

## Spring 容器是如何初始化的

因查找 ht 项目中一个久未解决 spring 内部异常，翻了一段时间源码。以此文总结 springIOC，容器初始化过程。

语言背景是 C#。网上有一些基于 java 的 spring 源码分析文档，大而乱，乱而不全，干脆自己梳理下。

废话不多说，进正题。

打开 spring.core.dll，这是核心库，找到 ContextRegistry 类，此类为密封类，无继承，本类实现对 spring 容器进行管理，获取一个容器均会通过此类来打交道，以，此类相当于我们使用 IOC 容器的入口。

注意我圈红的地方

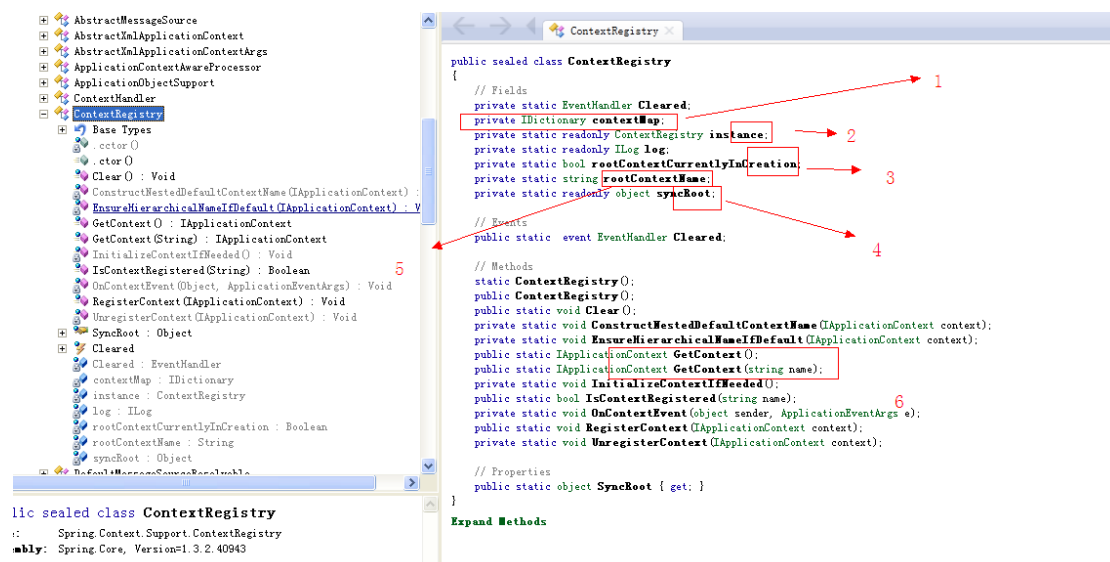


图 1

1 处是一个字典，管理父子容器，IOC 容器中可以有任意类型，容器中有子容器这是允许的，虽然生产中极少看见，但是，我翻看 Spring 源码的时候，确实看到了这种情况。不多说。

2 处 Context 管理类 ContextRegistry 实例，单列模式。

3 处，bool 变量，标识根容器对象是否正在创建，默认 false

4 处，用于线程同步的资源锁，Object 类型即可

5 处，根容器对象的名称

6 处，这两个方法是我们用到最多的，我们获取 IOC 容器的入口

打开类型构造器

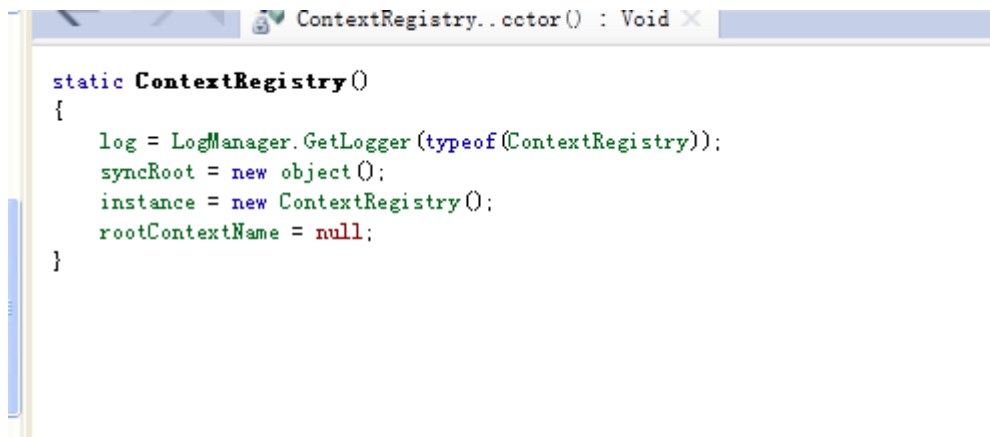


图 2

可以看见，对 12345 处的变量进行了初始化。

打开实例构造器

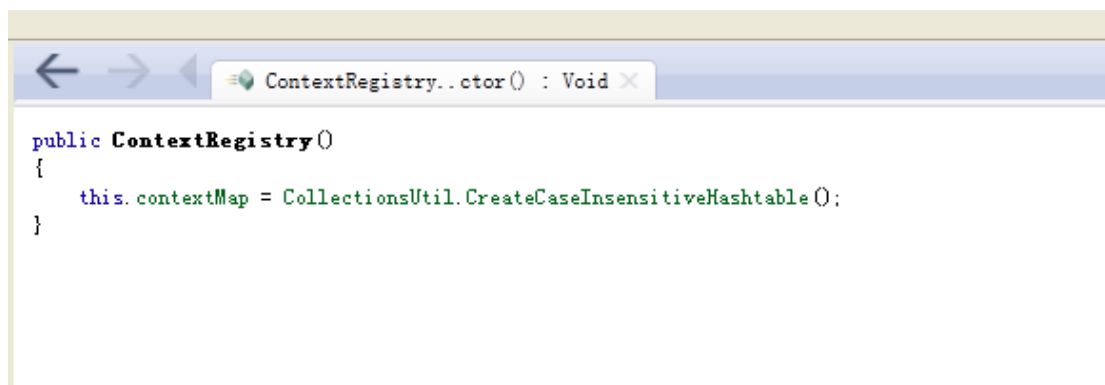


图 3

发现这里对管理容器的字典进行了实例化。

从字面上，毫无疑问，一个 hash 表，键对大小小敏感。

在我阅读的大量技术资料中，提到众多缓存组件均采用 hash 表这种数据结构。包括大名鼎鼎的分布式缓存 Memcache，redis 等，原因是 hash 表查找效率极高，易管理，并且线程安全。Spring 中对容器对象的管理也采用了 hash 表的数据结构，不多说。

dll 入口内容说完，看 spring 配置文件

这段代码是 spring 源码中的一段配置，形式已经固定。不多说。

在 configSections 中自定义配置节。并且配置节点处理器

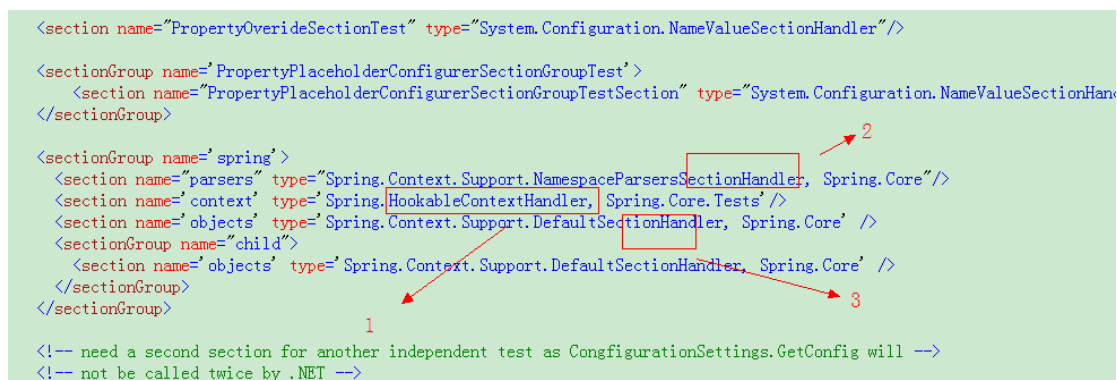


图 4

Huatong 生产中的配置，一样的

```
<configuration>
  <configSections>
    <sectionGroup name="spring">
      <section name="context" type="Spring.Context.Support.WebContextHandler, Spring.Web"/>
      <section name="parsers" type="Spring.Context.Support.NamespaceParsersSectionHandler, Spring.Core"/>
      <section name="objects" type="Spring.Context.Support.DefaultSectionHandler, Spring.Core"/>
    </sectionGroup>
    <section name="databaseSettings" type="System.Configuration.NameValueSectionHandler" />
  </configSections>

  <!--spring配置-->
  <spring xmlns="http://www.springframework.net">
    <parsers>
      <parser type="Spring.Data.Config.DatabaseNamespaceParser, Spring.Data" />
      <parser type="Spring.Transaction.Config.TxNamespaceParser, Spring.Data" />
    </parsers>
    <context>
      <resource uri="config://spring/objects" />
      <!--Service 配置-->
      <resource uri="assembly://Services/Services.Config/AdminService.xml" />
    </context>
  </spring>
</configuration>
```

