**[Web APi之过滤器创建过程原理解析【一】（十）](http://www.cnblogs.com/CreateMyself/p/4853134.html)**

http://www.cnblogs.com/CreateMyself/p/4853134.html

**前言**

Web API的简单流程就是从请求到执行到Action并最终作出响应，但是在这个过程有一把【筛子】，那就是过滤器Filter，在从请求到Action这整个流程中使用Filter来进行相应的处理从而作出响应，这对于认证以及授权等是及其重要的，所以说过滤器应用是Web API框架中非常重要的一种实现方式，我们有必要去探讨其原理。

Web API框架提供了一个请求、响应的消息处理管道，并且其框架极具扩展性，通过其扩展可以对执行的流程进行适当的干预，其扩展点就体现在以下三个方面：

* **将自定义的HttpMessageHandler注册到消息处理管道中。**
* **将自定义的标准化组件注册到当前的服务容器（ServicesContainer）或者（HttpConfiguration）上。**
* **将自定义的过滤器Filter注册到控制器或者Action方法上。**

**Filter创建原理解析**

上一节我们讲到了HttpActionDescriptor，顾名思义是在控制器方法上返回的是对控制器方法的描述类型，它封装了控制器方法的一切信息，而过滤器Filter就是在这个类中初始化和创建的，我们首先来看看这个类中我们会用到的方法及其属性：

[复制代码](javascript:void(0);)

1 public abstract class HttpActionDescriptor

2 {

3 // Fields

4 private HttpActionBinding \_actionBinding;

5 private HttpConfiguration \_configuration;

6 private HttpControllerDescriptor \_controllerDescriptor;

7 private IActionResultConverter \_converter;

8 private readonly Lazy<Collection<FilterInfo>> \_filterPipeline;

9 private readonly ConcurrentDictionary<object, object> \_properties;

10 private static readonly ResponseMessageResultConverter \_responseMessageResultConverter;

11 private readonly Collection<HttpMethod> \_supportedHttpMethods;

12 private static readonly VoidResultConverter \_voidResultConverter;

13

14 // Methods

15 static HttpActionDescriptor();

16 protected HttpActionDescriptor();

17 protected HttpActionDescriptor(HttpControllerDescriptor controllerDescriptor);

18 private static bool AllowMultiple(object filterInstance);

19 public abstract Task<object> ExecuteAsync(HttpControllerContext controllerContext, IDictionary<string, object> arguments, CancellationToken cancellationToken);

20 public virtual Collection<T> GetCustomAttributes<T>() where T: class;

21 public virtual Collection<FilterInfo> GetFilterPipeline();

22 public virtual Collection<IFilter> GetFilters();

23 public abstract Collection<HttpParameterDescriptor> GetParameters();

24 internal static IActionResultConverter GetResultConverter(Type type);

25 private Collection<FilterInfo> InitializeFilterPipeline();

26 private static IEnumerable<FilterInfo> RemoveDuplicates(IEnumerable<FilterInfo> filters);

27 }

[复制代码](javascript:void(0);)

我们首先看看这个此类的构造函数  protected HttpActionDescriptor();

[复制代码](javascript:void(0);)

protected HttpActionDescriptor()

{

this.\_properties = new ConcurrentDictionary<object, object>();

this.\_supportedHttpMethods = new Collection<HttpMethod>();

this.\_filterPipeline = new Lazy<Collection<FilterInfo>>(new Func<Collection<FilterInfo>>(this.InitializeFilterPipeline));

}

[复制代码](javascript:void(0);)

从上知过滤器管道的创建就是在此类的构造函数中进行。首先我们将构造函数进行搁置，我们先来了解过滤器的一些基本的信息。

**过滤器接口（IFilter）**

|  |  |
| --- | --- |
| 1  2  3  4  5 | public interface IFilter  {      // Properties      bool AllowMultiple { get; }  } |

