 54 02 16 57 72 61 70 4E 6F 6E 45 78   // ....T..WrapNonEx

63 65 70 74 69 6F 6E 54 68 72 6F 77 73 01 )       // ceptionThrows.
  .hash algorithm 0x00008004
  .ver 0:0:0:0
}
.module Hello.exe
// MVID: {7C2770DB-1594-438D-BAE5-98764C39CCCA}
.imagebase 0x00400000
.file alignment 0x00000200
.stackreserve 0x00100000
.subsystem 0x0003       // WINDOWS_CUI
.corflags 0x00000001    //  ILONLY
// Image base: 0x00600000
```

The following table describes each directive in the assembly manifest of the Hello.exe assembly used in the example.

DIRECTIVE	DESCRIPTION
.assembly extern < *assembly name* >	Specifies another assembly that contains items referenced by the current module (in this example, `mscorlib`).
.publickeytoken < *token* >	Specifies the token of the actual key of the referenced assembly.
.ver < *version number* >	Specifies the version number of the referenced assembly.
.assembly < *assembly name* >	Specifies the assembly name.
.hash algorithm < *int32 value* >	Specifies the hash algorithm used.
.ver < *version number* >	Specifies the version number of the assembly.

DIRECTIVE	DESCRIPTION
.module < *file name* >	Specifies the name of the modules that make up the assembly. In this example, the assembly consists of only one file.
.subsystem < *value* >	Specifies the application environment required for the program. In this example, the value 3 indicates that this executable is run from a console.
.corflags	Currently a reserved field in the metadata.

An assembly manifest can contain a number of different directives, depending on the contents of the assembly. For an extensive list of the directives in the assembly manifest, see the ECMA documentation, especially "Partition II: Metadata Definition and Semantics" and "Partition III: CIL Instruction Set". The documentation is available online; see ECMA C# and Common Language Infrastructure Standards on MSDN and Standard ECMA-335 - Common Language Infrastructure (CLI) on the Ecma International Web site.

See Also

Application Domains and Assemblies
Application Domains and Assemblies How-to Topics
Ildasm.exe (IL Disassembler)

Type Forwarding in the Common Language Runtime

5/2/2018 • 1 min to read • Edit Online

Type forwarding allows you to move a type to another assembly without having to recompile applications that use the original assembly.

For example, suppose an application uses the `Example` class in an assembly named `Utility.dll`. The developers of `Utility.dll` might decide to refactor the assembly, and in the process they might move the `Example` class to another assembly. If the old version of `Utility.dll` is replaced by the new version of `Utility.dll` and its companion assembly, the application that uses the `Example` class fails because it cannot locate the `Example` class in the new version of `Utility.dll`.

The developers of `Utility.dll` can avoid this by forwarding requests for the `Example` class, using the TypeForwardedToAttribute attribute. If the attribute has been applied to the new version of `Utility.dll`, requests for the `Example` class are forwarded to the assembly that now contains the class. The existing application continues to function normally, without recompilation.

> **NOTE**
> In the .NET Framework version 2.0, you cannot forward types from assemblies written in Visual Basic. However, an application written in Visual Basic can consume forwarded types. That is, if the application uses an assembly coded in C# or C++, and a type from that assembly is forwarded to another assembly, the Visual Basic application can use the forwarded type.

Forwarding Types

There are four steps to forwarding a type:

1. Move the source code for the type from the original assembly to the destination assembly.
2. In the assembly where the type used to be located, add a TypeForwardedToAttribute for the type that was moved. The following code shows the attribute for a type named `Example` that was moved.

```
[assembly:TypeForwardedToAttribute(typeof(Example))]
```

```
[assembly:TypeForwardedToAttribute(Example::typeid)]
```

3. Compile the assembly that now contains the type.
4. Recompile the assembly where the type used to be located, with a reference to the assembly that now contains the type. For example, if you are compiling a C# file from the command line, use the /reference (C# Compiler Options) option to specify the assembly that contains the type. In C++, use the #using directive in the source file to specify the assembly that contains the type.

See Also

TypeForwardedToAttribute
Type Forwarding (C++/CLI)
#using Directive