图 5

不同仅仅在于 context 节点的节点处理器不一样，这个后面再说。

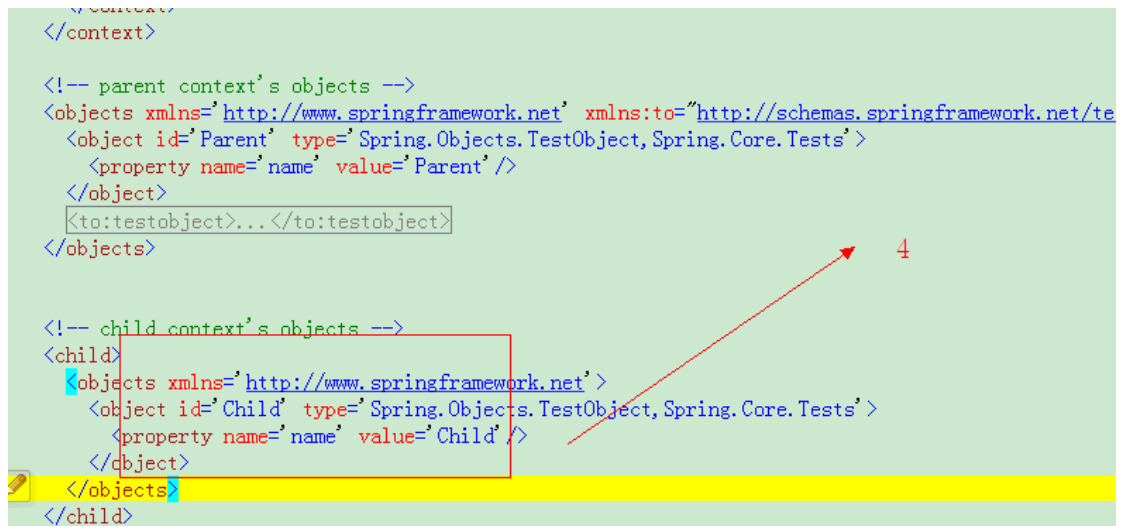
因为前面源码中的那个配置，是跑单元测试要用的 config，所以自定义了一个节点处理器。这里跟我要讲的不会有很大关系。不多说。

在看下 context 和 object，还有 parser 节点的详细配置

```
<spring>
  <parsers>
    <parser namespace="http://schemas.springframework.net/testobject"
      type="Spring.Context.Support.TestObjectConfigParser, Spring.Core.Tests"
      schemaLocation="assembly://Spring.Core.Tests/Spring.Context.Support/testobject.xsd"/>
  </parsers>

  <!-- parent context -->
  <context
    type="Spring.Context.Support.XmlApplicationContext, Spring.Core"
    name="Parent">
    <resource uri="config://spring/objects" />
    <!-- child context -->
    <context name="Child">
      <resource uri="config://spring/child/objects" />
    </context>
  </context>

  <!-- parent context's objects -->
  <objects xmlns="http://www.springframework.net" xmlns:to="http://schemas.springframework.net/testobject">
    <object id="Parent" type="Spring.Objects.TestObject, Spring.Core.Tests">
      <property name="name" value="Parent" />
    </object>
    <to:testobject>
      <to:age>12</to:age>
      <to:name>John</to:name>
    </to:testobject>
  </objects>
</spring>
```



```

</context>

<!-- parent context's objects -->
<objects xmlns='http://www.springframework.net' xmlns:to='http://schemas.springframework.net/te
  <object id='Parent' type='Spring.Objects.TestObject, Spring.Core.Tests'>
    <property name='name' value='Parent' />
  </object>
  <to:testobject>...</to:testobject>
</objects>

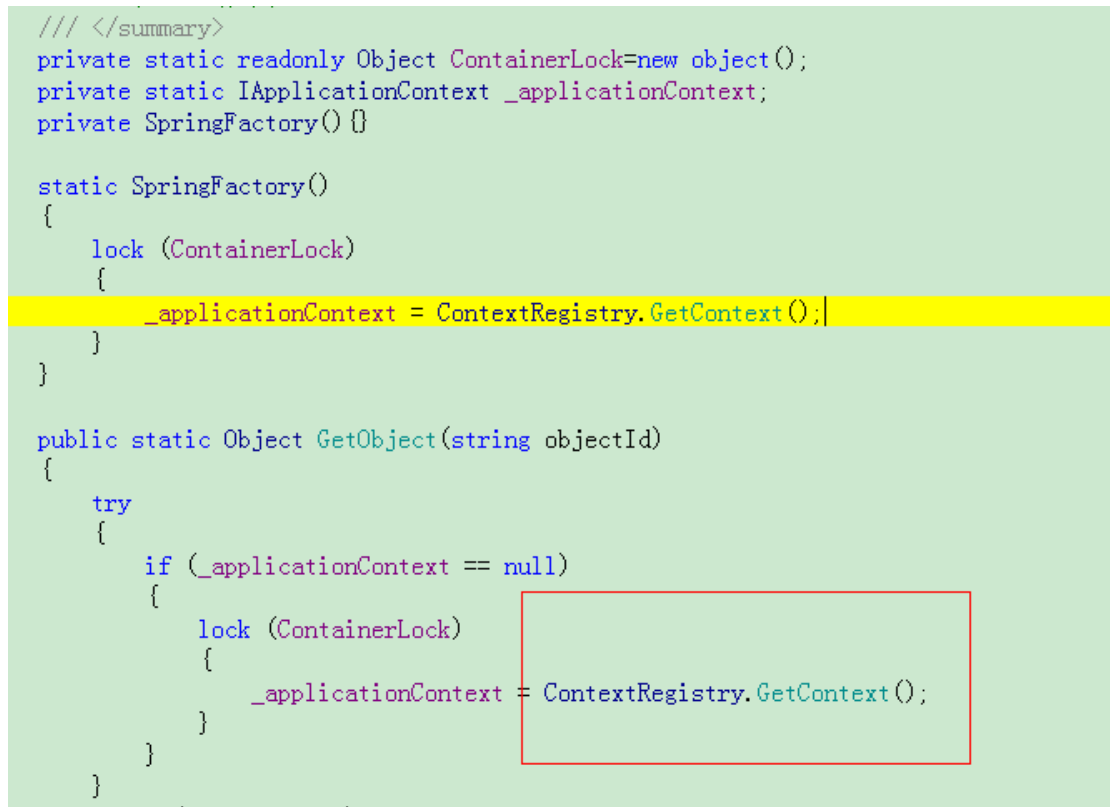
<!-- child context's objects -->
<child>
  <objects xmlns='http://www.springframework.net'>
    <object id='Child' type='Spring.Objects.TestObject, Spring.Core.Tests'>
      <property name='name' value='Child' />
    </object>
  </objects>
</child>

```

图 6

- 1 处资源解析器
- 2 容器配置，注意本处出现了容器嵌套，注意 resource 节点，指定资源为 config 类型，并且指定了 child 和 parent 下的 objects
- 3 处，指定父容器所管理配置的对象
- 4 处，容器所管理的对象