**此接口只有一个只读属性AllowMultiple，它表示多个同类的过滤器类型是否允许指定应用到同一个目标对象上，如果允许指定，则该值为true，否则为false，默认是false。**

**过滤器特性（FilterAttribute）**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | [AttributeUsage(AttributeTargets.Method | AttributeTargets.Class, Inherited=true, AllowMultiple=true)]  public abstract class FilterAttribute : Attribute, IFilter  {      // Fields      private static readonly ConcurrentDictionary<Type, bool> \_attributeUsageCache;        // Methods      static FilterAttribute();      protected FilterAttribute();      private static bool AllowsMultiple(Type attributeType);        // Properties      public virtual bool AllowMultiple { get; }  } |

**由上知，此过滤器特性类为过滤器的基类，当我们设置 AttributeUsage 中的 AllowMultiple 来指定是否允许多个特性应用到相同的对象上，同时通过其属性AllowMultiple来返回其值，当我们想要实现自定义的过滤器特性类时仅仅继承此类是不够的，因为此特性类只是特性中一个【属性】而已，而过滤器中的行为是通过IActionFilter来提供的。**

**过滤器行为接口（IActionFilter）**

|  |  |
| --- | --- |
| 1  2  3  4  5 | public interface IActionFilter : IFilter  {      // Methods      Task<HttpResponseMessage> ExecuteActionFilterAsync(HttpActionContext actionContext, CancellationToken cancellationToken, Func<Task<HttpResponseMessage>> continuation);  } |

**过滤器作用域（FilterScope）**

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | public enum FilterScope  {      Action = 20,      Controller = 10,      Global = 0  } |

**该类为枚举类，它有三个作用域选项： Action = 20　 在控制器之后指定一个操作。 Controller = 10 在Global之后，Action方法之前指定一个操作。  Global = 0 在控制器之前指定一个操作。**

**过滤器信息（FilterInfo）**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | public sealed class FilterInfo  {      // Methods      public FilterInfo(IFilter instance, FilterScope scope);        // Properties      public IFilter Instance { get; private set; }      public FilterScope Scope { get; private set; }  } |

显然，此类就是一个对过滤器封装的对象，通过它我们不仅获得封装的Filter并且可以获得其作用域范围。在此类中有两个只读属性 Instance 和 Scope 。

好了，说完了有关过滤器一些基本的概念，下面我们继续回到HttpActionDescriptor的构造函数中。构造函数中用来生成过滤器管道的一句，如下：

 this.\_filterPipeline = new Lazy<Collection<FilterInfo>>(new Func<Collection<FilterInfo>>(this.InitializeFilterPipeline));

在此类中的属性  \_filterPipeline 是一个FliterInfo的集合，由上知，此时还需进行初始化过滤器管道，我们继续查看此方法的实现：

[复制代码](javascript:void(0);)

private Collection<FilterInfo> InitializeFilterPipeline()

{

return new Collection<FilterInfo>(RemoveDuplicates((from fp in this.\_configuration.Services.GetFilterProviders() select fp.GetFilters(this.\_configuration, this)).OrderBy<FilterInfo, FilterInfo>(f => f, FilterInfoComparer.Instance).Reverse<FilterInfo>()).Reverse<FilterInfo>().ToList<FilterInfo>());

}

[复制代码](javascript:void(0);)

我们从最外面到最里面进行讲解，利用 RemoveDuplicates() 方法来移除重复的，下面会讲到，我们来看看此方法 this.\_configuration.Services.GetFilterProviders()

|  |  |
| --- | --- |
| 1  2  3  4 | public static IEnumerable<IFilterProvider> GetFilterProviders(this ServicesContainer services)  {      return services.GetServices<IFilterProvider>();  } |

首先我们还需要看一个对象FilterPrivoder对象:

**过滤器提供者（FilterPrivoder）**

**FilterProvider是过滤器管道机制中的核心对象，它根据指定的HttpActionDescriptor对象将其应用到对应Action上的过滤去Filter中，他们都实现了IFilterProvider接口，并且还定义了唯一的一个GetFilters方法，最终返回过滤器信息的集合列表，两个参数分别是表示当前的HttpConfiguration对象以及描述Action方法的HttpActionDescriptor对象。**

|  |  |
| --- | --- |
| 1  2  3  4  5 | public interface IFilterProvider  {      // Methods      IEnumerable<FilterInfo> GetFilters(HttpConfiguration configuration, HttpActionDescriptor actionDescriptor);  } |

从HttpConfiguration实例中的属性Services来获取实现了接口IFilterProvider的注册服务，既然如此我们去看看我们早已经很熟悉的DefaultServices子类，下面我们来看看FilterProvider。

this.SetMultiple<IFilterProvider>(new IFilterProvider[] { new ConfigurationFilterProvider(), new ActionDescriptorFilterProvider() });

由上知，FilterProvider是 ConfigurationFilterProvider 和 ActionDescriptorFilterProvider 两个对象。下面我们就一一来看。

* **ConfiguraionFilterProvider**

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | public class ConfigurationFilterProvider : IFilterProvider  {      // Methods      public ConfigurationFilterProvider();      public IEnumerable<FilterInfo> GetFilters(HttpConfiguration configuration, HttpActionDescriptor actionDescriptor);  } |

我们查看其方法的实现：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | public IEnumerable<FilterInfo> GetFilters(HttpConfiguration configuration, HttpActionDescriptor actionDescriptor)  {      if (configuration == null)      {          throw Error.ArgumentNull("configuration");      }      return configuration.Filters;  } |

我们继续看看HttpConfiguration中的属性Filters具体是什么：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | public HttpFilterCollection Filters  {      get      {          return this.\_filters;      }  } |

知，Filters是过滤器的集合HttpFilterCollection，我们再来详细看看此类

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | public class HttpFilterCollection : IEnumerable<FilterInfo>, IEnumerable  {      // Fields      private readonly List<FilterInfo> \_filters;        // Methods      public HttpFilterCollection();      public void Add(IFilter filter);      private void AddInternal(FilterInfo filter);      public void Clear();      public bool Contains(IFilter filter);      public IEnumerator<FilterInfo> GetEnumerator();      public void Remove(IFilter filter);      IEnumerator IEnumerable.GetEnumerator();        // Properties      public int Count { get; }  } |

**由上知，在HttpCofiguraiton上属于全局范围的Filters可以直接注册到HttpConfiguration上，此Filters是一个只读属性，其类型为HttpFilterCollection为FilterInfo对象的集合，此HttpFilterCollection类上的Add、Remve等方法真正作用的不是FilterInfo，而是有FilterInfo封装的Filter对象，当我们传入一个Filter进去时，通过调用Add方法，则此对象的FilterInfo则将添加到HttpFilterCollection集合列表中，则这些全局注册的Filter的FilterScope应该就是Global，不信我们查看其Add方法便知：**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | public void Add(IFilter filter)  {      if (filter == null)      {          throw Error.ArgumentNull("filter");      }      this.AddInternal(new FilterInfo(filter, FilterScope.Global));  } |

与我们猜想一致。所以上述ConfigutaionFilterProvider旨在通过FilterInfo来封装所有注册的Filter，所以最终返回的就是 configuration.Filters

* **ActionDescriptorFilterProvider**

由于都是继承于IFilterProvider，定义必定一样，只是实现不一样，我们只需查看具体的实现方法即可：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | public IEnumerable<FilterInfo> GetFilters(HttpConfiguration configuration, HttpActionDescriptor actionDescriptor)  {      if (configuration == null)      {          throw Error.ArgumentNull("configuration");      }      if (actionDescriptor == null)      {          throw Error.ArgumentNull("actionDescriptor");      }      IEnumerable<FilterInfo> first = from instance in actionDescriptor.ControllerDescriptor.GetFilters() select new FilterInfo(instance, FilterScope.Controller);      IEnumerable<FilterInfo> second = from instance in actionDescriptor.GetFilters() select new FilterInfo(instance, FilterScope.Action);      return first.Concat<FilterInfo>(second);  } |

由上知，由ActionDescriptorFilterProvider对象提供的FilterInfo封装的Filter具有两个来源：

1. **作用于Action方法上的Filter**
2. **作用于Action方法所对应的控制器的Filter**

对于这二者其对应的FilterScope属性分别为Action和Controller。

最终整个初始化过滤器管道以及创建过程就已经完成，在HttpActionDesciptor类上的 GetFilterPipeline 方法，如下：

|  |  |
| --- | --- |
| 1  2  3  4 | public virtual Collection<FilterInfo> GetFilterPipeline()  {      return this.\_filterPipeline.Value;  } |