DII 预览和 config 配置预览结束。来看下生产环境是如何用 spring.net IOC 的，看图



```

/// </summary>
private static readonly Object ContainerLock=new object();
private static IApplicationContext _applicationContext;
private SpringFactory() {}

static SpringFactory()
{
    lock (ContainerLock)
    {
        _applicationContext = ContextRegistry.GetContext();
    }
}

public static Object GetObject(string objectId)
{
    try
    {
        if (_applicationContext == null)
        {
            lock (ContainerLock)
            {
                _applicationContext = ContextRegistry.GetContext();
            }
        }
    }
}

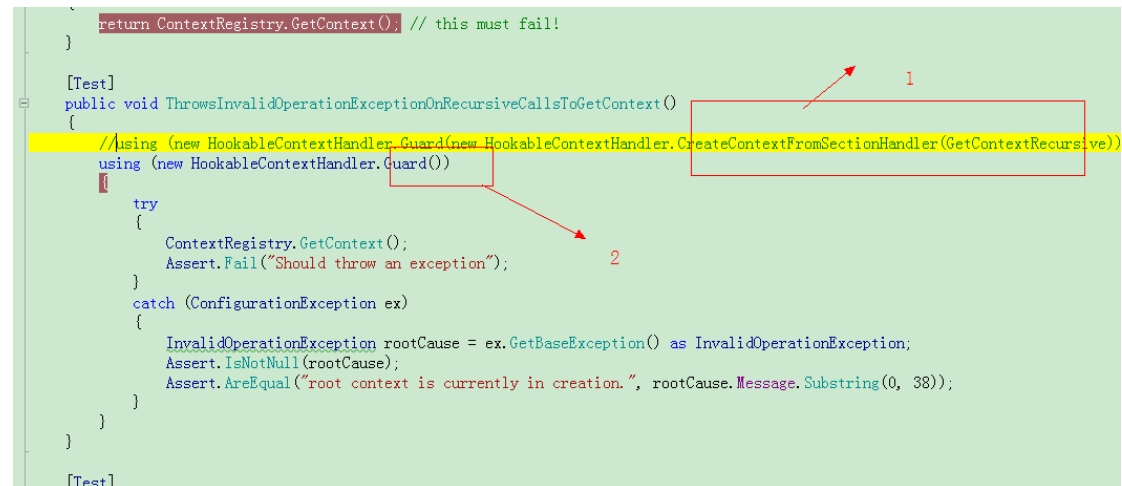
```

图 7

图 7 圈起来的就是我们用容器的入口，在这个方法内部，就进行了容器初始化处理。

要讲的就是整个容器是如何一步步初始化的。

拿源码中的单元测试代码



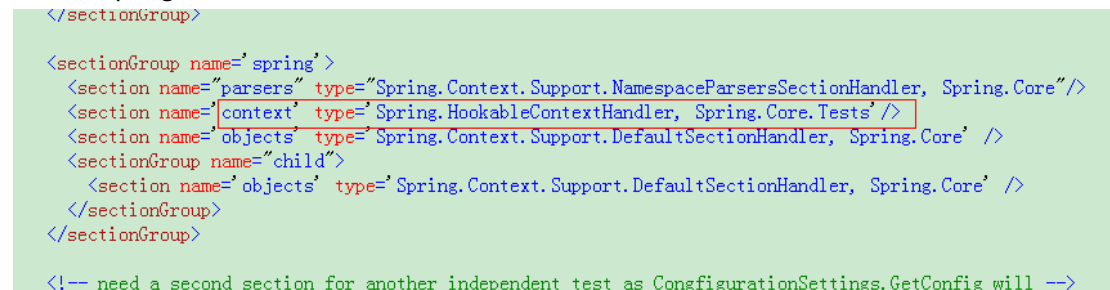
```
return ContextRegistry.GetContext(); // this must fail!
}

[Test]
public void ThrowsInvalidOperationExceptionOnRecursiveCallsToGetContext()
{
    //using (new HookableContextHandler.Guard(new HookableContextHandler.CreateContextFromSectionHandler(GetContextRecursive)))
    using (new HookableContextHandler.Guard())
    {
        try
        {
            ContextRegistry.GetContext();
            Assert.Fail("Should throw an exception");
        }
        catch (ConfigurationException ex)
        {
            InvalidOperationException rootCause = ex.GetBaseException() as InvalidOperationException;
            Assert.IsNotNull(rootCause);
            Assert.AreEqual("root context is currently in creation.", rootCause.Message.Substring(0, 38));
        }
    }
}

[Test]
```

图 8

这个是单元测试的入口，要注意的 就是 1 处。1 处是原开发者写的测试代码，2 处是我改过的。这里的 using，无非就是重新指定 context 的节点处理器，完全就是指定一个委托实例。还记得 spring\context 节点处理器么



```
</sectionGroup>

<sectionGroup name=' spring' >
  <section name='parsers' type='Spring.Context.Support.NamespaceParsersSectionHandler, Spring.Core' />
  <section name='context' type='Spring.HookableContextHandler, Spring.Core.Tests' />
  <section name='objects' type='Spring.Context.Support.DefaultSectionHandler, Spring.Core' />
  <sectionGroup name='child'>
    <section name='objects' type='Spring.Context.Support.DefaultSectionHandler, Spring.Core' />
  </sectionGroup>
</sectionGroup>

<!-- need a second section for another independent test as ConfigurationSettings.GetConfig will -->
```

图 9

打开 guard 方法，guard 方法是分配 context 的委托实例。

```

HookableContextHandler.Guard
{
    private IConfigurationSectionHandler baseHandler = new ContextHandler();

    /// <summary>
    /// May be used to wrap codeblocks within using(new Guard(new CreateContextFromSectionHandle
    /// </summary>
    public class Guard : IDisposable
    {
        private readonly CreateContextFromSectionHandler _prevInst;

        /// <summary>
        /// Initializes a new instance of the <see cref="T:System.Object"></see> class.
        /// </summary>
        public Guard(CreateContextFromSectionHandler sectionHandler)
        {
            NUnit.Framework.Assert.IsNotNull(sectionHandler);
            _prevInst = SetSectionHandler(sectionHandler);
        }

        public Guard()
        {
        }

        public void Dispose()
        {
            SetSectionHandler(_prevInst);
        }
    }
}

```

图 10

1 处指一个默认的 context 节点处理器。这个是 ContextHandle 是所有 context 节点处理器的父类。自己是可以自定义节点处理器的。

回到单元测试。现在我们通过调用我无参数 guid()方法，不分配具体的委托实例，那么 context 节点处理器则会按照 HookableContextHandler 内部默认处理器实例来处理。

对了， 在开始调试之前，有的人对配置的节点处理器有疑问。简单说下，是这样的。我自己查阅资料并实验。发现，当 ConfigManager.GetSection(string name)这个方法调用的时候，代码会触发进入到 ContextHandle 内部。说明什么呢？说明 getSection 内部有一个委托，而我们配置的节点处理器，都是满足这个委托的类型（签名和返回值），一旦 getSecion 读取指定节点，那么将会回调对应的 Handle，相当于一个触发的作用。

具体就是，当我们从 config 从读取 context 节点内容时候，将调用 context 节点处理器处理读取 Object 节点的时候，将调用 Objects 节点处理器，当读取 parser 节点的时候，将调用 parser 节点处理器。

那么既然这样。我们可以预言，spring 容器的初始化一定是在 contextHandle 内部完成的，真的如此么，擦亮眼睛，一步步看。

开始调试，睁大眼睛

```

78     }
79
80     [Test]
81     public void ThrowsInvalidOperationExceptionOnRecursiveCallsToGetContext()
82     {
83         //using (new HookableContextHandler.Guard(new HookableContextHandler.CreateContext))
84         using (new HookableContextHandler.Guard())
85         {
86             try
87             {
88                 ContextRegistry.GetContext();
89                 Assert.Fail("Should throw an exception");
90             }
91             catch (ConfigurationException ex)
92             {
93                 InvalidOperationException rootCause = ex.GetBaseException() as InvalidOperationException;
94                 Assert.IsNotNull(rootCause);
95                 Assert.AreEqual("root context is currently in creation.", rootCause.Message);
96             }
97         }
98     }

```

代码已经进来，单步，go

```

245         /// </p>
246         /// </remarks>
247         /// <returns>The root application context.</returns>
248         public static IApplicationContext GetContext()
249         {
250             lock (syncRoot)
251             {
252                 InitializeContextIfNeeded();
253                 if (rootContextName == null)
254                 {
255                     throw new ApplicationContextException(
256                         "No context registered. Use the 'RegisterContext' method or
257                     }
258                     return GetContext(rootContextName);
259                 }
260             }
261         }

```

Context 初始化，go

```

372
373
374     private static bool rootContextCurrentlyInCreation;
375
376     private static void InitializeContextIfNeeded()
377     {
378         if (rootContextName == null)
379         {
380             if (rootContextName == null)
381             {
382                 if (rootContextCurrentlyInCreation)
383                 {
384                     throw new InvalidOperationException("root context is currently in creation. You
385                 }
386
387                 rootContextCurrentlyInCreation = true;
388                 try
389                 {
390                     ConfigurationUtils.GetSection(ApplicationContext.ContextSectionName);
391                 }
392                 finally
393                 {
394                     rootContextCurrentlyInCreation = false;
395                 }
396             }
397         }
398     }