通过此方法来获得FilterInfo封装的Filter集合。

**过滤器生成管道顺序以及优先级**

说了这么久，它们生成的顺序到底是怎样的呢？它们的优先级又是怎样的呢？我们通过对上述整个创建过滤器过程的了解，我们来手动实现并看看其生成顺序以及优先级：

**（1）生成管道顺序**

我们自定义以下三个过滤器：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | public class OneFilter :System.Web.Http.Filters.ActionFilterAttribute  {    }  public class TwoFilter : System.Web.Http.Filters.ActionFilterAttribute  {    }  public class ThreeFilter : System.Web.Http.Filters.ActionFilterAttribute  {    } |

在Web API配置文件中添加一个全局过滤器：

|  |  |
| --- | --- |
| 1 | config.Filters.Add(new ThreeFilter()); |

最后实现调用并查看其顺序：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | [OneFilter]   public class ProductController : ApiController   {         [TwoFilter]       public IEnumerable<Tuple<string, System.Web.Http.Filters.FilterScope>> GetFilter()       {           var actionSelector = this.Configuration.Services.GetActionSelector();           var actionDesciptor = actionSelector.SelectAction(this.ControllerContext);           foreach (var filterInfo in actionDesciptor.GetFilterPipeline())           {              yield return new Tuple<string, System.Web.Http.Filters.FilterScope>(filterInfo.Instance.GetType().Name, filterInfo.Scope);           }       }  } |

最后访问生成如图所示结果：



**过滤器生成顺序：Global--------------------------------------->Controller------------------------------------->Action**

**（2）优先级**

首先我们重新建立一个过滤器特性类，如下：

[复制代码](javascript:void(0);)

[AttributeUsage(AttributeTargets.Method | AttributeTargets.Class, AllowMultiple = true)]

public class OneFilter :System.Web.Http.Filters.FilterAttribute,System.Web.Http.Filters.IActionFilter

{

public System.Threading.Tasks.Task<HttpResponseMessage> ExecuteActionFilterAsync(System.Web.Http.Controllers.HttpActionContext actionContext, System.Threading.CancellationToken cancellationToken, Func<System.Threading.Tasks.Task<HttpResponseMessage>> continuation)

{

return continuation();

}

}

[复制代码](javascript:void(0);)

接着将控制器以及Action方法进行如下修改：

[复制代码](javascript:void(0);)

[OneFilter]

public class ProductController : ApiController

{

[OneFilter]

public IEnumerable<Tuple<string, System.Web.Http.Filters.FilterScope>> GetFilter()

{

var actionSelector = this.Configuration.Services.GetActionSelector();

var actionDesciptor = actionSelector.SelectAction(this.ControllerContext);

foreach (var filterInfo in actionDesciptor.GetFilterPipeline())

{

yield return new Tuple<string, System.Web.Http.Filters.FilterScope>(filterInfo.Instance.GetType().Name, filterInfo.Scope);

}

}

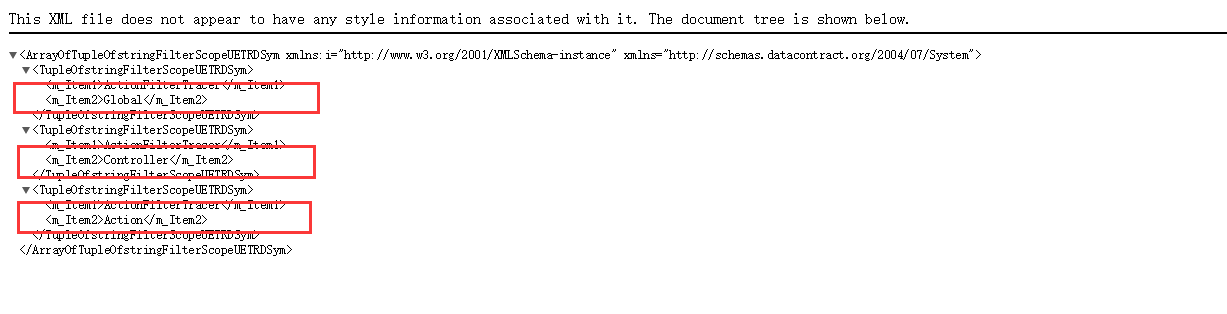
}

[复制代码](javascript:void(0);)

然后我们将配置文件也进行修改：

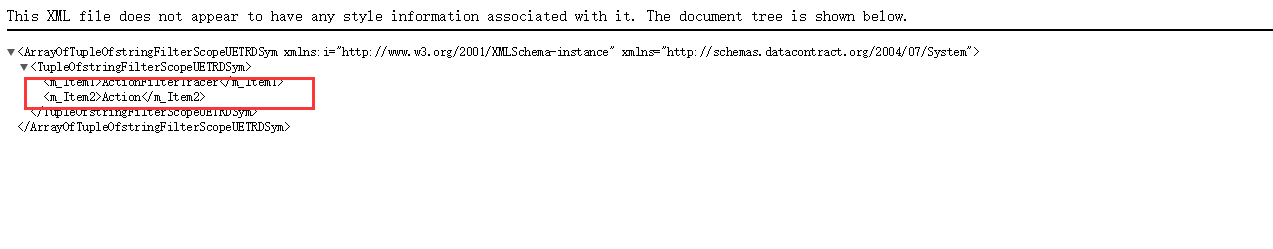
config.Filters.Add(new OneFilter());

此时结果如下：

似乎有点不太对劲，我们在全局和控制器对应的Action方法都使用同一特性，此时并没有起到过滤的作用，这是在我们意料之外，接下来我们将上述自定义过滤器特性的应用范围进行如下修改：

[AttributeUsage(AttributeTargets.Method | AttributeTargets.Class, AllowMultiple = false)]

我们仅仅是将 AllowMultiple = true 修改为了 AllowMultiple = false ，我们知道位于 AttributeUsage 上的AllowMutiple属性只能控制我们是否将多个目标特性多次应用到同一对象上而已，接下来我们再来看看其结果。



**由上知，当使用同一过滤器特性来进行修饰时，Action优先执行。**

我们先不管将 AllowMultiple 修改为false，为什么出现如此结果，我们在上述修改基础上，来修改控制器及Action方法的过滤器特性而配置文件中不变：

[复制代码](javascript:void(0);)

[OneFilter]

public class ProductController : ApiController

{

public IEnumerable<Tuple<string, System.Web.Http.Filters.FilterScope>> GetFilter()

{

List<string> list = new List<string>();

var actionSelector = this.Configuration.Services.GetActionSelector();

var actionDesciptor = actionSelector.SelectAction(this.ControllerContext);

foreach (var filterInfo in actionDesciptor.GetFilterPipeline())

{

yield return new Tuple<string, System.Web.Http.Filters.FilterScope>(filterInfo.Instance.GetType().Name, filterInfo.Scope);

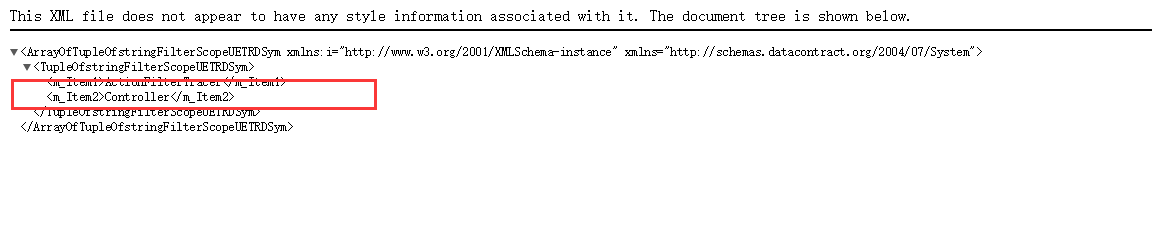
}

}

}

[复制代码](javascript:void(0);)

由上知，我们只是去掉了Action方法上的特性，接下来我们来再看看其结果：



**至此，我们可以得出如下结论**

**Web API框架多次利用同一Filter在同一目标元素时，所采用唯一性Filter策略，并且Filter总是采用FilterScope最大的那一个，换言之，对于其过滤器作用域（FilterScope）的优先级是：Action-----------------> Controller ------------------>Global**