```

```

(rootContextName == null)

if (rootContextCurrentlyInCreation)
{
    throw new InvalidOperationException("root context is currently in creation. You must not call ContextRegistry.GetContext() from e.g. co
}

rootContextCurrentlyInCreation = true;
try
{
    ConfigurationUtils.GetSection(AbstractApplicationContext.ContextSectionName);
}
}

```

初始根容器对象名称为 null, 从结构上看, 修改容器初始化标, rootContextCurrentlyInCreation, 初始是 false。

注意我圈红的地方, contextSectionname, 默认为"spring/context", 打开 spring.core.dll, 看下

```

private static void InitializeContextIfNeeded()

{
    if (rootContextName == null)
    {
        if (rootContextCurrentlyInCreation)
        {
            throw new InvalidOperationException("root context is currently in creation. You must not call Con
        }
        rootContextCurrentlyInCreation = true;
        try
        {
            ConfigurationUtils.GetSection("spring/context");
        }
        finally
        {
            rootContextCurrentlyInCreation = false;
        }
    }
}

```

看源码

```

/// an object in the context, with the special, well-known-name of
/// <c>"messageSource"</c>. Else, message resolution is delegated to the
/// parent context.
/// </p>
/// </remarks>
/// <author>Rod Johnson</author>
/// <author>Juergen Hoeller</author>
/// <author>Griffin Caprio</author>
/// <seealso cref="Spring.Objects.Factory.Config.IObjectPostProcessor"/>
/// <seealso cref="Spring.Objects.Factory.Config.IObjectFactoryPostProcessor"/>
public abstract class AbstractApplicationContext
: ConfigurableResourceLoader, IConfigurableApplicationContext, IObjectDefinitionRegistry
{
    #region Constants

    /// <summary>
    /// Name of the .Net config section that contains Spring.Net context definition.
    /// </summary>
    public const string ContextSectionName = "spring/context";

    /// <summary>
    /// Default name of the root context.
    /// </summary>
    public const string DefaultRootContextName = "spring.root";

    #endregion
}

```

都是作为常量定死的。我看到了大神 rod Johnson, 和 griffin caprio, 的名字, spring 的缔造者。众多 geek 的偶像。神一般的人物, 后者现在是一家公司的 cto, 刚翻墙去加了他的 twitter。



嗯，写了不少技术和管理的文章。最厉害的是 Johnson，学音乐的，我擦，居然成了码神，让我等码农情何以堪。不吐槽了，继续正题。

马上进入 try 块，开始读取配置节，单步 go，此时将读取 context 配置节，将进入 contexthandler

```
/// <summary>
/// Replace the original context handler with this hookable version for testing ContextRegistry
/// </summary>
/// <author>Erich Eichinger</author>
public class HookableContextHandler : IConfigurationSectionHandler
{
    private IConfigurationSectionHandler baseHandler = new ContextHandler();

    /// <summary>
    /// May be used to wrap codeblocks within using(new Guard(new CreateContextFromSectionHandler(MyTestSectionHandler))) { ... }
    /// </summary>
    public class Guard : IDisposable
    {
        private readonly CreateContextFromSectionHandler _prevInst;

        /// <summary>
        /// Initializes a new instance of the <see cref="T:System.Object"></see> class.
        /// </summary>
        public Guard(CreateContextFromSectionHandler sectionHandler)
        {
            NUnit.Framework.Assert.IsNotNull(sectionHandler);
            prevInst = SetSectionHandler(sectionHandler);
        }
    }
}
```

代码进来，默认的节点处理是 ContextHandle 实例，继续 go

```
79         s_callback = sectionHandler;
80         return prevInstance;
81     }
82
83     /// <summary>
84     /// Creates a configuration section.
85     /// </summary>
86     ///
87     /// <returns>
88     /// The created section object.
89     /// </returns>
90     ///
91     /// <param name="parent">Parent object.</param>
92     /// <param name="section">Section XML node.</param>
93     /// <param name="configContext">Configuration context object.</param>
94     public object Create(object parent, object configContext, XmlNode section)
95     {
96         if (s_callback != null)
97         {
98             return s_callback(parent, configContext, section);
99         }
100
101         return baseHandler.Create(parent, configContext, section);
102     }
103 }
```

进来，代码进到 create 方法内部来了。也就是说，当读取 config 指定节点的时候，会调用 Handle 处理器中的 Create 方法。

三个参数，parent 父对象，configcontext 配置上下文对象，section，指定节点下所有的内容。这个是 ms 封装好的。自定义节点处理器必须实现 IConfigurationSectionHandler 接口

```
/// Replace the original context handler with this hookable version for testing Co
/// </summary>
/// <author>Erich Eichinger</author>
public class HookableContextHandler : IConfigurationSectionHandler
{
    private IConfigurationSectionHandler baseHandler = new ContextHandler();
}
```

其实就是实现上面提到的 create 方法。不多说，继续 go

```

    ///Creates a configuration section.
    ///</summary>
    ///
    ///</returns>
    ///The created section object.
    ///</returns>
    ///
    ///<param name="parent">Parent object.</param>
    ///<param name="section">Section XML node.</param>
    ///<param name="configContext">Configuration context object.</param><filterpriority>2</filterpriority>
    public object Create(object parent, object configContext, XmlNode section)
    {
        if (s_callback != null)
        {
            return s_callback(parent, configContext, section);
        }
        return baseHandler.Create(parent, configContext, section);
    }
}

```

2 处，未发现自定义 context 节点处理器（无委托方法），使用基类 ContextHandle 处理器处理。Go

The screenshot shows the Visual Studio IDE with the `Spring.Context.Support.ContextHandler` class open. The `Create` method is highlighted, and the `InnerXml` property of the `XmlElement` object is being inspected. The `InnerXml` property is circled in red, and the `contextElement` object is also circled in red. The `InnerXml` property is of type `string` and its value is `<context names='Child'><resource uri='config://spring/child/objects' /></context>`.

注意我圈红的地方，查看 innerxml，发现 context 节点内部的内容，ok，看下 config 是否一致

```

</parsers>

<!-- parent context -->
<context
  type='Spring.Context.Support.XmlApplicationContext, Spring.Core'
  name='Parent'
  <resource uri='config://spring/objects' />
  <!-- child context -->
  <context name='Child'
    <resource uri='config://spring/child/objects' />
  </context>
</context>

<!-- parent context's objects -->
<objects uri='http://www.springframework.org/test' xmlns='http://www.springframework.org/test' />

```

嗯，完全一致，继续，go

```

#endregion

// determine name of context to be created
string contextName = GetContextName(configContext, contextElement);
if (!StringUtils.HasLength(contextName))
{
    contextName = AbstractApplicationContext.DefaultRootContextName;
}

#region Instrumentation
if (Log.IsDebugEnabled) Log.Debug(string.Format("creating contextElement {Element, Name=\"context\"}"));
#endregion

IApplicationContext context = null;
try
{
    IApplicationContext parentContext = parent as IApplicationContext;

    // determine context type
    //获取context的真实类型
    Type contextType = GetContextType(contextElement, parentContext);

```

对 context 名字进行处理，即将进入容器配置的对象加载过程，睁大眼睛  
Go

```

IApplicationContext context = null;
try
{
    IApplicationContext parentContext = parent as IApplicationContext;

    // determine context type
    //获取context的真实类型
    Type contextType = GetContextType(contextElement, parentContext);

    // determine case-sensitivity
    ///读取config中context声明的caseSensitive属性，是否对大小写敏感，默认为true
    bool caseSensitive = GetCaseSensitivity(contextElement);

    // get resource-list
    //获取配置资源文件
    IList<string> resources = GetResources(contextElement);

    // finally create the context instance
    context = InstantiateContext(parentContext, configContext, contextName, contextType, caseSensitive);
    // and register with global context registry
    if (AutoRegisterWithContextRegistry && !ContextRegistry.IsContextRegistered(context.Name))
    {
        ContextRegistry.RegisterContext(context);
    }
}

```

从上到下，初始化要返回的 context，看我的注释，好理解。最重要的是 resources 的获取，睁大眼睛，go

```

}

// <summary>
// Extracts the context-type from the context element.
// If none is specified, returns the parent's type.
// </summary>
private Type GetContextType(XmlElement contextElement, IApplicationContext parentContext)
{
    Type contextType;
    if (parentContext != null)
    {
        // set default context type to parent's type (allows for type inheritance)
        contextType = GetConfiguredContextType(contextElement, parentContext.GetType());
    }
    else
    {
        contextType = GetConfiguredContextType(contextElement, this.DefaultApplicationContextType);
    }
    return contextType;
}

```

获取 context 类型，本处默认 xmlApplicationContext

继续 Go，读取 context 的一个的配置类型属性，再次去读 context 配置节，再次触发 context 节点处理器，代码会进对应的 handle，获取真实类型

Go

```

// determine context type
//获取context的真实类型
Type contextType = GetContextType(contextElement, parentContext);
// determine case-sensitivity
//读取config中context声明的caseSensitive属性，是否对大小写敏感，默认为true
bool caseSensitive = GetCaseSensitivity(contextElement);

// get resource-list
//获取配置资源文件
IList<string> resources = GetResources(contextElement);

```

1 处观察到真实类型为 xmlApplicationContext，2 处读 config，读 context 节点配置是否大小写敏感。默认 true

Go

```

// determine context type
//获取context的真实类型
Type contextType = GetContextType(contextElement, parentContext);

// determine case-sensitivity
//读取config中context声明的caseSensitive属性，是否对大小写敏感，默认为true
bool caseSensitive = GetCaseSensitivity(contextElement);

// get resource-list
//获取配置资源文件
IList<string> resources = GetResources(contextElement);

// finally create the context instance
context = InstantiateContext(parentContext, configContext, contextName, contextType, caseSensitive, resour

```

取到值是 true

Go

接下来是非常重要的一步，加载 context 节点下配置的 objects，这里 context 节点下读取的配置支持多重协议，http，uri，config，ftp 等等，resources 就是要将这些所有支持的配置协议类型文件中配置的对象全部读取出来。

Go

```
/// Returns the array of resources containing object definitions for
/// this context.
/// </summary>
private IList<string> GetResources( XElement contextElement )
{
    List<string> resourceNodes = new List<string>(contextElement.ChildNodes.Count);
    foreach (XmlNode possibleResourceNode in contextElement.ChildNodes)
    {
        XElement possibleResourceElement = possibleResourceNode as XElement;
        if(possibleResourceElement != null &&
            possibleResourceElement.LocalName == ContextSchema.ResourceElement)
        {
            string resourceName = possibleResourceElement.GetAttribute(ContextSchema.URIAttribute);
            if(StringUtils.HasText(resourceName))
            {
                resourceNodes.Add(resourceName);
            }
        }
    }
}
```

注意 contextElement

Go

```
List<string> resourceNodes = new List<string>(contextElement.ChildNodes.Count);
foreach (XmlNode possibleResourceNode in contextElement.ChildNodes)
```

发现 context 下有三个子节点

Config 中是不是有三个子节点，看 config

```
<!-- parent context -->
<context
  type='Spring.Context.Support.XmlApplicationContext, Spring.Core'
  name='Parent'>
  <resource uri='config://spring/objects' />
  <!-- child context -->
  <context name='Child'>
    <resource uri='config://spring/child/objects' />
  </context>
</context>
```

圈起来的 123，两个资源节点，一个文本节点

Go

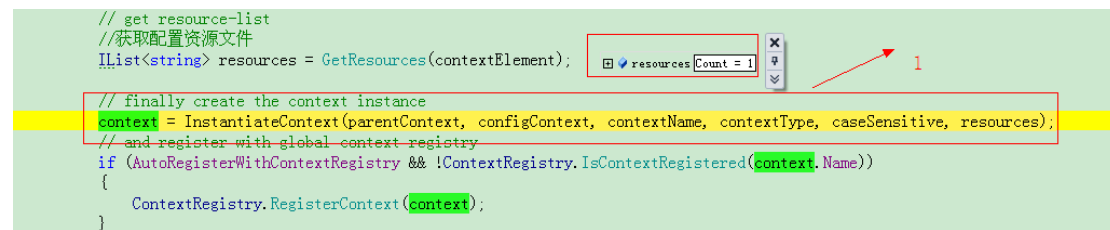
```
456 /// this context.
457 /// </summary>
458 private IList<string> GetResources( XElement contextElement )
459 {
460     List<string> resourceNodes = new List<string>(contextElement.ChildNodes.Count);
461     foreach (XmlNode possibleResourceNode in contextElement.ChildNodes)
462     {
463         XElement possibleResourceElement = po
464         if(possibleResourceElement != null &&
465             possibleResourceElement.LocalName == ContextSchema.ResourceElement)
466         {
467             string resourceName = possibleResourceElement.GetAttribute(ContextSchema.URIAttribute);
468             if(StringUtils.HasText(resourceName))
469             {
470                 resourceNodes.Add(resourceName);
471             }
472         }
473     }
474     return resourceNodes;
475 }
```

这里发现 resourceNodes 返回只有一个节点，那么意味着，父容器配置的资源节点全部被检索出来了，而子容器配置的资源节点和文本节点均被舍弃。注意，当前初始化的容器是父容器。现在要做的工作是，为父容器注入要管理的对象类型

Go

```
// get resource-list
//获取配置资源文件
IList<string> resources = GetResources(contextElement);

// finally create the context instance
context = InstantiateContext(parentContext, configContext, contextName, contextType, caseSensitive, resources);
// and register with global context registry
if (AutoRegisterWithContextRegistry && !ContextRegistry.IsContextRegistered(context.Name))
{
    ContextRegistry.RegisterContext(context);
}
```



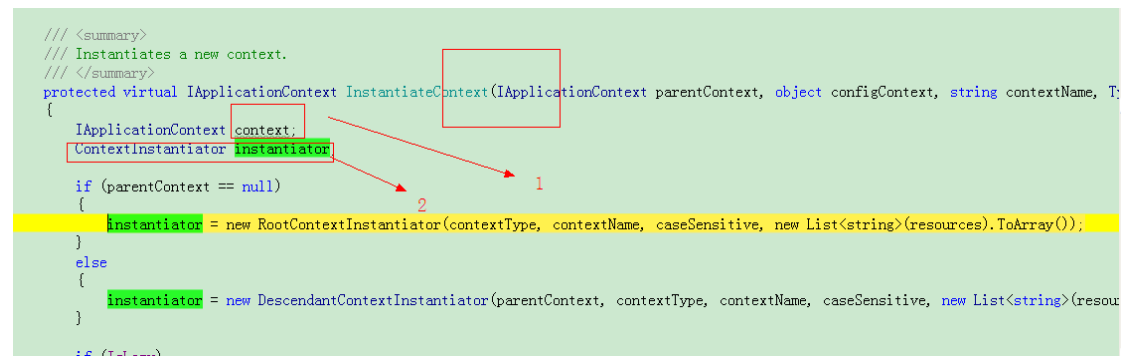
1 处初始化父容器，go

```
/// <summary>
/// Instantiates a new context.
/// </summary>
protected virtual IApplicationContext InstantiateContext(IApplicationContext parentContext, object configContext, string contextName, T
{
    IApplicationContext context;
    ContextInstantiator instantiator;

    if (parentContext == null)
    {
        instantiator = new RootContextInstantiator(contextType, contextName, caseSensitive, new List<string>(resources).ToArray());
    }
    else
    {
        instantiator = new DescendantContextInstantiator(parentContext, contextType, contextName, caseSensitive, new List<string>(resou
    }

    if (!IsLazy)
    {
        context = instantiator.InstantiateContext();
    }
    else
    {
        context = new Lazy<IApplicationContext>(() => instantiator.InstantiateContext());
    }

    return context;
}
```



进入 InstantiateContext 内部，1 处定义要返回的容器对象，2 处定义一个容器初始化器

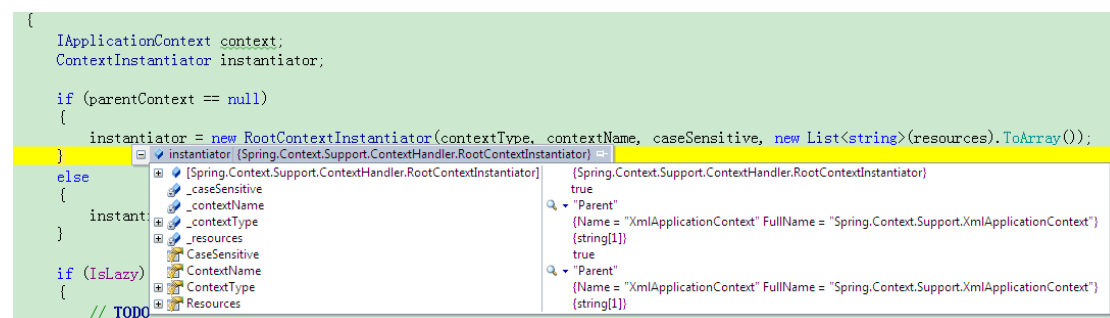
继续 go。

```
{
    IApplicationContext context;
    ContextInstantiator instantiator;

    if (parentContext == null)
    {
        instantiator = new RootContextInstantiator(contextType, contextName, caseSensitive, new List<string>(resources).ToArray());
    }
    else
    {
        instantiator = new DescendantContextInstantiator(parentContext, contextType, contextName, caseSensitive, new List<string>(resou
    }

    if (!IsLazy)
    {
        context = instantiator.InstantiateContext();
    }
    else
    {
        context = new Lazy<IApplicationContext>(() => instantiator.InstantiateContext());
    }

    return context;
}
```