**那么问题来了，对于这种唯一性Filter策略是如何实现的呢？请往下看。**

**唯一性Filter策略实现原理**

在实现唯一性Filter策略也就是在初始化过滤器的管道过程中，如下：

[复制代码](javascript:void(0);)

private Collection<FilterInfo> InitializeFilterPipeline()

{

return new Collection<FilterInfo>(RemoveDuplicates((from fp in this.\_configuration.Services.GetFilterProviders() select fp.GetFilters(this.\_configuration, this)).OrderBy<FilterInfo, FilterInfo>(f => f, FilterInfoComparer.Instance).Reverse<FilterInfo>()).Reverse<FilterInfo>().ToList<FilterInfo>());

}

[复制代码](javascript:void(0);)

至于是利用上述如何来实现的，我们只需关注，上述三个红色标记即可。下面我们一一进行叙述。

* **GetFilters**

这个就不用说了，通过注册了服务容器并实现了其过滤器接口的服务实例，并利用HttpConfiguration中封装了Filter的FilterInfo的集合列表来进行Linq筛选。

* **FilterInfoComparer**

通过其单词意思我们就能猜想得到，是进行FilterInfo比较，下面我们具体来看看此类的实现：

[复制代码](javascript:void(0);)

internal sealed class FilterInfoComparer : IComparer<FilterInfo>

{

// Fields

private static readonly FilterInfoComparer \_instance;

// Methods

static FilterInfoComparer();

public FilterInfoComparer();

public int Compare(FilterInfo x, FilterInfo y);

// Properties

public static FilterInfoComparer Instance { get; }

}

[复制代码](javascript:void(0);)

**在初始化管道中利用 FilterInfoComparer.Instance 实例进行比较，此Instance就是获取在上面此类的静态构造函数中实例并赋给字段\_instance。最重要的就是实现其比较接口的Compare方法了，我们来看看：**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | public int Compare(FilterInfo x, FilterInfo y)  {      if ((x == null) && (y == null))      {          return 0;      }      if (x == null)      {          return -1;      }      if (y == null)      {          return 1;      }      return (int) (x.Scope - y.Scope);  } |

接下来我们先进行搁置，最后来看看最后一步。

* **RemoveDuplicates**

从字面意思是去除重复，我们一起来看看：

[复制代码](javascript:void(0);)

private static IEnumerable<FilterInfo> RemoveDuplicates(IEnumerable<FilterInfo> filters)

{

HashSet<Type> iteratorVariable0 = new HashSet<Type>();

foreach (FilterInfo iteratorVariable1 in filters)

{

object instance = iteratorVariable1.Instance;

Type type = instance.GetType();

if (!iteratorVariable0.Contains(type) || AllowMultiple(instance))

{

yield return iteratorVariable1;

iteratorVariable0.Add(type);

}

}

}

[复制代码](javascript:void(0);)

我们再来看看上述红色标记的AllowMutiple方法：

[复制代码](javascript:void(0);)

private static bool AllowMultiple(object filterInstance)

{

IFilter filter = filterInstance as IFilter;

if (filter != null)

{

return filter.AllowMultiple;

}

return true;

}

[复制代码](javascript:void(0);)

到这里为止，想必大家知道了是怎样来进行筛选的了，筛选过程分为两类：

* **相同控制器上及Action方法有不同的Filter特性时**

此时通过上述比较方法并利用初始化管道中的OrderBy从小到大排序即Global->Controller->Action，然后通过Reverse方法进行扭转即Action->Controller->Global，接着利用RemoveDuplicates方法来去除重复，去除重复是利用过滤器的类型进行筛选，此时不同的Filter都会添加到其中，最后通过Reverse继续扭转最终得到就是Global->Controller->Action。