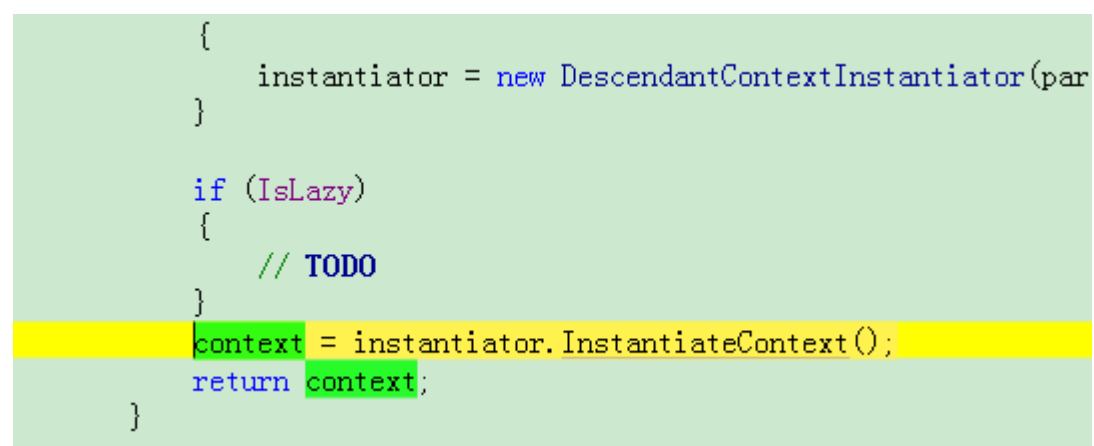


用已经得到的资源和相关参数，创建一个容器初始化器实例，紧接着干什么，用容器初始化器进行容器初始化，

```
{
    instantiator = new DescendantContextInstantiator(par
}

if (IsLazy)
{
    // TODO
}

context = instantiator.InstantiateContext();
return context;
}
```



Go, 进入 InstantiateContext 方法

```
    }
    _resources = resources;

    public IApplicationContext InstantiateContext()
    {
        ConstructorInfo ctor = GetContextConstructor();
        if (ctor == null)
        {
            string errorMessage = "No constructor with string[] argument found for context type [" + ContextType.Name + "]";
            throw ConfigurationUtils.CreateConfigurationException(errorMessage);
        }
        IApplicationContext context = InvokeContextConstructor(ctor);
        return context;
    }

    protected abstract ConstructorInfo GetContextConstructor();
```

1 处, 发现, 这里需要容器的构造器信息, 我们要初始化一个容器, 必然要知道他的构造器是什么样子的, 1 处的这个方法, 就是获取容器构造器信息, 到这里, 代码越来越难, 越来越超过我们的学习范围, 没事, 走流程, 看懂每一步就 ok

进入 GetContextConstructor

```
    : base(contextType, contextName, caseSensitive, resources)
    {
    }

    protected override ConstructorInfo GetContextConstructor()
    {
        return ContextType.GetConstructor(new Type[] { typeof(string), typeof(bool), typeof(string[]) });
    }

    protected override IApplicationContext InvokeContextConstructor(
        ConstructorInfo ctor)
    {
        return (IApplicationContext)(new SafeConstructor(ctor).Invoke(new object[] { ContextName, CaseSensitive, Resources }));
    }
}
```

传入参数类型数组作为参数, ContextType 为 XmlApplicationContext 类型

Go, 进入 GetContextConstructor 内部

```
506     }
507
508     public IApplicationContext InstantiateContext()
509     {
510         ConstructorInfo ctor = GetContextConstructor();
511         if (ctor == null)
512         {
513             string errorMessage = "No constructor with string[] argument found for context type [" + ContextType.Name + "]";
514             throw ConfigurationUtils.CreateConfigurationException(errorMessage);
515         }
516         IApplicationContext context = InvokeContextConstructor(ctor);
517         return context;
518     }
519 }
```

属性	值
CallingConvention	Standard   HasThis
ContainsGenericParameters	false
DeclaringType	{Name = "XmlApplicationContext" FullName = "Spring.Context.Support.XmlApplicationContext"}
IsSecurityCritical	true
IsSecuritySafeCritical	false
IsSecurityTransparent	false
MemberType	Constructor
MetadataToken	100668346
MethodHandle	{System.Runtime.MethodHandle}

拿到构造器信息 ctor, 得到类型和参数

继续 go，调用 InvokeContextConstructor，传入构造器信息 ctor

```
    }  
  
    protected override IApplicationContext InvokeContextConstructor(  
        ConstructorInfo ctor)  
    {  
        return (IApplicationContext)(new SafeConstructor(ctor).Invoke(new object[] { ContextName, CaseSensitive, Resources }));  
    }  
}
```

继续 go，接下来的代码是 我有耳闻但是从来不知道是什么的东西代码，继续

拿 ctor 创建一个 safeConstructor，究竟他是干嘛的，我也不清楚，继续

```
#endregion  
  
private ConstructorDelegate constructor;  
  
/// <summary>  
/// Creates a new instance of the safe constructor wrapper.  
/// </summary>  
/// <param name="constructorInfo">Constructor to wrap.</param>  
public SafeConstructor(ConstructorInfo constructorInfo)  
{  
    this.constructorInfo = constructorInfo;  
    this.constructor = GetOrCreateDynamicConstructor(constructorInfo);  
}  
  
/// <summary>
```

从代码和源码注释可以看处，这里有两个字段，分别接受构造器信息，通过构造器信息动态创建一个构造器，应该属于反射的内容

继续，go 进入 GetOrCreateDynamicConstructor

```
#region Constructors  
  
private static readonly IDictionary<ConstructorInfo, ConstructorDelegate> constructorCache = new Dictionary<ConstructorInfo, ConstructorDelegate>();  
  
/// <summary>  
/// Obtains cached constructor info or creates a new entry, if none is found.  
/// </summary>  
private static ConstructorDelegate GetOrCreateDynamicConstructor(ConstructorInfo constructorInfo)  
{  
    ConstructorDelegate method;  
    if (!constructorCache.TryGetValue(constructorInfo, out method))  
    {  
        method = DynamicReflectionManager.CreateConstructor(constructorInfo);  
        lock (constructorCache)  
        {  
            constructorCache[constructorInfo] = method;  
        }  
    }  
    return method;  
}  
  
#endregion
```

有一个构造方法委托，首先进来从缓存中取构造方法，没有的话，通过 DynamicReflectionManager.CreateConstructor(constructorInfo)得到一个构造函数，并且安全缓存下来

继续 go，进 CreateConstructor 内部



```

///<summary>
/// Creates a new delegate for the specified constructor.
///</summary>
///<param name="constructorInfo">the constructor to create the delegate for</param>
///<returns>delegate that can be used to invoke the constructor.</returns>
public static ConstructorDelegate CreateConstructor(ConstructorInfo constructorInfo)
{
    AssertUtils.ArgumentNotNull(constructorInfo, "You cannot create a dynamic constructor for a null value.");

    bool skipVisibility = true; //!IsPublic(constructorInfo);
    System.Reflection.Emit.DynamicMethod dmGetter;
    Type[] argumentTypes = new Type[] { typeof(object[]) };
    dmGetter = CreateDynamicMethod(constructorInfo.Name, typeof(object), argumentTypes, constructorInfo, skipVisibility);
    ILGenerator il = dmGetter.GetILGenerator();
    EmitInvokeConstructor(il, constructorInfo, false);
    ConstructorDelegate ctor = (ConstructorDelegate)dmGetter.CreateDelegate(typeof(ConstructorDelegate));
    return ctor;
}

```

返回类型是一个委托类型。内部的代码，应该是使用 emit 直接写 IL 代码。查看 ctor

式 (E):

ystem.Delegate) (ctor))

?:

名称	值
ctor	(Method = (System.Object _dynamic_Spring.Context.Support.XmlApplicationContext..ctor(System.Object[]))
base	(Method = (System.Object _dynamic_Spring.Context.Support.XmlApplicationContext..ctor(System.Object[]))
Method	(Method = (System.Object _dynamic_Spring.Context.Support.XmlApplicationContext..ctor(System.Object[]))
Target	(System.Object _dynamic_Spring.Context.Support.XmlApplicationContext..ctor(System.Object[]))
非公共成员	null
非公共成员	
(Spring.Reflection.Dynamic.ConstructorDelegate)	(Method = (System.Object _dynamic_Spring.Context.Support.XmlApplicationContext..ctor(System.Object[]))
base	(Method = (System.Object _dynamic_Spring.Context.Support.XmlApplicationContext..ctor(System.Object[]))
invocationCount	0
invocationList	null

我也看不懂。继续，返回一个委托类型，这里已经可以看到 ctor 的类型了。

继续 go

```

#endregion

private ConstructorDelegate constructor;

/// <summary>
/// Creates a new instance of the safe constructor wrapper.
/// </summary>
/// <param name="constructorInfo">Constructor to wrap.</param>
public SafeConstructor(ConstructorInfo constructorInfo)
{
    this.constructorInfo = constructorInfo;
    this.constructor = GetOrCreateDynamicConstructor(constructorInfo);
}

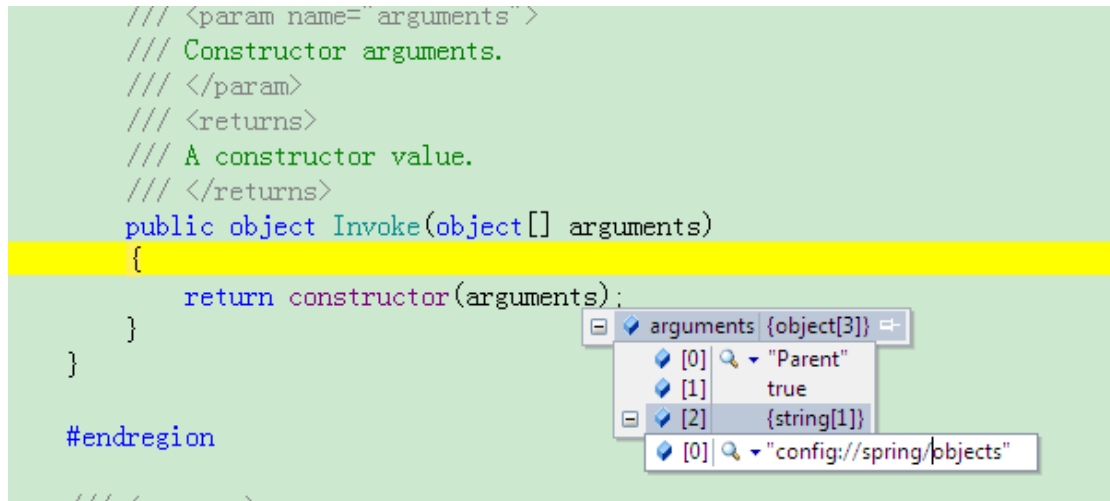
// this.constructor (Method = (System.Object _dynamic_Spring.Context.Support.XmlApplicationContext..ctor(System.Object[]))
// (System.MulticastDelegate)(this.constructor)) (Method = (System.Object _dynamic_Spring.Context.Support.XmlApplicationContext..ctor(System.Object[]))

/// <summary>
/// Invokes dynamic constructor.
/// </summary>
/// <param name="arguments">

```

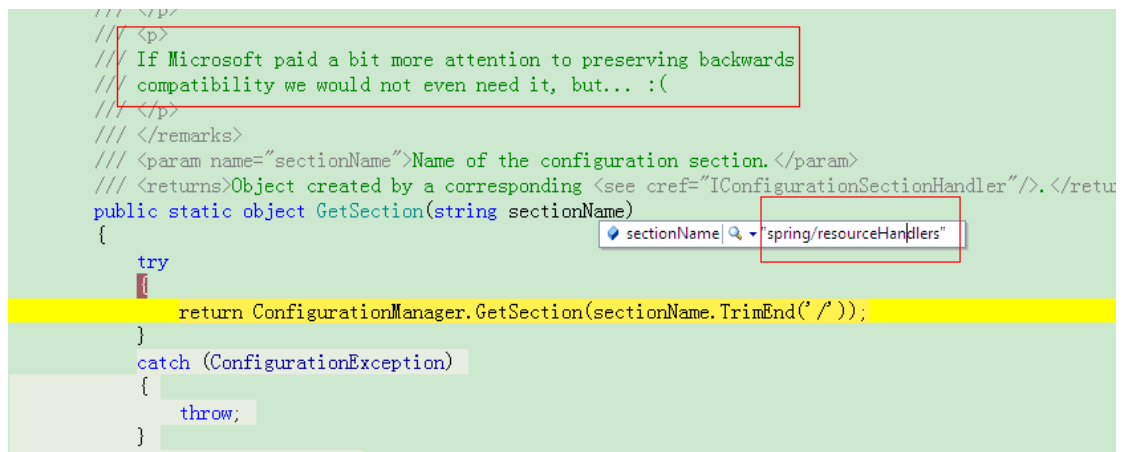
SafeConstructor 构造结束，俩属性分别接收了构造器信息和构造器

继续 go



调用 `SafeConstructor` 的 `Invoke` 方法，内部是拿构造器，和参数进行 `XmlApplicationContext` 实例创建的过程。参数中是有对象配置项的。看索引为 2 的参数。可以预料的是，这个时候，创建 `context` 实例。必然会再次读 `object` 配置节。会再次触发 `handle`

继续，获取配置的 `resource` 资源项解析器



大神也卖萌，看注释。

进入 `resourcehandle` 注册管理类



发现有各种资源项的处理器，ftp，config，file，assembly 等、继续  
注意 getSection，当我进入这个方法内部的时候，并没有调入到某个 handle 里面，为什么呢？  
是因为我的 config 文件中，spring 节点下并没有配置 resourcehandlers 节点，那么委托没有实例。也就是读这个节点的时候，不会触发之前我们看到的那种操作。不会进入到某个 handlerle 处理程序里。

继续 go

```
/// </para>
/// </remarks>
/// <param name="protocolName">Name of the protocol to get the handler for.</param>
/// <returns>Resource handler constructor for the specified protocol name.</returns>
/// <exception cref="ArgumentNullException">If <paramref name="protocolName"/> is <c>null</c>.</exception>
public static IDynamicConstructor GetResourceHandler(string protocolName)
{
    AssertUtils.ArgumentNotNull(protocolName, "protocolName");
    IDynamicConstructor constructor;
    resourceHandlers.TryGetValue(protocolName, out constructor);
    return constructor;
}
```

当前的 context 中的是资源配置项，协议走的是 config，

```
</parsers>

<!-- parent context -->
<context
  type='Spring.Context.Support.XmlApplicationContext, Spring.Core'
  name='Parent'>
  <resource uri='config://spring/objects' />
  <!-- child context -->
  <context name='Child'>
    <resource uri='config://spring/child/objects' />
  </context>
</context>

<!-- parent context's objects -->
```

那么根据协议类型获取一个 SafeConstructor

```
133 // to create an instance of the <see cref="IResource"/>-derived type by passing
134 // resource location as a parameter.
135 // </para>
136 // </remarks>
137 // <param name="protocolName">Name of the protocol to get the handler for.</param>
138 // <returns>Resource handler constructor for the specified protocol name.</returns>
139 // <exception cref="ArgumentNullException">If <paramref name="protocolName"/> is <c>null</c>.</exception>
140 public static IDynamicConstructor GetResourceHandler(string protocolName)
141 {
142     AssertUtils.ArgumentNotNull(protocolName, "protocolName");
143     IDynamicConstructor constructor;
144     resourceHandlers.TryGetValue(protocolName, out constructor);
145     return constructor;
146 }
```

100 %

resourceHandlers  
(new System.Collections.G

名称	值
静态成员	
resourceHandlers	System.Reflection.ConstructorInfo Count = 6
[0]	[[config, Spring.Reflection.Dynamic.SafeConstructor]]
[1]	[[file, Spring.Reflection.Dynamic.SafeConstructor]]
[2]	[[http, Spring.Reflection.Dynamic.SafeConstructor]]
[3]	[[https, Spring.Reflection.Dynamic.SafeConstructor]]
[4]	[[ftp, Spring.Reflection.Dynamic.SafeConstructor]]
[5]	[[assembly, Spring.Reflection.Dynamic.SafeConstructor]]
原始视图	