* **相同控制器上及Action方法有相同的Filter特性时**

这个和上述执行到RemoveDuplicate之前是是一致的，当执行到RemoveDuplicates时，此时的通过第一个Reverse顺序依然是Action->Controller->Global，我们能想到过滤器特性类型都是一样的，由于在 AttributeUsage 中的属性AllowMutiple默认是true，当检测到第二个类型一样时，此时第一个条件就没进行过滤，而第二个检测到true则依然如上述一样全部被执行并最终利用Reverse扭转依然是Global->Controller->Action，如我们所演示，将其AllowMultiple设置为false，此时第二个条件也就不再起作用，也就是说不会添加到FilterInfo集合列表中，换言之，相同的过滤器类型应用到控制器及方法或者是全局中时，此时只有第一个即Action会添加到FilterInfo集合列表中，最终进行Reverse扭转依然只有一个那就是优先级最高的Action，当然了前提是我们在Action方法使用了自定义特性，若未使用而是在控制器上使用最终输出的当然也就是优先级次之的Controller了，以此类推。

**总结**

至此，过滤器的初始化以及创建过程中的筛选就全部完成，不去了解尚未知，一旦去研究了，乐趣还是挺多。

**过滤器类型（Filter Type）**

Filter类型有五种，它能够在Action执行的不同时期被调用，下面列出：

**AuthenticationFilter**

**AuthcnticationFiIter会在执行Action方法之前被调用以实现对请求的认证,所有的AuthenticationFilter类型均实现了IAuthenticationFilter接口。**

**AuthorizationFilter**

**应用的AuthorticationFilter会在Action执行之前将被调用以针对当前请求的授权检查，所有的AuthorizationFilter类型均实现了IAuthorizationFilter接口。**

**ActionFilter**

**ActionFilter注册的回调操作会在执行Action方法前后被调用，所有的ActionFilter类型均实现了IActionFilter接口。**

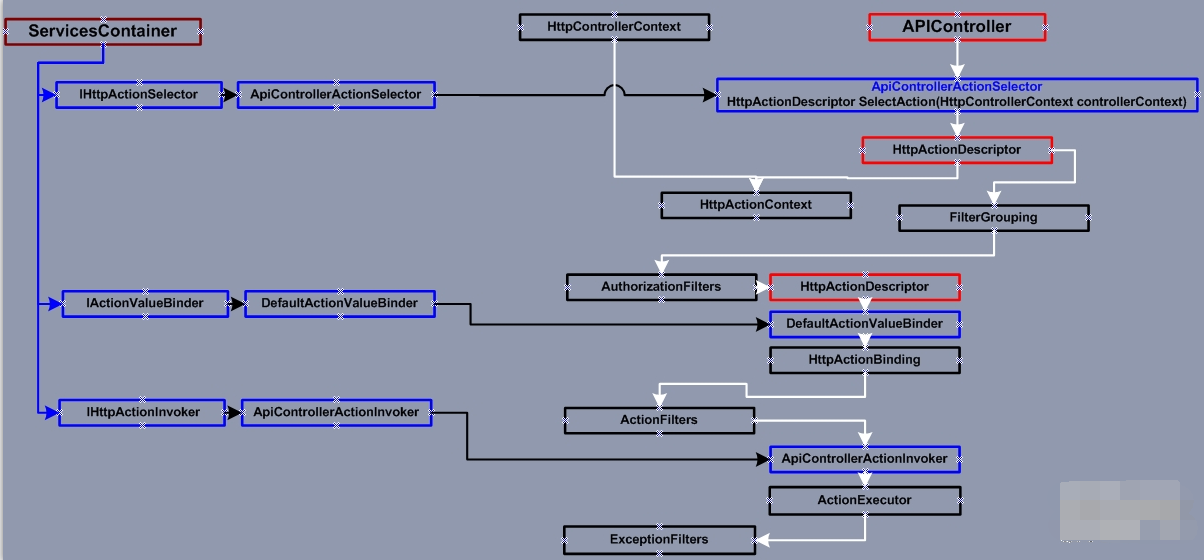
**ExceptionFilter**

**当在执行Action过程中抛出异常，ExceptionFilter会被调用来实现对异常的处理，所有的ExceptionFilter类型均实现了IExceptionFilter接口。**

**OverrideFilter**

**在默认情况下，全局注册的Filter和注册到控制器类上的Filter都会应用到Action方法上，如果我们希望某个Action方法屏蔽这些外层注册的Filter，可以在该方法上实现OverrideFilter，反之亦然，所有的OverrideFilter类型均实现了IOverrideFilter接口。**

**总结**



图片来源：[过滤器创建过程](http://jinyuan.blog.51cto.com/8854733/1548571)