看前 3 图，预注册了 6 种协议的资源处理器，这里只需要根据 config 类型，取出一个 SafeConstructor 就行了，跟前面介绍的一样。有了这个东西，我们可以以反射的形式初始化一个实例，而且他是通过 Emit 直接写的 IL，效率很高，虽然不怎么懂，但是感觉很牛逼☺

继续

```
/// </exception>
/// <seealso cref="ResourceHandlerRegistry.RegisterResourceHandler(string, Type)"/>
public IResource GetResource(string resourceName)
{
    string protocol = GetProtocol(resourceName);
    if (protocol == null)
    {
        protocol = DefaultResourceProtocol;
        resourceName = protocol + ProtocolSeparator + resourceName;
    }

    IDynamicConstructor handler = ResourceHandlerRegistry.GetResourceHandler(protocol);
    if (handler == null)
    {
        throw new UriFormatException("Resource handler for the '" + protocol + "' protocol is not supported");
    }

    return (IResource) handler.Invoke(new object[] { resourceName });
}
```

resourceName | config://spring/objects"

返回一个 IResource 资源对象

继续

```
111. ConfigurationUtilities
public static object GetSection(string sectionName)
{
    /// compatibility we would not even need it, but... :(
    /// </p>
    /// </remarks>
    /// <param name="sectionName">Name of the configuration section.</param>
    /// <returns>Object created by a corresponding <see cref="IConfigurationSectionHandler"/>.</returns>
    public static object GetSection(string sectionName)
    {
        try
        {
            return ConfigurationManager.GetSection(sectionName.TrimEnd('/'));
        }
        catch (ConfigurationException)
        {
            throw;
        }
        catch (Exception ex)
        {
            throw CreateConfigurationException(string.Format("Error reading section {0}", sectionName), ex);
        }
    }
}
```

sectionName | "spring/objects"

读 spring/objects 节点下的内容

和前面一样，触发一个 handle，代码进入，为什么这里会触发呢，因为我在 config 中对 objects 节点配置了节点处理器

```

<sectionGroup name=' spring'>
  <section name="parsers" type="Spring.Context.Support.NamespaceParsersSectionHandler, Spring.Core"/>
  <section name='context' type=' Spring.HookableContextHandler, Spring.Core.Tests' />
  <section name='objects' type=' Spring.Context.Support.DefaultSectionHandler, Spring.Core' />
  <sectionGroup name="child">
    <section name=' objects' type=' Spring.Context.Support.DefaultSectionHandler, Spring.Core' />
  </sectionGroup>
</sectionGroup>

```

再次证明这里是指定委托类型。读取节点触发调用。

拿到 object 的 xml

```

/// </param>
/// <exception cref="System.ArgumentNullException">
///   If the supplied <paramref name="resourceName"/> is
///   <see langword="null"/> or contains only whitespace character(s).
/// </exception>
public ConfigSectionResource(string resourceName) : base(resourceName)
{
    AssertUtils.ArgumentHasText(resourceName, "resourceName");
    sectionName = GetResourceNameWithoutProtocol(resourceName);
    configElement = (XmlElement) ConfigurationUtils.GetSection(sectionName);
}

#endregion

```

装载配置的 Object 对象，返回配置的 object 的个数

```

s.Factory.Support.C.Ast.ObjectDefinitionsFromResource
LoadObjectDefinitions(string location)
/// The number of object definitions found
/// </returns>
public int LoadObjectDefinitions(string location)
{
    if (ResourceLoader == null)
    {
        throw new ObjectDefinitionStoreException("Cannot import object definitions from location
                                                    ":" +
                                                    " no ResourceLoader available");
    }
    IResource resource;
    try
    {
        resource = ResourceLoader.GetResource(location);
    } catch (Exception e)
    {
        throw new ObjectDefinitionStoreException("Could not resolve resource location [" + locati
    }
    int loadCount = LoadObjectDefinitions(resource);

    if (log.IsDebugEnabled)
    {
        log.Debug("Loaded " + loadCount + " object definitions from location [" + location + "]")
    }
    return loadCount;
}

```

继续

在读取以 xml 信息存在的 object 时候，发现读取了 parser 节点，进入

```
Spring.Core.Tests.dll.config | NamespaceParsersSectionHandler.cs | ConfigurationUtils.cs | XmlObjectDefinitionReader.cs | AbstractObjectDefinitionReader.cs
Spring.Context.Support.NamespaceParsersSectionHandler
Create(object parent, object configContext, XmlNode section)
95     /// The configuration context when called from the ASP.NET
96     /// configuration system. Otherwise, this parameter is reserved and
97     /// is <see langword="null"/>.
98     /// </param>
99     /// <param name="section">
100     /// The <see cref="System.Xml.XmlNode"/> for the section.
101     /// </param>
102     /// <returns>
103     /// This method always returns <see langword="null"/>, because parsers
104     /// are registered as a side-effect of this object's execution and there
105     /// is thus no need to return anything.
106     /// </returns>
107     public object Create(object parent, object configContext, XmlNode section)
108     {
109         if (section != null)
110         {
111             XmlNodeList parsers = ((XmlElement)section).GetElementsByTagName(ParserElementName);
112             foreach (XmlElement parserElement in parsers)
113             {
114                 string parserTypeName = GetRequiredAttributeValue(parserElement, TypeAttributeName, sect
115                 string xmlNamespace = parserElement.GetAttribute(NamespaceAttributeName);
116                 string schemaLocation = parserElement.GetAttribute(SchemaLocationAttributeName);
117
118                 Type parserType = TypeResolutionUtils.ResolveType(parserTypeName);
119                 NamespaceParserRegistry.RegisterParser(parserType, xmlNamespace, schemaLocation);
120             }
121         }
122     }
```

看注释说，这个方法没啥用，不管了。

继续

```
/// </exception>
/// <exception cref="Spring.Objects.ObjectsException">
/// In the case of errors encountered reading any of the resources
/// yielded by either the <see cref="ConfigurationLocations"/> or
/// the <see cref="ConfigurationResources"/> methods.
/// </exception>
protected virtual void LoadObjectDefinitions(XmlObjectDefinitionReader objectDefinitionRe
{
    string[] locations = ConfigurationLocations;
    if (locations != null)
    {
        objectDefinitionReader.LoadObjectDefinitions(locations);
    }

    [Resource] resources = ConfigurationResources;
    if (resources != null)
    {
        objectDefinitionReader.LoadObjectDefinitions(resources);
    }
}
```

在这里方法里面，里面的层级非常深，以前跟踪进去过。代码很复杂，就不跟踪进去了。我知道的是，他会装载所有的配置的 object 对象

继续，代码最后回到这里

```

1 {
2 }
3 protected override ConstructorInfo GetContextConstructor()
4 {
5     return ContextType.GetConstructor(new Type[] { typeof(string), typeof(bool), typeof(string[]) });
6 }
7
8 protected override IApplicationContext InvokeContextConstructor(
9     ConstructorInfo ctor)
10 {
11     return (IApplicationContext)(new SafeConstructor(ctor).Invoke(new object[] { ContextName, CaseSensitive, Resources }));
12 }
13
14 endregion
15

```

当所有 object 对象装载完毕以后，那么这里一个 xmlApplicationContext 实例就构造出来了，转换成 IApplicationContext，返回

```

    _resources = resources;
}

public IApplicationContext InstantiateContext()
{
    ConstructorInfo ctor = GetContextConstructor();
    if (ctor == null)
    {
        string errorMessage = "No constructor with string[] argument found for context";
        throw ConfigurationUtils.CreateConfigurationException(errorMessage);
    }
    IApplicationContext context = InvokeContextConstructor(ctor);
    return context;
}

```

看到这里，context'容器对象被构造出来了。紧接着，无非是堆栈地址弹出一返回。当返回到 context 配置节的时候，会检测子节点有无 context，如果有，那么重复以上过程，构造子容器对象，并装填子容器管理的对象 Objects，然后返回。

在子容器对象创建的时候，父子容器管理的数据结构是一棵树结构，可以层层深入，并且这些容器对象都被注册在一个容器字典里，hash 查找非常之快，不同容器内的对象互不影响。而且这种容器管理结构线程安全。内部在创建构造器，资源解析器，还有对象装填，均使用了缓存。效率也不会低。

一步步跟踪了 spring 容器的创建代码。各种设计模式的灵活使用，缓存和同步方法的使用，在安全，高效率的前提下保证了巧妙和灵活。

兴许看了源码，你才能体会到 spring 真正精髓所在。

这种代码，我此生都难以企及。。